Imperial College London
Department of Computing

Reasoning About POSIX File Systems

Gian Ntzik

September 2016

Supervised by Philippa Gardner

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London
and the Diploma of Imperial College London

Declaration

I herewith certify that all material in this thesis which is not my own work has been properly acknowledged.

Gian Ntzik

Copyright

Abstract

POSIX is a standard for operating systems, with a substantial part devoted to specifying file-system operations. File-system operations exhibit complex concurrent behaviour, comprising multiple actions affecting different parts of the state: typically, multiple atomic reads followed by an atomic update. However, the standard's description of concurrent behaviour is unsatisfactory: it is fragmented; contains ambiguities; and is generally under-specified. We provide a formal concurrent specification of POSIX file systems and demonstrate scalable reasoning for clients. Our specification is based on a concurrent specification language, which uses a modern concurrent separation logic for reasoning about abstract atomic operations, and an associated refinement calculus. Our reasoning about clients highlights an important difference between reasoning about modules built over a heap, where the interference on the shared state is restricted to the operations of the module, and modules built over a file system, where the interference cannot be restricted as the file system is a public namespace. We introduce specifications conditional on *context invariants* used to restrict the interference, and apply our reasoning to lock files and named pipes.

Program reasoning based on separation logic has been successful at verifying that programs do not crash due to illegal use of resources, such invalid memory accesses. The underlying assumption of separation logics, however, is that machines do not fail. In practice, machines can fail unpredictably for various reasons, such as power loss, corrupting resources or resulting in permanent data loss. Critical software, such as file systems and databases, employ recovery methods to mitigate these effects. We introduce an extension of the Views framework to reason about programs in the presence of such events and their associated recovery methods. We use concurrent separation logic as an instance of the framework to illustrate our reasoning, and explore programs using write-ahead logging, such a stylised ARIES recovery algorithm.

To my loved ones forever lost,
but in my heart forgotten not.

Contents

1. Introduction **19**

1.1. Contributions and Dissertation Outline . . . 21

2. File Systems and POSIX **23**

2.1. File Systems . . . 23

2.1.1. Structure and Implementation . . . 24

2.2. The POSIX Standard . . . 27

2.2.1. POSIX File-System Structure and Interface Overview . . . 28

2.3. Concurrency in POSIX File Systems . . . 32

2.3.1. Thread Safety and Atomicity . . . 32

2.3.2. Path Resolution and Client Applications . . . 36

2.3.3. Atomicity of directory operations . . . 38

2.3.4. Non-determinism . . . 39

2.3.5. File Systems are a Public Namespace . . . 39

2.4. Conclusions . . . 40

3. Reasoning about Concurrent Modules **42**

3.1. Separation Logic . . . 42

3.2. Concurrent Separation Logic . . . 44

3.3. Abstract Separation Logic and Views . . . 45

3.4. Fine-grained Concurrency . . . 46

3.5. Atomicity . . . 47

3.5.1. Contextual Refinement and Separation Logic . . . 48

3.5.2. Atomic Hoare Triples . . . 49

3.6. Conclusions . . . 51

4. Formal Methods for File Systems **52**

4.1. Model Checking . . . 52

4.2. Testing . . . 52

4.3. Specification and Refinement to Implementation . . . 53

4.4. Program Reasoning . . . 53

5. Modelling the File System **55**

6. Concurrent Specifications and Client Reasoning **59**

6.1. Specifications . . . 59

6.1.1. Operations on Links . . . 59

6.1.2. Operations on Directories . . . 68
6.1.3. I/O Operations on Regular Files . . . 70
6.2. Client Reasoning . . . 75
6.3. Extending Specifications . . . 84
6.3.1. Symbolic Links and Relative Paths . . . 84
6.3.2. File-Access Permissions . . . 87
6.3.3. Threads and Processes . . . 90
6.4. Conclusions . . . 91

7. Atomicity and Refinement **93**
7.1. Specification Language . . . 93
7.2. Model . . . 97
7.3. Operational Semantics . . . 105
7.4. Refinement . . . 107
7.4.1. Contextual Refinement . . . 107
7.4.2. Denotational Refinement . . . 108
7.4.3. Adequacy . . . 111
7.5. Refinement Laws . . . 112
7.6. Abstract Atomicity . . . 117
7.7. Syntax Encodings . . . 129
7.8. Conclusions . . . 130

8. Client Examples **132**
8.1. CAP Style Locks . . . 132
8.2. Coarse-grained vs Fine-grained Specifications for POSIX . . . 134
8.2.1. Behaviour with Coarse-grained Specifications . . . 135
8.2.2. Behaviour with POSIX Specifications . . . 140
8.2.3. Correcting the Email Client-Server Interaction . . . 149
8.3. Case Study: Named Pipes . . . 156
8.3.1. Specification . . . 156
8.3.2. Implementation . . . 159
8.3.3. Verification . . . 166
8.4. Conclusions . . . 187

9. Fault Tolerance **188**
9.1. Faults and Resource Reasoning . . . 188
9.2. Motivating Examples . . . 190
9.2.1. Naive Bank Transfer . . . 190
9.2.2. Fault-tolerant Bank Transfer: Implementation . . . 190
9.2.3. Fault-tolerant Bank Transfer: Verification . . . 191
9.3. Journaling File Systems . . . 196
9.4. FTCSL Program Logic . . . 196
9.4.1. Unsound Rules . . . 199

9.5. Example: Concurrent Bank Transfer . . . 199
9.6. Semantics and Soundness . . . 202
9.6.1. Fault-tolerant Views . . . 202
9.6.2. Fault-tolerant Concurrent Separation Logic . . . 202
9.7. Case Study: ARIES . . . 203
9.8. Related Work . . . 215
9.9. Conclusions . . . 216

10. Conclusions **217**
10.1. Future Work . . . 217
10.1.1. Mechanisation and Automation . . . 218
10.1.2. Total Correctness . . . 218
10.1.3. Helping and Speculation . . . 218
10.1.4. File-System Implementation and Testing . . . 219
10.1.5. Fault-tolerance of File Systems . . . 219
10.1.6. Atomicity? Which Atomicity? . . . 219

A. POSIX Fragment Formalisation **228**
A.1. Path Resolution . . . 230
A.2. Operations on Links . . . 230
A.3. Operations on Directories . . . 234
A.4. I/O Operations on Regular Files . . . 235
A.5. I/O Operations on Directories . . . 236

B. Atomicity and Refinement Technical Appendix **238**
B.1. Adequacy Addendum . . . 238
B.1.1. Operational traces are denotational traces . . . 246
B.1.2. Denotational traces are operational traces . . . 249
B.2. Proofs of General Refinement Laws . . . 256
B.3. Primitive Atomic Refinement Laws Proofs . . . 268
B.4. Proofs of Abstract Atomic Refinement Laws . . . 274
B.5. Proofs of Hoare-Statement Refinement Laws . . . 288

C. Fault-Tolerant Views Framework **291**
C.1. General Framework and Soundness . . . 291
C.2. FTCSL . . . 298

List of Tables

2.1. Atomic POSIX file-system operations. 33

6.1. A selection of flags controlling I/O behaviour. 71
6.2. File-access permission bits. 87

List of Figures

2.1. Example of a file-system tree structure. . . . 25
2.2. Example of a file-system acyclic graph structure. . . . 26
2.3. Example of a file-system cyclic graph structure. . . . 26
2.4. Example of a file-system acyclic graph structure with a symbolic link. . . . 27
2.5. Example of "." and ".." links in the file-system structure. . . . 29
2.6. Example of `unlink`(/usr/bin/git) when file-system is not modified in parallel. . . . 32
2.7. Evolution of the file-system state during `unlink`(/usr/bin/git). . . . 34
2.8. Example of concurrent `unlink`(/usr/bin/git). . . . 35
2.9. Example where `unlink`(/usr/bin/git) succeeds, even when the path never actually exists. 37
2.10. Unsafe email client-server interaction. . . . 38

6.1. Example snapshot of the file-system graph. . . . 60
6.2. Specification of atomic operations for link lookup, insertion and deletion. . . . 63
6.3. Specification of error cases in operations on links. . . . 63
6.4. Specification of `rename`. . . . 66
6.5. Specification of atomic link moving operations. . . . 67
6.6. Additional error case specifications for `rename`. . . . 68
6.7. Specification of atomic operations that create and open regular files. . . . 72
6.8. Specification of atomic write actions. . . . 72
6.9. Specification of heap operations. . . . 75
6.10. Proof sketch of `lock` refining its specification. . . . 80
6.11. Refinement of `open`(lf, `O_CREAT|O_EXCL`) used in `lock`(lf). . . . 81
6.12. Path resolution specification accounting for symbolic links. . . . 85

7.1. Behaviour of a program satisfying the general form of the atomic Hoare triple. . . . 118

8.1. Proof that `lock` satisfies the CAP-style specification. . . . 134
8.2. Proof that `unlock` satisfies the CAP-style specification. . . . 134
8.3. Unsafe email client-server interaction. . . . 135
8.4. Coarse-grained specification for the operations used in figure 8.3. . . . 136
8.5. Proof of email client-server interaction using a coarse-grained file-system specification. 137
8.6. Email client refinement using a coarse-grained file-system specification. . . . 138
8.7. Proof of email server assuming a coarse-grained specification. . . . 139
8.8. POSIX specification of `stat`. . . . 140
8.9. Proof of email client-server interaction using the POSIX file-system specification. . . . 142
8.10. Derivation of the email client's specification using the POSIX file-system specification. 143
8.11. Derivation of specification for client's `resolve`(/mail/42, ι_0). . . . 144

8.12. Derivation of email server's specification using the POSIX file-system specification. . . 145
8.13. Derivation of server's `mkdir`(/mail/42) using the POSIX file-system specification. . . . 146
8.14. Derivation of server's `rename`(/mail/tmp/42.msg, /mail/42/unsafe.eml) using the POSIX file-system specification 147
8.15. Derivation of specification statement for `resolve`(/mail/tmp/42.msg, ι_0) used in email server's `rename`(/mail/tmp/42.msg, /mail/42/unsafe.eml). 148
8.16. Derivation of specification statement for `resolve`(/mail/42, ι_0) used in email server's `rename`(/mail/tmp/42.msg, /mail/42/unsafe.eml). 148
8.17. A safe email client-server interaction. 149
8.18. Proof of safe email client-server interaction assuming $\mathsf{EmlCtx}(\iota, j, k)$. 152
8.19. Refinement to the safe email client assuming $\mathsf{EmlCtx}(\iota, j, k)$. 153
8.20. Refinement to the safe email server assuming $\mathsf{EmlCtx}(\iota, j, k)$. 155
8.21. Specification of atomic named-pipe operations. 157
8.22. Specification of named-pipe specific error cases. 157
8.23. Implementation of `fifo_open`. 161
8.24. Specification of POSIX `pwrite` and `pread` 162
8.25. Implementation of `fifo_read`. 163
8.26. Implementation of `fifo_write`. 165
8.27. Naped-pipe header and data predicates. 166
8.28. Intermediate refinement of `mkfifo`, replacing `open` with its specifcation. 169
8.29. Proof of `mkfifo` satisfying its specification. 170
8.30. Refinement to `write_fifo_header`. 171
8.31. Proof of `fifo_open` satisfying its specification. 172
8.32. Proof sketch of the `O_RDONLY` branch of `fifo_open`. 173
8.33. Proof of `fifo_incr_readers` abstraction. 175
8.34. Proof of `fifo_mk_desc` abstraction. 176
8.35. Proof of `fifo_read` in the case of `fifo_read_nowriters`. 178
8.36. Proof of `fifo_get_mode` and `fifo_get_fd` abstractions. 179
8.37. Proof of `fifo_get_name` abstraction. 179
8.38. Proof of `fifo_get_writers` abstraction. 180
8.39. Derivation of specification for `fifo_is_empty`. 180
8.40. Proof of `fifo_read` in the case of `fifo_read_partial`. 182
8.41. Proof of `fifo_read_contents` abstraction in the case of a partial read. 183
8.42. Proof of `fifo_write` in the case of available readers. 185
8.43. Proof of `fifo_write_contents` abstraction. 186

9.1. Specification of a simplified journaling file system. 192
9.2. Proof of transfer operation using write-ahead logging. 193
9.3. Proof that the transfer recovery operation guarantees atomicity. 195
9.4. Selected proof rules of FTCSL. 198
9.5. Sketch proof of two concurrent transfers over the same accounts. 201
9.6. Abstract model of the database and ARIES log, and predicates. 204
9.7. ARIES recovery: high level structure. 204

9.8. Proof of the high level structure of ARIES recovery. 211
9.9. Implementation and proof of the ARIES analysis phase. 212
9.10. Implementation and proof of the ARIES redo phase. 213
9.11. Implementation and proof of the ARIES undo phase. 214

Acknowledgements

Foremost I am immensely grateful to my supervisor, Philippa Gardner, for giving me the opportunity to pursue this research and for her guidance, support and continued encouragement. I am equally grateful to my academic brother and collaborator Pedro da Rocha Pinto for his constant support, advice and insightful discussions that have been invaluable through the course of this research. I would also like to thank all my colleagues in our Imperial College research group for creating an amazing environment to work with.

My research would not have been possible without the support of my family and my friends. In particular, I would like to thank my parents and my brother for their unconditional support in pursuing my interests. I am grateful to my dear friends Manolis Michalas, Vaso Satafida and Aris Georgakopoulos for their continued encouragement and believing in me. I am equally grateful to Aggelos Biboudis for his friendship, sound advice and his calming influence. I would also like to thank all my former colleagues in Nessos IT for their continued support and especially Nick Palladinos for his mentorship and inspiration.

Finally, I am indebted to my dear friends John Mavrantonakis, Irene Schinaraki and George Zargiannakis for their moral support and help in the most difficult of times. Thank you for keeping me going.

1. Introduction

POSIX is a standard developed for maintaining compatibility between multi-process operating systems [7]. A substantial part of POSIX is devoted to specifying the interface for concurrent file systems. The English standard is mature, and widely implemented in operating systems and software libraries. However, its description of concurrent behaviour is unsatisfactory: it is fragmented; contains ambiguities; and is generally under-specified.

Many formal specifications of POSIX file systems have been proposed [11, 52, 71, 44, 45, 83, 26]. Most of these works focus on implementation reasoning by refining the specification to a verified implementation. Only recently, specifications based on separation logic [81] have been proposed for scalable reasoning about POSIX file-system clients [47, 72]. However, all the aforementioned specifications simplify concurrency, either by restricting to sequential fragments or by taking a coarse-grained view of concurrency that is equivalent to sequential behaviour.

The POSIX standard describes complex behaviour for the concurrent access of file systems. Although poorly described in the standard, there is a consensus between major file-system implementations on the intentions of the standard with respect to concurrency. File-system operations (such as unlinking files) typically traverse paths to identify the files or directories on which they will act. Path traversal comprises a sequence of atomic reads[1], each looking up a component of the path within a directory. Other operations exhibit more complex behaviour if, for example, they need to resolve multiple paths. Since POSIX does not specify the order in which multiple paths are resolved, the atomic reads of multiple path traversals can be arbitrarily interleaved. After the path resolution, other atomic actions perform the intended update of the file-system operation. In summary, file-system operations are not atomic, but sequential and parallel combinations of atomic actions.

If one simplifies the operations as being atomic, this suggests additional synchronisation on file-system accesses. This may be true for some specific implementations, or if the client explicitly adds synchronisation, but it does not hold in general. In fact, in most major file-system implementations path resolution is not atomic. Therefore, such simplified specifications carry unverified assumptions and are not reliable for client reasoning.

We provide a concurrent specification of POSIX file systems and demonstrate scalable reasoning for clients. To tackle the complexity of the concurrent behaviour, we specify operations with *specification programs*, inspired by refinement calculi [13]. We introduce a concurrent specification language and an associated refinement calculus, combined with a separation logic for fine-grained concurrency and *abstract atomicity*. The specification language provides a mechanism for specifying sequential and parallel combinations of atomic actions and the refinement calculus allows reasoning about clients and implementations of a specification. Our approach is inspired by the earlier work of Turon and Wand [94], which introduced a combination of a refinement calculus and separation logic for reasoning about atomicity. However, in their work, an operation is proven to be atomic with respect to all

[1] Atomic in the sense of *linearisability* [51], where operations appear to take effect at a single discrete point in time.

possible contexts. We demonstrate that this type of reasoning does not work for file systems. Recent developments in separation logics for fine-grained atomicity focus on abstract atomicity [32], where operations can be proven to be atomic only for some contexts. We combine our specification language and refinement calculus with TaDA [30, 28], a separation logic for abstract atomicity. Using this combination we give the first formal specification of POSIX file-system operations that properly captures their complex concurrent behaviour, and reason about client examples including a lock-file and named-pipe client module.

Reasoning about POSIX clients is subtle. Modules built over a heap typically restrict the interference on the shared resource they encapsulate, by only allowing access to the resource via the module operations. This is not the case with modules built over a file system. A file system is a public namespace of the operating system, allowing any process to access any path of its choosing regardless of whether it is used by other processes for their own purposes. File access permissions can only enforce restrictions to sets of processes. Therefore, a module that expects certain files to exist at certain paths can only do so when all of the processes (the context or environment) co-operate in maintaining the module's file-system invariants. For example, consider a lock-file module, whose operations insert (lock) and remove (unlock) a file in a directory identified by a path. The lock-file module is not able to restrict the interference on the path or the file. Therefore we develop specifications conditional on *context invariants* which restrict what interference is possible. In the case of the lock-file module, the environment cannot change the path to the lock file and only the module operations can be used to lock and unlock the lock. In this dissertation, we mainly study two examples: lock files and named pipes. Lock files provide a simple example to introduce context invariants. Named pipes provide a more complex example to demonstrate the scalability of our reasoning, building on our specification of lock files.

The primary purpose of file systems is to persist data to some persistent storage medium such as a hard disk. However, file-system updates that from the point of view of the client are atomic are actually implemented in terms of multiple updates to the storage medium. Host failures such as a power loss may occur at any point in time during the execution of an update to the state of the file system, which means that the update may not be fully committed to persistent storage. This may lead to the file system state becoming corrupt and to the loss of application data. For this reason, file system implementations strive to recover from such unavoidable events and provide fault-tolerance guarantees of some form. The same is true for database systems and practically for any application that considers data integrity to be critical.

Program logics based on separation logic have been successful in reasoning about sequential fragments of file systems [47, 72] and concurrent indexes [29] on which databases and file systems depend. However, resource reasoning, as introduced by separation logic [81], is a method for verifying that programs do not fail. A triple $\{P\}\,\mathbb{C}\,\{Q\}$ is given a *fault-avoiding*, partial correctness interpretation. This means that, assuming the precondition P holds then, if program $\mathbb{C}$ terminates, it must be the case that P does not fail and has all the resource necessary to yield a result which satisfies postcondition Q. Such reasoning guarantees the correct behaviour of the program, ensuring that the software does not crash itself due to bugs, e.g. invalid memory access. However, it assumes that there are no other failures of any form. To reason about programs that can recover from host failures, we must change the underlying assumptions of resource reasoning.

We swap the traditional resource models with one that distinguishes between *volatile* and *durable* resource: the volatile resource (e.g. in RAM) does not survive crashes; whereas the durable resource (e.g. on the hard drive) does. Recovery operations use the durable state to repair any corruptions caused by the host failure. We develop a general reasoning framework, by extending the Views framework [35], for reasoning about programs in the presence of host failures and their associated recovery operations. We introduce a new fault-tolerant Hoare triple judgement of the form:

$$S \vdash \{P_V \mid P_D\}\, \mathbb{C}\, \{Q_V \mid Q_D\}$$

which has a partial-correctness, *resource fault-avoiding* and *host-failing* interpretation. From the standard resource fault-avoiding interpretation: assuming the precondition $P_V \mid P_D$ holds, where the volatile state satisfies P_V and the durable P_D, then if $\mathbb{C}$ terminates and there is no host failure, the volatile and durable resource will satisfy Q_V and Q_D respectively. From the host-failing interpretation: when there is a host failure, the volatile state is lost, and after potential recovery operations, the remaining durable state will satisfy the *fault-condition* S. We instantiate our framework with a fault-tolerant extension of concurrent separation logic [76] and reason about a simplified ARIES recovery algorithm [69], widely used in database systems.

1.1. Contributions and Dissertation Outline

This dissertation contains three main contributions. First, we provide a formal specification of the concurrent behaviour of POSIX file systems, capturing the complex concurrent behaviour found in real-world implementations and described poorly in the standard. We hope that our formalisation will be useful for future revisions of the POSIX standard. We expect that our specification of sequential and parallel combinations of atomic actions will be useful for other examples, such as iterators in `java.util.concurrent`. Second, we highlight an important difference between reasoning about modules built over a heap, where the interference on the shared state is restricted to the operations of the module, and modules built over a file system, where the interference cannot be restricted as the file system is a public namespace. We develop specifications conditional on context invariants to restrict the interference, and apply our reasoning to the examples of lock files and named pipes. Third, we extend separation-logic style reasoning to reason about programs in the presence of host failures, and verify fault-tolerance properties.

- In chapter 2, we give an overview of file systems and their standardisation by POSIX. We analyse the concurrent behaviour of POSIX file systems as specified informally by the standard and identify two fundamental challenges for their formal specification: file-system operations perform complex sequences of atomic steps; and file systems are a public namespace.

- In chapter 3, we give an overview of various methods for reasoning about concurrent programs and modules in the literature. We focus primarily on concurrent separation logics and atomicity specifications, and explore the applicability of each approach to a formal specification of POSIX file systems and their concurrent behaviour.

- Chapter 4 gives an overview of related work in the formal specification and verification of file

systems from the fields of model checking, specification and refinement to implementation, testing and program logics.

- In chapter 5 we introduce our formal model of the POSIX file-system structure and definitions of associated types such as paths and error codes that subsequent chapters will be using.

- In chapter 6 we introduce our formal specification of POSIX file systems that accounts for the complex concurrent behaviour informally specified in the POSIX standard. We use examples of key file-system operations to introduce important features of our specification language, its refinement calculus and demonstrate how we formalise the behaviour of POSIX file-system operations. We then introduce how our approach is used for client reasoning by specifying and verifying a lock-file module. In this example, we introduce specifications conditional on context invariants that serve to restrict the interference from the environment on the file system resources used by the module's implementation, addressing the challenge of the file system being a public namespace. Additionally, we demonstrate several examples of how our formal specifications can be extended to larger fragments of POSIX file systems, thus demonstrating the flexibility of our specification approach.

- Chapter 7 formalises our specification language, its semantics, the refinement calculus and soundness proof. We define a core specification language in which the primitive statements specify primitive atomic actions regardless of interference from the environment. We define general refinement laws for our specification language and specific refinement laws for primitive atomicity. Within this core specification language and refinement calculus we define atomic specification statements for abstract atomicity in the style of the TaDA program logic[30, 28] as a derived construct, and give refinement laws for abstract atomicity.

- In chapter 8 we apply our reasoning to more file-system client examples. We study an example of a concurrent email client and server interaction, demonstrating that formal specifications that simplify the concurrent behaviour of file systems lead to unsafe client reasoning. As a case study of our reasoning we use an implementation of named pipes within our core POSIX file-system fragment and verify its correctness.

- In chapter 9 we extend resource reasoning, as introduced by separation logic, to account for host failures and recovery operations. We develop a general framework for reasoning about concurrent programs in the presence of host failures by extending the Views framework [35]. In this chapter we introduce fault-tolerant concurrent separation logic as instance of this framework and reason about the fault-tolerance of simple bank account transfer and fault-tolerance properties of a stylised ARIES recovery algorithm.

- Chapter 10 summarises our contributions and presents interesting directions for future work.

Collaboration

Chapters 5, 6, 7 and 8 are based on joint work with da Rocha Pinto and Gardner. Chapter 9 is heavily based on work done in collaboration with da Rocha Pinto and Gardner, previously published in *Fault-tolerant resource reasoning* [73].

2. File Systems and POSIX

We give an overview of file systems and their standardisation by POSIX. In section 2.1, we give an overview of variations in the structure of file systems and their implementations. In section 2.2, we discuss the POSIX standard and its informal file-system specification. Finally, in section 2.3, we discuss how POSIX specifies the concurrent behaviour of file systems and identify the fundamental challenges to its formal specification.

2.1. File Systems

Originally, the term "file system" referred to a method for organising storage and access to paper documents using filing cabinets in offices. With the advent of computing, systems for organising digital information stored and processed in computers were developed. These systems adopted the filing cabinet metaphor. Related pieces of information, in the form of binary data, are organised in distinct groups. Following the filing cabinet metaphor, each group of binary data is called a *file*. Each file is given a name, commonly referred to as a *filename*, by which it can be easily identified. Typically, files are further organised into a hierarchy of *directories* or, in filling cabinet terms, folders. Each directory may contain other files or directories. In the field of computing, a *file system* is the method for organising the storage of files and directories and access to their data [90].

File systems are ubiquitous. They are a critical component of an operating system. General purpose operating systems for desktop and server systems, such as Mac OS X, Linux and Windows, and special purpose operating systems for mobile and embedded device rely heavily on file systems. User and application data, the executable code of applications and even the executable code of the operating system are all stored in a file system. Most modern operating systems are multi-user and multi-process, meaning that many applications and users can access the file system concurrently. Since one of the primary roles of a file systems is to persist data in a storage medium, many file systems strive to preserve the integrity of their structure and the data stored in files in the event of hardware and software failures.

In effect, a file system is an abstraction over how data is organised, stored and accessed in a storage medium. Typically this abstraction is achieved with a layered approach. The term "file system" is often used to refer to two related but different layers of this abstraction. The first layer defines the hierarchical structure of files and directories that is visible by the users of the file system. Furthermore, this layer defines the application programming interface (API) through which users access this structure and the data stored within. The second layer is the actual implementation of this structure, including the data structures, algorithms and means to access the physical storage medium. Henceforth, to avoid ambiguity we use the terms *file-system structure* and *file-system interface* to refer to the hierarchical structure and programming interface of the first layer, and the term *file-system implementation* to refer to the second layer.

There are many different file-system structure, interfaces and implementations. File systems in early operating systems for microcomputers and personal computers had a flat file-system structure, where all files are contained within a single directory. Today, practically all major operating systems for personal computers use hierarchical file-system structures which allow nested directories. Even so, hierarchical file-system structures come in different flavours: plain trees, directed acyclic graphs, or general directed graphs. File-system interfaces differ in the behaviour of operations accessing the file-system structure, especially with respect to concurrent accesses and security policies. Even so, file-system interfaces derived from the UNIX family of operating systems are to a large extend compatible, which led to their standardisation by POSIX [7]. File-system implementations vary greatly in the data structures and algorithms, with choices typically governed by considerations on performance and fault tolerance. Some file-system implementations are designed for specific storage media. For example, the ISO 9660 file-system implementation was developed specifically targeting optical discs [12].

In this dissertation we focus on the file-system structure and interface. In particular, we study the file-system structure and interface specified by prose in POSIX.

2.1.1. Structure and Implementation

We now discuss file-system structures and related implementations in more detail. We base our descriptions on the file-system structures traditionally found in the UNIX family of operating systems, such in Linux, Mac OS X and FreeBSD, as well as in the POSIX standard. The details may differ to file-system structures found in other operating systems, such as Windows, however most of them paint a similar picture.

Intuitively, when as human users we explore the file-system structure by using a file manager application, such as the Windows Explorer or the Mac OS X Finder, we perceive the hierarchical structure of the file system as a tree. Each file and directory has a name, a directory contains other directories or files, and files are the leaf nodes of the hierarchy. However, this is only a high level view presented to users by applications hiding several additional details of the file-system structure.

In the hierarchical file-system structure every node is a file. Each file contains data and is associated with metadata. File data are stored as sequences of bytes. File metadata are additional attributes and properties of the file and data it stores. A typical example of file metadata are the size of a file, timestamps recording its creation and last modification, and file access permissions that control if and how users can access a file. However, the most important pieces of metadata are the *unique identifier* and *file type* of a file.

In file systems developed for operating systems derived from UNIX, and consequently POSIX implementations, metadata attributes are managed via the *inode* structure [87, 90]. From the point of view of the file-system implementation, an inode represents any node in the file-system structure. In addition to metadata, it manages information about how the data is stored in the storage medium, such as a hard disk. Typically, the storage medium organises data into blocks of a fixed size. The inode structure for a file maintains information about how file data are mapped to blocks in the storage medium. Each inode structure has an address, a unique identifier typically referred to as an *inode number*, which remains unchanged during the lifetime of the file the inode structure represents. Essentially, an inode number uniquely identifies both a file and its associated inode structure. In POSIX, the inode number is referred to as *file serial number* ([7],XBD,3.176). In this dissertation we

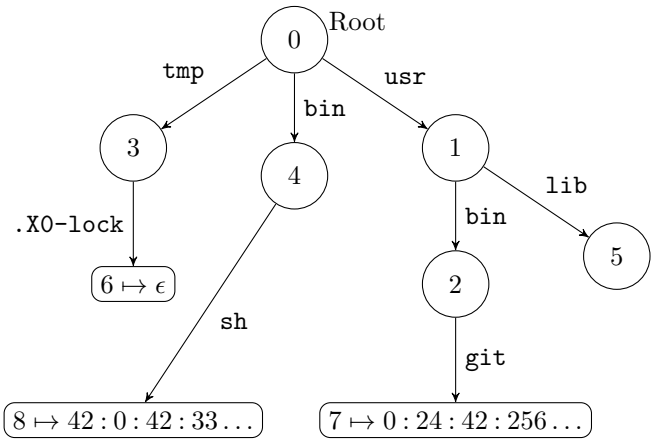

Figure 2.1.: Example of a file-system tree structure.

use the term *inode* to refer to the inode number, and the term *inode structure* to refer to the structure identified by an inode number.

File systems support various file types. The most common file types are that of a *directory file* and *regular file*. A regular file is the type of file we intuitively perceive as a "file", such as text document. The interpretation of the data stored in a regular file is up to the client applications that use them. On the other hand, directories are special types of file. The data stored in a directory file are managed and interpreted by the file-system implementation itself. A directory file stores a list of directory entries, or *links*. A link is a pair of a filename and an inode number of a file in the file system. Effectively, a link is a pointer to a file that has a name. Within each directory, the name of each link must be distinct. The implication of this organisation is that the name of file, is not part of the file itself. The name of a file is defined by the link that points to it.

Figure 2.1 shows an example of a file-system structure. Each node in this structure is a file. Directory files are depicted as circles, whereas regular files are depicted as rounded rectangles. Each file has unique identifier, the inode, which in the figure is represented by an integer. The contents of regular files are byte sequences, where each byte is represented by an integer and the empty byte sequence is written as ϵ. Directory files store links to other files, which are depicted as outgoing edges from directory nodes. The label on each edge depicts the name of the link. We elide the inode structure associated with each file and focus on the hierarchical structure induced by the directory contents.

In figure 2.1 the hierarchical structure is that of a tree, as each file is linked only once. This aligns with the general intuition we obtain from exploring the file-system structure as users: each directory contains other directories or regular files. However, several file systems allow files to be linked more than once. An example of such a file-system structure is given in figure 2.2, where the file with inode 7 has two incoming links. In this case the file actually has two names. If we were to explore the file-system structure in figure 2.2 in a file browsing application, or even a command line shell, we would observe two different entries: one named `git` within the `/usr/bin` directory with inode 2, and the other named `vc` within the `/bin` directory with inode 4. However, the two entries would be exactly the same file. Even though the names differ, and the location in which we observe them differ, they are the names of links pointing to the same file. Usually additional links to a file are called *hard links*, albeit there is no semantic difference between hard links and links. Typically, the term is used to indicate the presence of multiple links to a file.

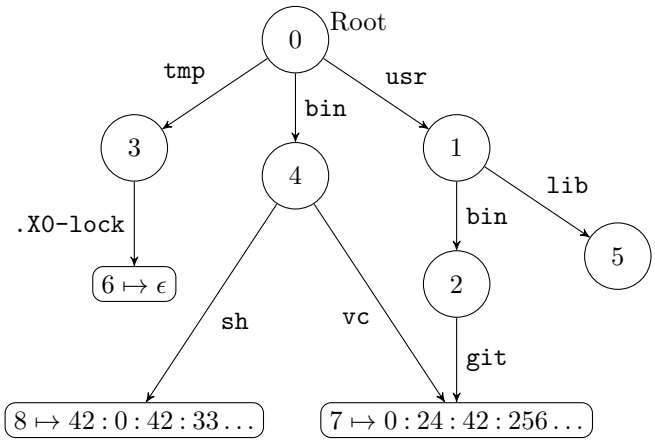

Figure 2.2.: Example of a file-system acyclic graph structure.

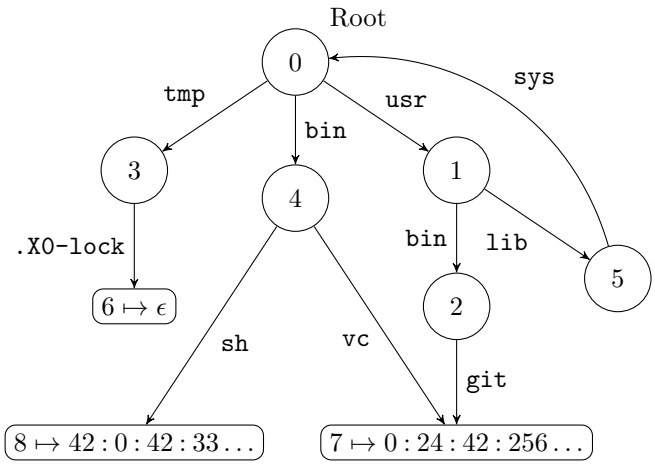

Figure 2.3.: Example of a file-system cyclic graph structure.

When the file system allows files to be linked more than once, the file-system hierarchical structure is that of a directed graph. In figure 2.2 this directed graph is acyclic, since only regular files are linked more than once. However, hard links to directories may introduce cycles in the graph as in the example shown in figure 2.3. Such loops are highly problematic, especially for applications that recursively traverse the file-system structure. It is not possible to differentiate hard links pointing to the same file, thus in the presence of loops such applications will never terminate their traversal. For this reason, most modern operating systems severely restrict the use of hard links to directories, or forbid them altogether, even if a file-system implementation supports it [87]. For example, Linux systems prohibit users from creating hard links to directories [4].On the other hand, directory hard links are useful for certain applications, if used with extra care. For example, starting from version 10.5 Leopard, Mac OS X utilises hard links to directories in the Time Machine backup software [2].

Another important file type supported by many file systems is that of a *symbolic link*. Symbolic links are another method for creating additional names to files. The contents of a symbolic link file are a path: a /-separated sequence of filenames. The purpose of this path is to act as a "pointer" to another file. An example of a file-system structure containing a symbolic link file is given in figure 2.4. The symbolic link file with inode 9 is depicted as a square rectangle containing the path `/usr/bin`. When an application attempts to access the symbolic link file by following the link named `app`, the file

system redirects the access to the file addressed by the path `/usr/bin`. In contrast to normal links, symbolic links are allowed to dangle: the path stored in a symbolic link is not required to be valid.

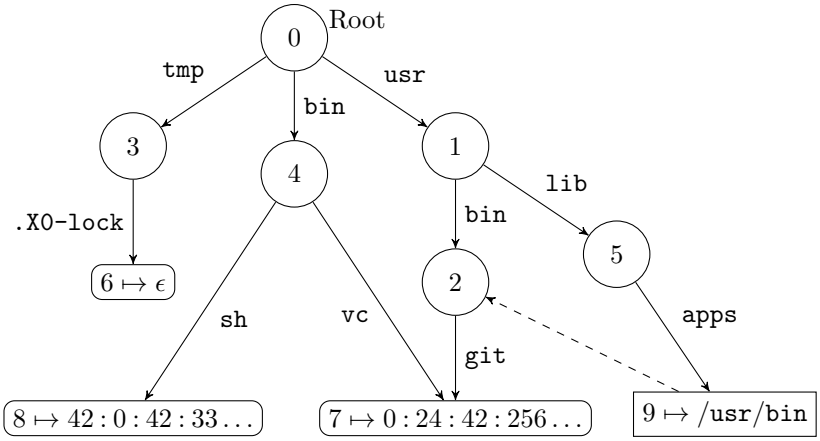

Figure 2.4.: Example of a file-system acyclic graph structure with a symbolic link.

It is the responsibility of a filesystem implementation to manage the filesystem structure in the terms of directory entries and inodes that we have just described. It is also responsible for physically storing a file in a storage medium. Different implementations employ different techniques and algorithms to maintain the filesystem structure as well as different physical stores. For example, ReiserFS internally uses a B+ tree structure [80], while FAT relies on an Allocation Table [5, 6, 1], tmpfs utilises the computer RAM as a storage medium [86], while ext2,3,4 typically use a hard disk [67].

2.2. The POSIX Standard

POSIX (Portable Operating System Interface) is a standard for maintaining compatibility between different operating systems [7]. The standard specifies the environment of the operating system, including its programming interface, the command line shell and utility applications. The aim of POSIX is to maximise the source code portability of client applications across different POSIX implementations. Features found in the UNIX family of operating systems form the basis of the standard, as it was developed out of the need to standardise the common set of features from the variety of operating systems derived from UNIX [87]. However, POSIX is not limited to UNIX-like systems. Other operating systems, such as Microsoft Windows (some versions) and Plan 9, provide optional compatibility sub-systems and libraries [3, 92].

Originally published in 1988, as IEEE Std 1003.1-1988, the standard has been through several revisions. The latest version of the standard, the 2016 edition of IEEE Std 1003.1-2008[1], is freely available online. It is maintained and developed for future revisions by the Austin Group, a joint working group between the IEEE, The Open Group and the ISO/IEC JTC 1 organisations [8]. The intended audience of POSIX is both application developers and implementers of operating systems.

The POSIX standard documents are divided into four volumes:

- XBD: definitions of general terms, concepts and features common to all volumes.
- XSH: specifications of the behaviour of the operations comprising the POSIX API.

[1]Originally published in 2008, revised in 2013 and 2016.

- XCU: specifications for a command line interface (shell) and common utility applications.
- XRAT: rationale behind some of the design decisions in the development of the other volumes.

Not surprisingly, since they are a critical part of operating systems, POSIX devotes a significant portion of its volumes to file systems. The most relevant volumes for file systems are the XBD, for the specification of the file-system structure, and XSH, for the file-system interface. The XCU volume specifies the behaviour of several utility applications operating on the file-system structure, however these are client applications implemented using the file-system interface specified in the XSH.

Specifications are given in English prose and small examples of how the POSIX API may be used. The ramification of this is that the standard's descriptions of file-system behaviour contain ambiguities, errors and imprecision. As we will demonstrate in chapter 2.3, this is especially evident in how POSIX specifies the concurrent behaviour of the file-system interface.

Henceforth, we will frequently reference specific informal specifications in the POSIX standard to justify our claims. We do this in the following format: ([7],volume name,section), where *volume name* identifies the volume of the standard, such as XSH, and *section* identifies the section within the volume that we reference.

2.2.1. POSIX File-System Structure and Interface Overview

In this dissertation we focus on formalising the POSIX file-system structure and reasoning about applications using the file-system interface in a concurrent environment. To aid the understanding of subsequent chapters, we give an overview of the POSIX file-system structure and interface. We do not discuss every detail of the POSIX standard relating to file systems here. We give additional details when discussing the challenges of formalising the concurrent behaviour of POSIX file system in section 2.3 and when we present our formal specifications and reasoning in chapter 6.

POSIX file systems follow the structure outlined in section 2.1.1. Here, we discuss some additional POSIX-specific requirements.

Hard links to regular file are always allowed. On the other hand, implementations are allowed to restrict or forbid the creation of hard links to directories ([7],XSH,3.link). Furthermore, the specification of an *empty directory* has implications on the structure of the file-system hierarchy, according to the following definitions:

Dot ([7],XBD,3.136,4.13): A special link within a directory, named ".", linking the same directory.
Dot-dot ([7],XBD,3.137,4.13): A special link in a directory, named "..", linking its parent directory.
Empty Directory ([7],XBD,3.144): A directory that contains at most, one dot link and one dot-dot link, and has only one link to it, other than "." (if "."exists).

Additionally, new directories are created as empty directories ([7],XSH,3.mkdir).

This means that POSIX file-systems are structured as directed graphs and due to ".." links they contain cycles. An example of this is shown in figure 2.5. Note that the directories `/tmp`, `/bin` and `/lib` are, according to the POSIX definition, empty. Even though this structure is cyclic, loops introduced by "." and ".." are harmless to applications because they can be easily distinguished from all other links by name. Note that the file-system interface does not allow applications to modify the "." and ".." links ([7],XSH,3.link,3.unlink), they are managed solely by the implementation.

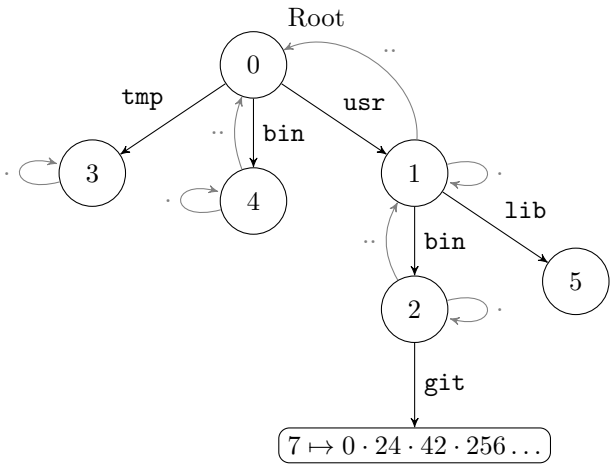

Figure 2.5.: Example of "." and ".." links in the file-system structure.

When manipulating the file-system structure, the file-system interface operations identify files by using paths.

Path ([7],XBD,3.271): A /-separated sequence of filenames. A filename is a string. Filenames include the strings "." and "..". A string is a sequence of bytes. Each filename in the path is also referred to as a *path component* ([7],XBD,3.272).

Absolute path ([7],XBD,3.2): A path that begins with a /.

Relative path ([7],XBD,3.324): A path that does not begin with a /.

Path prefix ([7],XBD,3.273): The path up to, but not including the last path component. If the path is just /, then its path prefix is also just /.

The process of following the path from some initial directory until the last filename in the path is referred to as *path resolution*.

Path resolution ([7],XBD,4.13): Each filename in the pathname is located in the current lookup directory. This means that a link named as the filename must exist in the current lookup directory. For example, in `usr/lib`, the link named `lib` is located in the directory linked by `usr`. The initial lookup directory from which path resolution starts depends on whether the path is absolute or relative. For absolute paths the initial lookup directory is the root directory of the file-system structure, whereas for relative paths it is the directory designated as the current working directory of the current process ([7],XBD,3.447). If a symbolic link is encountered during the resolution, then the remaining unresolved path is prefixed to the path contained in the symbolic link file and the process continues with the combined path. If a filename is not found in the current lookup directory, resolution fails by triggering and error.

The POSIX file-system interface is specified in the XSH volume of the standard. Each operation of the interface is defined as a C function declaration, followed by an English description of its behaviour and return values. File-system operations always return. If an operation is unable to perform its task an error code is returned. The specification of each operation includes a list of the error codes for each operations, and descriptions under which circumstances an error is triggered. Additionally, for some operations the standard includes small examples of how they can be used, as well as rationale behind some of the specification requirements. As a rule, file-system operations do not modify the file system in case of an error.

We can distinguish file-system operations into two broad categories: operations that manipulate and query the file-system structure, and operations that perform input and output (I/O) to files. A brief overview of the most basic file-system operations follows.

- `mkdir`(*path*) ([7],XSH,3.mkdir): Creates a new directory at path *path*. The operation first resolves the path prefix of *path* and if the path prefix resolves to a directory, creates a new empty directory with named by the last path component. If a link named after the last component in *path* already exists, then `mkdir` returns an error. If the path prefix of *path* does not resolve to a directory, then `mkdir` returns an error. If `mkdir` succeeds, it returns the value 0.

- `rmdir`(*path*) ([7],XSH,3.rmdir): Removes the empty directory resolved by the path *path*. If *path* does not resolve to a directory, or if the directory is not empty, then `rmdir` returns an error.

- `link`(*source*, *target*) ([7],XSH,3.link): Creates a new link at the path *target* to the file at the path *source*. The operation resolves *source* to the file for which we want to create the new link. Additionally, it resolves the path prefix of *target*, and if the path prefix resolves to a directory, `link` creates a new link named by the last component of *target* to the file resolved by *source*. If the path prefix of *target* does not resolve to a directory file, or if the path *source* does not resolve, then `link` returns an error. POSIX allows implementations of `link` to return an error if *source* resolves to a directory and the implementation does not support hard links to directories.

- `unlink`(*path*) ([7],XSH,3.unlink): Removes the link to the file at path *path*. The operation resolves the path prefix of *path*, and if it resolves to a directory, removes the link named after the last component of *path* from that directory. If *path* does not resolve to a file, then an error is returned. If the identified link is to a directory file, `unlink` is allowed to return an error, if the implementation does not support hard links to directories.

- `rename`(*source*, *target*) ([7],XSH,3.rename): Moves the link to the file at path *source*, so that it becomes a link to the same file at path *target*. The operation behaves differently depending on whether *source* resolve to a regular file, or a directory file.
  - *source* resolves to a regular file: If *target* resolves to a file, that file must also be a regular file. In that case, the link to the regular file resolved by *source*, is moved to the directory resolved by the path prefix of target, replacing the existing link. Otherwise, an error is returned. If *target* does not resolve to a file, then the link is moved to the directory resolved by the path prefix of *target*, and its name is changed to that of the last component of *target*.
  - *source* resolves to a directory file: If *target* resolve to a file, that file must be an empty directory file. In that case, the link to the regular file resolved by *source*, is moved to the directory resolved by the path prefix of target, replacing the existing link. Otherwise, an error is returned. If *target* does not resolve to a file, then the link is moved to the directory resolved by the path prefix of *target*, and its name is changed to that of the last component of *target*.

  If *source* and *target* resolve to the same file, nothing happens. If *source* does not resolve to a file, or if the path prefix of *target* does not resolve to a directory, `rename` returns an error. Note that `rename` does not physically move a file, only the link to the file.

- `stat`(*path*) ([7],XSH,3.stat): Returns metadata about the file resolve by *path*. The metadata returned includes the file type, inode number and file access permissions. If *path* does not resolve to a file, `stat` returns an error. This operation is frequently used to test the existence of a file.

- `open`(*path*, *flags*) ([7],XSH,3.open): Open the file resolved by *path* for I/O. The argument *flags* controls the behaviour of `open` and subsequent I/O operations. Depending on *flags*, if *path* does not resolve to a file, `open` can create a new empty regular file in the directory resolved by the path prefix of *path*. The operation returns a *file descriptor* to the file being opened. A file descriptor acts as a reference to the opened file ([7],XBD,3.166,3.258), and is associated with additional information controlling the behaviour of I/O operations on the file, the most important of which is the *file offset* ([7],XBD,3.172). The file offset records the byte position in a regular file from which any subsequent I/O operation begins.

- `read`(*fd*, *ptr*, *sz*) ([7],XSH,3.read): Reads at most *sz* number of bytes from the file opened with file descriptor *fd*. The read begins at the position of the file offset associated with the file descriptor. The bytes read from the file are written to the heap buffer with address *ptr* overwriting any previous contents. The heap buffer *ptr* must be of at least *sz* size. The operation will read less than *sz* number of bytes from the file, if less than *sz* bytes are stored in the file from the position of the file offset. When `read` completes, it returns the number of bytes read.

- `write`(*fd*, *ptr*, *sz*) ([7],XSH,3.write): Writes *sz* number of writes from the heap buffer *ptr* to the file opened with file descriptor *fd*. The bytes are written to the file starting at the position of the file offset associated with the file descriptor. The operation returns the number of bytes written.

- `lseek`(*fd*, *off*, *whence*) ([7],XSH,3.lseek): Modifies the file offset associated with the file descriptor *fd* according to *off* and *whence* as follows:
  - If *whence* is the flag `SEEK_SET`, then the file offset is set to *off*.
  - If *whence* is the flag `SEEK_CUR`, then the file offset is set to the current file offset plus *off*.
  - If *whence* is the flag `SEEK_END`, then the file offset is set to the size of the file plus *off*.

  The operation returns the new file offset associated with the file descriptor. `lseek` does not cause any I/O to take place on the file, it only sets the file offset. Note that the file offset may be moved to a position greater that the file's size. In that case, a subsequent `write` will extend the file. This is commonly referred to as creating a hole in the file [87], since this creates a byte range within the file that has not been written to. Bytes within this range are read as 0. Implementations are not required to physically store holes in a file.

- `close`(*fd*) ([7],XSH,3.close): Closes the file descriptor *fd*.

- `opendir`(*path*) ([7],XSH,3.fdopendir): Opens the directory at *path* for reading its contents. If *path* does not resolve to a directory, `opendir` returns an error. Otherwise, the operation returns a *directory stream*. A directory stream is an opaque structure that represents the links stored in a directory ([7],XBD,3.131). Effectively, a directory stream acts like an iterator over the contents of a directory.

- `readdir`(*dir*) ([7],XSH,3.readdir): Reads the next link from the directory stream *dir*. If there are no more links in the directory stream to read, `readdir` returns `null`.

- `closedir`(*dir*) ([7],XSH,3.closedir): Closes the directory stream *dir*.

2.3. Concurrency in POSIX File Systems

Understanding the concurrent behaviour of file systems as intended by the POSIX standard text is, in itself, a challenge. Information relevant to concurrency is fragmented throughout the text, in some cases contains ambiguities, while in other cases it is inadequate. Therefore, apart from the POSIX standard text, we also examine how the standard is interpreted in practice. The concurrent behaviour of POSIX file systems is complex. Path resolution poses a challenge with respect to atomicity guarantees. Operations that resolve paths are not atomic, but comprise several atomic actions. This has wider implications on both specification and reasoning. Furthermore, the nature of the file system as a public namespace poses a significant challenge for reasoning about client applications.

2.3.1. Thread Safety and Atomicity

The file system is a public service provided by the operating system, available for use to all processes. It is a public namespace. Therefore, all operations that access the file system are specified as *thread safe* ([7],XBD,3.407). Thread-safety guarantees that concurrent use of file-system operations always preserves the validity of the file system. File-system operations never fault; they either succeed or return an error code indicating the reason for failing.

The only exception to thread-safety of file-system operations, are few operations that move data between the file system and the process heap that may, for historical reasons, use statically allocated structures in memory, such as `readdir` ([7],XSH,3.readdir). However, even in these cases the file-system access is still safe, in that the validity of the file system is preserved. The unsafe behaviour is limited to the process heap. Furthermore, for each thread-unsafe operation, POSIX defines a thread-safe variant ([7],XRAT,A.4.18). For example, in the case of `readdir`, the thread-safe variant is `readdir_r`. For these reasons, we will not consider thread-unsafe behaviours and when discussing

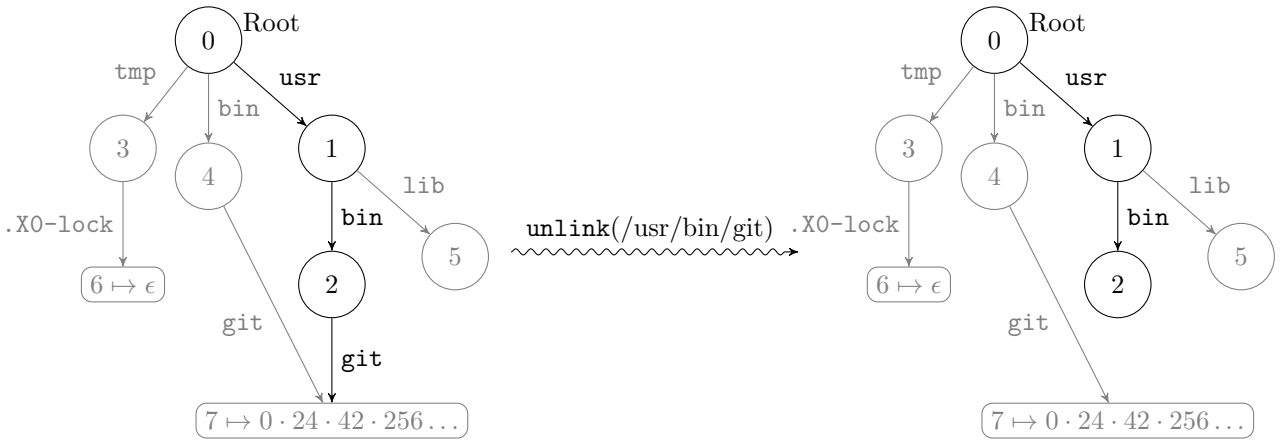

Figure 2.6.: Example of `unlink`(/usr/bin/git) when file-system is not modified in parallel.

`close`	`lseek`	`pread`	`pwrite`	`symlink`	`unlinkat`
`creat`	`lstat`	`read`	`rename`	`symlinkat`	`write`
`link`	`open`	`readlink`	`renameat`	`truncate`	
`linkat`	`openat`	`readlinkat`	`stat`	`unlink`	

Table 2.1.: List of operations specified to behave atomically on regular files and symbolic links.

any operation which may be unsafe, we limit the discussion to only the thread-safe variant of said operation.

The atomicity guarantees for POSIX file-system operations are not that simple. Meanwhile, the POSIX standard is not particularly helpful in understanding what they are. To appreciate this point, recall the English description of the `unlink` operation from section 2.2.1. If we ignore concurrency, for the moment, its behaviour is simple. In summary, according to its English specification ([7],XSH,3.unlink), `unlink`(*path*) removes the link identified by the *path* argument. For example, in the file-system structure of figure 2.6, `unlink`(/usr/bin/git) will first resolve the path /usr/bin, starting from the root directory, following the links usr and bin to their respective directories, and then it will remove the link named git from the directory with inode 2. If `unlink` is unable to resolve the path, because, for example, one of the names in the path does not exist in the appropriate directory, it returns an error. The English description for `unlink`, does not specify how the operation behaves concurrently.

In another section of the standard, POSIX defines a list of operations that must behave atomically, when operating on regular files and symbolic links ([7],XSH,2.9.7). We list some of those operations in table 2.1, from which we have excluded operations not considered in this dissertation. Note that `unlink` is included in this list. This may lead us to think that the whole process of resolving the path and removing the link to the identified file is logically indivisible, i.e. atomic. However, in other places the standard has wording to suggest otherwise.

POSIX defines a variant of `unlink`, called `unlinkat`. In summary, `unlinkat`(*fd*, *path*) is specified to behave in the same way as `unlink`(*path*), except when the *path* is a relative path, in which case the path resolution does not begin from the current working directory of the invoking process, but from the directory associated with the file descriptor *fd*. In the rationale section for `unlink` and `unlinkat` ([7],XSH,3.unlink.RATIONALE), we find the following:

> The purpose of the unlinkat() function is to remove directory entries in directories other than the current working directory without exposure to race conditions. Any part of the path of a file could be changed in parallel to a call to unlink(), resulting in unspecified behavior. By opening a file descriptor for the target directory and using the unlinkat() function it can be guaranteed that the removed directory entry is located relative to the desired directory.

Every -at variant of a POSIX file-system operation has an analogous paragraph in its rationale section. The wording admits race conditions between path resolution and any operation that modifies the structure along the path being resolved, leading to unspecified behaviour. In this context, “unspecified” does not mean unsafe since, as previously stated, file-system operations are always thread-safe. Thread-safety guarantees that in the presence of such race conditions, `unlink` will either succeed or fail by returning one of the specified error codes. It is not possible to determine the exact outcome

of `unlink` in the presence of such race conditions, even if the file-system state right before the invocation of `unlink` is known. We can only know the set of possible outcomes. In conclusion, the rationale behind `unlinkat` suggests that `unlink` is not actually atomic, contrary to what is stated in ([7],XSH,2.9.7).

POSIX exhibits this ambiguity only for the operations in table 2.1 that resolve paths. It is important to understand what the intentions of the standard are with respect to their behaviour. We suspect that POSIX does intend for these operations to have an atomic *effect*, but with consideration to implementation performance. A truly atomic implementation, where both the path resolution and the effect at the end of the path takes place in a single observable step, would require synchronisation over the entire file-system graph. For most implementations, the performance impact of this coarse-grained behaviour would be unacceptable. Therefore, the wording of the standard allows path resolution to be implemented non-atomically, as a sequence of atomic steps, where each looks up where the next name in the path leads to. The specification of path resolution ([7],XBD,4.13), is silent on this matter.

Our interpretation of the standard's intentions is verified in the Austin Group mailing list [9]. [2] Path resolution itself consists of a sequence of atomic lookups that traverse the file-system graph by following the path. In the case of `unlink`, the effect of removing the resolved link from the file-system graph is atomic. In fact, this is part of a common tenet followed by virtually all major file-system implementations: removing (`unlink`), adding (`open`, `creat`, `link`), moving (`rename`) and looking up individual links in a directory are implemented atomically. In other words, when accounting for concurrency, POSIX operations that resolve paths are sequences of atomic operations.

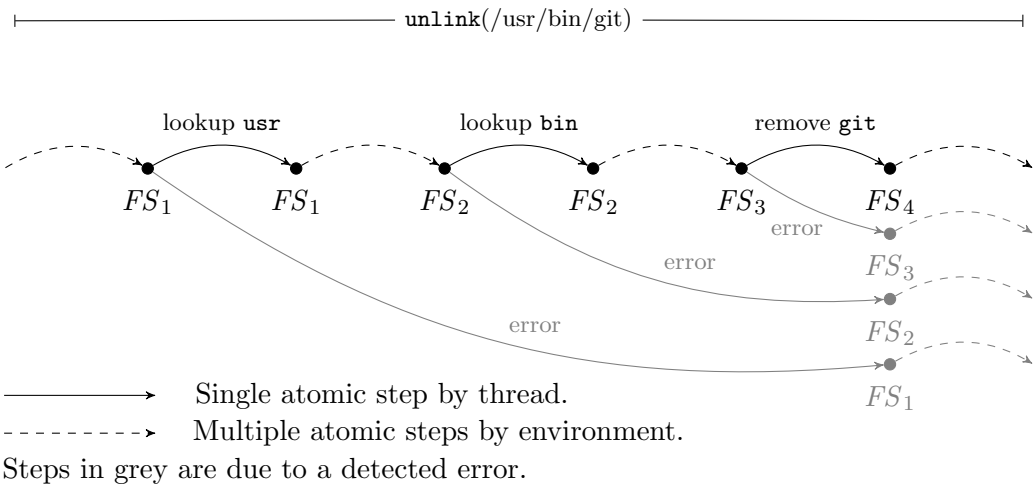

Figure 2.7.: Evolution of the file-system state during `unlink`(/usr/bin/git).

In the example of figure 2.6, where there are no concurrent updates, `unlink`(/usr/bin/git) will perform 3 atomic steps: one for looking up the link usr within the root directory, one for looking up the link bin within the directory with inode 1, and finally one for removing the link git from the directory with inode 2. In figure 2.7, we can see how the file-system state can evolve during the execution of `unlink`(/usr/bin/git), in a concurrent environment. The points $FS_1, \ldots, FS_4$ denote the discrete file-system states observed by `unlink`. In between each of the atomic steps, the environment

[2] Thread: "Atomicity of path resolution", Date: 21 Apr 2015. At the time of writing, this thread is not accessible via the mailing list archive. However, it is accessible via the Usenet gateway gmane.com.

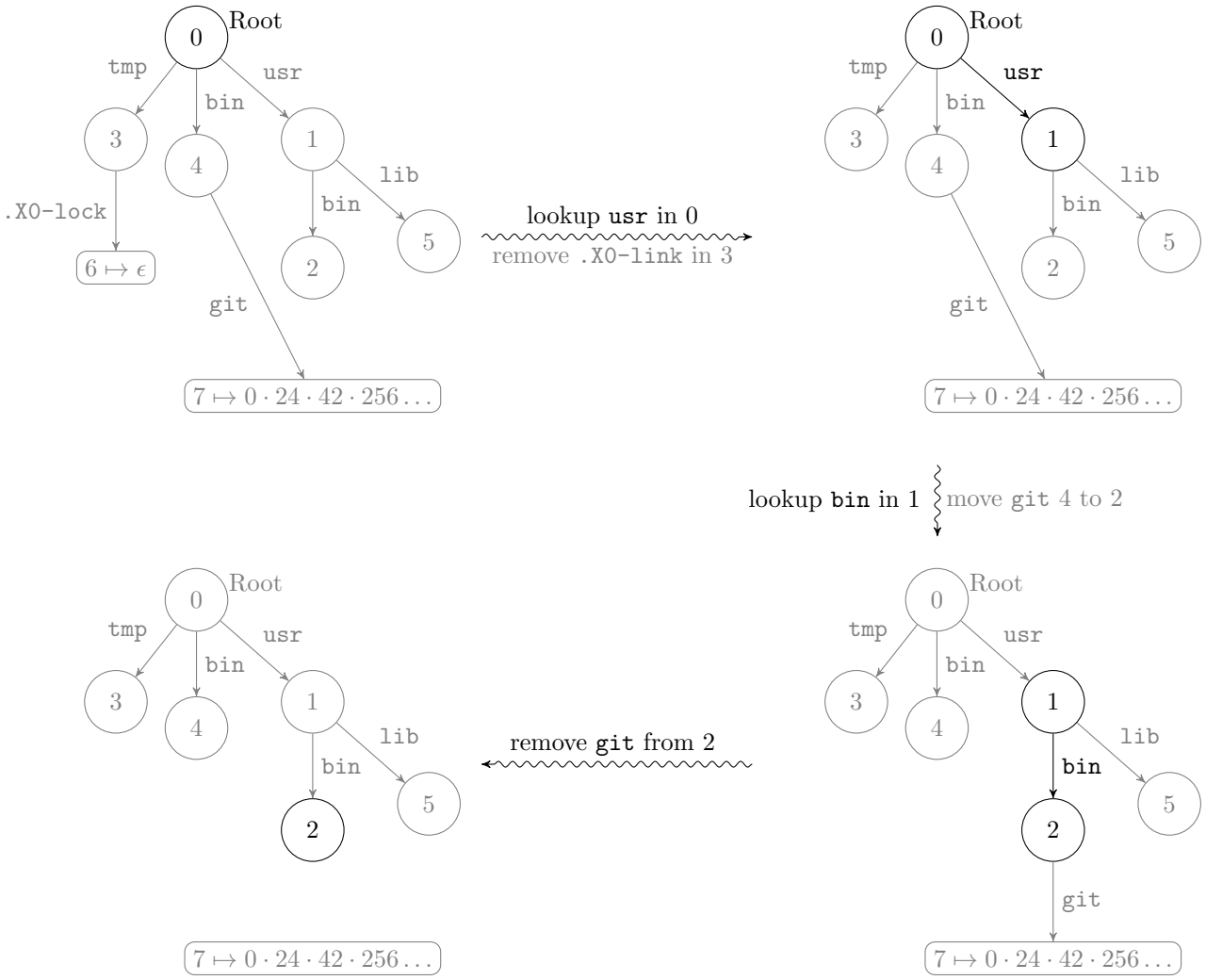

Figure 2.8.: Example of concurrent `unlink`(/usr/bin/git).

can modify the file-system state in any possible way. If `unlink` cannot find any of the names along the path, or if they link files of the wrong type, for example, if usr or bin are not directory files, then it does not modify the file system and the appropriate error code is returned to the application.

In figure 2.8, we show one possible evolution of the file system during `unlink`(/usr/bin/git), where the environment is concurrently modifying it. Note that in the beginning the path /usr/bin/git does not exist. In the first step, `unlink` looks-up the link usr within the root directory, and follows it to the directory with inode 1. Meanwhile, the environment removes the link $.X0-lock$ from the directory with inode 3. In the second step, `unlink` looks-up the link bin within the directory with inode 1 and follows it to the directory with inode 2. Meanwhile, the environment creates the link git within it. In the final step, `unlink` atomically checks that the link git exists in the directory with inode 2 and removes it. If the environment did not create the git link, `unlink` would not succeed.

In the example of figure 2.8, even though the path /usr/bin/git does not exist in the beginning, `unlink` can still succeed because the environment was able to create the missing link before `unlink`'s final atomic step. Similarly, even if the path existed in the beginning, the environment can cause the operation to error by modifying the links along the path. More importantly, the environment

can cause the operation to succeed even if the path does not exist at any single point in time during the operation's execution. We give an example of this behaviour in figure 2.9. Note that the path /usr/bin/git never actually exists at any single point in time. The environment renames existing links and creates new ones in between the steps of `unlink`, such that no error is triggered.

2.3.2. Path Resolution and Client Applications

The fact that POSIX intends path resolution to be a sequence of atomic steps is clearly important for file system implementations, as it allows conforming performant implementations. A reasonable question is if applications need to be aware of this behavioural complexity, or if they can simply assume a more coarse-grained semantics, such as those in the operational specification of Ridge *et al.* [83], where file-system operations behave atomically.

In the example of figure 2.9, we saw that a file-system operation that resolves a path may succeed even if the path being resolved never exists in a single point in time. Therefore, the success of an operation does not imply that the path used existed, merely that the operation was able to resolve it. On the other hand, client applications typically assign some functional meaning to the existence or non-existence of paths, and it is not uncommon for applications to use file-system operations to test for their existence, for example using `stat`. However, such operations also resolve paths. This means that applications test the existence of a path, using operations that, in general, do not provide any guarantee about the existence of the path tested! This is a problem for the correctness of client applications. In order for these tests to be meaningful, applications either have to place restrictions on the environment in which they are being used, or they have to introduce additional synchronisation.

If we assume the simpler, coarse-grained semantics for applications, then this problem is hidden by the assumption that file-system operations have additional internal synchronisation. In this setting, a successful path resolution does imply the existence of the path, at the point in time in which the resolution took place. Assuming a coarse-grained semantics ignores potential behaviours which can easily lead to applications which appear correct under the simpler semantics, but are in truth wrong.

Let us consider an example of an interaction between an email client and an email server that demonstrates this point. The email server is responsible for maintaining a directory structure in which emails arrive, are processed and are delivered. The email client reads emails by scanning this directory structure. Each newly arrived email message is assigned a numerical id, and is initially stored under the path /mail/tmp. For example, the newly arrived message with id 42 is stored initially under the path /mail/tmp/42.msg. To deliver the message, in order for the email client to be able to read it, the server must eventually move it to the path /mail/42/msg.eml. Before delivering the message, the email server has to do some processing, for example, using an anti-virus package to make sure it does not contain a virus. To scan the message, the server moves it to a quarantine directory. A message should be delivered only if it is virus free, and the client should not be able to read an unscanned message. As the delivery of an email message involves several file-system operations, the implementation of the server must take the necessary precautions that they do not accidentally allow the email client to access undelivered, and thus potentially virus infected, messages.

In figure 2.10, we show a possible implementation of the delivery process for the email message with numerical id 42, on the right, in parallel with an email client trying to test if this message is delivered with `stat`, on the left. The `stat` operation of the email client could be part of loop, continuously

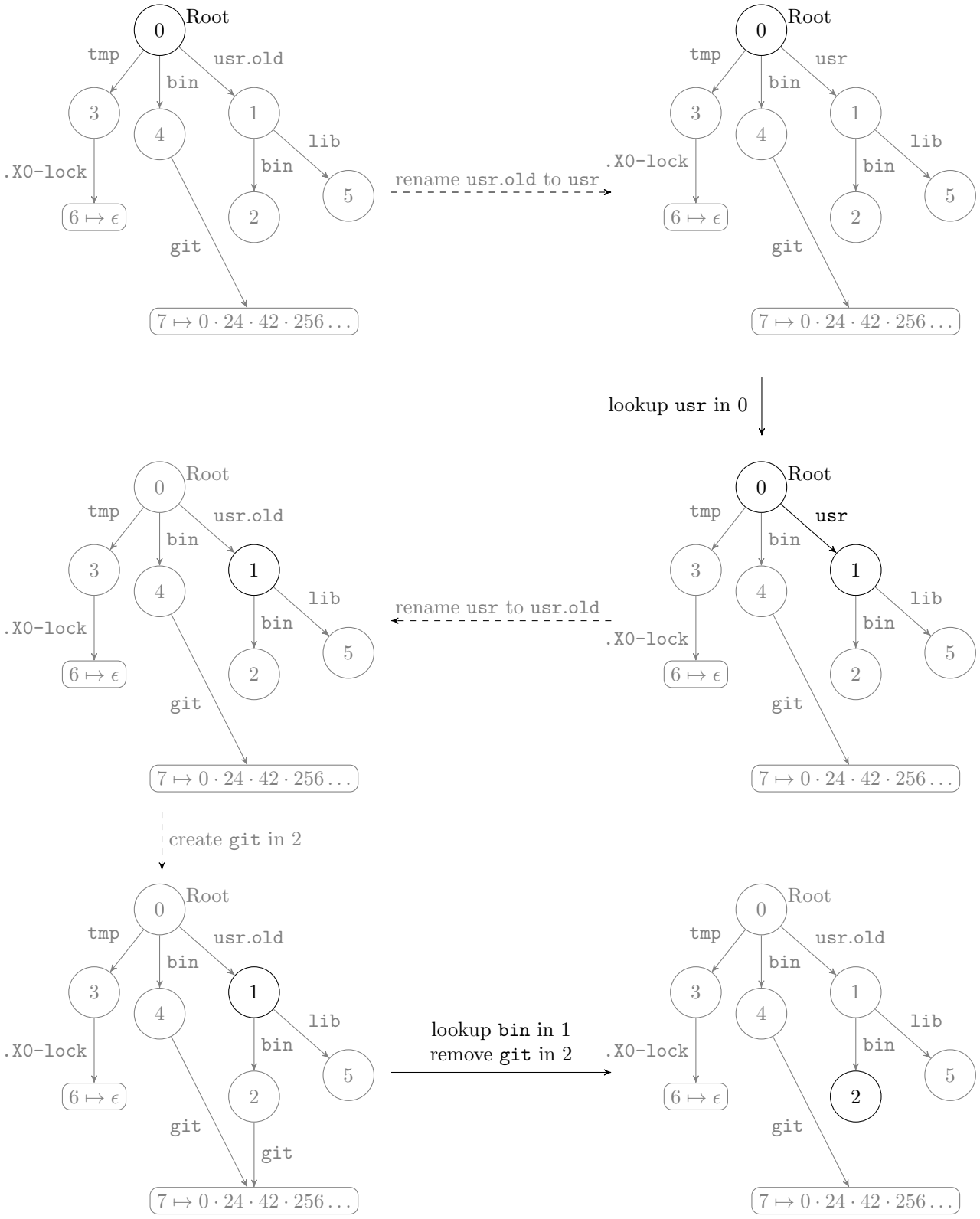

Figure 2.9.: Example where `unlink`(/usr/bin/git) succeeds, even when the path never actually exists.

`stat`(/mail/42/msg.eml);	`mkdir`(/mail/42); `rename`(/mail/tmp/42.msg, /mail/42/unsafe.eml); `rename`(/mail/42, /mail/quarantine); `rename`(/mail/quarantine/unsafe.eml, /mail/quarantine/msg.eml); **let** *is_not_virus* = `run_av_scan`(); **if** *is_not_virus* **then** `rename`(/mail/quarantine, /mail/42); **fi**

Figure 2.10.: Unsafe email client-server interaction.

scanning and testing for delivered messages. Note that the path /mail/42/msg.eml only exists if the last `rename` operation is executed, which in turn is only possible when the message is virus-free. Assume a coarse-grained semantics, where the operations involved are atomic. Then, the internal steps of the `stat` operation of the client will never be interleaved with the operations of the server. Thus, if `stat` succeeds, we are guaranteed for the email message to have been delivered virus-free.

Now consider what happens when the operations involved are not atomic, but sequences of atomic steps, as described in the previous section. In this case, each atomic step comprising `stat` is interleaved with the operations performed by the server on the right. It is then possible for the following interleaving to take place. First, the scheduler decides to execute the first two operations of the server, thus rendering the path /mail/42 resolvable. Next, the scheduler decides to execute the first two atomic steps of `stat`, that resolve the path-prefix /mail/42. However, before taking the final step, the scheduler decides to execute the server again, and the server executes the second and third `rename` operations. At this point, a link named msg.eml exists within the directory that has been previously reached by `stat`. Finally, the scheduler decides to go back to the client and execute the remaining step of `stat`, at which point `stat` will succeed. Therefore, the client succeeds in testing that the message exists, before the server has determined that it is virus free!

The example in figure 2.10 can be made safe by introducing synchronisation between the client and the server, or adding synchronisation between each individual file-system operation involved. This additional synchronisation will essentially guarantee the coarse-grained behaviour that was previously assumed to exist. In conclusion, it is unsafe for applications to assume that file-system operations exhibit more synchronisation that what POSIX actually intends. Instead, applications must explicitly introduce the additional synchronisation when necessary.

2.3.3. Atomicity of directory operations

The operations in table 2.1 are specified to have an atomic effect only on regular files and symbolic links. The operations `mkdir` and `rmdir` are not included. Directories contain the special links "." and "..", linking the directory to itself and to its parent respectively. When a new directory is created, the "." and ".." links must be created as well. When a directory is being moved, its ".." link must be updated to link to the new parent directory. Many modern file-system implementations operate on the directory and its "." and ".." links in a single atomic step. Other implementations operate on "." and ".." in separate atomic steps, potentially exposing intermediate states in which a directory

may contain only of "." and ".." or none at all [9][3]. POSIX does not require directory operations to have an atomic "effect" in order to allow for such implementations. However, as an atomic step is still required to create (`mkdir`), remove (`rmdir`) or move (`rename`) the link to the affected directory, applications can still rely on that step happening atomically. Note that POSIX specifies an empty directory as one that contains at most "." and ".." ([7],XBD,3.144), thus intermediate states in which an empty directory is even emptier are allowed.

2.3.4. Non-determinism

Many POSIX file-system operations are highly non-deterministic. Non-determinism is due to concurrency, optional behaviours allowed by POSIX and specification looseness.

As discussed in section 2.3.1, for file-system operations that resolve paths, even if we know the file-system state right at the invocation point, in general, we do not know whether the operation is going to succeed or return an error. The result depends on the context in which the operation is used and scheduling decisions. Therefore, such operations are non-deterministic on their result. This is an instance of *demonic non-determinism*: success or failure is not controlled by the implementation, but by the scheduler. The schedule acts as a demon, non-deterministically selecting one of the possible interleavings between the file-system operation and the concurrent context to execute.

For some file-system operations, POSIX allows implementations to make different choices. For example, the `unlink` operation is optionally allowed to remove links to directory files ([7],XSH,3.unlink). Analogously, the `link` operation is optionally allowed to create links to directory files ([7],XSH,3.link). Several implementations, such as those found in Linux systems, do not allow the creation and removal of directory links via `link` and `unlink` respectively, while others, such as Mac OS, do. This is an instance of *angelic non-determinism*. The implementation acts as an angel, choosing which of the specified behaviours is executed.

Several file-system operations accept more than one path as arguments. POSIX does not specify the order in which an operation must resolve multiple paths. They can be resolved in any order, or even their resolutions can be interleaved in all possible ways. From the point of view of an implementation, this is an extreme form of angelic non-determinism.

2.3.5. File Systems are a Public Namespace

A public namespace is any type of memory in which the entire address space is known to be accessible by everyone. In a public namespace no part of the memory can be privately known to a single thread. The POSIX file system is a public namespace. In POSIX file systems any process can, at any time, access and modify any part of the file-system graph it chooses. The moment a file is created, it is accessible to everyone. As we explain shortly, file-access permissions have a limited effect and cannot in general prevent this. In contrast, the heap is not a public namespace. Heap allocated objects are by default only known to the thread that allocates them. A program dereferencing random heap addresses has undefined behaviour. The only way for a heap object to be accessible by multiple threads is if the thread that allocated the object chooses to somehow share the object's address with other threads.

The fact that the file system is a public namespace has important ramifications on the functional correctness of client applications, especially those that rely on the existence of certain paths, or work

[3]Thread: "Rationale behind no atomicity guarantees for directory operations.", Date: 02 Sep 2016.

with elaborate directory hierarchies, such as databases, email servers and mail delivery agents. All client applications, both sequential and concurrent, are executed within a context where many other applications access the file system concurrently. Applications that expect certain paths to exist will not work correctly in all possible contexts, as some of those contexts may interfere with the paths they intend to work with. Such applications are developed with the assumption that the context will not interfere with those parts of the file-system they consider their own. However, POSIX offers no mechanism to enforce such assumptions across the entire operating system. Therefore, client applications behave correctly only within restricted contexts, that respect their assumptions.

POSIX does specify file-access permissions, which can restrict access to files to only processes running with the appropriate privileges. We discuss file-access permissions in more detail in chapter 6, section 6.3.2. However, POSIX file-access permissions cannot reliably restrict access to a file to just one process. It is possible for many processes to run with the same file-access permissions, and thus, this does not stop them from interfering with each other in unwanted ways. Furthermore, processes, such as administrator scripts, may run with super-user privileges, effectively ignoring file-access permission restrictions altogether. This means that even with file-access permissions, applications still behave correctly only in certain contexts: those that have the appropriate file-access permissions set on the file-system graph and with non-interfering concurrent processes. Instead of file-access permissions, applications can use other techniques, such as isolation and compartmentalisation, chroot and object capabilities, such techniques, however, are not specified in POSIX and are beyond the scope of this dissertation.

In conclusion, when reasoning about the correctness of file-system applications, we must be explicit in the assumptions an application has about the context.

2.4. Conclusions

File systems are an integral part of operating systems. We have given an overview of their structure, implementation and programming interfaces found in the UNIX family of operating systems. In particular we focus on the file-system structure and behaviour of programming interfaces that is informally specified by the POSIX standard using English language descriptions.

POSIX file systems are inherently concurrent. The concurrent behaviour of file systems is the most difficult aspect of understanding how their operations behave overall. The POSIX standard text is not particularly helpful in gaining a thorough understanding of concurrency in file systems. The relevant information is scattered throughout different sections and is often incomplete, as is the case for example in understanding the concurrent behaviour of path resolution, or even ambiguous. This means that we have to often turn to other sources of information, in particular the Austin Group mailing list, for clarifications and interpretations of the standard's text. This very fact is suggestive of the need for better specifications than English descriptions, a need that precise formal specifications can meet.

We have given an overview of the most challenging aspects of the concurrent behaviour of POSIX file systems that is specified in the standard's text with the additional interpretations from the Austin Group mailing list. File system operations are thread safe, in that they always succeed or return an error in any given context. However, they are not generally atomic. Operations that resolve paths perform multiple atomic actions: sequences of atomic directory lookups to resolve a path followed by

at least one atomic update to the file-system structure. This fact is not only important for file-system implementations, but as we informally discussed, using an example of a concurrent email client-server interaction, it is also something that client applications must not ignore. We revisit this example formally in chapter 8 demonstrating that specifications that simplify away this behavioural complexity lead to unsafe client reasoning. File system operations exhibit high levels of non-determinism, both in terms of their results due to concurrency and due to different implementation behaviours allowed by the standard. Finally, POSIX file systems are a public namespace, meaning that the entire file-system structure is accessible by any process at all times, which complicates client reasoning since it is generally not possible for an individual process to enforce exclusive access to the parts of the file system it requires. Any system for formally specifying and reasoning about POSIX file systems must tackle these challenges.

We demonstrate the approach developed in this dissertation in chapter 6 with examples of formal specifications for the core file-system operations and examples of reasoning about client applications. The formal development of our approach is presented in chapter 7.

3. Reasoning about Concurrent Modules

The POSIX file-system interface is a concurrent module. In this chapter we give an overview of various methods for reasoning about concurrent programs in general, and concurrent modules in particular, primarily focusing on concurrent separation logics. With an eye towards a formal POSIX specification, we explore the viability of each approach.

We begin with separation logic in section 3.1, as it forms a fundamental basis for the contributions of this dissertation as well as a large portion of related work. In section 3.2 we discuss the extension of separation logic to coarse-grained concurrency in the form of concurrent separation logic. We then discuss how separation-logic style reasoning about the heap was generalised to arbitrary resource models in section 3.3. In section 3.4 we discuss the development of program logics for fine-grained concurrency and in section 3.5 we expand our discussion on fine-grained concurrency to cover reasoning about atomicity. Finally, we present our conclusions in section 3.6.

3.1. Separation Logic

Hoare Logic [53, 54] by, C.A.R. Hoare, provided an axiomatic foundation for reasoning rigorously about imperative programs. In Hoare Logic, the semantics of programs are given in the form of Hoare triples, $\{P\}\,\mathbb{C}\,\{Q\}$, where $\mathbb{C}$ is a program, and P, Q are first-order logic assertions, referred to as the *precondition* and *postcondition* respectively. When the program's state satisfies the precondition P, executing the program $\mathbb{C}$ yields a state that satisfies the postcondition Q. Hoare triples are typically given a *fault avoiding partial correctness interpretation*. By this interpretation, the triple $\{P\}\,\mathbb{C}\,\{Q\}$ specifies that if before executing $\mathbb{C}$ the program's state satisfies the precondition P, then the execution of $\mathbb{C}$ will not fault, and if $\mathbb{C}$ terminates, then the program's state will satisfy the postcondition Q. Hoare Logic consists of axioms for the basic commands of an imperative language, such as assignment, and inference rules for imperative constructs, such as while loops and conditional statements, that are used to derive Hoare triples for programs.

Several attempts were made to extend this foundation to reason about programs that manipulate heap-based data structures with pointer aliasing [16, 82]. However, these approaches proved problematic, suffering from reasoning complexity, lack of compositionality, and did not scale effectively to larger programs.

The reason behind this lies in the fact that traditional Hoare Logic assertions describe the entire state of a program. From the programmer's perspective, this is counter-intuitive. In fact, programmers typically reason informally about their code in a more local manner, restricting to only the fragment of the state that their code manipulates. O'Hearn, Reynolds and Yang, with their work on separation logic [74, 81], extended Hoare Logic with exactly this intuition of locality for reasoning about programs that manipulate heap cells:

> "To understand how a program works, it should be possible for reasoning and specification to be confined to the cells that the program actually accesses. The value of any other cell will automatically remain unchanged."

Separation logic consists of classical first-order connectives but, more importantly, introduces additional connectives for describing the shape of the heap. The most important is the *separating conjunction*, $*$. An assertion $P * Q$, is satisfied when P and Q are satisfied by disjoint fragments of the heap.

A simple example of a separation logic specification is the following:

$$\{\mathtt{x} \mapsto 5\}\,[\mathtt{x}] := 1\,\{\mathtt{x} \mapsto 1\}$$

where $[\mathtt{x}] := 1$ is the assignment statement, assigning the value 1 to the contents of the heap cell with address given by variable $\mathtt{x}$. The heap cell assertion $\mathtt{x} \mapsto 5$ is treated as a *resource*. The assignment statement in the example acts only on the resource identified by the address $\mathtt{x}$. The assertion $\mathtt{x} \mapsto 5$ in the precondition of the specification above, confers *ownership* of the heap cell resource at address $\mathtt{x}$. Separation logic specifications, such as the one above, are local, in the sense that they only require ownership of the resources that the program is accessing.

When reasoning about a program accessing several resources, such local resource specifications can be extended to work with more resources via the *frame rule*, introduced in separation logic:

$$\frac{\{P\}\,\mathbb{C}\,\{Q\}}{\{P * R\}\,\mathbb{C}\,\{Q * R\}} \quad mod(\mathbb{C}) \cap free(R) = \emptyset$$

The specification of the premiss requires ownership of the resource in assertion P, which the program $\mathbb{C}$ updates to Q. In the conclusion, the specification requires ownership of the resource in assertion P and the resource in assertion R, and only P is update to Q; the resource in R is unmodified. The separating conjunction $P * R$, requires that the resource in P is disjoint from the resource in R. In the case of heap resource, this means that P and R are satisfied by disjoints fragments of the heap. The side condition is required so that any program variables mutated by the $\mathbb{C}$ are not referenced in R, since this would entail an update on R as well. As demonstrated by Bornat *et al.* [18], this side condition can be elided, if variables themselves are also treated as resource.

With the specification example given earlier, we can use the frame rule to extend the assignment to a larger heap, such as the following:

$$\{\mathtt{x} \mapsto 5 * \mathtt{y} \mapsto 42\}\,[\mathtt{x}] := 1\,\{\mathtt{x} \mapsto 1 * \mathtt{y} \mapsto 42\}$$

where we extend the resource required by the specification with the heap cell resource $\mathtt{y} \mapsto 42$. We see that the additional resource remains unaffected. Moreover, the two heap cells are disjoint, *i.e.* $\mathtt{x} \neq \mathtt{y}$.

The frame rule and separating conjunction of separation logic enables compositional verification. The verification of a large program can be decomposed to the verification of smaller fragments, such as individual procedures. Each fragment can be verified independently, only in terms of local resource, *i.e.* the heap cells it accesses, and the frame rule allows the fragment's specification to be extended to the resource used by the larger program. The compositional nature of this approach has led to the

development of an array of automated reasoning and verification tools based on separation logic. The first separation-logic based tool, Smallfoot [14], was able to prove properties of programs manipulating heap-based data structures such as lists and binary trees. Later tools such as SpaceInvader [99] and jStar [38, 19] were able to automatically verify memory safety properties of large bodies of code, including close to 60% of the Linux kernel [37]. A further development in the same line of tools, Infer [22, 23], is being used in production at Facebook, to verify memory safety properties in Facebook's codebase. Infer utilises the compositional nature of separation logic to automatically verify memory safety properties of each individual new commit to the codebase, using the verification results of previous commits as a specification repostitory. Finally, Verifast [59], a separation-logic based proof assistant, has been used to verify functional properties of complex programs, including device drivers.

3.2. Concurrent Separation Logic

Concurrent programs comprise multiple threads of control manipulating the program state. In this setting, the actions of one thread may interfere with the actions of another. *Data races* occur when multiple threads update the same resource concurrently, which may lead to state corruption. On the other hand, when the threads comprising the concurrent program work with disjoint resources data races never occur; such programs are *data-race free*. In separation logic resource disjointness is expressed with the separating conjunction. Therefore, separation logic is suitable for reasoning about data-race free concurrent programs. This is expressed through the *parallel rule* of concurrent separation logic [76], given below:

$$\frac{\{P_1\}\,\mathbb{C}_1\,\{Q_1\} \qquad \{P_2\}\,\mathbb{C}_2\,\{Q_2\}}{\{P_1 * P_2\}\,\mathbb{C}_1 \parallel \mathbb{C}_2\,\{Q_1 * Q_2\}}$$

The rule states that when two threads $\mathbb{C}_1$ and $\mathbb{C}_2$ update disjoint resources P_1 to Q_1 and P_2 to Q_2 respectively, the results of their individual updates are combined to produce the overall effect of their running in parallel: the resource $P_1 * P_2$ is updated to $Q_1 * Q_2$. This type of concurrency is known as *disjoint concurrency*.

In the setting of disjoint concurrency threads never share resources. Not all programs are data race free through, and certainly POSIX does not impose such restrictions; the file system is a shared resource. Bornat *et al.* introduced *permission systems* to allow a form of resource sharing [17]. A commonly used permission system is Boyland's *fractional permissions* [20], in which a resource is associated with a permission: a rational number in the interval $(0, 1]$. Permission 1 is commonly referred as the full permission, whereas a permission in the interval $(0, 1)$ is referred to a fractional permission. Using the system of fractional permissions, a resource with some initial permission π can be split into two parts, each with permission $\frac{\pi}{2}$. For example, when we associate fractional permissions with heap cells, we get the following axiom:

$$\mathtt{x} \overset{\pi_1+\pi_2}{\longmapsto} v \iff \mathtt{x} \overset{\pi_1}{\longmapsto} v * \mathtt{x} \overset{\pi_2}{\longmapsto} v \quad \text{if } \pi_1 + \pi_2 \leq 1$$

This means that a full permission 1 confers total ownership of the resource, whereas a fractional permission confers shared ownership of the resource with other threads. Typically, in this permission

system a thread can update the resource only if it has full ownership, otherwise it can only read it. Therefore, threads can share the resource as long as no thread updates it.

Concurrent separation logic (CSL) allows threads to update a shared resource, but only if access to the resource is protected by mutual exclusion. In CSL mutual exclusion is directly implemented in the programming language in the form of conditional critical regions [76]. For the current discussion we simplify conditional critical regions to *atomic blocks* $\langle \mathbb{C} \rangle$, a form of unconditional critical region. Effectively, an atomic block enforces sequential access to the shared resource. This is captured in the following atomic rule for reasoning about atomic blocks:

$$\frac{\{P * \mathsf{Inv}\}\, \mathbb{C}\, \{Q * \mathsf{Inv}\}}{\boxed{\mathsf{Inv}} \vdash \{P\}\, \langle \mathbb{C} \rangle\, \{Q\}}$$

Initially, ownership of the shared resource accessed by the atomic block is placed into the shared invariant $\boxed{\mathsf{Inv}}$. Once the atomic block is entered, ownership of the shared resource is transferred to the thread executing the code $\mathbb{C}$, within the block. Then, $\mathbb{C}$ proceeds to update the shared state, with the proviso that in the end it satisfies the invariant. When exiting the atomic block, ownership of the shared state is transferred back to the shared invariant. The pattern of transferring ownership of a shared resource, exhibited in CSL, is commonly referred to as *ownership transfer*.

3.3. Abstract Separation Logic and Views

The initial development of separation logic used the heap memory as a resource model. A general framework for separation-logic based program reasoning, in the form of *abstract separation logic*, was later introduced by Calcagno *et al.* [25]. In abstract separation logic a resource model is an instance of a *separation algebra*: a partial, associative, commutative and cancellative monoid $(H, *, u)$, where H is a set of resources, $* : H \rightharpoonup H$ is the binary resource composition operator and u is the unit of composition. In the resource model of heaps, H is the set of finite partial functions from heap addresses to values, $*$ is the union of functions with disjoint domains, and u is a function with an empty domain. Constructions that combine different types of resource are possible, for example by taking the cross product of different separation algebras.

In abstract separation logic the resource model is also the model of states that programs manipulate. This restricts program reasoning to states that are easily separable and composable, which is not always desirable.

The *Views Framework* by Dinsdale-Young *et al.* [35] is a descendant of abstract separation logic. In contrast to abstract separation logic, the Views Framework makes a distinction between the machine states M, manipulated by programs, and *views*: resources that denote composable abstract representations of machine states used in program reasoning. A view is an instance of a commutative semi-group $(\textsc{View}, *)$, where $p_1 * p_2$ is the composition operator on views $p_1, p_2 \in \textsc{View}$. When the view semi-group also has a unit element u, then it becomes a *view monoid* $(\textsc{View}, *, u)$. View monoids can be constructed from *view separation algebras*, a generalised form of separation algebras. A view separation algebra is a partial commutative monoid $(V, \bullet, I)$, with multiple units $I \subseteq V$. In contrast to separation algebras, view separation algebras do not require cancellativity. Each view separation algebra $(V, \bullet, I)$ induces a *separation view monoid* $(\mathcal{P}(V), *, I)$, where $p_1 * p_2 \triangleq \{m_1 \bullet m_2 \mid m_1 \in p_1, m_2 \in p_2\}$.

View resources often include instrumentation over machine states that solely serves to enable separation-logic style program reasoning. Examples include: fractional permissions [17]; contextual information about tree fragments [97], such that tree fragments can be composed to a complete tree; shared resource invariants [76]; and even information about interference from other threads [60, 36]. Views are related to machine states through *reification*: a function $[\![p]\!] : \textsc{View} \to \mathcal{P}(M)$, mapping views to sets of machine states. Note that the term "view" is on the point; each view represents a particular way to look at the current machine state, which in the reasoning is treated as a resource.

3.4. Fine-grained Concurrency

In CSL a thread can update only the resources it owns. At the same time, as we saw in the parallel rule, resource disjointness enforces that one thread does not interfere with the resources of another thread and vice-versa. In this sense, resource ownership can be seen as a form of *interference abstraction*. However, this type of interference abstraction is too strong and limits reasoning about concurrent programs to coarse-grained concurrency.

A general method for interference abstraction was first introduced with *rely-guarantee* by Jones [60]. Rely-guarantee specifications explicitly constrain the interference between a program and its concurrent environment. A rely-guarantee specification, $R, G \vdash \{P\}\,\mathbb{C}\,\{Q\}$, associates the standard Hoare-triple with two relations: the *rely relation* R, and the guarantee relation G. The rely relation abstracts the interference on the shared state by the concurrent environment, whereas the guarantee relation abstracts how the program updates the shared state. The precondition and postcondition assertions must be *stable* with respect to the rely relation, *i.e.* they must be robust with respect to environmental interference. When reasoning about concurrent threads, the guarantee of one thread must be included in the rely of the others. This is captured by the parallel composition rule of rely-guarantee:

$$\frac{R \cup G_2, G_1 \vdash \{P_1\}\,\mathbb{C}_1\,\{Q_1\} \qquad R \cup G_1, G_2 \vdash \{P_2\}\,\mathbb{C}_2\,\{Q_2\}}{R, G_1 \cup G_2 \vdash \{P_1 \wedge P_2\}\,\mathbb{C}_1 \parallel \mathbb{C}_2\,\{Q_1 \wedge Q_2\}}$$

Rely-guarantee does not restrict program reasoning to coarse-grained concurrency. Using this method we can specify the operations of a module and explicitly define the amount of interference from the environment the operations tolerate. However, in several cases this leads to specifications that are weaker than intended. In their survey paper da Rocha Pinto *et al.* [32] demonstrate this by examples of a ticket lock and a concurrent counter. More importantly, using rely-guarantee to specify the POSIX file-system interface leads to extremely weak specifications. File-system operations do not restrict the interference from the concurrent environment in any way. Therefore, the rely relation for each operation would have to be maximal, containing all possible updates to the file-system structure. The ramification of this is that the precondition and postcondition of each operation would simply have to state the existence of an arbitrary file-system structure in order to remain stable. Such specifications would only state the thread-safety; all other functional correctness properties are lost.

With the advent of concurrent separation logic, subsequent separation logics such as RGSep [96, 95], local rely-guarantee [41] and deny-guarantee [39] added support for fine-grained concurrency by combining resource ownership with interference abstraction, albeit with different forms. The amalgamation of these methods together with abstract predicates [77] led to the development of *concurrent abstract*

predicates (CAP) by Dinsdale-Young *et al.* [36], which introduced abstractions over shared resources.

CAP has been used to reason about implementations and clients of concurrent data structures such as sets and indexes [29]. For example, the specification of a concurrent set module involves the abstract predicates $\mathsf{in}(\mathtt{x}, \mathtt{v})$ and $\mathsf{out}(\mathtt{x}, \mathtt{v})$, which assert the existence and non-existence of value `v` in the set `x` respectively. The implementation of these predicates can be in terms of a linked list or a B-tree containing all the elements contained in the set. However, the granularity of the abstraction is at the level of individual elements. A specification for an operation that inserts elements to the set can be given with the following Hoare triples:

$$\{\mathsf{out}(\mathtt{x}, \mathtt{v})\}\ \mathtt{insert}(\mathtt{x}, \mathtt{v})\ \{\mathsf{in}(\mathtt{x}, \mathtt{v})\} \qquad\qquad \{\mathsf{in}(\mathtt{x}, \mathtt{v})\}\ \mathtt{insert}(\mathtt{x}, \mathtt{v})\ \{\mathsf{in}(\mathtt{x}, \mathtt{v})\}$$

Treating the abstract predicates in and out as resource we can use the specification to reason about concurrent clients. The granularity of the abstraction allows multiple threads to update the set concurrently, as long as they work with disjoint elements, *i.e.* in the setting of disjoint concurrency. In order for multiple threads to work on the same element, we would have to employ ownership transfer introducing synchronisation. To remedy this da Rocha Pinto *et al.* [29] proposed abstract predicates of a similar abstraction granularity for concurrent index incorporating permission systems as a means to grant threads the capability to concurrently update the same resources. However, these capabilities are predetermined, built into the specification and are only able to express certain concurrent interactions with the shared resource. Apart from the lack of atomicity specification, the same shortcoming of CAP-style specifications applies to file systems as well. Even though it is possible to give a file-system specification in this style, it restricts client reasoning to only those types of clients the choice of permission system allows.

In their survey paper, da Rocha Pinto *et al.* [32] made the following observation on such combinations of resource ownership and interference abstraction:

> "While it is an effective tool, and can be used to give elegant specifications, something more is required to provide the strong specifications we are seeking."

The same authors identify the missing ingredient to be the specification of atomicity.

3.5. Atomicity

In concurrency, *atomicity* is the property exhibited by an operation, in which it takes effect in a single, discrete point in time. Truly atomic operations in this sense are rare. In modern multi-processor architectures only few hardware instructions are atomic, such as *compare-and-swap* (CAS). Nevertheless, by means of these few instructions it is possible to build operations that *appear* to be atomic, even though they achieve this illusion by executing several atomic and non-atomic hardware instructions. If we can prove that such operations do appear to behave atomically, then it is safe to use them as if they were truly atomic. Concurrent modules of apparently atomic operations are common; examples include lock modules or the concurrent data structures in `java.util.concur`.

Linearisability [51] is a well established correctness condition used to prove that operations of module appear to behave atomically. In this approach, each operation of the module is given a sequential specification of its effect. Then, we must prove that each operation behaves atomically

with respect to the other operations of the module. This holds if for each operation we can find a point during its execution at which it appears to take effect. This point is commonly referred to as the *linearisation point*. Effectively, this amounts to showing that the concurrent behaviour of the operations is equivalent to a sequential interleaving of their effects.

As identified in chapter 2.3, several operations in POSIX file systems are not atomic, but are sequences of atomic steps. This effectively means that the POSIX file-system module is not linearisable. Linearisability could be used for the individual atomic steps comprising the non-atomic operations, however this would require a language for specifying sequences of linearisation points. Furthermore, linearisability requires that each operation of a concurrent module tolerates the interference from all other operations. No other interference, and specifically interference on the internal state of the module's implementation is not permitted. In other words, the module's interface is the boundary of the allowed interference. This is problematic for client modules built on top of the file system. The file system is a public namespace, and therefore any process can access the file system, possibly interfering with the internal state of a module using the file system. Therefore, the module's interface cannot be the boundary for interface abstraction. Reasoning about file-system based modules using the file system requires further restrictions on the concurrent context.

Filipovic *et al.* [43] demonstrated that linearisability is related to contextual refinement. Assuming that the programming language properly encapsulates the internals of a concurrent module, *i.e.* a client cannot access memory locations in which internal structures are held, linearisability implies contextual refinement. Contextual refinement means that in any concurrent context, we can replace a complex implementation of a concurrent data structure with a more abstract and simpler specification, possibly given as code, without loosing any observable behaviour.

3.5.1. Contextual Refinement and Separation Logic

Turon and Wand in their paper [94], developed a method for proving atomicity of operations with contextual refinement by combining a refinement calculus [13, 70] with separation logic. In this work, an atomic operation is specified as an *atomic action* $\langle \forall x.\, P,\, Q \rangle$, where P and Q are precondition and postcondition assertions in separation logic respectively. Intuitively, an atomic action represents any program satisfying the separation logic Hoare triple $\{P\} - \{Q\}$. This is akin to the linearisability approach where each atomic operation is given a sequential specification.

Atomic actions form the building block of a specification programming language, given below, which includes sequential composition (;), parallel composition ($\|$), angelic non-deterministic choice ($\sqcup$), existential quantification, as well as first-order functions and recursion (not given below).

$$\phi, \psi, \theta ::= \phi; \psi \mid \phi \parallel \psi \mid \phi \sqcup \psi \mid \exists x.\, \phi \mid \langle \forall x.\, P,\, Q \rangle \mid \ldots$$

Both programs and specifications are written in this language, with programs being the most concrete of specifications. Reasoning about specification programs in this language is performed by contextual refinement; when $\phi \sqsubseteq \psi$, we say that ϕ contextually refines ψ. The authors develop a refinement calculus with refinement laws designed to prove contextual refinements of atomic actions. This is demonstrated with two examples: an increment operation of a non-blocking counter; and a version of Treiber's non-blocking stack [91].

The concurrent increment operation $\texttt{incr}(x)$ is proven to be atomic by deriving the contextual refinement: $\texttt{incr}(x) \sqsubseteq \langle \forall n.\, x \mapsto n,\ x \mapsto n+1 \rangle$. This is a simple example in that the operation is shown to be atomic regardless of the interference on the heap cell x. For more complex concurrent data structures, such a concurrent stack, the interference must be restricted. Specifically, the environment should not interfere with the internals of the implementation, for example the representation of an abstract stack as a Treiber stack in memory, except by invoking the module operations that manipulate the abstract structure. To facilitate this style of reasoning, the authors introduce *fenced refinement*: $I, \theta \vdash \phi \sqsubseteq \psi$. Here, I is a representation invariant that interprets the abstract data structure used by clients into its in-memory representation used in the implementation, and θ is a specification program abstracting the interference from the environment, akin to the rely relation in rely-guarantee. Fenced refinement states that $\phi \sqsubseteq \psi$ with the proviso that all memory is either part of the shared resource described by I, or it is private, and interference on the shared resources is bounded by θ. In order to use the operations of a concurrent module as atomic, we must first show by fenced refinement that these operations are atomic with respect to each other, using all the module's operations as the "rely" θ. As in linearisability, the interference abstraction is the module boundary.

With the specification language of Turon and Wand, we can specify operations as sequences of atomic steps, as required by POSIX file-system operations. However, fenced refinement is prohibitive for reasoning about file-system clients. Fenced refinement carries the assumption that the module's state can only ever be changed by the module's operations. For modules that are implemented using the file system we cannot assume that the environment does not interfere with those parts of the file system used by the module. The file system is a public namespace and offers no mechanism for controlling file-system parts that are private to some thread or process.

CaReSL [93] is a concurrent separation logic for fine-grained concurrency used to prove contextual refinement between programs. The principle of CaReSL is that a highly optimised fine-grained implementation of a concurrent data structure is proven to contextually refine a much simpler coarse-grained reference implementation. Then, the reference implementation can be used to prove properties about clients. In this dissertation we are not interested in file-system implementations but the specification of the POSIX file-system interface. With CaReSL this specification would need to be given as a reference implementation. Even if we give a simple reference implementation, where each atomic step is given a coarse-grained implementation, it would still be significantly more complex than a sequence of atomic specifications in the style of Turon and Wand's atomic actions.

Liang *et al.* developed a separation logic for proving termination preserving refinements of concurrent programs [65]. Even though termination is obviously a desired property for file-system implementations, the POSIX standard does not mandate file-system operations to terminate. In order to require operations to terminate in all possible concurrent contexts, POSIX would also have to mandate scheduling behaviour properties, such as fairness. POSIX purposefully does not specify such properties to allow more flexibility in implementations.

3.5.2. Atomic Hoare Triples

The program logic TaDA by da Rocha Pinto *et al.* [30, 28] is a concurrent separation logic for fine-grained concurrency that introduced *atomic Hoare triples*, which are triples that specify the atomicity

of an operation. In TaDA an atomic Hoare triple is of the following simplified form:

$$\mathbb{V}x \in X.\, \langle P(x) \rangle\, \mathbb{C}\, \langle Q(x) \rangle$$

Intuitively, the atomic Hoare triple specifies that the program $\mathbb{C}$ atomically updates a state satisfied by the precondition $P(x)$ to a state satisfied by the postcondition $Q(x)$ for some $x \in X$. However, the semantics are bit more subtle. From the point of view of a client, the precondition $P(x)$ and postcondition $Q(x)$ specify the point at which the atomic update appears to take effect: the linearisation point. The binder $\mathbb{V}x \in X$, also referred to as the universal pseudo-quantifier, servers to restrict the interference by the environment. During the execution of $\mathbb{C}$ and until the atomic update takes effect, the environment may change the value of x, as long it remains with X, and as long as $P(x)$ still holds. After the atomic update takes effect, the environment is no longer restricted; it may use $Q(x)$ in any way. From the point of view of the implementation, $P(x)$ initially holds for some $x \in X$. The implementation must tolerate the interference from the environment: $P(x)$ may change to $P(x')$ for any $x, x' \in X$. The implementation must update $P(x)$ to $Q(x)$ (for some $x \in X$) at some point, after which the implementation can no longer access $Q(x)$ since the environment may be using it.

The atomicity captured by TaDA's atomic Hoare triples can be viewed as a generalisation of the atomicity specified by linearisability and Turon and Wand's atomic actions. Notice that an atomic action itself tolerates all possible interference on the shared resource: in $\langle \forall x.\, P,\, Q \rangle$ the value of x is fixed during its execution. Semantically an atomic action is sequentially interleaved with the atomic actions of its concurrent context. The same observation is made for linearisability. On the other hand, atomic Hoare triples can restrict the interference tolerated by the atomic specification.

Interference restriction allows atomic Hoare triples to specify blocking atomic operations. This is demonstrated in TaDA through the example of locking a lock. The state of a lock can be represented by the abstract predicate $\mathsf{Lock}(\mathtt{x}, v)$, where $\mathtt{x}$ is the address of the lock and $v \in \{0, 1\}$ is an abstract representation of the state of the lock: 0 when the lock is unlocked; and 1 when the lock is locked. Intuitively, a lock can become locked, only if it was previously unlocked. While the lock is unlocked, the $\mathtt{lock}(\mathtt{x})$ operation can immediately lock the lock. On the other hand, while the lock is locked, the operation has to wait until the lock becomes unlocked. This intuition is formally specified with the following atomic Hoare triple:

$$\mathbb{V}v \in \{0, 1\}\,.\, \langle \mathsf{Lock}(\mathtt{x}, v) \rangle\, \mathtt{lock}(\mathtt{x})\, \langle \mathsf{Lock}(\mathtt{x}, 1) * v = 0 \rangle$$

From the pseudo quantifier and the precondition we see that the environment is allowed to freely change the state of the lock until the operation locks the lock for the current thread. By the $v = 0$ in the postcondition we see that the only way for the lock to become locked for the current thread is if it was previously in the unlocked state: either it was unlocked in the first place; or some other thread has unlocked the lock during the execution of `lock`. Note that the atomic actions of Turon and Wand are unable to specify such blocking behaviour.

Locks are an interesting example for reasoning about file-system clients. A popular implementation of locks in file systems is through the use of *lock files*. A lock file is a file with a known path, typically a regular file. If the lock file exists the lock is locked; otherwise, it is unlocked. The same blocking behaviour in locking the lock is expected for lock files.

In TaDA the atomicity of each operation is specified with respect to an abstraction, such as $\mathsf{Lock}(\mathtt{x}, v)$. Therefore, it is possible to extend a module with additional operations without the need to re-verify the existing operations. Crucially, the operations of a concurrent module do not have to be atomic with respect to each other. In contrast, atomicity in both fenced refinement and linearisability is a whole module property. Henceforth, in order to distinguish between the linearisability-style atomicity of Turon and Wand and the atomicity specified by TaDA's atomic Hoare triples, we will refer to the former as *primitive atomicity* and to the latter as *abstract atomicity*.

Following earlier developments in concurrent separation logics for fine-grained concurrency [93, 88] TaDA utilises a system of generalised capabilities for defining protocols by which clients can atomically update a shared resource. Such protocols can be used to reason about file-system clients in spite of the file system being a public namespace. If we define a protocol that denies the environment certain capabilities on the shared file system then, if the environment follows the protocol, the client's use of the file system will not be subject to interference. For example in the case of lock files, if we deny the environment the capability to alter the path to the lock file, the environment will not interfere with the path used by the lock-file module implementation.

Several other recent developments in concurrent separation logics for fine-grained concurrency have incorporated support for atomicity specification. In contrast to TaDA where atomicity is built in to the logic, the logics of Jacobs and Piessens [58], HOCAP [89], iCAP [88] and Iris [62] encode atomicity through higher-order specifications. An alternative approach to abstract atomicity is the use of histories to maintain the changes that were performed to a module.

3.6. Conclusions

We have examined various approaches for specifying and reasoning about concurrent modules. To achieve modular specifications requires a combination of resource ownership, interference abstraction and atomicity. The fundamental challenge to a concurrent specification of POSIX file systems is the fact that file-system operations perform sequences of atomic steps, whereas the nature of the file system as public namespace presents a challenge to client reasoning. None of the approaches examined in this chapter are sufficiently capable of addressing both. At the same time however, we have gained some insight on how tackle both challenges. The specification language of Turon and Wand [94] is suited for specifying operations as sequences of atomic steps, but their atomic specifications are insufficient to reason about file-system clients. On the other hand, atomic Hoare triples in the style of TaDA [30, 28] are well suited for reasoning about file-system clients, but are unable to specify sequences of atomic steps. In conclusion, it clear that in order to address the challenges in specifying and reasoning about POSIX file systems, a combination of both approaches is needed.

4. Formal Methods for File Systems

There has been substantial work on the specification and formal verification of key fragments of POSIX file systems. File systems have even been the subject of a verification mini-challenge by Joshi and Holzmann [61], as part of the verification grand-challenge set by Hoare [55].

We can broadly distinguish the work in this field by the following approaches: model checking (section 4.1), testing (section 4.2), refinement from specification (section 4.3), and program reasoning (section 4.4). The majority of this work is focused on the verification of file-system implementations. Only recently, program reasoning approaches based on Separation Logic have been proposed, to allow reasoning about the behaviour of programs using the POSIX file-system interface. More importantly, the majority of the related work either completely ignores concurrency assuming a sequential fragment of POSIX, or assumes a coarse-grained concurrent behaviour where file-system operations take a single atomic step.

4.1. Model Checking

Model checking has been used to find bugs in file-system implementations with some success. Galloway *et al.* [46] apply model checking tools to the Linux VFS and verify structural integrity properties, deadlock freedom and liveness. By careful examination of the implementation they define a model for Linux VFS in a subset of C. Thus, this model is focused on the internals of a particular implementation and not the POSIX file-system interface. Even though inclusive of concurrency, it is simplified to remain amenable to model checking.

Yang *et al.* [100] develop FiSC, a model checking based tool aimed at discovering bugs in file-system implementations. They applied the tool to four widely used file-system implementations, JFS, ReiserFS, ext3 and XFS, and managed to discover in total 33 bugs of a serious nature. The bugs discovered include permanent data loss after a host failure, crashes due to invalid memory accesses, security issues and memory leaks. FiSC extracts models directly from the implementations, but again these are simplified to remain amenable to model checking. Finally, this work is not focused on concurrency, albeit it does consider fault-tolerance properties of file-system implementations.

4.2. Testing

On of the significant challenges highlighted in model-checking approaches for file systems is the complexity of the search space. This prompted Groce *et al.* [50] to use randomised differential testing to verify the implementation of file system for flash memory. In this approach, the behaviour of the tested implementation is compared with that of a more trusted reference implementation of similar functionality through randomly generated tests. Any difference between the behaviour of the tested system and the reference implementation is considered a potential bug in the tested system. However,

if the reference implementation itself is not verified, both implementations may exhibit the same bug that goes unreported, or bugs reported for the tested system may be false positives.

Ridge *et al.* [83] develop a formal specification of POSIX file systems in the form of operational semantics. They use the specification both to generate a substantial test suite, and as the test oracle in their testing framework. The testing framework has been used on several popular file-system implementations identifying deviations from the POSIX specification. The specification accounts for concurrency but in a coarse-grained manner. The effect of every POSIX operation on the file system is specified through a single transition in the operational semantics. In other words, the specification assumes that every file-operation behaves atomically. As discussed in chapter 2, this assumption is not valid for POSIX file systems.

4.3. Specification and Refinement to Implementation

Morgan and Sufrin developed the first formal specification of a UNIX file system using Z notation [71]. This work predates the development of the POSIX standard and does not consider concurrency. Freitas *et al.* [44, 45] adapt the original Z notation specification of Morgan and Sufrin to align with a fragment of POSIX file systems, mechanise the specification in Z/Eves and finally refine the specification to a Java-based implementation. As in the original specification by Morgan and Sufrin this line of work does not consider concurrency and only specifies sequential file system behaviour.

Damchoom *et al.* [34] develop a formal specification for a tree structured file system in Event-B and Rodin. This work is not intended as a specification of the POSIX file-system interface. Event-B and Rodin are further used by Damchoom and Butler [33] in the verification of a flash-based file system. Again, the formal specification does not aim to follow POSIX, but focuses on the operations of a file-system implementation for flash memory. Both of the aforementioned Event-B specifications account for concurrency by specifying file-system operations as atomic events. The verification of flash-based file systems is also the subject of other approaches [42, 63]. However, these specifications are idealised and not aimed at specifying POSIX file systems.

Hesselink and Lali [52] formalise an abstract hierarchical file system by means of partial functions from paths to data and give specifications for a set of update operations. However, this work does not consider the POSIX file-system interface, but rather a more generic file store. Arkoudas *et al.* [11], develop an abstract file-system specification and an implementation that they prove correct using the Athena theorem prover. The file-system structure specified in this work is flat and does not distinguish between directories and regular files.

4.4. Program Reasoning

Program reasoning using separation logic has recently started to be used both for the specification and verification of file systems, but also to reason about file-system clients.

Ernst *et al.* [40] have used separation logic to verify parts of the Linux VFS. Chen *et al.* [27, 26] use separation logic to build a verified fault-tolerant file-system implementation in the Coq theorem prover. Both works are on sequential fragment of POSIX file systems and do not handle concurrency. In order to prove fault-tolerance of file-system operations this work extends separation logic to reason about the state of the file system after a host failure and the implementation's recovery procedure.

This work was developed concurrently to our own approach for reasoning about fault tolerance that we present in chapter 9. We defer the comparison to chapter 9. Both the work of Ernst *et al.* and Chen *et al.* , are on sequential fragment of POSIX file systems and do not handle concurrency.

Ntzik *et al.* [47] use *structural separation logic* to formally specify a fragment of POSIX file systems. Structural separation logic [97] is a separation logic designed for abstract reasoning about structured data such as trees, dags and lists, combining earlier concepts from context logics [24, 101, 49, 48] with Views [35]. This work motivates the use of separation logic style specifications for POSIX file systems by showing that resource ownership and disjointness allow client reasoning to scale significantly better than the first-order global reasoning. The authors demonstrate their client reasoning by verifying properties of a stylised software installer. The specification in this work only consider simple paths that do not use ".", ".." or symbolic links.

Subsequently, Ntzik and Gardner [72] demonstrate that structural separation logic is not capable to extend the POSIX specification to such paths. The fundamental problem of with such paths is that local updates to the file-system structure have a global effect on the path structure. Even though an update may change a single directory or regular file, the same update may invalidate the path used to identify or even other paths to other files. To remedy this, the authors developed *fusion logic*, a separation logic in which nodes of the file-system structure are associated with permissions based on Boyland's fractional permissions [20]. The permission on each node determines whether it can be updated or only read, and also how it is affected by a local update if used as part of a path. In order for these global effects to be propagated across disjoint resources, fusion logic introduces a novel variation of the frame rule which allows the effects of a local update to be propagated to resources that are framed off. The global effects on paths may cause bugs in client programs using the file system. An example of this is the command line utility $\texttt{rm} - \texttt{r}$ which recursively removes a directory and its contents. The authors applied their approach to popular implementations of this utility and discovered bugs caused by incorrectly handling paths with ".." or symbolic links.

Both the structural separation logic and fusion logic file-system specifications are given for sequential fragments of POSIX. The only type of concurrency reasoning that these specifications can support is disjoint concurrency.

5. Modelling the File System

We formally define the model of the file-system structure used in the subsequent development of our POSIX file-system specification. We define the file-system structure as a directed graph, first in a simpler form without symbolic links, then extended to incorporate symbolic links. We use the basic structure in chapter 6 to introduce our concurrent specification with simplified paths, after which we use the extended structure to demonstrate extended specifications with arbitrary paths.

We model the contents of regular files as sequences of bytes, where a byte is a natural number within the range of natural numbers that can be represented with 8 bits.

Definition 1 (Bytes)**.** *The set of byte values is defined as:*

$$\textsc{Bytes} \triangleq \left\{ n \in \mathbb{N} \;\middle|\; 0 \leq n < 2^8 \right\}$$

We denote the set of byte sequences with $\textsc{Bytes}^*$, where ϵ is the empty byte sequence, and use the notation $\overline{y}$ to denote a byte sequence. We also denote the set of non-empty byte sequences with $\textsc{Bytes}^?$. Additionally, we denote a byte sequence with head y and tail $\overline{y}$ as $y : \overline{y}$, the concatenation of byte sequences $\overline{y}, \overline{y}'$ as $\overline{y} :: \overline{y}'$, and the length of a sequence $\overline{y}$ as $\mathsf{len}(\overline{y})$.

Furthermore, we interpret characters in strings as bytes, according to some implementation defined character encoding, such as ASCII, albeit restricted to a single byte per character. Consequently, strings are just sequences of bytes. For example, the string "hello" is a sequence of 5 bytes, determined by the character encoding. Note that we do not consider strings to be null terminated, as in C. The reason to treat strings as byte sequences is to allow reasoning about file-system clients that store string data in regular files.

Similarly, we want to reason about storing integer values in regular files. For this, we define bounded integers that can be represented as a fixed-length byte sequence. The representation of bounded integers to bytes sequences is implementation defined.

Definition 2 (Bounded Integer Values)**.** *Let* `INT_MAX` *bet the maximum implementation-defined integer value. Let* $\textsc{Int} \triangleq \{-\texttt{INT\_MAX}, -\texttt{INT\_MAX} + 1, \ldots -1, 0, 1, \ldots, \texttt{INT\_MAX} - 1, \texttt{INT\_MAX}\}$ *be a finite continuous range of integers. All elements of* Int *are representable as byte sequences of the same implementation-defined size* `sizeof(int)`*. Let* $\textsc{IBytes} \triangleq \left\{ \overline{y} \in \textsc{Bytes}^? \;\middle|\; \mathsf{len}(\overline{y}) = \texttt{sizeof(int)} \right\}$ *be the set of byte sequences of length* `sizeof(int)`*. The implementation-defined function,* $\mathsf{i2b} : \textsc{Int} \to \textsc{IBytes}$*, converts a bounded integer to its byte-sequence representation of size* `sizeof(int)`*. The implementation-defined function,* $\mathsf{b2i} : \textsc{IBytes} \to \textsc{Int}$*, converts byte sequences to a bounded integer. The two functions are mutually inverse.*

Filenames are strings, used to name the links in the file-system graph. POSIX allows implementations to restrict the characters allowed in filename strings, as long as characters in the *portable filename character set* are always allowed ([7],XBD,3.278).

Definition 3 (Filenames). *Let* $\textsc{FNames} \subseteq \textsc{Bytes}^?$ *be the countable set of all filenames, ranged by* $a, b, \ldots$.

Definition 4 (Dot and dot-dot). *The names dot and dot-dot, denoted by "." and "..", are distinct from filenames.*

As explained in chapter 2, inode numbers, simply referred to as *inodes*, serve as unique identifiers for the nodes in a file-system graph.

Definition 5 (Inodes). *Let* $\textsc{Inodes}$ *be a countable set of* inode numbers*, ranged by* $\iota, j, \ldots$. *Let* $\iota_0 \in \textsc{Inodes}$ *denote the distinguished inode number of the root directory.*

The contents of directory files are modelled as a set of links, where each link associates a filename with an inode.

Definition 6 (Links). *Links are mappings from filenames, "." and "..", to inodes.*

$$\textsc{Links} \triangleq \textsc{FNames} \cup \{., ..\} \xrightarrow{fin} \textsc{Inodes}$$

The contents of regular files are modelled as sequences of bytes. However, regular files may contain gaps; that is, regions where bytes have not been written to. We represent these gaps as sequences of the distinguished value $\emptyset$.

Definition 7 (File Data). *Let* $\emptyset \notin \textsc{Bytes}$ *be a distinguished value, indicating the non-existence of a byte. The set of byte sequences stored in a regular file (including* $\emptyset$*) is defined by the following regular language:*

$$\textsc{FileData} \triangleq \textsc{Bytes}^*; (\{\emptyset\}^*; \textsc{Bytes}^?)^*$$

With all these components in place, we define file-system graphs as a mapping from inodes to either directory contents or regular file contents.

Definition 8 (Basic File-system Graphs). *A basic file-system graph,* $FS \in \mathcal{FS}^-$*, is defined as a mapping from inodes to either directories or regular files.*

$$\mathcal{FS}^- \triangleq \textsc{Inodes} \xrightarrow{fin} \textsc{Links} \cup \textsc{FileData}$$

A file-system graph is well-formed, written $\mathsf{wffs}(FS)$*, if:*

- *It does not contain dangling links.*
- *Every "." links the directory that contains it.*
- *Every ".." links* a *directory that has a link to the directory that contains it.*

$$\begin{aligned}\mathsf{wffs}(FS) \overset{def}{\iff} \forall \delta \in \mathrm{codom}(FS) \cap \textsc{Links}.\ & \\ & \mathrm{codom}(\delta) \subseteq \mathrm{dom}(FS) \\ & \wedge \exists \iota.\, \delta(.) = \iota \Rightarrow FS(\iota)(.) = \iota \\ & \wedge \exists \iota.\, \delta(..) = \iota \Rightarrow \exists a.\, FS(FS(\iota)(a))(..) = \iota\end{aligned}$$

The set of well-formed file-system graphs, $FS \in \mathcal{FS}$ *is defined as:*

$$\mathcal{FS} \triangleq \left\{ FS' \in \mathcal{FS}^- \;\middle|\; \mathsf{wffs}(FS') \right\}$$

Several file-system operations use paths to identify files within the file-system structure.

Definition 9 (Relative Paths)**.** *A relative path, $p \in$ RPaths, is a sequence of filenames, "." and "..", defined by the following grammar.*

$$\begin{array}{rll} p ::= & \varnothing_p & \textit{the empty path} \\ \mid & a & \textit{filename } a \in \textsc{FNames} \cup \{., ..\} \\ \mid & a/p & a \in \textsc{FNames} \cup \{., ..\} \textit{ followed by } p \end{array}$$

Henceforth, we implicitly omit $\varnothing_p$ after the last / in a path. For example, we write $a/b/$ to mean $a/b/\varnothing_p$.

Definition 10 (Absolute paths)**.** *Absolute paths are defined by the set:*

$$\textsc{APaths} \triangleq \left\{ \varnothing_p/p \;\middle|\; p \in \textsc{RPaths} \right\}$$

Henceforth, we implicitly omit $\varnothing_p$ before the first / in an absolute path. For example, we write $/a/b$ to mean $\varnothing_p/a/b$. The set of all paths is defined as: $\textsc{Paths} \triangleq \textsc{RPaths} \cup \textsc{APaths}$.

A symbolic link is a link that maps a filename to a path, instead of an inode. We will discuss how symbolic links affect the resolution of a path within a file-system graph in chapter 6, section 6.3.1.

Definition 11 (File-system Graphs with Symbolic Links)**.** *A file-system graph with symbolic links, $FS \in \mathcal{FS}_s$, is a mapping from inodes to either directories, regular files or paths.*

$$\mathcal{FS}_s \triangleq \textsc{Inodes} \xrightarrow{fin} \textsc{Links} \cup \textsc{FileData} \cup \textsc{Paths}$$

File-system graphs with symbolic links are well formed under the same conditions as basic file-system graphs.

Note that symbolic links may be dangling. The path they map to does not have to exist within the file-system graph.

Files are associated with metadata, which include information such as their type and file access permissions. In this dissertation we focus on file-type metadata and discuss how our POSIX fragment specification can be extended with file-access permissions in chapter 6, section 6.3.2.

Definition 12 (File Types)**.** *File types,*

$$\textsc{FTypes} \triangleq \{\textsc{FType}, \textsc{DType}, \textsc{SType}\}$$

are labels denoting the type of file in a file-system graph, where FType *denotes the regular file type,* DType *denotes the directory file type and* SType *denotes the symbolic link file type.*

When file-system operations fail, they return an error. Each error is identified by an error code associated with the situation that caused the particular failure.

Definition 13 (Error Codes)**.** *The set of error codes is defined as:*

$$\textsc{Errs} \triangleq \{\texttt{ENOENT}, \texttt{ENOTDIR}, \texttt{ENOTEMPTY}, \ldots\}$$

We do not define a complete list of the error codes here. Instead, we discuss the error codes at their point of use, when we develop formal specifications for POSIX operations.

Finally, we define some useful shorthand notation.

Notation 1 (File-system model notation)**.**

$$\mathsf{isfile}(o) \triangleq o \in \textsc{FileData} \qquad \mathsf{iserr}(o) \triangleq o \in \textsc{Errs} \qquad \mathsf{isdir}(o) \triangleq o \in \textsc{Links}$$

$$\mathsf{isempdir}(o) \triangleq \mathsf{isdir}(o) \wedge \mathrm{dom}(o) \subseteq \{\text{“.”}, \text{“..”}\} \qquad \mathsf{ishl}(o) \triangleq o \in \textsc{FileData} \cup \textsc{Links}$$

$$\mathsf{issl}(o) \triangleq o \in \textsc{Paths} \qquad \iota \in FS \triangleq \iota \in \mathrm{dom}(FS) \qquad a \in FS(\iota) \triangleq a \in \mathrm{dom}(FS(\iota))$$

$$a \notin FS(\iota) \triangleq a \notin \mathrm{dom}(FS(\iota)) \qquad \mathsf{ftype}(o) \triangleq \begin{cases} \textsc{FType} & \textit{if } o \in \textsc{FileData} \\ \textsc{DType} & \textit{if } o \in \textsc{Links} \\ \textsc{SType} & \textit{if } o \in \textsc{Paths} \end{cases}$$

$$\mathsf{descendants}(FS, \iota) \triangleq \begin{cases} \emptyset & \textit{if } \neg\mathsf{isdir}(FS(\iota)) \\ \mathrm{codom}(FS(\iota)) \cup \bigcup_{a \in \mathrm{dom}(FS(\iota))} \mathsf{descendants}(FS, FS(\iota)(a)) & \textit{otherwise} \end{cases}$$

$$\mathsf{isabspath}(p) \triangleq p \in \textsc{APaths} \qquad \mathsf{isrelpath}(p) \triangleq p \in \textsc{RPaths}$$

6. Concurrent Specifications and Client Reasoning

So far, we have given an overview of the concurrent behaviour of POSIX file-systems and identified the fundamental challenges to its formal specification in chapter 2. We have also discussed various existing approaches in reasoning about concurrent modules and atomicity, evaluating their applicability to POSIX specifications in chapter 3. In this chapter, we present our approach to a formal specification of POSIX file systems that accounts for the complex concurrent behaviour informally specified in the POSIX standard.

To account for file-system operations that perform sequences of atomic actions on the file-system graph, our specifications take the form of "programs", written in a simple specification language. Our specification language is of a similar style to that of Turon and Wand [94], but with one major and fundamental difference. For atomic actions, we do not rely on *primitive* atomicity, but on *abstract* atomicity as developed in recent program logics for fine-grained concurrency. In particular, we base the specification of atomic actions on the atomic Hoare triples introduced in the program logic TaDA [30].

In section 6.1, we introduce the key features of our specification language and its associated refinement calculus, through characteristic examples of file-system operations. This is done in the context of a simplified POSIX file-system fragment. In section 6.2, we demonstrate how our specification language and refinement calculus are used to reason about client applications by considering the example of a lock-file module. In particular, we demonstrate how we address file systems being a public namespace through specifications conditional on context invariants. Finally, in section 6.3 we demonstrate the flexibility of our approach by considering various extensions of our specification to larger POSIX file-system fragments. We defer the formal definition of our specification language, its semantics and associated refinement calculus to chapter 7. Their use in specifying POSIX file-system operations and reasoning about client programs here provides the intuition for the formal treatment later.

6.1. Specifications

To simplify the presentation and focus on the specification of the concurrent behaviour of POSIX file-system operations, we will work with a simplified fragment of POSIX file systems in this section. We work with the basic file-system graphs of definition 8 and define the specifications in this section in terms of absolute paths. In section 6.3, we discuss extensions of the specifications developed here to larger POSIX fragments.

6.1.1. Operations on Links

In section 2.3.1, we have informally described the behaviour of the `unlink`(*path*) operation. It performs a sequence of atomic steps which resolve the argument *path* and removes the link to the file the path

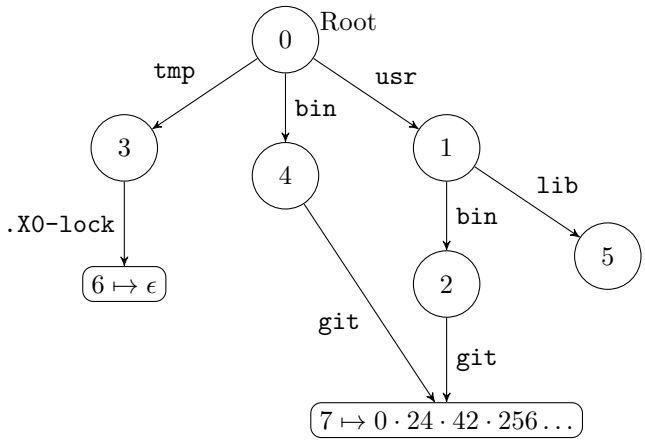

Figure 6.1.: Example snapshot of the file-system graph.

identifies. We define the specification of `unlink` in the form of the following *specification program*:

$$
\begin{aligned}
\textbf{let}\ \texttt{unlinkSpec}(\mathit{path}) \triangleq\ & \textbf{let}\ p = \texttt{dirname}(\mathit{path}); \\
& \textbf{let}\ a = \texttt{basename}(\mathit{path}); \\
& \textbf{let}\ r = \texttt{resolve}(p, \iota_0); \\
& \textbf{if}\ \neg\mathsf{iserr}(r)\ \textbf{then} \\
& \quad \texttt{return}\ \texttt{link\_delete}(r, a) \\
& \qquad\qquad \sqcup\, \texttt{link\_delete\_notdir}(r, a) \\
& \textbf{else}\ \texttt{return}\ r\ \textbf{fi}
\end{aligned}
$$

The specification program initially splits the *path* argument to the path prefix p and last name a, using `dirname` and `basename` respectively. If *path* contains only one name, then `dirname` returns `null`. The path-prefix p is then resolved by calling $\texttt{resolve}(p, \iota_0)$. The second argument to `resolve` is the inode number of the directory from which to start the path resolution. In figure 6.1, this would be the directory with inode 0. To simplify the presentation in this section, we assume that paths are absolute (chapter 5, definition 10), and therefore we always start the resolution from the root directory, which has the known fixed inode ι_0. If the resolution fails with an error code, we return it. If the resolution succeeds, the return value is the inode of the directory containing the link we want to remove. As we previously discussed in chapter 2.3, POSIX gives implementations the option to prevent attempts to remove links to directories. This freedom of choice given to implementations introduces *angelic non-determinism*. An implementation is allowed to choose which behaviour it implements. On the other hand, clients must be robust with respect to both behaviours. Essentially, this means that using `unlink` on directories is not portable across different implementations. To account for this, we use the non-deterministic *angelic choice* operator, $\sqcup$, to compose the operations `link_delete` and `link_delete_notdir`. We will define `link_delete` to remove any link, and `link_delete_notdir` to return an error for a directory link.

The `resolve` operation is defined as a function that recursively follows *path*, starting from the initial

directory with inode ι:

$$
\begin{array}{l}
\textbf{letrec}\ \texttt{resolve}(\mathit{path}, \iota) \triangleq \\
\quad \textbf{if}\ \mathit{path} = \texttt{null}\ \textbf{then}\ \texttt{return}\ \iota\ \textbf{else} \\
\quad\quad \textbf{let}\ a = \texttt{head}(\mathit{path}); \\
\quad\quad \textbf{let}\ p = \texttt{tail}(\mathit{path}); \\
\quad\quad \textbf{let}\ r = \texttt{link\_lookup}(\iota, a); \\
\quad\quad \textbf{if}\ \mathsf{iserr}(r)\ \textbf{then}\ \texttt{return}\ r \\
\quad\quad \textbf{else}\ \texttt{return}\ \texttt{resolve}(p, r)\ \textbf{fi} \\
\quad \textbf{fi}
\end{array}
$$

The `head` and `tail` operations return the first name and the path postfix of the path argument. Note that if *path* is a single name, then `tail` returns `null`. In each step, `resolve` calls `link_lookup`(ι, a) to get the inode of the file the link named a points to, if that link exists in the directory with inode ι. If the link a does not exist in the ι directory, or if the ι file is not a directory, `link_lookup` returns an error, the resolution stops and the error is immediately returned. The procedure returns the resolved inode when there is no more path to resolve, i.e. the postfix p of the *path* argument is `null`.

Elements of our specifications are reusable. Any specification that resolves paths will use this definition. In line with programming practice, we reuse specification code as much as possible.

The `unlink` operation must behave in at most the same way as the specification program `unlinkSpec`. In other words a correct implementation must be a refinement of our specification program:

$$\texttt{unlink}(\mathit{path}) \sqsubseteq \texttt{unlinkSpec}(\mathit{path})$$

The refinement relation, $\sqsubseteq$, is contextual: in any concurrent context, every behaviour of `unlink` is a behaviour of `unlinkSpec`. Therefore to reason about a client (a particular context), we can replace the implementation with its specification. If C is a particular context, then $C[\texttt{unlink}(\mathit{path})] \sqsubseteq C[\texttt{unlinkSpec}(\mathit{path})]$. Then we can define another specification program ϕ, to be the specification of the client, so that $C[\texttt{unlinkSpec}(\mathit{path})] \sqsubseteq \phi$. To facilitate such reasoning, we develop a refinement calculus for our specification language, consisting of an array of refinement laws that allow us to prove such refinements.

Our `unlink` specification is incomplete. We have yet to define the aforementioned operations `link_lookup`, `link_delete` and `link_delete_notdir` that lookup and delete a link in a directory. Note that these are not POSIX operations, but abstract operations corresponding to atomic actions that POSIX operations perform.

Our aim is to define the abstract atomic action performed by an operation, under any possible interference from the concurrent environment. Recall from chapter 3, section 3.5, the *atomic Hoare triples* introduced by TaDA[30], of the form:

$$\forall x \in X.\, \langle P(x) \rangle\, \mathbb{C}\, \langle Q(x) \rangle$$

Intuitively, the atomic triple specifies that the program $\mathbb{C}$ atomically updates the state satisfying the precondition $P(x)$ to a state satisfying the postcondition $Q(x)$, for an arbitrary $x \in X$. The pseudo-quantifier, $\forall x \in X$, serves to restrict the interference from the environment that the specification can tolerate: before the atomic update is committed, the environment must preserve $P(x)$, but is allowed

to change x within the set X; after the atomic update is committed, all constraints on the environment are lifted. The atomic triple thus constitutes a contract between clients and implementations: the clients can assume that the precondition holds for some $x \in X$ until the update is committed.

Atomic Hoare triples explicitly mention the code $\mathbb{C}$ implementing the specification. For an abstract operation, such as `link_delete`, we only care about its behaviour, not the code that implements it. In our specification language we use *atomic specification statements*, of the form $\mathbb{V}x \in X.\, \langle P(x),\ Q(x)\rangle$, to specify atomic behaviour, without referencing the implementation. Intuitively, an atomic specification statement specifies the behaviour of any program satisfying the triple $\mathbb{V}x \in X.\, \langle P(x)\rangle - \langle Q(x)\rangle$. In the setting of our refinement calculus, an atomic Hoare triple $\mathbb{V}x \in X.\, \langle P(x)\rangle\, \mathbb{C}\, \langle Q(x)\rangle$ corresponds to a refinement $\mathbb{C} \sqsubseteq \mathbb{V}x \in X.\, \langle P(x),\ Q(x)\rangle$.

Now to complete our specification of the `unlink` operation, we define `link_lookup`, `link_delete` and `link_delete_notdir` as the functions given in figure 6.2. Consider `link_delete` used in the definition of `unlinkSpec` earlier, repeated below:

$$
\begin{aligned}
&\textbf{let } \texttt{link\_delete}(\iota, a) \triangleq \\
&\quad \mathbb{V}FS.\, \langle \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota))\,,\ a \in FS(\iota) \Rightarrow \mathsf{fs}(FS[\iota \mapsto FS(\iota) \setminus \{a\}]) * \texttt{ret} = 0\rangle \\
&\quad \sqcap\, \texttt{return enoent}(\iota, a) \\
&\quad \sqcap\, \texttt{return enotdir}(\iota)
\end{aligned}
$$

There are three cases composed with $\sqcap$, which we will explain shortly. Consider the atomic specification statement of the first case. In the precondition $\mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota))$, the abstract predicate $\mathsf{fs}(FS)$ states that the file-system structure is given by the file-system graph FS, and the pure predicate $\mathsf{isdir}(FS(\iota))$ states that a directory with inode ι must exist in that file-system graph. The pseudo-quantifier[1], $\mathbb{V}FS$, states that the environment may arbitrarily change the file system, as along as the precondition is satisfied, which, in this case, means that a directory with inode ι exists. The postcondition states that if, at the point the atomic update takes effect, the link named a exists in the directory with inode ι, the link is removed and the return variable `ret` is bound to 0. As a convention, we use the variable `ret` within a function to bind its return value.

The other two cases specify erroneous behaviour via the functions `enoent` and `enotdir`, defined in figure 6.3. Consider the `enoent` function repeated below:

$$
\textbf{let } \texttt{enoent}(\iota, a) \triangleq \mathbb{V}FS.\, \langle \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota))\,,\ a \notin FS(\iota) \Rightarrow \mathsf{fs}(FS) * \texttt{ret} = \texttt{ENOENT}\rangle
$$

The atomic specification statement specifies that, if a link named a does not exist in the directory with inode ι, the file system is not modified and the return variable is bound to the error code `ENOENT`. The second error case defined by `enotdir` specifies that if the inode ι does not identify a directory, the file system remains unchanged and the error code `ENOTDIR` is returned. Most of the file-system related errors defined by POSIX are common to most file-system operations, and, similarly to `resolve`, we treat functions that specify error cases as reusable components for specifications.

The three specification cases in `link_delete` are composed with the non-deterministic *demonic choice* operator $\sqcap$. We use demonic choice to account for the non-determinism of a specification due to scheduling behaviour. In the case of `link_delete`, which of the three possible behaviours we observe

[1] We write $\mathbb{V}x$ when x ranges over its entire domain instead of some subset.

$$\textbf{let}\ \texttt{link\_lookup}(\iota, a) \triangleq$$
$$\quad \forall FS.\, \langle \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota))\,,\ a \in FS(\iota) \Rightarrow \mathsf{fs}(FS) * \texttt{ret} = FS(\iota)(a) \rangle$$
$$\quad \sqcap\, \texttt{return enoent}(\iota, a)$$
$$\quad \sqcap\, \texttt{return enotdir}(\iota)$$

$$\textbf{let}\ \texttt{link\_delete}(\iota, a) \triangleq$$
$$\quad \forall FS.\, \langle \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota))\,,\ a \in FS(\iota) \Rightarrow \mathsf{fs}(FS[\iota \mapsto FS(\iota) \setminus \{a\}]) * \texttt{ret} = 0 \rangle$$
$$\quad \sqcap\, \texttt{return enoent}(\iota, a)$$
$$\quad \sqcap\, \texttt{return enotdir}(\iota)$$

$$\textbf{let}\ \texttt{link\_delete\_notdir}(\iota, a) \triangleq$$
$$\quad \forall FS.\, \langle \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota))\,,\ \mathsf{isfile}(FS(FS(\iota)(a))) \Rightarrow \mathsf{fs}(FS[\iota \mapsto FS(\iota) \setminus \{a\}]) * \texttt{ret} = 0 \rangle$$
$$\quad \sqcap\, \texttt{return enoent}(\iota, a)$$
$$\quad \sqcap\, \texttt{return enotdir}(\iota)$$
$$\quad \sqcap\, \texttt{return err\_nodir\_hlinks}(\iota, a)$$

$$\textbf{let}\ \texttt{link\_insert}(\iota, a, j, b) \triangleq$$
$$\quad \forall FS.\, \left\langle \begin{array}{l} \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota)) \wedge \mathsf{isdir}(FS(j))\,, \\ a \in FS(\iota) \wedge b \notin FS(j) \Rightarrow \mathsf{fs}(FS[j \mapsto FS(j)[b \mapsto FS(\iota)(a)]]) * \texttt{ret} = 0 \end{array} \right\rangle$$
$$\quad \sqcap\, \texttt{return enoent}(\iota, a)$$
$$\quad \sqcap\, \texttt{return eexist}(j, b)$$
$$\quad \sqcap\, \texttt{return enotdir}(\iota)$$
$$\quad \sqcap\, \texttt{return enotdir}(j)$$

$$\textbf{let}\ \texttt{link\_insert\_notdir}(\iota, a, j, b) \triangleq$$
$$\quad \forall FS.\, \left\langle \begin{array}{l} \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota)) \wedge \mathsf{isdir}(FS(j))\,, \\ \mathsf{isfile}(FS(FS(\iota)(a))) \wedge b \notin FS(j) \Rightarrow \mathsf{fs}(FS[j \mapsto FS(j)[b \mapsto FS(\iota)(a)]]) * \texttt{ret} = 0 \end{array} \right\rangle$$
$$\quad \sqcap\, \texttt{return enoent}(\iota, a)$$
$$\quad \sqcap\, \texttt{return eexist}(j, b)$$
$$\quad \sqcap\, \texttt{return enotdir}(\iota)$$
$$\quad \sqcap\, \texttt{return enotdir}(j)$$
$$\quad \sqcap\, \texttt{return err\_nodir\_hlinks}(\iota, a)$$

Figure 6.2.: Specification of atomic operations for link lookup, insertion and deletion.

$$\textbf{let}\ \texttt{enoent}(\iota, a) \triangleq \forall FS.\, \langle \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota))\,,\ a \notin FS(\iota) \Rightarrow \mathsf{fs}(FS) * \texttt{ret} = \texttt{ENOENT} \rangle$$

$$\textbf{let}\ \texttt{enotdir}(\iota) \triangleq \forall FS.\, \langle \mathsf{fs}(FS) \wedge \neg\mathsf{isdir}(FS(\iota))\,,\ \mathsf{fs}(FS) * \texttt{ret} = \texttt{ENOTDIR} \rangle$$

$$\textbf{let}\ \texttt{eexist}(\iota, a) \triangleq \forall FS.\, \langle \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota))\,,\ a \in FS(\iota) \Rightarrow \mathsf{fs}(FS) * \texttt{ret} = \texttt{EEXIST} \rangle$$

$$\textbf{let}\ \texttt{err\_nodir\_hlinks}(\iota, a) \triangleq$$
$$\quad \forall FS.\, \langle \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota))\,,\ \mathsf{isdir}(FS(FS(\iota)(a))) \Rightarrow \mathsf{fs}(FS) * \texttt{ret} = \texttt{EPERM} \rangle$$

Figure 6.3.: Specification of error cases in operations on links.

in a particular execution depends not only on the environment, but also on which of the possible interleavings the scheduler decides to execute. Thus, we consider the scheduler to act as a demon and we call such specifications demonic. A client of a demonic specification must be robust against all the demonic cases. For example, `link_delete` handles errors by returning the error code to the client. When reasoning about a particular client, if we have information that restricts the environment, for example by requiring some path to always exist, we can elide the cases that are no longer applicable. On the other hand, an implementation of a demonic specification must implement all the demonic cases. For example, an implementation of `link_delete` must implement all three atomic specification statements. The definition of `link_delete_notdir` is similar, except that it succeeds only when the link being removed does not link a directory, and an extra error case is added for when it does.

The final piece required to complete the specification of `unlink` is the abstract operation `link_lookup`, which we use in `resolve` and define in figure 6.2. The operation succeeds if it is able to find a link named a within the directory identified by inode ι. The operation fails, by returning an error, in the same way as `link_delete`. This means that by the error codes alone, a client only determines why `unlink` fails, and not when, whether it was during the path resolution, or during the attempted update.

Through the `unlink` example we have seen so far, we demonstrate how our refinement-based approach, of using specification programs, succeeds in tackling some of the challenges of developing a concurrent specification of POSIX file systems. With sequential composition of atomic specification statements we specify file-system operations performing multiple atomic steps. With demonic choice composition we address the non-determinism due to concurrent interleavings. However, with angelic choice composition alone, we cannot efficiently address all the non-determinism present due to optional implementation behaviours.

Consider the `link`(*source*, *target*) operation, which acts as the dual of `unlink`. Informally, it creates a new link identified by the path *target* to the file identified by *source*, if it does not already exist. The operation has to resolve two paths before the actual linking is attempted. POSIX does not specify the order in which multiple path arguments are resolved. Since path resolution is a sequence of atomic steps, the choice space for implementations is substantially larger than if it was atomic. To address this fact we employ parallel composition, $\|$, and compose multiple path resolutions in parallel. Formally, we give the following refinement specification to `link`:

$$
\begin{aligned}
&\texttt{link}(\mathit{source}, \mathit{target}) \\
&\quad \sqsubseteq \textbf{let } p_s = \texttt{dirname}(\mathit{source}); \\
&\qquad \textbf{let } a = \texttt{basename}(\mathit{source}); \\
&\qquad \textbf{let } p_t = \texttt{dirname}(\mathit{target}); \\
&\qquad \textbf{let } b = \texttt{basename}(\mathit{target}); \\
&\qquad \textbf{let } r_s, r_t = \texttt{resolve}(p_s, \iota_0) \parallel \texttt{resolve}(p_t, \iota_0); \\
&\qquad \textbf{if } \neg\mathsf{iserr}(r_s) \wedge \neg\mathsf{iserr}(r_t) \textbf{ then} \\
&\qquad\quad \texttt{return link\_insert}(r_s, a, r_t, b) \\
&\qquad\qquad\quad \sqcup\ \texttt{link\_insert\_notdir}(r_s, a) \\
&\qquad \textbf{else if } \mathsf{iserr}(r_s) \wedge \neg\mathsf{iserr}(r_t) \textbf{ then } \texttt{return } r_s \\
&\qquad \textbf{else if } \neg\mathsf{iserr}(r_s) \wedge \mathsf{iserr}(r_t) \textbf{ then } \texttt{return } r_t \\
&\qquad \textbf{else if } \mathsf{iserr}(r_s) \wedge \mathsf{iserr}(r_t) \textbf{ then } \texttt{return } r_s \sqcup \texttt{return } r_t \textbf{ fi}
\end{aligned}
$$

The link insertion is attempted when both resolutions succeed. In that case, analogously to `unlink`, we use angelic choice between `link_insert` and `link_insert_notdir`. The former allows the link to be created for any link, even to a directory, whereas the latter considers this erroneous. The atomic specification statements for both are defined in figure 6.2. Error handling must be robust against errors from both resolutions. Note that if both resolutions error, either error code is returned. In general, a client is unable to determine which path resolution triggered the error.

The parallel composition of the two path resolutions, $\texttt{resolve}(p_s, \iota_0) \parallel \texttt{resolve}(p_t, \iota_0)$, gives the maximum freedom of choice to implementations. Not only can the resolutions be implemented in any order, but they can also be interleaved in all possible ways. This fact is engraved in our refinement calculus. For any two specification programs ϕ and ψ, their sequential composition is a refinement of their parallel composition, as expressed by the SeqPar refinement law:

$$\phi; \psi \sqsubseteq \phi \parallel \psi$$

This means that we can replace (refine) the parallel composition of ϕ and ψ with a sequential composition, without introducing any additional behaviour. On the other hand, if we replace (abstract) the sequential composition with a parallel composition we are introducing additional behaviours, which may include undefined behaviours. In particular, if ϕ and ψ are not thread-safe, composing them in parallel will lead to undefined behaviour. Semantically, we treat undefined behaviour as faulting. Faulting acts as the most permissive of specifications; it can be refined by anything. Note that we treat refinement from a partial correctness perspective. Refinement does not preserve nor guarantee termination. We discuss faulting behaviour and termination when we formally define contextual refinement in chapter 7, section 7.4.1.

Since parallel composition commutes, its sequential implementation commutes as well. Furthermore, if ϕ and ψ are sequences, $\phi_1; \phi_2$ and $\psi_1; \psi_2$ respectively, then an implementation can process the sequences in parallel instead, as captured by Hoare's Exchange law [56]:

$$(\phi_1 \parallel \psi_1); (\phi_2 \parallel \psi_2) \sqsubseteq (\phi_1; \phi_2) \parallel (\psi_1; \psi_2)$$

Combining this with SeqPar and commutativity of parallel composition, the implementation can interleave the sequents in order it so pleases. Suffice to say an implementation `link_insert` may choose to resolve the two paths truly in parallel.

Finally, consider the $\texttt{rename}(\mathit{source}, \mathit{target})$ operation, which moves the link identified by the path *source* so that it becomes a link identified by the path *target*. It is arguably the most complex file-system operation acting on links as it exhibits different behaviour depending on whether it is working with links to directories or regular files, and whether the target already exists. Fortunately, the compositional nature of our specification programs aids, to some extend, in tackling its complexity. The specification for `rename` is given in figure 6.4. The path resolution is specified as in `link`, using parallel composition, to indicate the unordered resolution of multiple paths. If both path prefixes are successfully resolved, then the move is attempted. There are several success cases, defined in figure 6.5.

The first case, `link_move_noop`, specifies that when the source link named a and the target link named b, exist within the directories with inodes ι_s and ι_t respectively, and they link files of the same type, the operation will succeed without modifying the file-system graph. The subsequent success

```
rename(source, target)
  ⊑ let p_s = dirname(source);
    let a = basename(source);
    let p_t = dirname(target);
    let b = basename(target);
    let r_s, r_t = resolve(p_s, ι_0) ∥ resolve(p_t, ι_0);
    if ¬iserr(r_s) ∧ ¬iserr(r_t) then
      return //Success cases
             link_move_noop(r_s, a, r_t, b)
             ⊓ link_move_file_target_not_exists(r_s, a, r_t, b)
             ⊓ link_move_file_target_exists(r_s, a, r_t, b)
             ⊓ link_move_dir_target_not_exists(r_s, a, r_t, b)
             ⊓ link_move_dir_target_exists(r_s, a, r_t, b)
             //Error cases
             ⊓ enoent(r_s, a)
             ⊓ enotdir(r_s)
             ⊓ enotdir(r_t)
             ⊓ err_source_isfile_target_isdir(r_s, a, r_t, b)
             ⊓ err_source_isdir_target_isfile(r_s, a, r_t, b)
             ⊓ err_target_notempty(r_t, b)
             ⊓ err_target_isdescendant(r_s, a, r_t, b)
    else if iserr(r_s) ∧ ¬iserr(r_t) then return r_s
    else if ¬iserr(r_s) ∧ iserr(r_t) then return r_t
    else if iserr(r_s) ∧ iserr(r_t) then return r_s ⊔ return r_t fi
```

Figure 6.4.: Specification of `rename`.

cases specify the different behaviours on regular file links and directory links.

Consider the case of `link_move_file_target_not_exists`. The postcondition states that if at the point the atomic update takes effect, the link named a within the ι_s directory identifies a regular file, and a link name b does not exist in the directory with inode ι_t, then the link named a is moved into the ι_t directory and renamed to b. The `link_move_file_target_exists` case specifies that when the link named b does already exists, then it must link a regular file and the link is overwritten.

The `link_move_dir_target_not_exists` and `link_move_dir_target_exists` cases work analogously for directory links, albeit they place additional restrictions. Specifically, both require that the directory identified by inode ι_t is not a descendant of the directory identified by inode ι_s. Otherwise, the operation would potentially cause the ι_s directory to become disconnected from the graph. Additionally, in `link_move_dir_target_exists`, when the link named b already exists in the ι_t directory it must link an empty directory. Note that moving the directory link may update the ".." link within that directory so that it points to the correct parent.

The `rename` operation has numerous error cases. It shares the `enoent` and `enotdir` error cases with `link` and `unlink`, but also requires additional cases defined in figure 6.6, for returning the appropriate error code in case the success conditions of `link_move_file` and `link_move_dir` are not met. The `err_source_isfile_target_isdir` returns the error code `EISDIR` if the link named a identifies a file while the link named b identifies a directory, whereas `err_source_isdir_target_isfile` returns `ENOTDIR` in the opposite case. The `err_target_notempty` case applies when we attempt to move a

$$\textbf{let } \texttt{link\_move\_noop}(\iota_s, a, \iota_t, b) \triangleq$$
$$\mathbb{V}FS.\, \langle \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota_s)) \wedge \mathsf{isdir}(FS(\iota_t))\,,\ FS(\iota_s)(a) = FS(\iota_t)(b) \Rightarrow \mathsf{fs}(FS) * \texttt{ret} = 0 \rangle$$

$$\textbf{let } \texttt{link\_move\_file\_target\_not\_exists}(\iota_s, a, \iota_t, b) \triangleq$$
$$\mathbb{V}FS.\, \left\langle \begin{array}{l} \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota_s)) \wedge \mathsf{isdir}(FS(\iota_t))\,, \\ \mathsf{isfile}(FS(FS(\iota_s)(a))) \wedge b \notin FS(\iota_t) \\ \quad \Rightarrow \mathsf{fs}(FS[\iota_s \mapsto FS(\iota_s) \setminus \{a\}][\iota_t \mapsto FS(\iota_t)[b \mapsto FS(\iota_s)(a)]]) * \texttt{ret} = 0 \end{array} \right\rangle$$

$$\textbf{let } \texttt{link\_move\_file\_target\_exists}(\iota_s, a, \iota_t, b) \triangleq$$
$$\mathbb{V}FS.\, \left\langle \begin{array}{l} \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota_s)) \wedge \mathsf{isdir}(FS(\iota_t))\,, \\ \mathsf{isfile}(FS(FS(\iota_s)(a))) \wedge \mathsf{isfile}(FS(FS(\iota_t)(b))) \\ \quad \Rightarrow \mathsf{fs}(FS[\iota_s \mapsto FS(\iota_s) \setminus \{a\}][\iota_t \mapsto FS(\iota_t)[b \mapsto FS(\iota_s)(a)]]) * \texttt{ret} = 0 \end{array} \right\rangle$$

$$\textbf{let } \texttt{link\_move\_dir\_target\_not\_exists}(\iota_s, a, \iota_t, b) \triangleq$$
$$\mathbb{V}FS.\, \left\langle \begin{array}{l} \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota_s)) \wedge \mathsf{isdir}(FS(\iota_t))\,, \\ \mathsf{isdir}(FS(FS(\iota_s)(a))) \wedge \iota_t \notin \mathsf{descendants}(FS, \iota_s) \wedge b \notin FS(\iota_t) \\ \quad \Rightarrow \exists FS'.\, FS' = FS[\iota_s \mapsto FS(\iota_s) \setminus \{a\}][\iota_t \mapsto FS(\iota_t)[b \mapsto FS(\iota_s)(a)]] \\ \qquad \wedge (FS'(\iota_t)(b)(\text{“..”}) \neq \iota_s \Rightarrow \mathsf{fs}(FS')) \\ \qquad \wedge (FS'(\iota_t)(b)(\text{“..”}) = \iota_s \Rightarrow \mathsf{fs}(FS'[FS'(\iota_t)(b) \mapsto FS'(\iota_t)(b)[\text{“..”} \mapsto \iota_t]])) \\ \qquad * \texttt{ret} = 0 \end{array} \right\rangle$$

$$\textbf{let } \texttt{link\_move\_dir\_target\_exists}(\iota_s, a, \iota_t, b) \triangleq$$
$$\mathbb{V}FS.\, \left\langle \begin{array}{l} \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota_s)) \wedge \mathsf{isdir}(FS(\iota_t))\,, \\ \mathsf{isdir}(FS(FS(\iota_s)(a))) \wedge \iota_t \notin \mathsf{descendants}(FS, \iota_s) \wedge \mathsf{isempdir}(FS(FS(\iota_t)(b))) \\ \quad \Rightarrow \exists FS'.\, FS' = FS[\iota_s \mapsto FS(\iota_s) \setminus \{a\}][\iota_t \mapsto FS(\iota_t)[b \mapsto FS(\iota_s)(a)]] \\ \qquad \wedge (FS'(\iota_t)(b)(\text{“..”}) \neq \iota_s \Rightarrow \mathsf{fs}(FS')) \\ \qquad \wedge (FS'(\iota_t)(b)(\text{“..”}) = \iota_s \Rightarrow \mathsf{fs}(FS'[FS'(\iota_t)(b) \mapsto FS'(\iota_t)(b)[\text{“..”} \mapsto \iota_t]])) \\ \qquad * \texttt{ret} = 0 \end{array} \right\rangle$$

Figure 6.5.: Specification of atomic link moving operations.

$$\textbf{let } \texttt{err\_source\_isfile\_target\_isdir}(\iota_s, a, \iota_t, b) \triangleq$$
$$\forall FS. \left\langle \begin{array}{l} \mathsf{fs}(FS) \wedge \iota_s \in FS \wedge \iota_t \in FS, \\ \mathsf{isfile}(FS(FS(\iota_s)(a))) \wedge \mathsf{isdir}(FS(FS(\iota_t)(b))) \Rightarrow \mathsf{fs}(FS) * \texttt{ret} = \texttt{EISDIR} \end{array} \right\rangle$$

$$\textbf{let } \texttt{err\_source\_isdir\_target\_isfile}(\iota_s, a, \iota_t, b) \triangleq$$
$$\forall FS. \left\langle \begin{array}{l} \mathsf{fs}(FS) \wedge \iota_s \in FS \wedge \iota_t \in FS, \\ \mathsf{isdir}(FS(FS(\iota_s)(a))) \wedge \mathsf{isfile}(FS(FS(\iota_t)(b))) \Rightarrow \mathsf{fs}(FS) * \texttt{ret} = \texttt{ENOTDIR} \end{array} \right\rangle$$

$$\textbf{let } \texttt{err\_target\_notempty}(\iota_t, b) \triangleq$$
$$\forall FS. \langle \mathsf{fs}(FS) \wedge \iota_t \in FS,\ \mathsf{isempdir}(FS(FS(\iota_t)(b))) \Rightarrow \mathsf{fs}(FS) * \texttt{ret} \in \{\texttt{EEXIST}, \texttt{ENOTEMPTY}\} \rangle$$

$$\textbf{let } \texttt{err\_target\_isdescendant}(\iota_s, a, \iota_t, b) \triangleq$$
$$\forall FS. \langle \mathsf{fs}(FS) \wedge \iota_s \in FS \wedge \iota_t \in FS,\ \iota_t \in \mathsf{descendants}(FS, FS(\iota_s)(a)) \Rightarrow \mathsf{fs}(FS) * \texttt{ret} = \texttt{EINVAL} \rangle$$

Figure 6.6.: Additional error case specifications for `rename`.

directory link, but the existing link points to a non-empty directory. In this case, POSIX allows either `EEXIST` or `ENOTEMPTY` to be returned. Finally, `err_target_isdescendant` returns `EINVAL` if the target directory is a descendant of the source directory.

6.1.2. Operations on Directories

With the examples in the previous section we have seen how links to existing directories can be moved with `rename`, as well as added with `link` and removed with `unlink` (if implementations allow it). New directories are created with the `mkdir` operation. Informally, `mkdir`(*path*), creates a new empty directory named by the last component of *path*, within the directory resolved by the path prefix of *path*. Formally, we can give the following refinement specification to `mkdir`:

$$\begin{array}{l} \texttt{mkdir}(path) \\ \quad \sqsubseteq \textbf{let } p = \texttt{dirname}(path); \\ \quad\quad \textbf{let } a = \texttt{basename}(path); \\ \quad\quad \textbf{let } r = \texttt{resolve}(p, \iota_0); \\ \quad\quad \textbf{if } \neg\mathsf{iserr}(r) \textbf{ then} \\ \quad\quad\quad \texttt{return link\_new\_dir}(r, a) \\ \quad\quad\quad\quad\quad \sqcap\, \texttt{eexist}(\iota, a) \\ \quad\quad\quad\quad\quad \sqcap\, \texttt{enotdir}(\iota) \\ \quad\quad \textbf{else return } r \textbf{ fi} \end{array}$$

The path resolution is exactly as in `unlink`. If path resolution succeeds, either the directory is created by `link_new_dir`, or an error is returned according to `enoent` and `enotdir`, which are defined in figure 6.3. We define `link_new_dir` to atomically create the new empty directory as follows:

$$\textbf{let } \texttt{link\_new\_dir}(\iota, a) \triangleq$$
$$\forall FS. \left\langle \begin{array}{l} \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota)), \\ a \notin FS(\iota) \Rightarrow \exists \iota'. \mathsf{fs}(FS[\iota \mapsto FS(\iota)[a \mapsto \iota']] \uplus \iota' \mapsto \emptyset[\text{“.”} \mapsto \iota'][\text{“..”} \mapsto \iota]) * \texttt{ret} = 0 \end{array} \right\rangle$$

Note that the operation creates the new directory with "." linking itself and ".." linking its parent.

Recall the discussion in chapter 2.3.3 on the atomicity of directory operations. Even though in most modern implementations the directory is created in an atomic step, this is not mandated by POSIX. The rationale behind this underspecification is to allow for historical implementations where the directory is initially created *truly empty*, and the "." and ".." links are added in subsequent steps. We can easily formalise this behaviour as the following sequence of atomic steps:

$$\begin{aligned}&\exists \iota'.\, \forall FS.\, \langle \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota))\,,\ a \notin FS(\iota) \Rightarrow \mathsf{fs}(FS[\iota \mapsto FS(\iota)[a \mapsto \iota']] \uplus \iota' \mapsto \emptyset) * \mathtt{ret} = 0\rangle;\\ &\quad \forall FS.\, \langle \mathsf{fs}(FS) \wedge \text{“.”} \notin FS(\iota'),\ \mathsf{fs}(FS[\iota' \mapsto FS(\iota')[\text{“.”} \mapsto \iota']])\rangle;\\ &\quad \forall FS.\, \langle \mathsf{fs}(FS) \wedge \text{“..”} \notin FS(\iota'),\ \mathsf{fs}(FS[\iota' \mapsto FS(\iota')[\text{“..”} \mapsto \iota']])\rangle;\end{aligned}$$

The behaviour of this sequence is related to `link_new_dir`. Every observable behaviour of `link_new_dir` can also be observed in the above sequence. In fact, any sequence of atomic steps, is sometimes observed as happening atomically, simply because sometimes the scheduler does not interleave those steps with other threads. In trace semantics this is known as *mumbling* [21], and in our refinement calculus it is expressed as the AMumble refinement law, a simplified version of which is as follows:

$$\forall x \in X.\, \langle P(x),\ Q(x)\rangle \sqsubseteq \forall x \in X.\, \big\langle P(x),\ P'(x)\big\rangle; \forall x \in X.\, \big\langle P'(x),\ Q(x)\big\rangle$$

With this refinement law it is apparent that the single atomic step in `link_new_dir` is a refinement of the sequence of three atomic steps. Note however, that an implementation can choose in which order to add the "." and ".." links. Let us abstract each of those steps into the following function:

$$\textbf{let}\ \mathtt{link\_ins\_dir}(\iota, a, \iota') \triangleq \forall FS.\, \big\langle \mathsf{fs}(FS) \wedge a \notin FS(\iota),\ \mathsf{fs}\big(FS[\iota \mapsto FS(\iota)[a \mapsto \iota']]\big)\big\rangle$$

Now we can define the non-atomic directory creation operation as follows:

$$\begin{aligned}&\textbf{let}\ \mathtt{link\_new\_dir\_nonatomic}(\iota, a) \triangleq\\ &\quad \exists \iota'.\, \forall FS.\, \langle \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota))\,,\ a \notin FS(\iota) \Rightarrow \mathsf{fs}(FS[\iota \mapsto FS(\iota)[a \mapsto \iota']] \uplus \iota' \mapsto \emptyset) * \mathtt{ret} = 0\rangle;\\ &\qquad \mathtt{link\_ins\_dir}(\iota', \text{“.”}, \iota') \parallel \mathtt{link\_ins\_dir}(\iota', \text{“..”}, \iota)\end{aligned}$$

Note how we compose the operations creating the "." and ".." links in parallel to allow them to occur in any order, similarly to the parallel composition we used on path resolution in `link` and `rename`. The difference here is that they cannot be implemented truly in parallel, since `link_ins_dir`() is defined to be atomic.

Note that the existential quantification on the new inode ι' is before the whole sequence. This way the new inode is in scope for both the atomic statement that allocates it and the parallel `link_ins_dir`s that follow. This is in contrast to the atomic specification statement in `link_new_dir`, in which the new inode is existentially quantified in the postcondition. In `link_new_dir_nonatomic` the choice of the new inode ι' is an example of *early choice*: we chose the inode before the allocation. In `link_new_dir` the choice of the new inode ι' is an example of *late choice*: we chose the inode at the point of allocation. In our refinement calculus late and early choice are equivalent, as expressed with

the EAAtom refinement law. A simplified version of this law is as follows:

$$\frac{y \notin \texttt{free}(P)}{\exists y.\, \forall x \in X.\, \langle P(x),\, Q(x)\rangle \equiv \forall x \in X.\, \langle P(x),\, \exists y.\, Q(x)\rangle}$$

Effectively, this refinement law states that in contextual refinement there is no context that observes a difference between early and late choice. Intuitively, we can non-deterministically choose a value for y. The choice is correct if the postcondition of the atomic specification statement is satisfied for the chosen value; otherwise, the choice is wrong. All correct early choices for y are also correct when they are made late and vice versa. All wrong early choices for y are also wrong when they are made late and vice versa. The validity of both early and late choices is observable at the point the atomic update takes effect. For all choices, no matter when they are made, the same outcomes are observed.

With the AMumble and EAAtom refinement laws, and with SeqPar, from section 6.1.1, we can prove the following refinement between the two variants of directory creation:

$$\texttt{link\_new\_dir}(\iota, a) \sqsubseteq \texttt{link\_new\_dir\_nonatomic}(\iota, a)$$

Even though the atomic specification is arguably simpler, and more desirable to reason about modern implementations, the non-atomic specification is more inclusive and more aligned with the POSIX standard text. Portable clients applications, should be aware of the possibility that while creating a directory, they can observe intermediate states. Therefore, the client facing specification of `mkdir` should use `link_new_dir_nonatomic` instead of the simpler `link_new_dir`.

The same reasoning applies to all operations manipulating directories. Specifically, in the previous section we gave an atomic specification to the abstract operations moving directory links, such as `link_move_dir_target_not_exists`, as part of `rename`. This was a simplification. By AMumble and SeqPar we can derive more appropriate specifications for these operations.

Another consequence of mumbling is that we can coalesce all the steps taken by a file-system operation such as `mkdir` into just one atomic step, thus refining it to a coarse grained specification. Even though such atomic specifications can be used to reason about a subset of implementation behaviours, as for example in SibylFS [83], we have demonstrated in chapter 2.3, section 2.3.2, that they are completely unsuitable for client applications.

6.1.3. I/O Operations on Regular Files

POSIX defines `read` and `write` as the primitive operations for reading and writing data to regular files. The `read` operation reads a sequence of bytes from a regular file to the heap, whereas `write` writes a sequence of bytes stored in the heap to a regular file. These operations do not identify the file they act on with a path, but with a *file descriptor*. A file descriptor acts as a reference to a file and is associated with additional information controlling the behaviour of I/O operations acting on it.

In order to obtain a file descriptor to a file, a client must first open it for I/O with `open`. Incidentally, `open` is also used to create new regular files. The operation, `open`(*path*, *flags*), accepts two arguments: the *path* to the being opened, and *flags* controlling the behaviour of `open` and subsequent I/O operations such as `read` and `write`. POSIX defines a wide selection of flags. For presentation simplicity, we only consider the flags in table 6.1 in this section.

O_CREAT	Create the file if it does not exist.
O_EXCL	If used in conjunction with O_CREAT and the file already exists, return the EEXIST error. The test for the existence of the file and its creation if it does not already exist occur as a single atomic step.
O_RDONLY	Open the file for reading only.
O_WRONLY	Open the file for writing only.
O_RDWR	Open the file for both reading and writing.

Table 6.1.: A selection of flags controlling I/O behaviour.

We treat the *flags* argument as a subset of the flags controlling the operation. Mimicking the convention in C, we use its bitwise OR syntax to compose multiple flags. For example, we write O_CREAT|O_EXCL to mean {O_CREAT} ∪ {O_EXCL}.

Depending on the combination of the O_CREAT and O_EXCL flags, open behaves differently. We give the following refinement specification to open:

```
open(path, flags)
  ⊑ let p = dirname(path);
    let a = basename(path);
    let r = resolve(p, ι0);
    if ¬iserr(r) then
      if O_CREAT ∈ flags ∧ O_EXCL ∈ flags then
        return link_new_file(r, a, flags)
               ⊓ eexist(r, a)
      else if O_CREAT ∈ flags ∧ O_EXCL ∉ flags then
        return link_new_file(r, a, flags)
               ⊓ open_file(r, a, flags)
      else
        return open_file(r, a, flags)
               ⊓ enoent(r, a)
      fi
      ⊓ return enotdir(r)
    else return r fi
```

The path resolution is a usual for a single path. If the path prefix is resolved successfully, the specification branches depending on *flags*.

Consider the first branch, where both O_CREAT and O_EXCL exist in *flags*. According to table 6.1, this combination will cause the file to be created atomically if it does not already exist, and otherwise, the EEXIST error is returned. In the specification, this behaviour is expressed as the demonic choice between link_new_file, defined in figure 6.7, and enoent, defined previously in figure 6.3. Consider the definition of link_new_file. If a link named a does not already exist in the directory identified by the ι argument, link_new_file atomically creates a new empty regular file, adds a link named a to the file into the ι directory, and allocates and returns a new file descriptor for the file. In the postcondition, fd(ret, ι', 0, fdflags(*flags*)) is an abstract predicate stating that the value of newly allocated file descriptor is bound to ret, and that the file descriptor is associated with: the inode ι'

of the newly created file, the initial current file offset 0, and the flags given by $\mathsf{fdflags}(\mathit{flags})$. The latter is an expression that filters out any flags that do not affect subsequent I/O operations, such as `O_CREAT` and `O_EXCL`. We explain the use of the current file offset shortly.

$$\textbf{let } \texttt{link\_new\_file}(\iota, a, \mathit{flags}) \triangleq$$
$$\mathbb{V}FS. \left\langle \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota)),\ a \notin FS(\iota) \Rightarrow \begin{array}{c} \exists \iota'.\ \mathsf{fs}(FS[\iota \mapsto FS(\iota)[a \mapsto \iota']] \uplus \iota' \mapsto \epsilon) \\ * \mathsf{fd}(\texttt{ret}, \iota', 0, \mathsf{fdflags}(\mathit{flags})) \end{array} \right\rangle$$

$$\textbf{let } \texttt{open\_file}(\iota, a, \mathit{flags}) \triangleq$$
$$\mathbb{V}FS. \langle \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota)),\ \mathsf{isfile}(FS(FS(\iota)(a))) \Rightarrow \mathsf{fs}(FS) * \mathsf{fd}(\texttt{ret}, FS(\iota)(a), 0, \mathsf{fdflags}(\mathit{flags})) \rangle$$

Figure 6.7.: Specification of atomic operations that create and open regular files.

Consider the second branch, where `O_CREAT` is included in *flags*, but `O_EXCL` is not. In this case, if the file does not already exist, it is created with `link_new_file` as before. However, instead of returning an error if it already exists, it is directly opened with `open_file`, as defined in figure 6.7.

Finally, the third branch applies when `O_CREAT` is not included in *flags*. In this case, the file is opened according to `open_file` if it exists, or the `ENOENT` error is returned according to `enoent`. Note that the entire **if-then-else** is composed demonically with `enotdir`, since the `ENOTDIR` error is triggered in every branch, when the file resolved by the path prefix is not a directory.

Now that we have specified the operation with which we obtain a file descriptor to a file, consider the `write` operation with which we can write data to the file associated with an open file descriptor. We give the following refinement specification to `write`:

$$\texttt{write}(\mathit{fd}, \mathit{ptr}, \mathit{sz}) \sqsubseteq \texttt{return write\_off}(\mathit{fd}, \mathit{ptr}, \mathit{sz}) \sqcap \texttt{write\_badf}(\mathit{fd})$$

We specify the operation as the demonic choice between the `write_off` and `write_badf` abstract operations, defined in figure 6.8.

$$\textbf{let } \texttt{write\_off}(\mathit{fd}, \mathit{ptr}, \mathit{sz}) \triangleq$$
$$\mathbb{V}FS, o \in \mathbb{N}. \left\langle \begin{array}{l} \mathsf{fs}(FS) \wedge \mathsf{isfile}(FS(\iota)) * \mathsf{fd}(\mathit{fd}, \iota, o, \mathit{fl}) \wedge \mathsf{iswrfd}(\mathit{fl}) * \mathsf{buf}\big(\mathit{ptr}, \overline{b}\big) \wedge \mathsf{len}\big(\overline{b}\big) = \mathit{sz}, \\ \mathsf{fs}\big(FS[\iota \mapsto FS(\iota)[o \leftarrow \overline{b}]]\big) * \mathsf{fd}(\mathit{fd}, \iota, o + \mathit{sz}, \mathit{fl}) * \mathsf{buf}\big(\mathit{ptr}, \overline{b}\big) * \texttt{ret} = \mathit{sz} \end{array} \right\rangle$$

$$\textbf{let } \texttt{write\_badf}(\mathit{fd}) \triangleq \mathbb{V}o \in \mathbb{N}. \langle \mathsf{fd}(\mathit{fd}, \iota, o, \mathit{fl}) \wedge \texttt{O\_RDONLY} \in \mathit{fl},\ \mathsf{fd}(\mathit{fd}, \iota, o, \mathit{fl}) * \texttt{ret} = \texttt{EBADF} \rangle$$

Figure 6.8.: Specification of atomic write actions.

Consider the atomic specification statement of `write_off`. The precondition requires *fd* to be a file descriptor for the file with inode ι, with current file offset o and flags *fl*. Note that the current file offset is bound by the pseudo-universal quantifier, meaning that until `write_off` takes effect, the environment can concurrently modify it, with the proviso it remains a valid offset (a natural number).

The predicate $\mathsf{iswrfd}(fl) \triangleq \texttt{O\_WRONLY} \in fl \ \vee \texttt{O\_RDWR} \in fl$, states that file descriptor must have been opened for writing. Furthermore, the predicate $\mathsf{buf}(prt, \overline{b})$ states that ptr points to a heap based buffer storing the byte sequence $\overline{b}$. The postcondition states that the byte sequence $\overline{b}$ stored in the ptr buffer, is written to the file, offset from the start of the file (offset 0) by o. Any existing bytes from offset o onward, up to the length of $\overline{b}$, are overwritten. The current file offset associated with the file descriptor is incremented by the number of bytes written, which the operation also returns.

The `write_badf` abstract operation returns the `EBADF` error, if the file descriptor has not been opened for writing, and does not modify the file.

Recall the discussion in chapter 2.3 on the atomicity of I/O operations such as `write`. Partly due to the standard's ambiguity on the subject, and partly due to some, otherwise conforming, implementations not guaranteeing atomicity, whether client applications should rely on these operations being atomic is a matter of debate between application developers. It then stands to question, if atomicity is doubt, what is the specification that clients should rely on?

The obvious answer is to specify the operation as non-atomic. In our specification language, apart from atomic specification statements, we also use *Hoare specification statements* of the form:

$$\{P,\, Q\}$$

A Hoare specification statement specifies an update from a state satisfying the precondition P to a state satisfying the postcondition Q, without enforcing any atomicity guarantees. Intuitively, it specifies the behaviour of any program that satisfies the Hoare triple $\{P\} - \{Q\}$.

With Hoare specification statements, we can give a non-atomic specification to `write`, with the same preconditions and postconditions as in `write_off` and `write_badf`. For example, we can define the following non-atomic variant of `write_off`:

$$\textbf{let}\ \texttt{write\_off}(fd, ptr, sz) \triangleq \left\{\begin{array}{l} \mathsf{fs}(FS) \wedge \mathsf{isfile}(FS(\iota)) * \mathsf{fd}(fd, \iota, o, fl) \wedge \mathsf{iswrfd}(fl) * \mathsf{buf}\big(ptr, \overline{b}\big) \wedge \mathsf{len}\big(\overline{b}\big) = sz, \\ \mathsf{fs}\big(FS[\iota \mapsto FS(\iota)[o \leftarrow \overline{b}]]\big) * \mathsf{fd}(fd, \iota, o + sz, fl) * \mathsf{buf}\big(ptr, \overline{b}\big) * \texttt{ret} = sz \end{array}\right\}$$

In fact, by the same intuitive argument as in the AMumble refinement law, every atomic specification statement is a refinement of a (non-atomic) Hoare specification statement, as expressed by the following, slightly simplified, AWeaken2 refinement law:

$$\forall x \in X.\, \langle P(x),\, Q(x)\rangle \sqsubseteq \{P(x),\, Q(x)\}.$$

Using this law, we can trivially derive non-atomic specifications for I/O operations.

On the other hand, a non-atomic specification of I/O operations such as `write` may be too weak. Even if we do not rely on the reads or writes of bytes to files as being atomic, we could still rely on the update occurring to the current file offset associated with the file descriptor as atomic. This requires a combined specification, where some part of the state is updated atomically, while the other is updated non-atomically.

Following the general form of atomic Hoare triples in TaDA [30], our atomic specification statements

take the following general form[2]:

$$\mathbb{V}x \in X.\, \langle P_p \mid P(x),\ Q_p(x) \mid Q(x) \rangle$$

Here, the precondition and postcondition are split into two parts. The *private part*, on the left of the vertical separator, is updated non-atomically. The *public part*, on the right of the vertical separation, is updated atomically as in the simpler form atomic specification statements we have used so far. Note that since the private part has no atomicity guarantees, the private precondition P_p is not bound by the pseudo universal quantifier. In the private postcondition $Q_p(x)$, the variable x is bound to the value it has at the moment the atomic update on the public part takes effect.

Using the general form of atomic specification statements, we can rewrite the atomic `write_off` to the following partially atomic specification:

$$\textbf{let } \texttt{write\_off\_part\_atomic}(fd, ptr, sz) \triangleq$$
$$\mathbb{V}o \in \mathbb{N}. \left\langle \begin{array}{l|l} \exists FS, \overline{y}.\, \mathsf{fs}(FS[\iota \mapsto \overline{y}]) & \mathsf{fd}(fd, \iota, o, \mathit{fl}) \wedge \mathsf{iswrfd}(\mathit{fl}) * \mathsf{buf}\big(ptr, \overline{b}\big) \wedge \mathsf{len}\big(\overline{b}\big) = sz \\ \exists FS, \overline{y}.\, \mathsf{fs}\big(FS[\iota \mapsto \overline{y}[o \leftarrow \overline{b}]]\big) & \mathsf{fd}(fd, \iota, o + sz, \mathit{fl}) * \mathsf{buf}\big(ptr, \overline{b}\big) * \texttt{ret} = sz \end{array} \right\rangle$$

The current file offset in the file descriptor is update atomically, whereas the file referred to by the file descriptor is not. The existential quantification over the file-system graph FS in the private part, allows the environment to concurrently update the file-system graph during the non-atomic update of the file with inode ι. Note however, that the environment is not allowed to modify the byte sequence $\overline{y}$ stored in the file.

A client of this specification may rely on the update of the current file offset being atomic. However, in order to use the specification and derive strong properties about the contents of the file, the client is forced to employ concurrency control, so that other threads are prevented from interfering with the write. Otherwise, to use this specification a client would have to weaken the entire file-system graph, thus losing information about the information stored in the file.

Once more, this specification is related to the atomic specification `write_off`. An atomic update is a refinement of a partially atomic update, as expressed by the AWeaken1 refinement law, a slightly simplified version of which is given below:

$$\mathbb{V}x \in X.\, \langle P_p * P(x),\ Q_p(x) * Q(x) \rangle \sqsubseteq \mathbb{V}x \in X.\, \langle P_p \mid P(x),\ Q_p(x) \mid Q(x) \rangle$$

Thus with our refinement calculus it is easy to prove the following refinement between the specification variants:

$$\texttt{write\_off}(fd, ptr, sz) \sqsubseteq \texttt{write\_off\_part\_atomic}(fd, ptr, sz) \sqsubseteq \texttt{write\_off\_non\_atomic}(fd, ptr, sz)$$

Note that I/O operations involve copying byte sequences between regular files and buffers in the heap. Therefore, we need to combine reasoning about the file system with reasoning about the heap. In figure 6.9 we specify atomic operations on the heap. The `malloc`(*size*) operation atomically allocates a heap buffer of *size* bytes with arbitrary values and returns the address of the allocated heap buffer.

[2]For presentation simplicity at this point, this is still not the most general form of atomic specification statements. Atomic specification statements in their most general form are defined in chapter 7.

$$\textbf{let } \texttt{malloc}(\mathit{size}) \triangleq \langle \mathsf{true},\ \exists \overline{y}.\, \mathsf{buf}(\texttt{ret}, \overline{y}) \wedge \mathsf{len}(\overline{y}) = \mathit{size} \rangle$$

$$\textbf{let } \texttt{memset}(\mathit{ptr}, \mathit{byte}) \triangleq \langle \mathit{ptr} \mapsto -,\ \mathit{ptr} \mapsto \mathit{byte} \rangle$$

$$\textbf{let } \texttt{memget}(\mathit{ptr}) \triangleq \langle \mathit{ptr} \mapsto y,\ \mathit{ptr} \mapsto y * \texttt{ret} = y \rangle$$

$$\textbf{let } \texttt{memwrite}(\mathit{ptr}, \mathit{bytes}) \triangleq \langle \exists \overline{y}.\, \mathsf{buf}(\mathit{ptr}, \overline{y}) \wedge \mathsf{len}(\overline{y}) = \mathsf{len}(\mathit{bytes}),\ \mathsf{buf}(\mathit{ptr}, \mathit{bytes}) \rangle$$

$$\begin{aligned}&\textbf{let } \texttt{memread}(\mathit{ptr}, \mathit{type}) \triangleq \\ &\quad \langle \mathsf{buf}(\mathit{ptr}, \overline{y}) \wedge \mathsf{len}(\overline{y}) = \texttt{sizeof}(\texttt{int}) \wedge \mathit{type} = \texttt{int},\ \mathsf{buf}(\mathit{ptr}, \overline{y}) * \texttt{ret} = \mathsf{b2i}(\overline{y}) \rangle \\ &\sqcap\ \langle \mathsf{buf}(\mathit{ptr}, \overline{y}) \wedge \mathsf{len}(\overline{y}) = \mathit{sz} \wedge \mathit{type} = \texttt{STR}(\mathit{sz}),\ \mathsf{buf}(\mathit{ptr}, \overline{y}) * \texttt{ret} = \overline{y} \rangle\end{aligned}$$

$$\begin{aligned}&\textbf{let } \texttt{memcpy}(\mathit{ptrt}, \mathit{ptrs}, \mathit{size}) \triangleq \\ &\quad \langle \mathsf{buf}(\mathit{ptrt}, \overline{y}) \wedge \mathsf{len}(\overline{y}) = \mathit{size} * \mathsf{buf}(\mathit{ptrs}, \overline{y}') \wedge \mathsf{len}(\overline{y}') = \mathit{size},\ \mathsf{buf}(\mathit{ptrt}, \overline{y}') * \mathsf{buf}(\mathit{ptrs}, \overline{y}') \rangle\end{aligned}$$

where:

$$\mathsf{buf}(\mathit{ptr}, \overline{y}) \triangleq (\overline{y} = \epsilon) \vee (\exists y, \overline{y}'.\, \overline{y} = y : \overline{y}' \wedge \mathit{ptr} \mapsto y * \mathsf{buf}(\mathit{ptr} + 1, \overline{y}'))$$

Figure 6.9.: Specification of heap operations.

Note that a heap buffer is a consecutive sequence of heap cells storing bytes. $\texttt{memset}(\mathit{ptr}, \mathit{byte})$ sets the byte value of the heap cell with address ptr to byte, whereas $\texttt{memget}(\mathit{ptr})$ retrieves the byte value stored in the heap cell with address ptr. The operation $\texttt{memwrite}(\mathit{ptr}, \mathit{bytes})$ writes the sequence of bytes bytes to the heap buffer with address ptr. The $\texttt{memread}(\mathit{ptr}, \mathit{type})$ operation reads the sequence of bytes stored in the heap buffer at address ptr. The second parameter type determines how the byte sequence being read is interpreted. If $\mathit{type} = \texttt{int}$, then it is converted to a bounded integer. If $\mathit{type} = \texttt{STR}(\mathit{sz})$, it is interpreted as sequence of bytes of size sz. Finally, $\texttt{memcpy}(\mathit{ptrt}, \mathit{ptrs}, \mathit{size})$ copies the contents of the heap buffer with address ptrs of size size into the heap buffer with address ptrt.

6.2. Client Reasoning

So far we have developed examples of specifications for POSIX file-system operations, and introduced the key features of our specification language and of refinement between specification programs. The notion of refinement we employ is that of contextual refinement. If ϕ and ψ are specification programs and $\phi \sqsubseteq \psi$ then, for all contexts C, we have $C[\phi] \sqsubseteq C[\psi]$. Client applications are contexts of the POSIX operations we specified in the previous section. We now use our specifications *in context*, and demonstrate how our refinement calculus is used to prove strong functional properties for clients.

We introduce our development through the example of *lock files*. Lock files are a popular pattern for implementing mutual exclusion between different processes accessing the file system, with several library implementations in different programming languages.

The lock-file concept is simple. A lock file is a regular file, under a fixed path, representing the status of a lock. If the lock file exists, the lock it represents is locked. Otherwise, the lock is unlocked. For example, `/tmp/.X0-lock` is a typical lock file in contemporary Linux systems, and in figure 6.1, the lock it represents is locked.

Consider an implementation of a lock-file module with two operations: $\texttt{lock}(lf)$ and $\texttt{unlock}(lf)$, where lf is the path identifying the lock file. We implement the operations as follows:

$$
\begin{array}{l}
\textbf{letrec}\ \texttt{lock}(lf) \triangleq \\
\quad \textbf{let}\ fd = \texttt{open}(lf, \texttt{O\_CREAT}|\texttt{O\_EXCL}); \\
\quad \textbf{if}\ \mathsf{iserr}(fd)\ \textbf{then}\ \texttt{return}\ \texttt{lock}(lf) \\
\quad \textbf{else}\ \texttt{close}(fd)\ \textbf{fi} \\
\\
\textbf{let}\ \texttt{unlock}(lf) \triangleq \texttt{unlink}(lf)
\end{array}
$$

The `lock` operation attempts to create the lock file at path lf by invoking `open`. Recall from section 6.1.3 that the flag combination `O_CREAT|O_EXCL` causes `open` to create a file at the given path if one does not already exist; otherwise, an error is returned. Thus, if `open` returns an error we try again, with a recursive call to `lock`. If it succeeds, we invoke `close` to close the file descriptor returned by `open`. We give the following specification to `close`:

$$\texttt{close}(fd) \sqsubseteq \langle \mathsf{fd}(fd, \iota, -)\,,\ \mathsf{true} \rangle$$

By contextual refinement, we can replace the `open` and `unlink` with their specifications and thus derive a specification for `lock` and `unlock` respectively. However, this would not be useful for reasoning about lock files since it fails to capture their abstract behaviour as lock. Instead, we establish the following specification:

$$
\begin{array}{l}
\mathsf{LFCtx}(lf) \vdash \texttt{lock}(lf) \sqsubseteq \mathbb{V} v \in \{0,1\}\,.\,\langle \mathsf{Lock}(s, lf, v)\,,\ \mathsf{Lock}(s, lf, 1) * v = 0 \rangle \\
\mathsf{LFCtx}(lf) \vdash \texttt{unlock}(lf) \sqsubseteq \langle \mathsf{Lock}(s, lf, 1)\,,\ \mathsf{Lock}(s, lf, 0) \rangle
\end{array}
$$

The abstract predicate $\mathsf{Lock}(s, lf, v)$ states the existence of a lock represented by a lock file at path lf, with state v, the value of which is either 0, if the lock is unlocked, or 1 if the lock is locked. The first parameter, $s \in \mathbb{T}_1$, is a variable ranging over an abstract type. It serves to capture invariant information, specific to the implementation of the Lock predicate and is opaque to the client. The specification states that we can abstract each lock-file operation to a single atomic step that updates the state of the lock. In particular, the `lock` specification states that the environment can arbitrarily lock and unlock the lock, but the lock is atomically locked only when it is previously unlocked; the operation blocks while the lock is locked. The `unlock` specification states that the lock can only be atomically unlocked when the lock is locked. The specifications for `lock` and `unlock` hold under the invariant $\mathsf{LFCtx}(lf)$, the purpose of which we explain shortly.

In order to justify the module's specification, we must show that `lock` and `unlock` update the state of the lock, according to a protocol that determines how its state can change. We follow the approach of TaDA [30], using *shared regions* to encapsulate state that is shared by multiple threads, with the proviso that it can only be accessed atomically. We use $\mathbf{t}_\alpha(x)$, to denote a shared region α of type $\mathbf{t}$ and abstract state x. The abstract state x abstracts the shared state encapsulated by the region. The region is associated with an *interpretation function*, a *labelled transition system* and *guards*. The interpretation function maps abstract states to the encapsulated shared resources. The

labelled transition system defines the atomic updates that are allowed on the abstract region state. Transitions are labelled by guards. In order for a thread to change the state of a region, it must own the guard resource associated with the transition that defines the atomic update. Guard resources can be taken from any user-defined separation algebra [25]. Therefore, guards serve as generalised capabilities.

For our current example, we introduce the region type **Lock**. Regions of this type are parameterised by the lock-file path. The abstract state of the region corresponds to the state of the lock. We define a single guard G for the region. The labelled transition system associated with the region is as follows:

$$\mathsf{G} : 0 \rightsquigarrow 1 \qquad\qquad \mathsf{G} : 1 \rightsquigarrow 0$$

Ownership of the guard resource G provides the capability to lock the lock, by a transition from abstract state 0 to 1, and the capability to unlock the lock, by a transition from 1 to 0.

We now need to provide a concrete interpretation for each abstract state of the region, in terms of the file-system representation of the lock as a lock file. However, at this point, we first need to explain a fundamental difference between reasoning about typical heap-based modules and file-system based modules.

In a heap-based implementation of a lock, the interpretation of the region would be defined as the heap resource representing the lock in memory; for instance, a single heap cell with value either 0 or 1. Any update to the state of the lock must be allowed by the guards and transition system associated with the region. In this case, the region serves to restrict access to the heap resource implementing the lock only to the module operations. However, as we discussed in chapter 2.3, the file system is a public namespace: it is always shared as a whole between all possible threads and processes. Anyone at any time can access any path of their choosing. We cannot restrict access to the entire file system only to the operations of a module, since then the context would not be able to access the file system at all. Yet we do need some restrictions on the context. In the case of a lock-file module, we require the context not to change the path to the lock-file directory, and that the only way to create or remove the lock file is via the module operations. The lock-file module cannot enforce such restrictions on its own. Instead, these restrictions form a proof obligation for the context. We express such proof obligations with *context invariants*. For our lock-file module, LFCtx denotes the context invariant under which its specification holds.

In order to define LFCtx, we first encapsulate the file system within the global file-system shared region of type **GFS**. There is only a single instance of this region, with a known identifier which we keep implicit. All clients accessing the file system do so via this region. The region's state is a file-system graph, $FS \in \mathcal{FS}$, with the following straightforward interpretation:

$$I_r(\mathbf{GFS}(FS)) \triangleq \mathsf{fs}(FS)$$

We keep the guards and labelled transition system of this region open ended. However, we define LFCtx to place restrictions on the guards and transition system.

To aid our definition of LFCtx, we introduce some additional notation and predicates. $\mathcal{G}_{\mathbf{t}}$ denotes the set of guards associated with the region type $\mathbf{t}$; $\mathsf{G} \bullet \mathsf{G}'$ denotes the partial, associative and commutative composition of guards; $\mathsf{G}\#\mathsf{G}'$ states that the composition of guards G and G' is defined; and $\mathcal{T}_{\mathbf{t}}(\mathsf{G})^*$

denotes the transitions for guard G of the region type $\mathbf{t}$, where the superscript * denotes the reflexive-transitive closure. We also define the following auxiliary predicates:

$$!\mathsf{G} \in \mathcal{G}_{\mathbf{t}} \triangleq \mathsf{G} \in \mathcal{G}_{\mathbf{t}} \wedge \mathsf{G} \bullet \mathsf{G}\ \textbf{undefined}$$
$$(x, y)\ \dagger_{\mathbf{t}}\ \mathsf{G} \triangleq (x, y) \in \mathcal{T}_{\mathbf{t}}(\mathsf{G})^* \wedge \forall \mathsf{G}' \in \mathcal{G}_{\mathbf{t}}.\ \mathsf{G}' \# \mathsf{G} \Rightarrow (x, y) \notin \mathcal{T}_{\mathbf{t}}(\mathsf{G}')^*$$

The predicate $!\mathsf{G} \in \mathcal{G}_{\mathbf{t}}$ states that there is only one instance of the guard G for the region type $\mathbf{t}$, and $(x, y)\ \dagger_{\mathbf{t}}\ \mathsf{G}$ states that in regions of type $\mathbf{t}$, the transition from state x to y is defined only for G.

Let p/a be the path to a lock file. We define the predicate $p \overset{FS}{\longmapsto} \iota$ to assert that the path p resolves to the file with inode ι in the file-system graph FS as follows:

$$p \overset{FS}{\longmapsto} \iota \triangleq (p, \iota_0) \overset{FS}{\longmapsto} \iota$$
$$(a, \iota) \overset{FS}{\longmapsto} \iota' \triangleq FS(\iota)(a) = \iota'$$
$$(a/p, \iota) \overset{FS}{\longmapsto} \iota' \triangleq \exists \iota''.\ FS(\iota)(a) = \iota'' \wedge (p, \iota'') \overset{FS}{\longmapsto} \iota'$$

Additionally, we define the expression $FS\restriction_p$ to identify the sub-graph of FS that is formed by the path p as follows:

$$FS\restriction_p \triangleq FS\restriction_p^{\iota_0}$$
$$FS\restriction_a^{\iota} \triangleq \iota \mapsto (a \mapsto FS(\iota)(a))$$
$$FS\restriction_{a/p}^{\iota} \triangleq \iota \mapsto (a \mapsto FS(\iota)(a)) \uplus FS\restriction_p^{FS(\iota)(a)}$$

We define the file-system states that the lock-file module can be in as follows:

$$\mathcal{ULK}(p/a) \triangleq \left\{ FS \mid \exists \iota.\ p \overset{FS}{\longmapsto} \iota \wedge \mathsf{isdir}(FS(\iota)) \wedge a \notin FS(\iota) \right\}$$
$$\mathcal{LK}(p/a) \triangleq \left\{ FS \mid \exists \iota.\ p \overset{FS}{\longmapsto} \iota \wedge \mathsf{isdir}(FS(\iota)) \wedge a \in FS(\iota) \right\}$$
$$\mathcal{LF}(p/a) \triangleq \mathcal{ULK}(p/a) \cup \mathcal{LK}(p/a)$$

$\mathcal{ULK}$ denotes the unlocked states, where the lock file does not exist, whereas $\mathcal{LK}$ denotes the locked states, where the lock file exists. $\mathcal{LF}$ denotes both possibilities. Note that the directory containing the lock file always exists. The following predicates describe the updates that create and remove the lock file in its directory:

$$\mathsf{lk}(FS, FS', p/a) \triangleq \exists \iota, \iota'.\ p \overset{FS}{\longmapsto} \iota \wedge FS' = FS[\iota \mapsto FS(\iota)[a \mapsto \iota']][\iota' \mapsto \epsilon]$$
$$\mathsf{ulk}(FS, FS', p/a) \triangleq \exists \iota.\ p \overset{FS}{\longmapsto} \iota \wedge FS' = FS[\iota \mapsto FS(\iota) \setminus \{a\}]$$

We can now define the context invariant as follows:

$$\mathsf{LFCtx}(p/a) \triangleq$$
$$\exists FS \in \mathcal{LF}(p/a).\ \mathbf{GFS}(FS) \wedge !\,[\mathsf{LF}(p/a)] \in \mathcal{G}_{\mathbf{GFS}}$$
$$\wedge \forall FS \in \mathcal{ULK}(p/a).\ \exists FS'.\ \mathsf{lk}(FS, FS', p/a) \wedge (FS, FS')\ \dagger_{\mathbf{GFS}}\ \mathsf{LF}\,(p/a)$$
$$\wedge \forall FS \in \mathcal{LK}(p/a).\ \exists FS'.\ \mathsf{ulk}(FS, FS', p/a) \wedge (FS, FS')\ \dagger_{\mathbf{GFS}}\ \mathsf{LF}\,(p/a)$$
$$\wedge \forall \mathsf{G} \in \mathcal{G}_{\mathbf{GFS}}.\ \forall FS, FS' \in \mathcal{LK}(lf).\ (FS, FS') \in \mathcal{T}_{\mathbf{GFS}}(\mathsf{G})^* \Rightarrow FS\restriction_p = FS'\restriction_p$$

The first line of the definition restricts the states of the global file-system region to those in which the path to the lock-file directory exists and the lock file itself may exist or not. Additionally, it requires the indivisible guard $\mathsf{LF}\,(p/a)$ to be defined for the global file-system region. The second and third lines of the definition state that transitions creating or removing the lock file in its directory are only defined for the guard $\mathsf{LF}\,(p/a)$. Therefore, only ownership of this guard grants a thread the capability to transition between locked and unlocked states. Finally, the last line of the definition requires all transitions between lock-file states to maintain the same file-system sub-graph for the path p. This guarantees that the context does not modify the sub-graph such that the path is diverted to a different location.

Assuming the context satisfies $\mathsf{LFCtx}(lf)$, we can now define the interpretation of the **Lock** region as:

$$I_r(\mathbf{Lock}_\alpha(lf,0)) \triangleq \exists FS \in \mathcal{ULK}(lf).\,\mathbf{GFS}(FS) * [\mathsf{LF}(lf)]$$
$$I_r(\mathbf{Lock}_\alpha(lf,1)) \triangleq \exists FS \in \mathcal{LK}(lf).\,\mathbf{GFS}(FS) * [\mathsf{LF}(lf)]$$

Finally, we instantiate the Lock predicate and $\mathbb{T}_1$ as follows:

$$\mathbb{T}_1 \triangleq \textsc{RId}$$
$$\mathsf{Lock}(\alpha, lf, 0) \triangleq \mathbf{Lock}_\alpha(lf,0) * [\mathsf{G}]_\alpha$$
$$\mathsf{Lock}(\alpha, lf, 1) \triangleq \mathbf{Lock}_\alpha(lf,1) * [\mathsf{G}]_\alpha$$

where RId is the set of region identifiers.

It remains to prove the refinement between the implementation and our specification. First, to reduce the complexity of reasoning about `open`, we derive the following refinement specific to its use in `lock`(), where we focus only on the `O_CREAT|O_EXCL` behaviour.

$$\begin{aligned}
&\texttt{open}(path, \texttt{O_CREAT}|\texttt{O_EXCL})\\
&\quad\sqsubseteq \textbf{let } p = \texttt{dirname}(path);\\
&\qquad \textbf{let } a = \texttt{basename}(path);\\
&\qquad \textbf{let } r = \texttt{resolve}(p, \iota_0);\\
&\qquad \textbf{if } \neg\mathsf{iserr}(r) \textbf{ then}\\
&\qquad\quad \texttt{return link_new_file}(r, a)\\
&\qquad\qquad\qquad \sqcap\, \texttt{eexist}(r, a) \sqcap \texttt{enotdir}(r, a)\\
&\qquad \textbf{else return } r \textbf{ fi}
\end{aligned}$$

In figure 6.10 we give a sketch proof for the `lock` operation. Throughout the proof we assume that LFCtx holds. On the other hand, LFCtx is a proof obligation for the context.

We begin the refinement proof with the specification of `lock` at the bottom of figure 6.10. In the first refinement step we apply the MakeAtomic refinement law, discussed shortly, refining the atomic specification statement to a Hoare specification statement.

In section 6.1.3, we have introduced the AWeaken2 refinement law that allows a non-atomic Hoare specification statement to be refined by an atomic statement. The reverse, *atomicity abstraction*, only holds when we can prove that the state of a region is updated only once. This is captured by the

let $fd = \texttt{open}(lf, \texttt{O\_CREAT}|\texttt{O\_EXCL})$;

$\sqsubseteq$ by figure 6.11

$\forall FS \in \mathcal{LF}(lf).\left\langle \mathsf{fs}(FS) \wedge p \overset{FS}{\longmapsto} r,\ \begin{array}{l}(\mathsf{fs}(FS) * fd = \texttt{EEXIST}) \\ \vee\ (\exists FS'.\, \mathsf{lk}(FS, FS', lf) * \mathsf{fs}(FS') * \mathsf{fd}(fd, -, 0))\end{array}\right\rangle$

$\sqsubseteq$ by UseAtomic and ACons

$\forall FS \in \mathcal{LF}(lf).\left\langle \mathbf{GFS}(FS) * [\mathsf{LF}(lf)],\ \begin{array}{l}(\mathbf{GFS}(FS) * fd = \texttt{EEXIST}) \\ \vee\ (\exists FS' \in \mathcal{LK}(lf).\, \mathbf{GFS}(FS') * \mathsf{fd}(fd, -, -)) * [\mathsf{LF}(lf)]\end{array}\right\rangle$

$\sqsubseteq$ by AEElim

$\left\langle \exists FS \in \mathcal{LF}(lf).\, \mathbf{GFS}(FS) * [\mathsf{LF}(lf)],\ \begin{array}{l}(\exists FS \in \mathcal{LF}(lf).\, \mathbf{GFS}(FS) * fd = \texttt{EEXIST}) \\ \vee\ (\exists FS' \in \mathcal{LK}(lf).\, \mathbf{GFS}(FS') * \mathsf{fd}(fd, -, -)) * [\mathsf{LF}(lf)]\end{array}\right\rangle$

$\sqsubseteq$ by UpdateRegion

$\left\langle \exists v \in \{0,1\}.\, \mathbf{Lock}_\alpha(lf, v) * \alpha \Rrightarrow \blacklozenge,\ \begin{array}{l}\exists v \in \{0,1\}.\, \mathbf{Lock}_\alpha(lf, v) * (\alpha \Rrightarrow \blacklozenge * fd = \texttt{EEXIST}) \\ \vee\ (\alpha \Rrightarrow (0,1) * \mathsf{fd}(fd, -, -))\end{array}\right\rangle^{\mathcal{A}}$

$\sqsubseteq$ by AWeaken2

$\left\{ \exists v \in \{0,1\}.\, \mathbf{Lock}_\alpha(v) * \alpha \Rrightarrow \blacklozenge,\ \begin{array}{l}\exists v \in \{0,1\}.\, \mathbf{Lock}_\alpha(lf, v) * (\alpha \Rrightarrow \blacklozenge * fd = \texttt{EEXIST}) \\ \vee\ (\alpha \Rrightarrow (0,1) * \mathsf{fd}(fd, -, -))\end{array}\right\}^{\mathcal{A}}$

Seq

IfThenElse

if $\mathsf{iserr}(fd)$ **then**

$\texttt{return lock}(lf)$

$\sqsubseteq$ by assumption

$\{\exists v \in \{0,1\}.\, \mathbf{Lock}_\alpha(lf, v) * \alpha \Rrightarrow \blacklozenge,\ \alpha \Rrightarrow (0,1)\}^{\mathcal{A}}$

$\sqsubseteq$ by HCons (precondition strengthening)

$\{\exists v \in \{0,1\}.\, \mathbf{Lock}_\alpha(lf, v) * \alpha \Rrightarrow \blacklozenge * fd = \texttt{EEXIST},\ \alpha \Rrightarrow (0,1)\}^{\mathcal{A}}$

else

$\texttt{close}(fd)$

$\sqsubseteq$ by specification

$\langle \mathsf{fd}(fd, -, -)\,,\ \mathsf{true}\rangle$

$\sqsubseteq$ by AWeaken2

$\{\mathsf{fd}(fd, -, -)\,,\ \mathsf{true}\}$

$\sqsubseteq$ by HFrame and HCons (precondition strengthening)

$\{\exists v \in \{0,1\}.\, \mathbf{Lock}_\alpha(lf, v) * \alpha \Rrightarrow (0,1) * \mathsf{fd}(fd, -, -)\,,\ \alpha \Rrightarrow (0,1)\}^{\mathcal{A}}$

fi

$\sqsubseteq \left\{ \begin{array}{l}\exists v \in \{0,1\}.\, \mathbf{Lock}_\alpha(lf, v) * (\alpha \Rrightarrow \blacklozenge * fd = \texttt{EEXIST}) \\ \vee\ (\alpha \Rrightarrow (0,1) * \mathsf{fd}(fd, -, -))\end{array},\ \alpha \Rrightarrow (0,1)\right\}^{\mathcal{A}}$

$\sqsubseteq$ assuming $\texttt{lock}(lf)$ refines the same specification

$\{\exists v \in \{0,1\}.\, \mathbf{Lock}_\alpha(lf, v) * \alpha \Rrightarrow \blacklozenge,\ \alpha \Rrightarrow (0,1)\}^{\mathcal{A}}_k$

$\sqsubseteq$ by MakeAtomic

$\forall v \in \{0,1\}.\, \langle \mathbf{Lock}_\alpha(lf, v) * [\mathsf{G}]_\alpha\,,\ \mathbf{Lock}_\alpha(lf, 1) * [\mathsf{G}]_\alpha * v = 0\rangle$

Figure 6.10.: Proof sketch of $\texttt{lock}(lf)$ refining its specification, assuming $\mathsf{LFCtx}(lf)$ at every step.

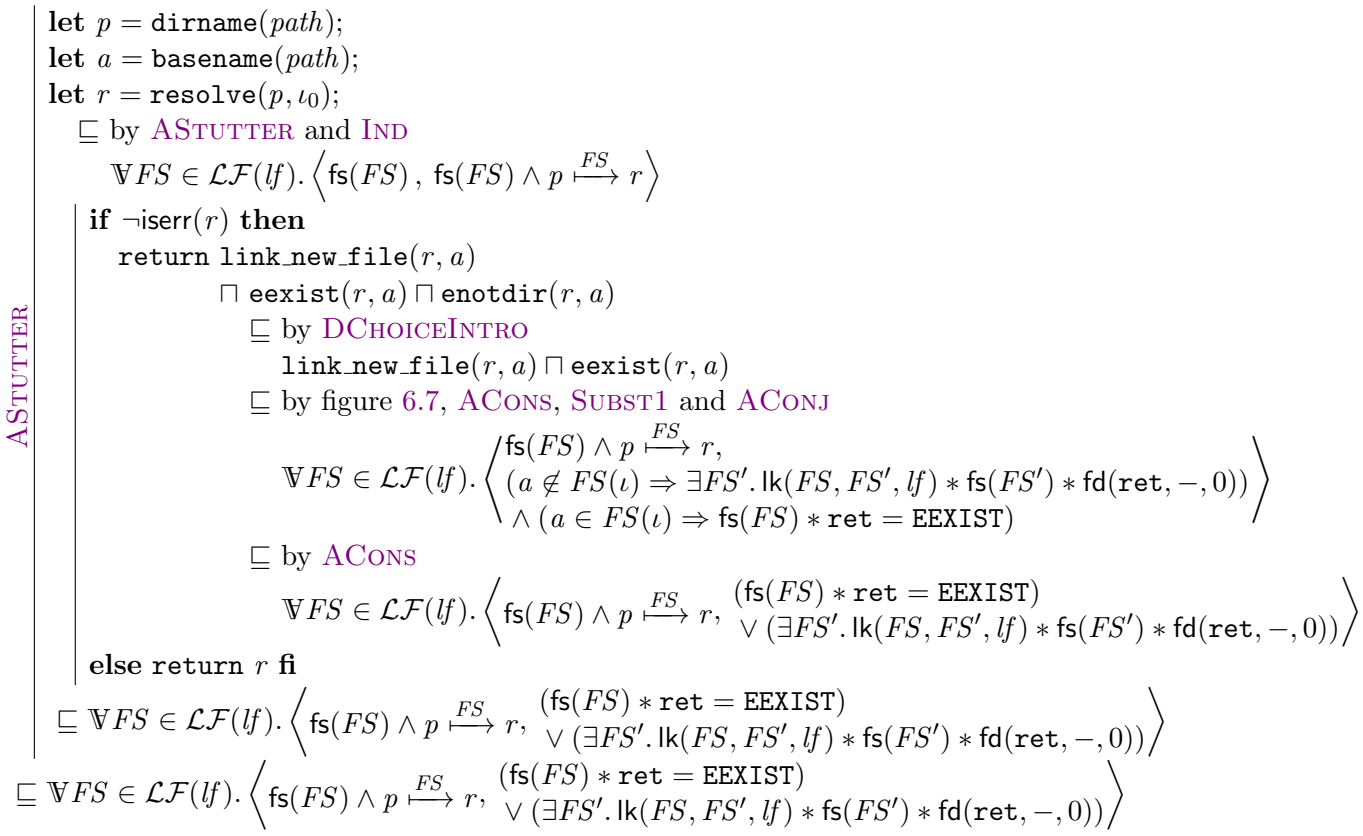

Figure 6.11.: Refinement of $\texttt{open}(lf, \texttt{O\_CREAT}|\texttt{O\_EXCL})$ used in $\texttt{lock}(lf)$.

MakeAtomic refinement law. A slightly simplified version of this law is as follows:

$$\frac{\{(x,y) \mid x \in X, y \in Y(x)\} \subseteq \mathcal{T}_{\mathbf{t}}(\mathsf{G})^*}{\begin{array}{c}\{\exists x \in X.\, \mathbf{t}_\alpha(x) * \alpha \Rightarrow \blacklozenge,\ \exists x \in X, y \in Y(x).\, \alpha \Rightarrow (x,y)\}^{\alpha : x \in X \rightsquigarrow Y(x)} \\ \sqsubseteq \forall x \in X.\, \langle \mathbf{t}_\alpha(x) * [\mathsf{G}]_\alpha\,,\ \mathbf{t}_\alpha(y) * [\mathsf{G}]_\alpha \rangle\end{array}}$$

The atomic specification statement of the conclusion specifies an atomic update on the state of the region α, of region type $\mathbf{t}$, from $x \in X$ to $y \in Y(x)$. The environment may change the state of the region before the atomic update takes effect, with the proviso that it remains in X. The update to the state of the region must be justified by the guard resource $[\mathsf{G}]_\alpha$. The premiss of the law requires that the update from x to y is allowed by the transition system for the guard G. The atomic specification statement is refined by a non-atomic Hoare specification that ensures the update happens atomically. The *atomicity context* in the superscript, $\alpha : x \in X \rightsquigarrow Y(x)$, records the update the specification statement must perform. The *atomic tracking resource*, $\alpha \Rightarrow \blacklozenge$, acts as a proxy to the guard. However, it permits only a single update to the region in accordance to the atomicity context. Until the update is performed, the region's state is guaranteed to be within the set X. When the update takes effect, the atomic tracking resource is simultaneously updated to $\alpha \Rightarrow (x,y)$, recording the actual update performed.

In the next step, we use the Seq refinement law, which allows us to refine a Hoare specification statement to a sequence of specification statements. A simplified version of this law is as follows:

$$\frac{\phi \sqsubseteq \{P,\ R\} \qquad \psi \sqsubseteq \{R,\ Q\}}{\phi ; \psi \sqsubseteq \{P(x),\ Q(x)\}}$$

With Seq, we split the Hoare specification that we obtained using MakeAtomic into a Hoare specification statement for `open` and a Hoare specification statement for the **if-then-else** statement.

Consider the refinement of the Hoare specification statement for `open`. In the first step, we apply AWeaken2 to refine the Hoare specification to an atomic specification statement. Then, we use the UpdateRegion law.

The UpdateRegion refinement law allows us to refine the atomic update required by an atomicity tracking resource for a region into an atomic update to the region's interpretation. A simplified version of the this refinement law is as follows:

$$\begin{array}{c}\forall x \in X.\ \left\langle I_r(\mathbf{t}_\alpha(x)) * P(x),\ \begin{array}{l}(\exists y \in Y(x).\, I_r(\mathbf{t}_\alpha(y)) * Q_1(x,y)) \\ \vee\, (I_r(\mathbf{t}_\alpha(x)) * Q_2(x))\end{array} \right\rangle \\ \sqsubseteq \\ \forall x \in X.\ \left\langle \mathbf{t}_\alpha(x) * P(x) * \alpha \Rightarrow \blacklozenge,\ \begin{array}{l}(\exists y \in Y(x).\, \mathbf{t}_\alpha(y) * Q_1(x,y) * \alpha \Rightarrow (x,y)) \\ \vee\, (\mathbf{t}_\alpha(x) * Q_2(x) * \alpha \Rightarrow \blacklozenge)\end{array} \right\rangle^{\alpha : x \in X \rightsquigarrow Y(x)}\end{array}$$

Note that the atomic specification statement we obtain by UpdateRegion in figure 6.10 uses existential quantification on the file-system graphs. Existential quantification in atomic specification statement can be refined to pseudo universal quantification according to the AEElim refinement law,

a simplified version of which is as follows:

$$\mathbb{\forall} x \in X.\, \langle P(x),\, Q(x)\rangle \sqsubseteq \langle \exists x \in X.\, P(x),\, \exists x \in X.\, Q(x)\rangle$$

In the final refinement step to `open` in figure 6.10, we justify the atomic update of `open` according to the corresponding transition on the global file-system region **GFS** with the UseAtomic refinement law. A simplified version is as follows:

$$\frac{\forall x \in X.\, (x, f(x)) \in \mathcal{T}_{\mathbf{t}}(\alpha)^*}{\begin{array}{c}\mathbb{\forall} x \in X.\, \langle I_r(\mathbf{t}_\alpha(x)) * P(x) * [\mathsf{G}]_\alpha\,,\; I_r(\mathbf{t}_\alpha(f(x))) * Q(x)\rangle \\ \equiv \mathbb{\forall} x \in X.\, \langle \mathbf{t}_\alpha(x) * P(x) * [\mathsf{G}]_\alpha\,,\; \mathbf{t}_\alpha(f(x)) * Q(x)\rangle\end{array}}$$

This allows us to refine an atomic update to the abstract state of a region into an atomic update to the region's interpretation, as long as we have the required guard resource for which the update is allowed by the region's transition system.

Consider the refinement to **if-then-else** statement that follows `open`. Here, we are using the IfThenElse law to refine the specification of the branch to the two branches. This refinement law is analogous to the rule of **if-then-else** statements in Hoare logic and is given in detail in chapter 7, section 7.7. Note that in the **if**-branch, we assume that `lock` already refines the specification that we are refining, as per the Ind refinement law discussed in chapter 7.

In figure 6.11 we proceed to refine the atomic specification for `open` that we use in figure 6.10 into its POSIX specification program. This refinement proof relies on using the AStutter refinement law, to coalesce the multiple atomic steps in the POSIX specification of `open` into a single atomic step. A simplified version of this law is as follows:

$$\mathbb{\forall} x \in X.\, \langle P(x),\, P(x)\rangle; \mathbb{\forall} x \in X.\, \langle P(x),\, Q(x)\rangle \sqsubseteq \mathbb{\forall} x \in X.\, \langle P(x),\, Q(x)\rangle$$

This law allows us to refine an atomic update by adding steps that do not affect the shared state.

Note that the $\mathcal{LF}(lf)$ from the specification statement that we are refining guarantees that `resolve` in figure 6.11 succeeds. Thus we are only concerned with refining the **if**-branch. We begin the refinement from the bottom of the **if**-branch by strengthening the postcondition into a conjunction of two implications, one for the case where the lock file does not exist and one for the case where it does. This is achieved by using the ACons refinement law, which is directly analogous to the consequence rule of Hoare logics. We present the ACons law in detail in chapter 7.

In the next refinement step we split the atomic specification statement into two cases, corresponding to the earlier implications, composed with demonic choice. This is analogous to using the conjunction rule of Hoare logics, which in our refinement calculus corresponds to the AConj refinement law. A slightly simplified version of this law is as follows:

$$\mathbb{\forall} x \in X.\, \langle P_1(x),\, Q_1(x)\rangle \sqcap \mathbb{\forall} x \in X.\, \langle P_1(x),\, Q_1(x)\rangle \sqsubseteq \mathbb{\forall} x \in X.\, \langle P_1(x) \wedge P_2(x),\, Q_1(x) \wedge Q_2(x)\rangle$$

By using AConj followed by another use of the ACons law we refine the atomic specification statement to the demonic composition of `link_new_file` and `eexist`. In order to reach the specification program of `open`(lf, `O_CREAT|O_EXCL`) in the last refinement we also add the `enotdir` demonic case.

Refinement always allows us to add more demonic cases to a specification program, as captured by the following DChoiceIntro refinement law:

$$\phi \sqcap \psi \sqsubseteq \phi$$

6.3. Extending Specifications

6.3.1. Symbolic Links and Relative Paths

In section 6.1 we have given simplified specifications of POSIX file-system operations, where we assumed that file-system graphs do not use symbolic links, paths are absolute and do not use "." and "..". We now demonstrate how our specifications can be extended to remove these restrictions, by revisiting the specification of `unlink`.

Symbolic links, relative paths, and "." and ".." affect path resolution. To account for the existence of symbolic links, we use the file-system graphs of definition 11. Note that the definition of `resolve` in section 6.1.1 can already handle "." and "..". When `resolve` encounters a "." along the path, it follows the "." link within the current lookup directory, if it exists, to the same directory. When it encounters a "..", it follows the ".." link within the current lookup directory, provided it exists, to the parent directory. However, `resolve` must be extended in order to work with symbolic links.

A symbolic link is a file that stores an arbitrary path. During path resolution, if the current path component names a link to a file that is a symbolic link, then the path stored in the symbolic link file is prefixed to the remaining unresolved path, and path resolution restarts with the combined path. If the combined path is relative, then path resolution restarts from the current lookup directory, whereas if it is absolute, it restarts from the root directory.

In order to obtain the path stored in a symbolic link, we first extend the `link_lookup` abstract operation from figure 6.2, as follows:

$$\begin{aligned}
&\textbf{let } \texttt{link_lookup_sl}(\iota, a) \triangleq \\
&\quad \forall FS.\, \langle \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota))\,,\ a \in FS(\iota) \wedge \mathsf{ishl}(FS(FS(\iota)(a))) \Rightarrow \mathsf{fs}(FS) * \texttt{ret} = FS(\iota)(a) \rangle \\
&\quad \sqcap\ \forall FS.\, \langle \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota))\,,\ a \in FS(\iota) \wedge \mathsf{issl}(FS(FS(\iota)(a))) \Rightarrow \mathsf{fs}(FS) * \texttt{ret} = FS\,(FS(\iota)(a)) \rangle \\
&\quad \sqcap\ \texttt{return enoent}(\iota, a) \\
&\quad \sqcap\ \texttt{return enotdir}(\iota)
\end{aligned}$$

From the definition of `link_lookup`, we have split the atomic specification statement of the success case into two. In the first atomic specification statement, the predicate $\mathsf{ishl}(FS(FS(\iota)(a)))$ of the postcondition requires that the link named a within the directory with inode ι does not point to a symbolic link file. As before, if the link named a exists in the ι directory, the inode of the file it points to is returned. In the second atomic specification statement, the predicate $\mathsf{issl}(FS(FS(\iota)(a)))$ requires that the link named a within the ι directory points to a symbolic link file. In this case, if the link exists, the path stored in the symbolic link file is returned.

Symbolic links may introduce loops causing path resolution to not terminate. POSIX requires implementations to return the `ELOOP` error, in case of such loops. A naive implementation of this behaviour would be to detect the existence of cycles in the file-system graph along the path. For example, the resolution process could maintain a set of all the symbolic link paths resolved, and if a

symbolic link path is already in that set, return the error. However, this works only on a quiescent system, or if path resolution is atomic. Since path resolution is not atomic, the concurrent environment may always introduce loops that are not detectable by graph cycle detection algorithms. For this reason, POSIX specifies path resolution to follow a bounded number of symbolic links. This bound is defined by implementations as global constant `SYMLOOP_MAX`, with a minimum value of 8.

Using `link_lookup_sl` instead of `link_lookup` and adapting to account for `ELOOP`, we extend the specification of path resolution in figure 6.12. The additional parameter c counts the number of

```
letrec resolve_sl(path, ι, c) ≜
  if path = null then return ι else
    let a = head(path);
    let p = tail(path);
    let r = link_lookup_sl(ι, a);
    if iserr(r) then return r
    else if ispath(r) then
      if c ≤ SYMLOOP_MAX then
        if isabspath(r) then return resolve_sl(r/p, ι0, c + 1)
        else return resolve_sl(r/p, ι, c + 1) fi
      else return ELOOP fi
    else return resolve_sl(p, r, c) fi
  fi
```

Figure 6.12.: Path resolution specification accounting for symbolic links.

symbolic links encountered. If the current pathname component a names a link to a symbolic link file, then `link_lookup_sl` returns the symbolic-link path. In this case, provided that we are not exceeding the `SYMLOOP_MAX` limit, the resolution continues on the symbolic-link path prefixed to the remaining unresolved path. The path resolution continues from the root directory, if the symbolic-link path is absolute, or otherwise, from the current lookup directory, incrementing the counter c. If the `SYMLOOP_MAX` limit is reached, `ELOOP` is returned.

Every process is associated with a *current working directory*. If a relative path is given as an argument to a file-system operation such as `unlink`, its path resolution initiates from the current working directory of the invoking process. We introduce the abstract predicate $\mathsf{cwd}(pid, \iota)$, which states that the inode of current working directory, associated with the process with process identifier pid, is ι. With this abstract predicate, we define the following function, returning the inode of the current working directory of a process.

$$\textbf{let } \texttt{get\_cwd\_inode}(pid) \triangleq \forall \iota.\, \langle \mathsf{cwd}(pid, \iota)\,,\ \mathsf{cwd}(pid, \iota) * \texttt{ret} = \iota \rangle$$

Using `get_cwd_inode` and `resolve_sl`, we specify the resolution of an arbitrary path as the following function:

```
let resolve(path, ι) ≜
  if isabspath(path) then return resolve_sl(path, ι0, 0)
  else return resolve_sl(path, ι, 0)fi
```

If the given *path* is absolute, path resolution initiates from the root directory, which has the known

inode ι_0. Otherwise, *path* is a relative path. In this case, path resolution initiates from the directory with inode ι, which we can obtain via `get_cwd_inode`.

We now have the means in place to modify our simplified specification of `unlink` to one that can handle arbitrary paths, as follows:

```
unlink(path)
  ⊑ let p = dirname(path);
    let a = basename(path);
    let cwd = get_cwd_inode(pid);
    let r = resolve(p, cwd);
    if ¬iserr(r) then
      return link_delete(r, a)
             ⊔ link_delete_notdir(r, a)
    else return r fi
```

The difference from the original is the use of `get_cwd_inode` to obtain the inode of the current working directory, and using the extended `resolve`. Note that here, `pid` is a special variable, bound to the process id of the current process. All our other simplified specifications in section 6.1, can be analogously extended to work with arbitrary paths.

POSIX additionally defines `unlinkat`$(fd, path)$ as a variant of `unlink`$(path)$. The additional argument is a file descriptor opened for a directory. If *path* is an absolute path, `unlinkat` behaves in the same way as `unlink`. However, if *path* is relative, then the resolution of *path* initiates from the directory associated with the file descriptor *fd*. Alternatively, *fd* may take the value `AT_FDCWD`, in which case the resolution initiates from the current working directory of the process. We define the following function to return the inode associated with a file descriptor:

$$\textbf{let}\ \texttt{get\_fd\_inode}(fd) \triangleq \langle \mathsf{fd}(fd, \iota, \mathit{off}, \mathit{flags})\,,\ \mathsf{fd}(fd, \iota, \mathit{off}, \mathit{flags}) * \texttt{ret} = \iota \rangle$$

With `get_fd_inode` we can give the following refinement specification to `unlinkat`:

```
unlinkat(fd, path)
  ⊑ let p = dirname(path);
    let a = basename(path);
    let cwd = if fd = AT_FDCWD then return get_cwd_inode(pid)
              else return get_fd_inode(fd) fi;
    let r = resolve(cwd, p);
    if ¬iserr(r) then
      return link_delete(r, a)
             ⊔ link_delete_notdir(r, a)
    else return r fi
```

Note that it is easy to prove that `unlinkat`$(\texttt{AT\_FDCWD}, path) \equiv$ `unlink`$(path)$ in our refinement calculus.

6.3.2. File-Access Permissions

POSIX defines the traditional UNIX security model. The operating system has groups of users, where each user belongs to at least one group. Users and groups are identified by numerical identifiers, which we take from the set of natural numbers. Each user or group identifier is associated with a string, acting as the user or group name respectively. Every process is associated with a user identifier, and a set of group identifiers for the groups the user belongs in.

Permission bit	Capability
`S_IRUSR`	User can read.
`S_IWUSR`	User can write
`S_IXUSR`	User can execute.
`S_IRGRP`	Group can read.
`S_IWGRP`	Group can write.
`S_IXGRP`	Group can execute.
`S_IROTH`	Others can read.
`S_IWOTH`	Others can write.
`S_IXOTH`	Others can execute.

Table 6.2.: File-access permission bits.

Every file is associated with a user identifier, a group identifier and access permission bits. The user identifier identifies the file's owner. The group identifier identifies the group to which the file belongs. The access permission bits define the capabilities the owner, the group and the rest of the world have on the file. We summarise available permission bits and the associated capabilities in table 6.2. The execute permission bits are interpreted differently for directories and regular files. For example, `S_IXOTH` for a directory means that every user can traverse the directory during path resolution. On the other hand, the same permission bit for a regular file means that every user can run the file as an executable program.

Let PermBits denote the set of permission bits defined in table 6.2. We define an ownership and permission assignment environment:

$$\Pi \in \textsc{PermHeaps} \triangleq \textsc{Inodes} \xrightarrow{\text{fin}} \textsc{UserIds} \times \textsc{GroupIds} \times \mathcal{P}(\textsc{PermBits})$$

that associates inodes with a user identifier $uid \in \textsc{UserIds} \triangleq \mathbb{N}$, a group identifiers $gid \in \textsc{GroupIds} \triangleq \mathbb{N}$, and a set of active permission bits. We introduce the abstract predicate $\mathsf{fap}(\Pi)$, stating that the files comprising the file-system are given ownership and permission metadata according to Π.

To specify path resolution under file access permissions, all we need is to modify the definition of `link_lookup`. Specifically, when the link that is being followed identifies a directory, we must check that the process has the capability to traverse it, via an execute permission bit. A directory link can be followed if one of the following holds:

- the user identifier of the process is the user identifier of the directory's owner and the execute permission bit is set for the owner,
- one of the group identifiers of the process matches the group identifier of the directory and the execute permission bit set for the group, and

- the execute permission bit is set for others.

We encode these conditions with the following auxiliary predicate:

$$\begin{aligned}\mathsf{ix}(\mathit{fuid}, \mathit{fgid}, \mathit{bits}) \triangleq\ & (\texttt{S\_IXUSR} \in \mathit{bits} \wedge \mathit{fuid} = \texttt{uid}) \\ & \vee (\texttt{S\_IXGRP} \in \mathit{bits} \wedge \mathit{fgid} \in \texttt{gids}) \\ & \vee (\texttt{S\_IXOTH} \in \mathit{bits})\end{aligned}$$

The arguments *fuid*, *fgid* and *bits* correspond to the identifiers of the user and group owning the file, and the set of enabled permission bits for the file respectively. The variables `uid` and `gids` are special variables, binding the user and group identifiers associated with the current process.

We redefine `link_lookup` based on the definition given section 6.3.1, taking symbolic links into account, as follows:

$$\begin{aligned}&\textbf{let } \texttt{link\_lookup}(\iota, a) \triangleq \\ &\quad \forall FS, \Pi. \left\langle \begin{array}{l} \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota)) * \mathsf{fap}(\Pi)\,, \\ a \in FS(\iota) \wedge \mathsf{isdir}(FS(\iota)(a)) \wedge \mathsf{ix}(\Pi(FS(\iota)(a))) \Rightarrow \mathsf{fs}(FS) * \mathsf{fap}(\Pi) * \texttt{ret} = FS(\iota)(a) \end{array} \right\rangle \\ &\quad \sqcap\ \forall FS.\, \langle \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota))\,,\ a \in FS(\iota) \wedge \mathsf{isfile}(FS(\iota)(a)) \Rightarrow \mathsf{fs}(FS) * \texttt{ret} = FS(\iota)(a) \rangle \\ &\quad \sqcap\ \forall FS.\, \langle \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota))\,,\ a \in FS(\iota) \wedge \mathsf{issl}(FS(\iota)(a)) \Rightarrow \mathsf{fs}(FS) * \texttt{ret} = FS\,(FS(\iota)(a)) \rangle \\ &\quad \sqcap\ \texttt{return eaccess\_x}(\iota, a) \\ &\quad \sqcap\ \texttt{return enoent}(\iota, a) \\ &\quad \sqcap\ \texttt{return enotdir}(\iota)\end{aligned}$$

We distinguish the success case into three sub-cases based on the type file the links is pointing to. In the first case, an existing link to a directory is followed only if the process has the execute capability. In the second and third cases permissions are not checked. Specifically, the second case applies to a regular file link, at which point path resolution stops and some other action is subsequently performed on the file. It is the responsibility of the subsequent action to check that the process has the appropriate privileges. The third case covers symbolic links, at which point path resolution restarts with the symbolic-link path. Thus permissions will be checked for the symbolic-link path during its resolution.

The additional error case, `eaccess_x`, is defined as follows:

$$\begin{aligned}&\textbf{let } \texttt{eaccess\_x}(\iota, a) \triangleq \\ &\quad \forall FS, \Pi. \left\langle \begin{array}{l} \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota)) * \mathsf{fap}(\Pi)\,, \\ a \in FS(\iota) \wedge \mathsf{isdir}(FS(\iota)(a)) \wedge \neg\mathsf{ix}(\Pi(FS(\iota)(a))) \Rightarrow \mathsf{fs}(FS) * \mathsf{fap}(\Pi) * \texttt{ret} = \texttt{EACCESS} \end{array} \right\rangle\end{aligned}$$

When the process does not have execute permission on the directory the link points to, the `EACCESS` error is returned.

Consider the case of the `unlink` operation. After a successful path resolution, we want to remove a link from the resolved directory with either `link_delete` or `link_delete_notdir`. This entails writing to the directory, and therefore the process is required to have a write permission for it. We encode

the test for this permission as the following auxiliary predicate:

$$\begin{aligned}\mathsf{iw}(\mathit{fuid}, \mathit{fgid}, \mathit{bits}) \triangleq\ & (\texttt{S\_IWUSR} \in \mathit{bits} \wedge \mathit{fuid} = \texttt{uid}) \\ & \vee (\texttt{S\_IWGRP} \in \mathit{bits} \wedge \mathit{fgid} \in \texttt{gids}) \\ & \vee (\texttt{S\_IWOTH} \in \mathit{bits})\end{aligned}$$

To account for the permission check, we redefine `link_delete` and `link_delete_notdir` as follows:

$$\begin{aligned}&\textbf{let}\ \texttt{link\_delete}(\iota, a) \triangleq \\ &\quad \mathbb{V}FS, \Pi. \left\langle \begin{array}{l} \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota)) * \mathsf{fap}(\Pi)\,, \\ a \in FS(\iota) \wedge \mathsf{iw}(\Pi(\iota)) \Rightarrow \mathsf{fs}(FS[\iota \mapsto FS(\iota) \setminus \{a\}]) * \mathsf{fap}(\Pi) * \texttt{ret} = 0 \end{array} \right\rangle \\ &\quad \sqcap \texttt{return eaccess\_w}(\iota) \\ &\quad \sqcap \texttt{return enoent}(\iota, a) \\ &\quad \sqcap \texttt{return enotdir}(\iota)\end{aligned}$$

$$\begin{aligned}&\textbf{let}\ \texttt{link\_delete\_notdir}(\iota, a) \triangleq \\ &\quad \mathbb{V}FS, \Pi. \left\langle \begin{array}{l} \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota)) * \mathsf{fap}(\Pi)\,, \\ \mathsf{isfile}(FS(\iota)(a)) \wedge \mathsf{iw}(\Pi(\iota)) \Rightarrow \mathsf{fs}(FS[\iota \mapsto FS(\iota) \setminus \{a\}]) * \mathsf{fap}(\Pi) * \texttt{ret} = 0 \end{array} \right\rangle \\ &\quad \sqcap \texttt{return eaccess\_w}(\iota) \\ &\quad \sqcap \texttt{return enoent}(\iota, a) \\ &\quad \sqcap \texttt{return enotdir}(\iota) \\ &\quad \sqcap \texttt{return err\_nodir\_hlinks}(\iota, a)\end{aligned}$$

The success cases are amended with the permission check. The `eaccess_w` error case returns the `EACCESS` error when the check fails, and is defined as follows:

$$\textbf{let}\ \texttt{eaccess\_w}(\iota) \triangleq \mathbb{V}\Pi. \langle \mathsf{fap}(\Pi)\,,\ \neg\mathsf{iw}(\Pi(\iota)) \Rightarrow \mathsf{fap}(\Pi) * \texttt{ret} = \texttt{EACCESS} \rangle$$

POSIX additionally defines operations for modifying the ownership and permission bits of files. Consider the example of the `chmod` operation. Informally, `chmod`$(\mathit{path}, \mathit{bits})$, resolves *path* and atomically sets the permissions bits enabled for the resolved file to those in *bits*. Formally, we give the following refinement specification to the operation:

$$\begin{aligned}&\texttt{chmod}(\mathit{path}, \mathit{bits}) \\ &\quad \sqsubseteq \textbf{let}\ \mathit{cwd} = \texttt{get\_cwd\_inode}(\texttt{pid}); \\ &\qquad \textbf{let}\ r = \texttt{resolve}(\mathit{path}, \mathit{cwd}); \\ &\qquad \textbf{if}\ \neg\mathsf{iserr}(r)\ \textbf{then}\ \texttt{return update\_perms}(r, \mathit{bits}) \\ &\qquad \textbf{else}\ \texttt{return}\ r\ \textbf{fi}\end{aligned}$$

where `update_perms` performs the atomic update of the permission bits for the resolve file and is

defined as follows:

$$\begin{aligned}&\textbf{let } \texttt{update\_perms}(\iota, \mathit{bits}) \triangleq \\ &\quad \mathbb{V}\Pi.\, \langle \mathsf{fap}(\Pi)\,,\ \mathsf{iu}(\Pi(\iota)) \Rightarrow \mathsf{fap}(\Pi[\iota \mapsto \mathit{bits}]) * \texttt{ret} = 0\rangle \\ &\quad \sqcap \texttt{eperm\_u}(\iota)\end{aligned}$$

Updating the permission bits is allowed only if the process is running with a user identifier matching the owner of the file, or with one of its group identifiers matching the group of the file. We encode this permission check with the following auxiliary predicate:

$$\mathsf{iu}(\mathit{fuid}, \mathit{fgid}, \mathit{mode}) \triangleq \texttt{uid} = \mathit{fuid} \vee \mathit{fgid} \in \texttt{gids}$$

The `eperm_u` error case returns the `EPERM` error, if the permission check fails, as defined below:

$$\textbf{let } \texttt{eperm\_u}(\iota) \triangleq \mathbb{V}\Pi.\, \langle \mathsf{fap}(\Pi)\,,\ \neg\mathsf{iu}(\Pi(\iota)) \Rightarrow \mathsf{fap}(\Pi) * \texttt{ret} = \texttt{EPERM}\rangle$$

6.3.3. Threads and Processes

POSIX is a standard for mulit-process operating systems, where processes are running concurrently and each process may have concurrent threads of execution. In the previous sections we have been implicit about this distinction. We will now introduce what this formally means within the context of our specification language.

Every process, and consequently every thread, shares the same file system. On the other hand, every process has its own heap memory, as well as process based state associated with the file system, such as file descriptors. We will refer to the combination of heap memory, and process based state associated with the file system as the *process heap*. The process heap is shared by all threads within the same process. Every process has a unique process identifier (an integer) that is bound to the variable `pid`. Note that this variable is process-local: it is bound to a different value in different processes.

In order to distinguish between resources of different processes heaps, every process-heap predicate is indexed by a process identifier. For example, the heap cell predicate $x \overset{pid}{\mapsto} n$, asserts the existence of a heap cell with address x and value n within the process heap of the process with process identifier pid. Note that in $x \overset{pid}{\mapsto} n * x \overset{pid'}{\mapsto} n$, the heap cells are disjoint. Even though the heap cells have the same address, they belong to a different process heap. We omit the process identifier when the predicate is indexed by `pid`. For example, $\mathsf{fd}(\mathit{fd}, -, -, -) \triangleq \mathsf{fd}(\mathit{fd}, -, -, -)_{\texttt{pid}}$, and thus the predicate $\mathsf{fd}(\mathit{fd}, -, -, -)$, asserts the existence of the file descriptor fd within the current process.

In section 6.1.1, we have used the parallel composition $\phi \parallel \psi$, in the specifications of `link` and `rename`. This composition is thread-parallel, where ϕ and ψ are threads of the same process. For example, in $\mathbb{V}n \in \mathbb{N}.\, \langle x \mapsto n,\ x \mapsto n+1\rangle \parallel \mathbb{V}n \in \mathbb{N}.\, \langle x \mapsto n,\ x \mapsto n+1\rangle$, we have two threads, each atomically incrementing the same heap cell.

For parallel processes, we introduce the process-parallel composition $\phi \,|||\, \psi$, where ϕ and ψ run as different processes. In ϕ and ψ, `pid` is bound to a different value. Therefore, in $\mathbb{V}n \in \mathbb{N}.\, \langle x \mapsto n,\ x \mapsto n+1\rangle \,|||\, \mathbb{V}n \in \mathbb{N}.\, \langle x \mapsto n,\ x \mapsto n+1\rangle$, we have two processes, each incrementing a different heap cell.

We encode the process-parallel composition $\phi \,|||\, \psi$ in terms of thread-parallel composition as follows:

$$\phi \,|||\, \psi \triangleq \exists new\_pid.\, \{P * P(\texttt{pid}),\, \texttt{pid} \neq new\_pid * P * P(\texttt{pid}) * P(new\_pid)\}^{\mathcal{A}}_{k};\\ \phi \parallel (\lambda \texttt{pid}.\, \psi)\; new\_pid$$

Consider the first specification statement in the definition. Here, we use $P(pid)$ to denote the resources indexed by the process identifier pid, and we use P to denote resources that can be shared between processes, such as the file system. The statement creates $P(new\_pid)$ as a duplicate of the resources of the current process $P(\texttt{pid})$. The process identifier new_pid is chosen non-deterministically by the existential quantification. Note that in the postcondition ensures that $new\_pid \neq \texttt{pid}$. In the subsequent thread-parallel composition the specification ϕ uses `pid` as the process identifier. Effectively, ϕ takes the role of the parent process. In order for ψ to inherit the new process identifier new_pid, we wrap it a function that binds its argument to `pid` and apply it to new_pid. Thus, within this function `pid` is bound to a different value than that in ϕ. Finally, since in both ϕ and ψ the current process identifier variable is bound to a different value, each specification can only access $P(\texttt{pid})$ for their value of `pid`. This guarantees isolation between process based resources.

In this setting, process-based resources, such as heap cells, are abstract predicates implemented in terms of concurrent indexes. For example, in the case of heap cells we can use a concurrent index mapping process identifiers to normal heaps: $\mathcal{M} : \mathbb{N} \xrightarrow{\text{fin}} \textsc{Heap}$. Let $\mathsf{Map}(\mathcal{M})$ be an abstract predicate asserting the existence of the index $\mathcal{M}$. Then, we can implement a process-heap predicate as follows:

$$x \overset{pid}{\mapsto} y \triangleq \exists \mathcal{M}.\, \mathsf{Map}(\mathcal{M}) \wedge \mathcal{M}(pid) = x \mapsto y$$

Abstract specifications for concurrent indexes using abstract predicates such as Map, as well as an implementation to the concurrent skip-list of `java.util.concur` have been studied using TaDA [98]. Since the specification statements developed in this dissertation are based on TaDA it should be possible to use the same specifications here as well.

6.4. Conclusions

We have presented our approach for developing a concurrent specification for POSIX file-system operations and reasoning about client applications that use the file system, demonstrating that it successfully tackles the challenges we identified in chapter 2, section 2.3: operations perform multiple atomic actions, they exhibit high levels of non-determinism and that the file system is a public namespace. Our approach is based on a specification programming language with which file-system operations are specified as specification programs. The building blocks of such specification programs are the atomic and Hoare specification statements for specifying atomic and non-atomic actions respectively. Atomic and Hoare specification statements are based on the atomic and Hoare triples of the TaDA [30] program logic respectively. Specification statements can be composed to form more complex specifications via sequential, non-deterministic and parallel composition, which allow us to specify operations that perform sequences of atomic actions and exhibit non-deterministic behaviour.

Reasoning about our specifications is facilitated by a refinement calculus for proving contextual refinements between specification programs. Thus our approach can be viewed as a combination of

TaDA specifications and reasoning with contextual refinement. We have demonstrated how protocols describing the concurrent interaction between threads in the form of TaDA shared regions and guards are used to reason about client programs of the file system. In particular, the same mechanism is used to define context invariants that constraint the actions of the environment such that we can derive useful specifications for clients within the public-namespace nature of the file system.

We have informally introduced the key features of our specification language and its associated refinement calculus through select examples of specifying core file-system operations in section 6.1 and reasoning about a lock-file client module in section 6.2. The informal introduction provides the necessary intuition for the formal development of the language and calculus in chapter 7.

In order to focus on the challenges of formalising the concurrent behaviour in section 6.1 we focused on simplified core fragment of POSIX file-system operations, using simplified paths and ignoring features such as file-access permissions. The core fragment covers core operations for manipulating links, directories and regular file I/O. The complete specification of this fragment is given in appendix A. In section 6.3 we demonstrate how our specification can be extended to larger fragments, including symbolic links, relative paths, file-access permissions and reasoning about threads of different processes.

Sections 6.1 and 6.3 highlight a significant advantage of our approach: its flexibility. Our specifications can be easily adapted to account for different interpretations of the text in the POSIX standard, to capture implementation specification behaviour, or even to update the specification to future revisions of the standard. The granularity of atomicity in our specifications can be easily adapted. For example, in section 6.1.2 we discussed how directory creation can be split into multiple atomic actions and reciprocally coalesced into a single atomic step, as is expressed by mumbling in the refinement calculus. Following this pattern, more atomic steps can be added or removed from the specification whenever required. Furthermore, the atomicity guarantees offered by file-system operations, or individual steps comprising such operations, are adaptable. Atomic specification statements can be used to specify actions that are observed to update the file-system state atomically. In the absence of any atomicity guarantees, Hoare specification statements can be used to specify non-atomic operations. As discussed in section 6.1.3 using the example of I/O operations, the general form of atomic specification statements even allows us to specify actions that atomically update only some parts of the state and non-atomically update other parts. The laws of our refinement calculus easily allow us to adapt an atomic specification, choosing which parts of the state are updated atomically and which are not.

In summary, our approach is not only capable of giving a formal specification of POSIX file-system operations that faithfully captures their concurrent behaviour, but it can easily adapt to future changes in the POSIX standard and its implementations.

7. Atomicity and Refinement

We now present the technical details behind the specification and reasoning approach introduced in chapter 6. In section 7.1 we define the syntax of our core specification language and TaDA assertion used therein. The core specification language does not define atomic specification statements as a primitive. Instead, we define the *primitive atomic specification statement*, of the form $a(\forall \vec{x}.\, P,\, Q)$, which specifies an atomic action in terms of primitive atomicity, in a similar style to Turon and Wand's atomic actions and linearisability. In section 7.2 we define a general state model and the semantics of TaDA assertions. In section 7.3 we define operational and denotational semantics for our specification language and in section 7.4 we define the semantics of contextual refinement and present our soundness proof. In section 7.5 we introduce general refinement laws as well as refinement laws for primitive atomicity. In section 7.6 we encode atomic specification statements in terms of our core specification language for primitive atomicity and give refinement laws for abstract atomicity. Therefore, we develop abstract atomicity as a derived construct. Finally, in section 7.7 we define the syntactic constructs used in the specification given in chapter 6.

7.1. Specification Language

Specification programs and the assertions used in specification statements share the same variable environment. We do not distinguish between program variables and logical variables. For the general theory of atomicity and refinement that we develop in this chapter, we only use a basic set of boolean and integer values and associated expressions. We leave these definitions open-ended and applications of our theory, such as the POSIX specification, can extend these as appropriate.

Definition 14 (Variables and values)**.** *Let* Var *be a countable set of variables. Let* Val *be the set of values assigned to variables, at least comprising booleans, integers and the unit value,* $\mathbf{1} \triangleq \{()\}$*:*

$$\textsc{Val} \triangleq \mathbb{B} \cup \mathbb{Z} \cup \mathbf{1} \ldots$$

Variable stores, $\rho \in \textsc{VarStore} \triangleq \textsc{Var} \to \textsc{Val}$*, assign values to variables.*

The variable environment defined above is basic, and will require extensions when defining the semantics of assertions and specification programs. This is because some variables, for functions and recursion, will receive special treatment.

Definition 15 (Expressions)**.** *Expressions,* $e, e' \in \textsc{Expr}$*, are defined by the grammar:*

$$
\begin{array}{lrll}
\textit{Expressions} & e, e' ::= & v & \textit{value } v \in \textsc{Val} \\
 & & \mid x & \textit{variable } x \in \textsc{Var} \\
\textit{Boolean Expressions} & & \mid \neg e & \textit{negation} \\
 & & \mid e \wedge e' & \textit{conjunction} \\
 & & \mid e \vee e' & \textit{disjunction} \\
 & & \mid e = e' & \textit{equality} \\
 & & \mid e < e' & \textit{inequality} \\
\textit{Integer Expressions} & & \mid e + e' & \textit{addition} \\
 & & \mid e - e' & \textit{subtraction} \\
 & & \mid e \cdot e' & \textit{multiplication} \\
 & & \mid e \div e' & \textit{division} \\
 & & \mid \ldots &
\end{array}
$$

Here we have chosen the $\div$ for division, to distinguish from the path separator used in POSIX. Expressions have a standard, albeit partial, denotational semantics.

Definition 16 (Expression evaluation)**.** *Expression evaluation,* $[\![-]\!]^- : \textsc{VarStore} \to \textsc{Expr} \rightharpoonup \textsc{Val}$*, is defined as a partial function over expressions parameterised by a variable store:*

$$
\begin{aligned}
[\![v]\!]^\rho &\triangleq v \\
[\![x]\!]^\rho &\triangleq \rho(x) \\
[\![\neg e]\!]^\rho &\triangleq \neg\, [\![e]\!]^\rho \quad \textit{if } [\![e]\!]^\rho \in \mathbb{B} \\
[\![e \wedge e']\!]^\rho &\triangleq [\![e]\!]^\rho \wedge [\![e']\!]^\rho \quad \textit{if } [\![e]\!]^\rho \in \mathbb{B} \textit{ and } [\![e']\!]^\rho \in \mathbb{B} \\
[\![e \vee e']\!]^\rho &\triangleq [\![e]\!]^\rho \vee [\![e']\!]^\rho \quad \textit{if } [\![e]\!]^\rho \in \mathbb{B} \textit{ and } [\![e']\!]^\rho \in \mathbb{B} \\
[\![e = e']\!]^\rho &\triangleq [\![e]\!]^\rho = [\![e']\!]^\rho \\
[\![e < e']\!]^\rho &\triangleq [\![e]\!]^\rho < [\![e']\!]^\rho \quad \textit{if } [\![e]\!]^\rho \in \mathbb{Z} \textit{ and } [\![e']\!]^\rho \in \mathbb{Z} \\
[\![e - e']\!]^\rho &\triangleq [\![e]\!]^\rho - [\![e']\!]^\rho \quad \textit{if } [\![e]\!]^\rho \in \mathbb{Z} \textit{ and } [\![e']\!]^\rho \in \mathbb{Z} \\
[\![e \cdot e']\!]^\rho &\triangleq [\![e]\!]^\rho \cdot [\![e']\!]^\rho \quad \textit{if } [\![e]\!]^\rho \in \mathbb{Z} \textit{ and } [\![e']\!]^\rho \in \mathbb{Z} \\
[\![e \div e']\!]^\rho &\triangleq [\![e]\!]^\rho \div [\![e']\!]^\rho \quad \textit{if } [\![e]\!]^\rho \in \mathbb{Z} \textit{ and } [\![e']\!]^\rho \in \mathbb{Z} \setminus \{0\}
\end{aligned}
$$

In all other cases, the result is undefined.

When the denotation of an expression is undefined, a fault is triggered in the semantics of our specification language.

For the purposes of our POSIX fragment specification we extend the basic expressions further with additional operators and values. An example of this are sets and the standard set operators such as union and intersection. Sets have many uses in our POSIX specification, such as a more abstract treatment of flag arguments instead of low-level bitwise manipulation as seen in chapter 6, section 6.1.3. We discuss these extensions in detail in appendix A.

Our assertion language is based on TaDA [30, 28], extending intuitionistic separation logic [81] with

regions, guards, atomicity tracking resources and abstract predicates.

Definition 17 (Assertion Language)**.** *Assertions,* $P, Q, R \in \textsc{Assrt}$*, are defined by the grammar:*

Assertions	$P, Q, R ::=$	false	*falsehood*
		$\mid \mathsf{true}$	*truthfulness*
		$\mid P * Q$	*separating conjunction*
		$\mid P \wedge Q$	*conjunction*
		$\mid P \vee Q$	*disjunction*
		$\mid \neg P$	*negation*
		$\mid \exists x.\, P$	*existential quantification*
		$\mid \forall x.\, P$	*universal quantification*
		$\mid P \Rightarrow Q$	*implication (material)*
		$\mid e \mapsto e'$	*heap cell at* e *storing* e'
		$\mid \mathbf{t}^k_\alpha(\vec{e}, e')$	*shared region* α*, of type* $\mathbf{t}$*, region level* k*, parameterised by* $\vec{e}$ *and with abstract state* e'
		$\mid I_r(\mathbf{t}^k_\alpha(\vec{e}, e'))$	*shared-region interpretation*
		$\mid [\mathsf{G}(\vec{e})]_\alpha$	*guard* G *for region* α*, parameterised by* $\vec{e}$
		$\mid \alpha \Mapsto \blacklozenge$	*atomicity tracking resource for region* α*, allowing atomic update*
		$\mid \alpha \Mapsto \diamondsuit$	*atomicity tracking resource for region* α*, not allowing atomic update*
		$\mid \alpha \Mapsto (e, e')$	*atomicity tracking resource for region* α*, witnessing an atomic update from* e *to* e'
		$\mid \mathsf{ap}(\vec{e})$	*application of abstract predicate* ap *to parameters* $\vec{e}$
		$\mid I_a(\mathsf{ap}(\vec{e}))$	*interpretation of abstract predicate*
		$\mid \mathsf{pred}e$	*application of concrete predicate* pred *to* e
		$\mid e$	*expression*
Concrete Predicates	$\mathsf{pred} ::=$	$\lambda x.\, P$	*non-recursive predicate*
		$\mid \mu X.\, \lambda x.\, P$	*recursive predicate*
		$\mid X$	*recursion variable*

Recursion variables, $X \in \textsc{AssrtRecVars}$*, are taken from a countable set, disjoint from* Var*. The binding precedence, from strongest to weakest, is:* $\neg, *, \wedge, \vee, \forall, \exists, \Rightarrow$*.*

The specification language we informally presented in chapter 6 builds on atomic specification statements of the form, $\mathbb{V}x \in X.\, \langle P(x),\, Q(x)\rangle$, and Hoare specification statements of the form $\{P,\, Q\}$. However, the reality is more complicated. The atomic and Hoare specification statements are not primitive constructs of the specification language. The primitive construct for specifying an atomic action is the *primitive atomic statement* of the form, $a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_k$. Intuitively this specifies a physical atomic action on the shared state. Examples of such actions include compare-and-swap (CAS), and atomic reads and writes to heap cells. In contrast to the atomic specification statement, it always tolerates all possible interference; the specified action will always update the state of the precondition to the state of the postcondition atomically, irrespective of actions taken by the environment.

We define a core specification language, based on the primitive atomic statement. As we will see in section 7.6, the atomic and Hoare specification statements are defined in terms of sequences of primitive atomic statements.

Definition 18 (Specification Language)**.** *The language of specifications, $\mathcal{L}$, is defined by the following grammar:*

Specifications	$\phi, \psi ::=$	$\phi; \psi$	*Sequential composition*
		$\mid \phi \parallel \psi$	*Parallel composition*
		$\mid \phi \sqcup \psi$	*Angelic choice*
		$\mid \phi \sqcap \psi$	*Demonic choice*
		$\mid \exists x.\, \phi$	*Existential quantification*
		$\mid$ **let** $f = F$ **in** ϕ	*Function binding*
		$\mid Fe$	*Function application*
		$\mid a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_{k}$	*Primitive atomic statement*
Functions	$F ::=$	f	*Function variable*
		$\mid A$	*Recursion variable*
		$\mid F_l$	*Function literal*
Function Literals	$F_l ::=$	$\mu A.\, \lambda x.\, \phi$	*Recursive function*
		$\mid \lambda x.\, \phi$	*Function*

where $k \in$ RLevel and $\mathcal{A} \in$ AContext are defined in section 7.2. Recursion variables, $A \in$ RecVars, and function variables, $f \in$ FuncVars, are taken from disjoint countable sets, both disjoint from Var*, and cannot be bound nor free in P or Q. Predicate-recursion variables, $X \in$ AssrtRecVars, are also disjoint from* RecVars *and* FuncVars *and cannot be bound nor free in ϕ. The operator binding precedence, from strongest to weakest, is: $Fe, \mu A.\,, \lambda x.\,, \sqcap, \sqcup, \exists x.\,;, \parallel$, with parentheses used to enforce order.*

The specification language includes traditional programming constructs including sequential composition $\phi; \psi$, parallel composition $\phi \parallel \psi$, together with first-order functions and recursion. We include additional constructs to account for specification non-determinism. Angelic choice, $\phi \sqcup \psi$, behaves either as ϕ or as ψ. Demonic choice, $\phi \sqcap \psi$, behaves as ϕ and ψ. The `unlink` specification in section 6 is an example that uses both angelic and demonic non-determinism. Existential quantification, $\exists x.\, \phi$, behaves as ϕ for some choice of x.

In $a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_{k}$, the binder $\forall \vec{x}.$ binds variables across the precondition and postcondition. Free variables, anywhere in the specification language, are implicitly universally quantified by the context. The subscript k is analogous to the subscript in region assertions in definition 17. The region level is simply an integer which signifies that only regions below level k may be replaced by their interpretation (opened) in the refinement of a specification statement. Their purpose is to ensure that we cannot open the same region twice during a refinement derivation, as this could unsoundly duplicate resources encapsulated by the region. The atomicity context, $\mathcal{A}$, defines the atomic updates performed on regions. In all our examples, we only use an atomicity context to define an atomic update to a single

shared region. In general, we allow the atomicity context to update multiple regions with one atomic update per region.

We keep the specification language minimal. For simplicity and to keep specifications declarative, variables are immutable. Atomic and Hoare specification statements are defined in terms of this core specification language in section 7.6. Additional programming constructs used in the specifications given in chapter 6 are easily encoded. We formally define the encodings of these syntactic features in section 7.7.

7.2. Model

Our assertions about the shared state are based on those of TaDA. The same applies to the models of assertions which we develop here. Therefore, the contents of this chapter are heavily based on the model defined in the technical report of TaDA [31] and da Rocha Pinto's thesis [28].

Regions, guards, atomicity tracking resources, region levels, atomicity contexts and abstract predicates are merely instrumentation – also referred to as ghost state in the literature – of the concrete shared state, the purpose of which to enable scalable, modular and compositional reasoning for concurrent programs. We take the concrete state shared between threads to be the heap memory.

Definition 19 (Heaps)**.** *Let* $\textsc{Addr}$ *be the set of addresses such that* $\textsc{Addr} \subseteq \textsc{Val}$*. A heap,* $h \in \textsc{Heap} \triangleq \textsc{Addr} \xrightarrow{fin} \textsc{Val}$*, is a finite partial function from addresses to values. Heaps form a separation algebra* $(\textsc{Heap}, \uplus, \emptyset)$*, where* $\uplus$ *is the disjoint union of partial functions and* $\emptyset$ *is the partial function with an empty domain. Heaps are ordered by resource ordering,* $h_1 \leq h_2 \overset{def}{\iff} \exists h_3.\, h_1 \uplus h_3 = h_2$*.*

Recall from chapter 4, that when we refer to separation algebras, we mean separation algebras in the style of the Views framework [35]. These are different from the separation algebras initially introduced by Calcagno *et al.* [25], in that they do not require the cancellative property and allow multiple units. Even though we use heaps for concrete shared states, any separation algebra with resource ordering can be used instead. All instrumentation is eventually reified to concrete states.

Our POSIX specification is using abstract predicates to describe the state of the file system. For example we model the file system as a graph $FS \in \mathcal{FS}$ and use the abstract predicate $\mathsf{fs}(FS)$ to assert that the state of the file system is FS. This difference between the concrete states in the semantics of our specification language and abstract states in our POSIX specification is intentional. This will allow us to refine the POSIX specification to various heap based implementations in the future, something which would not have been possible if we used file-system graphs as the concrete states of the specification language.

As mentioned in chapter 6, guard resources represent capabilities to atomically update the state of a shared region. Guard resources can be taken from any user-defined separation algebra [25], with some additional properties that we require of guards. Therefore, we refer to such separation algebras as guard algebras.

Definition 20 (Guards and Guard Algebras)**.** *Let* $\textsc{Guard}$ *be a set containing all the possible guards. A guard algebra* $\zeta = (\mathcal{G}, \bullet, \mathbf{0}, \mathbf{1})$ *comprises:*

- *a guard carrier set* $\mathcal{G} \subseteq \textsc{Guard}$*,*

- *a partial, binary, associative and commutative operator* $\bullet : \mathcal{G} \times \mathcal{G} \rightharpoonup \mathcal{G}$,
- *an identity element,* $\mathbf{0} \in \mathcal{G}$, *such that* $\forall x \in \mathcal{G}.\, \mathbf{0} \bullet x = x$,
- *a maximal element,* $\mathbf{1} \in \mathcal{G}$, *such that* $\forall x \in \mathcal{G}.\, x \leq \mathbf{1}$, *where* $x \leq y \overset{def}{\iff} \exists z.\, x \bullet z = y$.

We denote the set of all guard algebras as GALG. A guard algebra is a separation algebra with a single unit, $\mathbf{0}$. Given guard algebra $\zeta \in \text{GALG}$ we denote the carrier set of guards with $\mathcal{G}_\zeta$, the identity (zero) guard with $\mathbf{0}_\zeta$, the maximal guard with $\mathbf{1}_\zeta$ and the resource ordering $\leq_\zeta$. Let $g_1, g_2 \in \mathcal{G}_\zeta$. We use the notation $g_1 \# g_2$ to denote the fact that $g_1 \bullet g_2$ is defined.

Regions denote shared abstract state. Each region is associated with a labelled transition system, the transitions of which define how the abstract state of the region can be atomically updated. Each transition is labelled by a guard. A thread has the capability to update the abstract state of a region, as long as it owns the guard resource that "guards" the transition by which the update is allowed.

Definition 21 (Abstract States and Transition Systems)**.** *Let* ASTATE *be a set containing all the possible region abstract states. Let* $\zeta \in \text{GALG}$. *A* guard-labelled transition system, $\mathcal{T} : \mathcal{G}_\zeta \overset{mono}{\to} \mathcal{P}(\text{ASTATE} \times \text{ASTATE})$, *is a function mapping guards to abstract state binary relations. The mapping is required to be monotone with respect to guard resource ordering* $(\leq_\zeta)$ *and subset ordering; having more guard resources permits more transitions. The set of all* ζ*-labelled transition systems is denoted by* ASTS_ζ.

Regions have types that associate regions of the same type with a guard algebra and a guard-labelled transition system.

Definition 22 (Abstract Region Types)**.** *Let* RTNAME *be the set of region type names. An* abstract region typing,

$$\mathbf{T} \in \text{ARTYPE} \triangleq \text{RTNAME} \to \biguplus_{\zeta \in \text{GALG}} \{\zeta\} \times \text{ASTS}_\zeta$$

maps region type names to pairs of guard algebras and guard-labelled transition systems. Let $\mathbf{t} \in$ RTNAME. *The guard labelled transition system of the region type name* $\mathbf{t}$ *is denoted by* $\mathcal{T}_\mathbf{t}$.

Following TaDA, as well as other separation logics for fine-grained concurrency [36, 88], we use abstract predicates as mechanism for data abstraction.

Definition 23 (Abstract Predicates)**.** *Let* APNAME *be the set of abstract predicate names. An* abstract predicate *ap* $\in \text{APNAME} \times \text{VAL}^*$, *comprises an abstract predicate name and a list of parameters. An* abstract predicate bag, $b \in \text{APBAG} \triangleq \mathcal{M}_{fin}(\text{APNAME} \times \text{VAL}^*)$, *is a finite multiset of abstract predicates. Abstract predicate bags form a separation algebra,* $(\text{APBAG}, \cup, \emptyset)$, *where* $\cup$ *is the multiset union and* $\emptyset$ *is the empty multiset. Abstract predicate bags are ordered by the subset order* $\subseteq$.

Regions may refer to other regions and circularities may arise. This is a problem for the refinement laws that allow us to open a region by replacing it with its interpretation. During the derivation of a refinement, if there is a circularity, then the refinement laws could be used to open the same region twice. This is unsound as it would replicate the resource encapsulated by the region. To prevent this unsoundness, we associate each region and specification statement with a region level.

Definition 24 (Region Levels)**.** *A* region level*,* $k \in \text{RLEVEL} \triangleq \mathbb{N}$*, is a natural number. Levels are ordered by the* $\leq$ *ordering on natural numbers.*

Intuitively, region levels track the nesting depth of regions. The region level associated with a region indicates how deeply the region is nested. The region level associated with a specification statement indicates how far we can look into regions. In order to open a region, we require that the region level associated with the region is less than the region level associated with the specification statement. When a region is opened, the region level associated with the specification statement containing it is decreased. Then, if the same region is encountered again, its region level will be greater that than of the specification statement, and thus it will not be possible to open it again.

Each region has a unique identifier, which is used to identify the region's type, region level and parameters.

Definition 25 (Region Assignments)**.** *Let* RID *be a countable set of region identifiers. A* region assignment*,* $r \in \text{RASS} \triangleq \text{RID} \xrightarrow{fin} \text{RLEVEL} \times \text{RTNAME} \times \text{VAL}^*$*, is a finite partial function from region identifiers to region levels and parameterised region type names. Region assignments are ordered by extension ordering:* $r_1 \leq r_2 \overset{def}{\iff} \forall \alpha \in \text{dom}(r_1).\, r_2(\alpha) = r_1(\alpha)$*.*

In the following definitions, we assume a fixed abstract region typing, $\mathbf{T} \in \text{ARTYPE}$. Each region in a region assignment is associated with guards from the guard algebra defined in the region typing.

Definition 26 (Guard Assignments)**.** *Let* $r \in \text{RASS}$ *be a region assignment. A* guard assignment*,*

$$\gamma \in \text{GASSN}_r \triangleq \prod_{\alpha \in \text{dom}(r)} \mathcal{G}_{\mathbf{T}(r(\alpha)\downarrow_2)\downarrow_1}$$

is a mapping from the regions declared in the region assignment r *to the guards of the appropriate type for each region. The guards assigned to a region with region identifier* α *are denoted by* $\gamma(\alpha)$*. Guard assignments form a separation algebra,* $(\text{GASSN}_r, \bullet, \lambda\alpha.\, \mathbf{0}_{\mathbf{T}(r(\alpha)\downarrow_2)\downarrow_1})$*, where* $\bullet$ *is the pointwise lift of guard composition:*

$$\gamma_1 \bullet \gamma_2 \triangleq \lambda\alpha.\, \gamma_1(\alpha) \bullet \gamma_2(\alpha)$$

For $\gamma_1 \in \text{GASSN}_{r_1}$ *and* $\gamma_2 \in \text{GASSN}_{r_2}$ *with* $r_1 \leq r_2$*, guard assignments are ordered extensionally:*

$$\gamma_1 \leq \gamma_2 \overset{def}{\iff} \forall \alpha \in \text{dom}(\gamma_1).\, \gamma_1(\alpha) \leq \gamma_2(\alpha)$$

Each region in a region assignment is associated with an abstract state: the abstract state of the region.

Definition 27 (Region States)**.** *Let* $r \in \text{RASS}$ *be a region assignment. A* region state

$$\beta \in \text{RSTATE}_r \triangleq \text{dom}(r) \to \text{ASTATE}$$

is a mapping from the regions declared in r *to abstract states. For* $\beta_1 \in \text{RSTATE}_{r_1}$ *and* $\beta_2 \in \text{RSTATE}_{r_2}$*, with* $r_1 \leq r_2$*, region states are ordered extensionally:*

$$\beta_1 \leq \beta_2 \overset{def}{\iff} \forall \alpha \in \text{dom}(\beta_1).\, \beta_1(\alpha) = \beta_2(\alpha)$$

Hitherto, we have given the semantic definitions required for regions. Now we proceed to develop the semantics definitions for the instrumented states that constitute the models of the assertion language of definition 17. We call these instrumented states *worlds*.

Definition 28 (Worlds)**.** *A* world

$$w \in \textsc{World} \triangleq \biguplus_{r \in \textsc{RAss}} (\{r\} \times \textsc{Heap} \times \textsc{APBag} \times \textsc{GAssn}_r \times \textsc{RState}_r)$$

consists of a region assignment, a heap, an abstract predicate bag, a guard assignment and a region state.

Worlds are composed, provided they agree on the region assignment and region state, by composing the heap, abstract predicate bag and guard assignment components in their respective separation algebras. Worlds form a multi-unit separation algebra $(\textsc{World}, \circ, \mathsf{emp})$*, where*

$$(r, h_1, b_1, \gamma_1, \beta) \circ (r, h_2, b_2, \gamma_2, \beta) \triangleq (r, h_1 \uplus h_2, b_1 \cup b_2, \gamma_1 \bullet \gamma_2, \beta) \qquad \mathsf{emp} \triangleq \left\{(r, \emptyset, \emptyset, \lambda\alpha.\, \mathbf{0}_{\mathbf{T}(r(\alpha)\downarrow_2)\downarrow_1}, \beta)\right\}$$

Worlds are ordered by product order:

$$(r_1, h_1, b_1, \gamma_1, \beta_1) \leq (r_2, h_2, b_2, \gamma_2, \beta_2) \overset{def}{\iff} r_1 \leq r_2 \wedge h_1 \leq h_2 \wedge b_1 \leq b_2 \wedge \gamma_1 \leq \gamma_2 \wedge \beta_1 \leq \beta_2$$

Thus, if $w_1 \leq w_2$*, we can get* w_2 *from* w_1 *by adding new regions, with arbitrary associated type name and state, and adding new heap, abstract predicates and guards.*

Another part of the instrumentation is the atomic tracking resource associated with a region. This can be in one of three states denoting the ability or inability to perform an atomic update and if the atomic update has or has not yet taken effect. These three states form a separation algebra.

Definition 29 (Atomic Tracking Component Separation Algebra)**.** *Let* $\blacklozenge$ *denote the right to perform an update to the abstract state of a region. Let* $\lozenge$ *denote the absence of a right to perform an update to the abstract state of a region. Let* $(x, y) \in \textsc{AState} \times \textsc{AState}$ *denote the update performed on the abstract state of a regions, from state* x *to state* y*. The* atomic tracking component separation algebra *is defined as:*

$$((\textsc{AState} \times \textsc{AState}) \uplus \{\blacklozenge, \lozenge\}, \centerdot, (\textsc{AState} \times \textsc{AState}) \uplus \{\lozenge\})$$

where $\centerdot$ *is defined by:*

$$\blacklozenge \centerdot \lozenge = \blacklozenge = \lozenge \centerdot \blacklozenge \qquad \lozenge \centerdot \lozenge = \lozenge \qquad (x, y) \centerdot (x, y) = (x, y)$$

and undefined otherwise. Resource ordering on atomic tracking components is defined by the following two rules:

$$\forall k \in (\textsc{AState} \times \textsc{AState}) \uplus \{\blacklozenge, \lozenge\}\,.\, k \leq k \qquad \lozenge \leq \blacklozenge$$

We now extend the worlds of definition 28 with an environment mapping region identifiers to atomic tracking components.

Definition 30 (Worlds with Atomic Tracking)**.** *Let $\mathcal{R} \subseteq_{fin} \text{RID}$, be a finite set of region identifiers. A* world with atomic tracking*, $w \in \text{AWORLD}_{\mathcal{R}} \triangleq \text{WORLD} \times (\mathcal{R} \to (\text{ASTATE} \times \text{ASTATE}) \uplus \{\blacklozenge, \lozenge\})$, consists of a world with a mapping from regions in $\mathcal{R}$ to atomic tracking resources.*

Let $d_1, d_2 \in \mathcal{R} \to (\text{ASTATE} \times \text{ASTATE}) \uplus \{\blacklozenge, \lozenge\}$. The composition of atomic tracking components is lifted to maps:

$$d_1 \centerdot d_2 \triangleq \lambda\alpha.\, d_1(\alpha) \centerdot d_2(\alpha)$$

Maps of atomic tracking components form the following separation algebra:

$$((\mathcal{R} \to (\text{ASTATE} \times \text{ASTATE}) \uplus \{\blacklozenge, \lozenge\}), \centerdot, \emptyset)$$

They are ordered by extension ordering: $d_1 \leq d_2 \overset{def}{\iff} \forall\alpha \in \text{dom}(d_1).\, d_2(\alpha) = d_1(\alpha)$. Consequently, worlds with atomic tracking components form a separation algebra, $(\text{AWORLD}_{\mathcal{R}}, \circ, \mathsf{emp}_{\mathcal{R}})$, where:

$$(w_1, d_1) \circ (w_2, d_2) \triangleq (w_1 \circ w_2, d_1 \centerdot d_2) \qquad \mathsf{emp}_{\mathcal{R}} = (\mathsf{emp}, \emptyset)$$

Worlds with atomic tracking are ordered by product order:

$$(w_1, d_1) \leq (w_2, d_2) \overset{def}{\iff} w_1 \leq w_2 \wedge d_1 \leq d_2$$

We consider that $\text{WORLD} = \text{AWORLD}_{\emptyset}$, and generally use the term world to refer to a world with (possibly empty) atomic tracking, unless explicitly stated otherwise. Let $w \in \text{AWORLD}_{\mathcal{R}}$ be a world. We denote its region assignment component with r_w, its heap component with h_w, its abstract predicates component with b_w, its guard assignment component with γ_w, its region states component with β_w, and its atomicity tracking components as d_w.

Definition 31 (World Predicates)**.** *Let $\mathcal{R} \subseteq_{fin} \text{RID}$, be a finite set of region identifiers. A* world predicate*, $p, q \in \text{WPRED}_{\mathcal{R}} \triangleq \mathcal{P}^{\uparrow}(\text{AWORLD}_{\mathcal{R}})$, is a set of worlds that is upwards closed with respect to the world ordering: $\forall w \in p.\, \exists w'.\, w \leq w' \Rightarrow w' \in p$. Composition of world predicates is obtained by lifting the composition of worlds[1]:*

$$p * q \triangleq \{w \mid \exists w' \in p, w'' \in q.\, w = w' \circ w''\}$$

World predicates form a separation algebra, $(\text{WPRED}, *, \text{WORLD})$.

Definition 32 (Atomicity Context)**.** *An* atomicity context*, $\mathcal{A} \in \text{ACONTEXT} \triangleq \text{RID} \overset{fin}{\rightharpoonup} \text{ASTATE} \rightharpoonup \mathcal{P}(\text{ASTATE})$, is a finite partial mapping from region identifiers to partial, non-deterministic abstract state transformers.*

The atomicity context records the abstract atomic update that is to be performed on the shared region and is used while proving a refinement of an atomic specification statement. This has implications on both how a thread can perform the update (guarantee) and what the environment is allowed to do on the same region (rely). Specifically, the environment is allowed to update a region for which it owns a guard, in any way allowed by the transition system for that guard. The guard owned by

[1] The result of the composition is upwards closed: any extension to the composition of two worlds can be tracked back and applied to one of the components.

the environment has to be compatible with the guard owned by the thread. If an atomic update is pending in the atomicity context, then the environment is only allowed to update the region to those states stated in the atomicity context. Environment interference is abstracted by the rely relation.

Definition 33 (Rely Relation)**.** *Let $\mathcal{A} \in$ AContext be an atomic context, with $\mathcal{R} = \mathrm{dom}(\mathcal{A})$, the rely relation, $R_{\mathcal{A}} \subseteq$ AWorld$_{\mathcal{R}} \times$ AWorld$_{\mathcal{R}}$, is the smallest reflexive and transitive relation that satisfies the following rules:*

$$\frac{g \# g' \qquad (s, s') \in \mathcal{T}_{\mathbf{t}}(g')^* \qquad \big(d(\alpha) \in \{\blacklozenge, \lozenge\} \Rightarrow s' \in \mathrm{dom}(\mathcal{A}(\alpha))\big)}{(r[\alpha \mapsto (k, \mathbf{t}, \vec{v})], h, b, \gamma[\alpha \mapsto g], \beta[\alpha \mapsto s], d) R_{\mathcal{A}} (r[\alpha \mapsto (k, \mathbf{t}, \vec{v})], h, b, \gamma[\alpha \mapsto g], \beta[\alpha \mapsto s'], d)}$$

$$\frac{(s, s') \in \mathcal{A}(\alpha)}{(r[\alpha \mapsto (k, \mathbf{t}, v)], h, b, \gamma, \beta[\alpha \mapsto s], d[\alpha \mapsto \lozenge]) R_{\mathcal{A}} (r[\alpha \mapsto (k, \mathbf{t}, v)], h, b, \gamma, \beta[\alpha \mapsto s'], d[\alpha \mapsto (s, s')])}$$

Consider the first rule. It states that the environment can update a region, if it owns a guard g' for which the update is allowed and as long as that guard is compatible with the thread's own guard g. If there is a pending atomic update for the region in the atomicity tracking component, then the environment is restricted to update the region to a state within one of those specified in the atomicity context. Now consider the second rule. It states that if the thread owns $\blacklozenge$ (the thread has not yet performed an update), the environment can update the state in accordance to the atomicity context.

Interference is explicitly confined to shared regions and atomicity tracking resources. In addition, extending the atomicity context restricts the interference.

Definition 34 (Guarantee Relation)**.** *Let $\mathcal{R} \subseteq$ RId, be a set of region identifiers. Let $k \in$ RLevel be a region level. Let $\mathcal{A} \in$ AContext be an atomicity context. The* guarantee relation*, $G_{k;\mathcal{A}} \subseteq$ AWorld$_{\mathcal{R}} \times$ AWorld$_{\mathcal{R}}$, is defined as:*

$$w \; G_{k;\mathcal{A}} \; w' \overset{def}{\iff}$$
$$\forall \alpha.\, (\exists k' \geq k.\, r_w(\alpha) = (k', -, -)) \Rightarrow \beta_w(\alpha) = \beta_{w'}(\alpha)$$
$$\wedge\, \forall \alpha \in \mathrm{dom}(\mathcal{A}).\, \left(\begin{array}{l} (d_w(\alpha) = d_{w'}(\alpha) \wedge \beta_w(\alpha) = \beta_{w'}(\alpha)) \\ \vee\, (d_w(\alpha) = \blacklozenge \wedge d_{w'}(\alpha) = (\beta_w(\alpha), \beta_{w'}(\alpha)) \wedge (\beta_w(\alpha), \beta_{w'}(\alpha)) \in \mathcal{A}(\alpha)) \end{array} \right)$$

The guarantee relation enforces that regions with level k or higher cannot be modified. It also enforces that regions mentioned in the atomicity context can only be updated according to what is specified in the atomicity context.

Definition 35 (Stable World Predicates and Views)**.** *Let $\mathcal{A} \in$ AContext, be an atomicity context. A* stable world predicate *is a world predicate that is closed under the rely relation.*

$$\mathcal{A} \vdash p \;\mathsf{stable} \overset{def}{\iff} R_{\mathcal{A}}(p) \subseteq p$$

Stable world predicates are referred to as views*. The set of views, with atomicity context $\mathcal{A}$, are denoted by* View$_{\mathcal{A}}$*.*

$$\textsc{View}_{\mathcal{A}} \triangleq \big\{ p \in \textsc{WPred}_{\mathrm{dom}(\mathcal{A})} \mid R_{\mathcal{A}}(p) \subseteq p \big\}$$

View *is shorthand for* View$_{\emptyset}$*. If $\mathcal{A}'$ is an extension of $\mathcal{A}$, there is a coercion from* View$_{\mathcal{A}}$ *to* View$_{\mathcal{A}'}$*, by extending the atomicity tracking component for the additional regions in every possible way. That*

is, if $p \in \textsc{View}_{\mathcal{A}}$ *and* $q \in \textsc{View}_{\mathcal{A}'}$*, with* $\mathcal{A} \leq \mathcal{A}'$*, then*

$$p \leq q \overset{def}{\iff} \forall w \in p, w' \in q.\, w \leq w'$$

The least upper bound of a set of views with atomicity context $\mathcal{A}$ *is denoted by* $\sqcup\textsc{View}_{\mathcal{A}}$*.*

Lemma 1. *Stable world predicates are closed under* $*$*,* $\cup$ *and* $\cap$*:*

$$\mathcal{A} \vdash p \text{ stable} \wedge \mathcal{A} \vdash q \text{ stable} \Rightarrow \mathcal{A} \vdash p * q \text{ stable}$$
$$\mathcal{A} \vdash p \text{ stable} \wedge \mathcal{A} \vdash q \text{ stable} \Rightarrow \mathcal{A} \vdash p \cup q \text{ stable}$$
$$\mathcal{A} \vdash p \text{ stable} \wedge \mathcal{A} \vdash q \text{ stable} \Rightarrow \mathcal{A} \vdash p \cap q \text{ stable}$$

In chapter 6, we have given examples of how regions are interpreted into the shared state they encapsulated and how abstract predicates are interpreted to their implementation through respective interpretation functions. We now formally define these interpretation functions.

Definition 36 (Region Interpretation)**.** *A* region interpretation

$$I_r \in \textsc{RInterp} \triangleq (\textsc{RLevel} \times \textsc{RTName} \times \textsc{Val}^*) \times \textsc{RId} \times \textsc{AState} \to \textsc{Assrt}$$

associates an assertion with each abstract state of each parameterised region.

Definition 37 (Abstract Predicate Interpretation)**.** *An* abstract predicate interpretation

$$I_a \in \textsc{APInterp} \triangleq \textsc{APName} \times \textsc{Val}^* \to \textsc{Assrt}$$

associates an assertion with each abstract predicate.

We give a denotational semantics to the assertions of definition 17 in terms of world predicates within a given atomicity context. We require assertions to be stable and thus the denotations of assertions are required to be views. To ensure the stability of the denotations we use the following auxiliary predicate:

$$\mathsf{stab}(\mathcal{A}, p) \triangleq \begin{cases} p & \text{if } p \in \textsc{View}_{\mathcal{A}} \\ \emptyset & \text{otherwise} \end{cases}$$

Let $\mathcal{A} \in \textsc{AContext}$ be an atomicity context. We extend the basic values of definition 14 with views and extend the variable stores analogously. Furthermore, to define the denotation of recursive predicates, we extend the variable stores so that predicate-recursion variables are mapped to functions from values to views.

$$\textsc{Val}_{\mathcal{A}} \triangleq \textsc{Val} \cup \textsc{View}_{\mathcal{A}} \qquad \textsc{VarStore}_{\mathcal{A}} \triangleq \textsc{Var} \to \textsc{Val}_{\mathcal{A}} \uplus (\textsc{AssrtRecVars} \to (\textsc{Val} \to \textsc{View}_{\mathcal{A}}))$$

Definition 38 (Assertion Interpretation)**.** *Let* $\mathcal{A} \in \textsc{AContext}$ *be a given atomicity context. The assertion interpretation function,* $(\!|-|\!)^{-}_{\mathcal{A}} : \textsc{Assrt} \to \textsc{VarStore}_{\mathcal{A}} \to \textsc{View}_{\mathcal{A}} \cup (\textsc{Val} \to \textsc{View}_{\mathcal{A}})$*, maps*

assertions to views, or functions from values to views, within a variable store.

$$
\begin{aligned}
(\!|\mathsf{false}|\!)^{\rho}_{\mathcal{A}} &\triangleq \emptyset \\
(\!|\mathsf{true}|\!)^{\rho}_{\mathcal{A}} &\triangleq \sqcup \textsc{View}_{\mathcal{A}} \\
(\!|P * Q|\!)^{\rho}_{\mathcal{A}} &\triangleq (\!|P|\!)^{\rho}_{\mathcal{A}} * (\!|Q|\!)^{\rho}_{\mathcal{A}} \\
(\!|P \wedge Q|\!)^{\rho}_{\mathcal{A}} &\triangleq (\!|P|\!)^{\rho}_{\mathcal{A}} \cap (\!|Q|\!)^{\rho}_{\mathcal{A}} \\
(\!|P \vee Q|\!)^{\rho}_{\mathcal{A}} &\triangleq (\!|P|\!)^{\rho}_{\mathcal{A}} \cup (\!|Q|\!)^{\rho}_{\mathcal{A}} \\
(\!|\neg P|\!)^{\rho}_{\mathcal{A}} &\triangleq (\sqcup \textsc{View}_{\mathcal{A}}) \setminus (\!|P|\!)^{\rho}_{\mathcal{A}} \\
(\!|\exists x.\, P|\!)^{\rho}_{\mathcal{A}} &\triangleq \bigcup_{v \in \textsc{Val}} (\!|P|\!)^{\rho[x \mapsto v]}_{\mathcal{A}} \\
(\!|\forall x.\, P|\!)^{\rho}_{\mathcal{A}} &\triangleq \bigcap_{v \in \textsc{Val}} (\!|P|\!)^{\rho[x \mapsto v]}_{\mathcal{A}} \\
(\!|P \Rightarrow Q|\!)^{\rho}_{\mathcal{A}} &\triangleq \left((\sqcup \textsc{View}_{\mathcal{A}}) \setminus (\!|P|\!)^{\rho}_{\mathcal{A}}\right) \cup (\!|Q|\!)^{\rho}_{\mathcal{A}} \\
(\!|e \mapsto e'|\!)^{\rho}_{\mathcal{A}} &\triangleq \mathsf{stab}\left(\mathcal{A}, \left\{ w \in \textsc{AWorld}_{\mathrm{dom}(\mathcal{A})} \;\middle|\; h_w([\![e]\!]^{\rho}) = [\![e']\!]^{\rho} \right\}\right) \\
(\!|\mathbf{t}^k_{\alpha}(\vec{e}, e')|\!)^{\rho}_{\mathcal{A}} &\triangleq \mathsf{stab}\left(\mathcal{A}, \left\{ w \in \textsc{AWorld}_{\mathrm{dom}(\mathcal{A})} \;\middle|\; r_w(\alpha) = \left(k, \mathbf{t}, \overrightarrow{[\![e]\!]^{\rho}}\right) \wedge \beta_w(\alpha) = [\![e']\!]^{\rho} \right\}\right) \\
(\!|I_r(\mathbf{t}^k_{\alpha}(\vec{e}, e'))|\!)^{\rho}_{\mathcal{A}} &\triangleq (\!|I_r((k, \mathbf{t}, \overrightarrow{[\![e]\!]^{\rho}}), \alpha, [\![e']\!]^{\rho})|\!)^{\rho}_{\mathcal{A}} \\
(\!|[\mathsf{G}(\vec{e})]_{\alpha}|\!)^{\rho}_{\mathcal{A}} &\triangleq \mathsf{stab}\left(\mathcal{A}, \left\{ w \in \textsc{AWorld}_{\mathrm{dom}(\mathcal{A})} \;\middle|\; \mathsf{G}\left(\overrightarrow{[\![e]\!]^{\rho}}\right) \leq \gamma_w(\alpha) \right\}\right) \\
(\!|\alpha \Mapsto \blacklozenge|\!)^{\rho}_{\mathcal{A}} &\triangleq \mathsf{stab}\left(\mathcal{A}, \left\{ w \in \textsc{AWorld}_{\mathrm{dom}(\mathcal{A})} \;\middle|\; d_w(\alpha) = \blacklozenge \right\}\right) \\
(\!|\alpha \Mapsto \lozenge|\!)^{\rho}_{\mathcal{A}} &\triangleq \mathsf{stab}\left(\mathcal{A}, \left\{ w \in \textsc{AWorld}_{\mathrm{dom}(\mathcal{A})} \;\middle|\; d_w(\alpha) = \lozenge \vee d_w(\alpha) = \blacklozenge \right\}\right) \\
(\!|\alpha \Mapsto (e, e')|\!)^{\rho}_{\mathcal{A}} &\triangleq \mathsf{stab}\left(\mathcal{A}, \left\{ w \in \textsc{AWorld}_{\mathrm{dom}(\mathcal{A})} \;\middle|\; d_w(\alpha) = ([\![e]\!]^{\rho}, [\![e']\!]^{\rho}) \wedge d_w(\alpha) \in \mathcal{A}(\alpha) \right\}\right) \\
(\!|\mathsf{ap}(\vec{e})|\!)^{\rho}_{\mathcal{A}} &\triangleq \mathsf{stab}\left(\mathcal{A}, \left\{ w \in \textsc{AWorld}_{\mathrm{dom}(\mathcal{A})} \;\middle|\; (\mathsf{ap}, \overrightarrow{[\![e]\!]^{\rho}}) \in b_w \right\}\right) \\
(\!|I_a(\mathsf{ap}(\vec{e}))|\!)^{\rho}_{\mathcal{A}} &\triangleq (\!|I_a(\mathsf{ap}, \overrightarrow{[\![e]\!]^{\rho}})|\!)^{\rho}_{\mathcal{A}} \\
(\!|\lambda x.\, P|\!)^{\rho}_{\mathcal{A}} &\triangleq \lambda v.\, (\!|P|\!)^{\rho[x \mapsto v]}_{\mathcal{A}} \\
(\!|\mu X.\, \lambda x.\, P|\!)^{\rho}_{\mathcal{A}} &\triangleq \bigcap \left\{ w_f \in \textsc{Val} \to \textsc{View}_{\mathcal{A}} \;\middle|\; (\!|\lambda x.\, P|\!)^{\rho[X \mapsto w_f]}_{\mathcal{A}} \leq w_f \right\} \\
(\!|X|\!)^{\rho}_{\mathcal{A}} &\triangleq \rho(X) \\
(\!|\mathsf{pred}(e)|\!)^{\rho}_{\mathcal{A}} &\triangleq \mathsf{stab}\left(\mathcal{A}, (\!|\mathsf{pred}|\!)^{\rho}_{\mathcal{A}}([\![e]\!]^{\rho})\right) \\
(\!|e|\!)^{\rho}_{\mathcal{A}} &\triangleq \begin{cases} \sqcup \textsc{View}_{\mathcal{A}} & \textit{if } [\![e]\!]^{\rho} = \mathsf{true} \\ [\![e]\!]^{\rho} & \textit{if } [\![e]\!]^{\rho} \in \textsc{View}_{\mathcal{A}} \\ \emptyset & \textit{otherwise} \end{cases}
\end{aligned}
$$

Note that on their own, concrete predicates are interpreted as functions from values to views.

Functions from values to views, $\textsc{Val} \to \textsc{View}_{\mathcal{A}}$, are ordered by pointwise extension of the ordering on $\textsc{View}_{\mathcal{A}}$. Together with the following lemma, this guarantees the existence of the least fixed point for recursive predicates.

Lemma 2. *For all assertions P and recursion variables X, the function, $(\!|P|\!)^{\rho[X \mapsto -]}_{\mathcal{A}} : (\textsc{Val} \to \textsc{View}_{\mathcal{A}}) \to \textsc{View}_{\mathcal{A}}$, is monotonic.*

Proof. By straightforward induction over P. □

Assertions are interpreted as views. These include all the instrumentation in terms of regions, guards, atomicity tracking components and abstract predicates. We now proceed to define the means by which all of the aforementioned instrumentation is reified to concrete heaps.

Definition 39 (Region Collapse)**.** *Let $\mathcal{R} \subseteq \text{RID}$. Let $I_r \in \text{RINTERP}$ be a given region interpretation. Given a region level $k \in \text{RLEVEL}$, and atomicity context, $\mathcal{A} \in \text{ACONTEXT}$, the* region collapse *of a world $w \in \text{AWORLD}_{\mathcal{R}}$, is a set of worlds given by:*

$$w{\downarrow_{k;\mathcal{A}}} \triangleq \left\{ w \circ (w', \emptyset) \;\middle|\; w' \in \circledast_{\{\alpha \mid \exists k' < k. r_w(\alpha) = (k', -, -)\}} (\!| I_r(r_w(\alpha), \alpha, \beta_w(\alpha)) |\!)^{\emptyset}_{\mathcal{A}} \right\}$$

Region collapse is lifted to world predicates as expected: $p{\downarrow_{k;\mathcal{A}}} \triangleq \bigcup_{w \in p} w{\downarrow_{k;\mathcal{A}}}$.

Definition 40 (Abstract Predicate Collapse)**.** *The one-step* abstract predicate collapse *of a world is a set of given worlds given by:*

$$(r, h, b, \gamma, \beta, d){\lfloor_{1;\mathcal{A}}} \triangleq \left\{ (r, h, \emptyset, \gamma, \beta, d) \circ (w, \emptyset) \;\middle|\; w \in \circledast_{ap \in b} (\!| I_a(ap) |\!)^{\emptyset}_{\mathcal{A}} \right\}$$

This is lifted to world predicates as expected: $p{\lfloor_{1;\mathcal{A}}} \triangleq \bigcup_{w \in p} w{\lfloor_{1;\mathcal{A}}}$. The one-step collapse gives rise to multi-step collapse: $p{\lfloor_{n+1;\mathcal{A}}} \triangleq (p{\lfloor_{n;\mathcal{A}}}){\lfloor_{1;\mathcal{A}}}$. The abstract predicate collapse of a predicate (view), applies the multi-step collapse until all abstract predicates are collapsed:

$$p{\lfloor_{\mathcal{A}}} \triangleq \{ w \mid \exists n.\, w \in p{\lfloor_{n;\mathcal{A}}} \wedge b_w = \emptyset \}$$

The above approach to interpreting abstract predicates effectively gives a step-indexed interpretation to the predicates. The concrete interpretation of a predicate is given by the finite unfoldings of the abstract predicate collapse. If a predicate cannot be made fully concrete by finite unfoldings, then it's interpreted as false.

Definition 41 (Reification)**.** *The* reification *operation on worlds collapses the regions and the abstract predicates, and then only retains the heap component:*

$$[\![w]\!]_{k;\mathcal{A}} \triangleq \left\{ h_{w'} \mid w' \in w{\downarrow_{k;\mathcal{A}}}{\lfloor_{\mathcal{A}}} \right\}$$

The operation is lifted to world predicates as expected: $[\![p]\!]_{k;\mathcal{A}} \triangleq \bigcup_{w \in p} [\![w]\!]_{k;\mathcal{A}}$.

7.3. Operational Semantics

The semantics of primitive atomic statements are defined via a state transformer on concrete states: the heaps of definition 19.

Definition 42 (Atomic Action State Transformer)**.** *Given an atomicity context, $\mathcal{A} \in \text{ACONTEXT}$, and region level, $k \in \text{RLEVEL}$, the* atomic action state transformer*, $\mathsf{a}(-,-)^{\mathcal{A}}_{k}(-) : \text{VIEW}_{\mathcal{A}} \times \text{VIEW}_{\mathcal{A}} \to \text{HEAP} \to \mathcal{P}(\text{HEAP})$, associates precondition and postcondition views to a non-deterministic state trans-*

former:

$$\mathsf{a}(p,q)^{\mathcal{A}}_{k}\,(h) \triangleq$$
$$\left\{ h' \in \textsc{Heap} \;\middle|\; \begin{array}{l} \forall r \in \textsc{View}_{\mathcal{A}}.\, \forall w \in p * r. \\ h \in [\![w]\!]_{k;\mathcal{A}} \wedge \exists w'.\, w\; G_{k;\mathcal{A}}\; w' \wedge h' \in [\![w']\!]_{k;\mathcal{A}} \wedge w' \in q * r \end{array} \right\}$$

Given a starting heap, $h \in \textsc{Heap}$, which is contained in the reification of p composed with all possible frames, $\mathsf{a}(p,q)^{\mathcal{A}}_{k}\,(h)$ returns the set of heaps that result from the reification of q composed with all possible frames, as long as the result is within the guarantee relation.

The use of a state transformer for the semantics of atomic actions is similar to the semantics of atomic actions in the Views framework [35]. The definition of the atomic action state transformer we have given here, corresponds to the definition of the primitive atomic satisfaction judgement in the semantics of TaDA [31], that defines the semantics of physically atomic actions.

View-shifts [35] are relations between assertions that reify to the same concrete states but may use different instrumentation. In other words, view-shifts allow us to change the view of the underlying state. Examples of view-shifts include the allocation/deallocation of shared regions and the opening/closing of abstract predicates.

Definition 43 (View Shift)**.** *View shifts are defined as follows:*

$$k; \mathcal{A} \vdash P \preccurlyeq Q \overset{def}{\iff}$$
$$\forall \rho.\, \forall r \in \textsc{View}_{\mathcal{A}}.\, \forall w \in (\!|P|\!)^{\rho}_{\mathcal{A}} * r.\, \forall h \in [\![w]\!]_{k;\mathcal{A}}.\, \exists w' \in w\; G_{k;\mathcal{A}}\; w' \wedge h \in [\![w']\!]_{k;\mathcal{A}} \wedge w' \in (\!|Q|\!)^{\rho}_{\mathcal{A}} * r$$

Lemma 3 (Implications are View Shifts)**.**

$$\frac{\mathcal{A} \vdash P \Rightarrow Q}{\mathcal{A}; k \vdash P \preccurlyeq Q}$$

We give a largely standard operational semantics for the specification language of definition 18.

Definition 44 (Operational Semantics)**.** *Let* $\lightning$ *denote fault. Let outcomes be* $o \in \textsc{Heap}^{\lightning} \triangleq \textsc{Heap} \uplus \{\lightning\}$*. Let configurations be* $\kappa \in \textsc{Configs} \triangleq \left((\mathcal{L} \times \textsc{Heap}) \cup \textsc{Heap}^{\lightning}\right)$*. The single-step operational transition relation,* $\rightsquigarrow \subseteq (\mathcal{L} \times \textsc{Heap}) \times \textsc{Configs}$*, is the smallest relation satisfying the rules:*

(7.1)
$$\frac{\phi, h \rightsquigarrow \phi', h'}{\phi; \psi, h \rightsquigarrow \phi'; \psi, h'}$$

(7.2)
$$\frac{\phi, h \rightsquigarrow h'}{\phi; \psi, h \rightsquigarrow \psi, h'}$$

(7.3)
$$\frac{\phi, h \rightsquigarrow \lightning}{\phi; \psi, h \rightsquigarrow \lightning}$$

(7.4)
$$\frac{\phi \parallel \psi, h \rightsquigarrow \kappa}{\psi \parallel \phi, h \rightsquigarrow \kappa}$$

(7.5)
$$\frac{\phi, h \rightsquigarrow \phi', h'}{\phi \parallel \psi, h \rightsquigarrow \phi' \parallel \psi, h'}$$

(7.6)
$$\frac{\phi, h \rightsquigarrow h'}{\phi \parallel \psi, h \rightsquigarrow \psi, h'}$$

(7.7)
$$\frac{\phi, h \rightsquigarrow \lightning}{\phi \parallel \psi, h \rightsquigarrow \lightning}$$

(7.8)
$$\frac{\phi_i, h \rightsquigarrow \kappa \qquad i \in \{1,2\}}{\phi_1 \sqcup \phi_2, h \rightsquigarrow \kappa}$$

(7.9)
$$\frac{\forall i \in \{1,2\}\,.\, \phi_i, h \rightsquigarrow \kappa}{\phi_1 \sqcap \phi_2, h \rightsquigarrow \kappa}$$

(7.10)
$$\frac{v \in \textsc{Val} \qquad \phi\,[v/x]\,, h \rightsquigarrow \kappa}{\exists x.\, \phi, h \rightsquigarrow \kappa}$$

(7.11)
$$\frac{\phi\,[F/f]\,, h \rightsquigarrow \kappa}{\textbf{let}\ f = F\ \textbf{in}\ \phi, h \rightsquigarrow \kappa}$$

(7.12)
$$\frac{(\lambda x.\, \phi\,[\mu A.\, \lambda x.\, \phi/A])e, h \rightsquigarrow \kappa}{(\mu A.\, \lambda x.\, \phi)e, h \rightsquigarrow \kappa}$$

$$\frac{\phi\left[[\![e]\!]/x\right], h \rightsquigarrow \kappa}{(\lambda x.\, \phi)e, h \rightsquigarrow \kappa} \tag{7.13}$$

$$\frac{h' \in \left\{ \mathsf{a}\left((\!|P|\!)_{\mathcal{A}}^{[\vec{x}\mapsto\vec{v}]}, (\!|Q|\!)_{\mathcal{A}}^{[\vec{x}\mapsto\vec{v}]} \right)_k^{\mathcal{A}} (h) \;\middle|\; \vec{v} \in \overrightarrow{\text{VAL}} \right\}}{a(\forall \vec{x}.\, P,\, Q)_k^{\mathcal{A}}, h \rightsquigarrow h'} \tag{7.14}$$

$$\frac{\forall \vec{v} \in \overrightarrow{\text{VAL}}.\, \mathsf{a}\left((\!|P|\!)_{\mathcal{A}}^{[\vec{x}\mapsto\vec{v}]}, (\!|Q|\!)_{\mathcal{A}}^{[\vec{x}\mapsto\vec{v}]} \right)_k^{\mathcal{A}} (h) = \emptyset}{a(\forall \vec{x}.\, P,\, Q)_k^{\mathcal{A}}, h \rightsquigarrow \lightning} \tag{7.15}$$

where: $[\![e]\!]$ *denotes the denotation of the expression* e *in the empty variable store, i.e.* e *has no variables;* $\vec{v} \in \overrightarrow{\text{VAL}}$ *denotes a vector of values; and* $[\vec{x} \mapsto \vec{v}]$ *denotes a function mapping each variable in the vector* $\vec{x}$ *to a value in the vector* $\vec{v}$*. The multi-step operational transition relation,* $\rightsquigarrow^*$*, is defined as the reflexive, transitive closure of* $\rightsquigarrow$*.*

The operational semantics is defined on closed specification programs. A specification program is closed when it has no free variables. This is largely for simplicity; variables are immutable. The operational semantics of a specification program with free variables can be defined with respect to all closing contexts. In section 7.4.2 we additionally define a denotational semantics for specification programs with respect to a variable store.

7.4. Refinement

In the previous sections, we have defined the syntax and semantics of specification programs with physical atomicity. As highlighted in chapter 6, we use specification programs to reason about concurrent behaviour by "comparing" them. We say that a relatively concrete specification program ϕ, *implements* a more abstract specification program ψ, when every behaviour of ϕ is also a behaviour of ψ. Then, any client or context interacting with ψ can also interact with ϕ in the same way, without observing different behaviour. Formally, this is expressed as *contextual refinement*, which we define in section 7.4.1.

Reasoning about contextual refinement involves reasoning about all possible contexts, which hinders our ability to derive useful refinement laws to include in a refinement calculus for atomicity. To overcome this difficulty, we additionally define a denotational trace semantics for specification programs, giving rise to a denotational version of refinement in section 7.4.2, which we prove sound with respect to contextual refinement in section 7.4.3. Then, by the compositional nature of denotational semantics, we are able to justify a large selection of refinement laws in section 7.5.

7.4.1. Contextual Refinement

We consider standard single-holed contexts of specifications. We denote a (single-holed) specification context by C and context application by $C[\phi]$.

Definition 45 (Contextual Refinement)**.** *Let* $h \in \text{HEAP}$*.*

$$\phi \sqsubseteq_{\text{op}} \psi \iff \forall C, h, h'. \begin{cases} C[\phi], h \rightsquigarrow^* \lightning \Rightarrow C[\psi], h \rightsquigarrow^* \lightning \\ C[\phi], h \rightsquigarrow^* h' \Rightarrow C[\psi], h \rightsquigarrow^* h' \vee\ C[\psi], h \rightsquigarrow^* \lightning \end{cases}$$

Contextual refinement is given a *partial correctness* interpretation. If ϕ terminates by faulting, then ψ must do the same. On the other hand, if ϕ terminates successfully, then ψ must either successfully terminate with the same result, or fault. Faults are treated as *unspecified behaviour*. They are the most permissible of specifications; everything is a valid refinement of unspecified behaviour. Finally, if ϕ does not terminate, it is still a refinement of ψ. Hence, $\phi \sqsubseteq_{\text{op}} \psi$ does not guarantee termination of either ϕ or ψ.

7.4.2. Denotational Refinement

Following the approach of Turon and Wand [94], the denotational model for specification programs is based on Brookes's transition trace model [21], adjusted to account for heaps and faults. A *transition trace* is finite sequence of pairs of heaps, (h, h'), called *moves*, possibly terminated by a fault, either due to the specification program faulting on its own accord, $(h, \lightning)$, or due to interference from the environment causing the specification program to fault, $(\lightning, \lightning)$.

Definition 46 (Transition Traces)**.** *Single successful transitions (moves) in a trace are:* $\textsc{Move} \triangleq \textsc{Heap} \times \textsc{Heap}$. *Faulty transitions in a trace are:* $\textsc{Fault} \triangleq \textsc{Heap}^{\lightning} \times \{\lightning\}$. *Transition traces are defined by the regular language:* $\textsc{Trace} \triangleq \textsc{Move}^*; \textsc{Fault}^?$. *The empty trace is denoted by* ϵ.

We use $s, t, u \in \textsc{Trace}$ to range over traces and $S, T, U \subseteq \textsc{Trace}$ to range over sets of traces. Note that sets of traces form a lattice: the powerset lattice. Due to the existence of faults, we extend concatenation of transition traces such that an early fault causes termination.

Definition 47 (Trace Concatenation)**.** *Let* $s, t \in \textsc{Trace}$. *Concatenation between traces is defined such that a fault on the left discards the trace on the right:*

$$st \triangleq \begin{cases} s & \text{if } \exists u \in \textsc{Trace}.\, s = u(h, \lightning) \vee s = u(\lightning, \lightning) \\ st & \text{otherwise} \end{cases}$$

Trace concatenation is lifted pointwise to sets of traces: $S;T \triangleq \left\{ st \;\middle|\; s \in S \wedge t \in T \right\}$.

Each move in a trace corresponds to a *timeslice* of the execution of a specification program ϕ, where we observe a starting and an ending state from the operational semantics. Arbitrary interference is allowed between discrete timeslices of execution.

Definition 48 (Multi-Step Observed Traces)**.** *The multi-step observed traces relation,* $\mathcal{O}[\![-]\!] \subseteq \mathcal{L} \times \mathcal{P}(\textsc{Trace})$, *is the smallest relation that satisfies the following rules:*

$$\frac{}{(\lightning, \lightning) \in \mathcal{O}[\![\phi]\!]} \quad (7.16) \qquad \frac{\phi, h \rightsquigarrow^* o}{(h, o) \in \mathcal{O}[\![\phi]\!]} \quad (7.17) \qquad \frac{\phi, h \rightsquigarrow^* \psi, h' \qquad t \in \mathcal{O}[\![\psi]\!]}{(h, h')t \in \mathcal{O}[\![\phi]\!]} \quad (7.18)$$

For example, a trace $(h_1, h_1')(h_2, h_2')$ for ϕ comprises two moves. The first, (h_1, h_1'), is a timeslice arising from an execution $\phi, h_1 \rightsquigarrow^* \psi, h_1'$. The second, (h_2, h_2'), is a timeslice arising from an execution $\psi, h_2 \rightsquigarrow^* \psi', h_2'$. In between the two timeslices, the environment executed some other specification program thus changing h_1' to h_2.

The denotational semantics, defined shortly, provide an alternative mechanism to $\mathcal{O}[\![\phi]\!]$ that is compositional on the structure of ϕ. However, to define the denotations of parallel composition, $\phi \parallel \psi$, in terms of the traces of ϕ and ψ, we need to non-deterministically interleave sets of traces.

Definition 49 (Trace Interleaving)**.** *Let $s, t \in \textsc{Trace}$. The non-deterministic interleaving of s and t, denoted by $s \parallel t$, is the smallest set of traces that satisfies the following rules:*

$$\text{(7.19)}\ \frac{s \in t \parallel u}{s \in u \parallel t} \qquad \text{(7.20)}\ \frac{s \in t \parallel u}{(h,h')s \in (h,h')t \parallel u} \qquad \text{(7.21)}\ \frac{}{(h,h')u \in (h,h') \parallel u} \qquad \text{(7.22)}\ \frac{}{(h,\lightning) \in (h,\lightning) \parallel u}$$

The interleaving is lifted pointwise to sets of traces: $T \parallel U \triangleq \left\{ s \in t \parallel u \;\middle|\; t \in T \wedge u \in U \right\}$.

In the model of Brookes, the transition traces $\mathcal{O}[\![\phi]\!]$ of ϕ are closed under *stuttering* and *mumbling* [21]. Stuttering adds a move (h,h) to a trace, whereas mumbling merges two moves with a common midpoint. For example, $(h,h')(h',h'')$ is merged by mumbling to (h,h''). The stuttering and mumbling closures correspond to the reflexivity and transitivity of $\rightsquigarrow^*$ respectively.

Definition 50 (Trace Closure)**.** *The trace closure of a set of traces T, denoted by $T^\dagger$, is the smallest set of traces that satisfies the following rules:*

$$\text{(7.23)}\ \frac{t \in T}{t \in T^\dagger} \qquad \textsc{clStutter}\ \frac{st \in T^\dagger}{s(h,h)t \in T^\dagger} \qquad \textsc{clMumble}\ \frac{s(h,h')(h',o)t \in T^\dagger}{s(h,o)t \in T^\dagger} \qquad \text{(7.24)}\ \frac{}{(\lightning,\lightning) \in T^\dagger} \qquad \text{(7.25)}\ \frac{t(h,\lightning) \in T^\dagger}{t(h,h')u \in T^\dagger}$$

Let $f : \textsc{Val} \to \mathcal{P}(\textsc{Trace})$. Trace closure is pointwise extended to functions from values to sets of traces: $f^\dagger \triangleq \lambda v.\, f(v)^\dagger$.

The last two rules regarding faults were added to the closure of Brookes [21] by Turon and Wand [94]. Intuitively, rule (7.24) captures the fact that the environment of a specification program ϕ, may cause it to fault at any given time. The rule (7.25) captures the fact that faulting behaviour is permissive: a specification program that terminates with a fault after a trace t, can always be implemented by a specification program that continues after t.

The denotational semantics of specification programs are defined as sets of (closed) traces. We extend the variable stores used for assertions, so that function and recursion variables are mapped to functions from values to sets of traces.

$$\textsc{VarStore}_\mu \triangleq \left(\bigcup_{\mathcal{A} \in \textsc{AContext}} \textsc{VarStore}_{\mathcal{A}} \right) \uplus (\textsc{FuncVars} \uplus \textsc{RecVars} \to (\textsc{Val} \to \mathcal{P}(\textsc{Trace})))$$

Definition 51 (Denotational Semantics)**.** *The denotational semantics of specification programs are given by the function, $[\![-]\!]^- : \textsc{VarStore}_\mu \to \mathcal{L} \to \mathcal{P}(\textsc{Trace})$, mapping specification programs to sets*

of traces, within a variable environment.

$$
\begin{aligned}
[\![\phi; \psi]\!]^\rho &\triangleq ([\![\phi]\!]^\rho \,;\, [\![\psi]\!]^\rho)^\dagger \\
[\![\phi \parallel \psi]\!]^\rho &\triangleq ([\![\phi]\!]^\rho \parallel [\![\psi]\!]^\rho)^\dagger \\
[\![\phi \sqcup \psi]\!]^\rho &\triangleq ([\![\phi]\!]^\rho \cup [\![\psi]\!]^\rho)^\dagger \\
[\![\phi \sqcap \psi]\!]^\rho &\triangleq ([\![\phi]\!]^\rho \cap [\![\psi]\!]^\rho)^\dagger \\
[\![\exists x.\, \phi]\!]^\rho &\triangleq \left(\bigcup_{v \in \textsc{Val}} [\![\phi]\!]^{\rho[x \mapsto v]} \right)^\dagger \\
[\![\textbf{let } f = F \textbf{ in } \phi]\!]^\rho &\triangleq [\![\phi]\!]^{\rho[f \mapsto [\![F]\!]^\rho]} \\
[\![Fe]\!]^\rho &\triangleq [\![F]\!]^\rho\, [\![e]\!]^\rho \\
[\![f]\!]^\rho &\triangleq \rho(f)^\dagger \\
[\![A]\!]^\rho &\triangleq \rho(A)^\dagger \\
[\![\mu A.\, \lambda x.\, \phi]\!]^\rho &\triangleq \bigcap \left\{ T_f \in \textsc{Val} \to \mathcal{P}(\textsc{Trace}) \;\middle|\; T_f = {T_{f'}}^\dagger \wedge [\![\lambda x.\, \phi]\!]^{\rho[A \mapsto T_{f'}]} \subseteq {T_{f'}}^\dagger \right\} \\
[\![\lambda x.\, \phi]\!]^\rho &\triangleq \lambda v.\, [\![\phi]\!]^{\rho[x \mapsto v]} \\
\left[\!\!\left[a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_k \right]\!\!\right]^\rho &\triangleq \left(\begin{array}{l} \left\{ (h, h') \in \textsc{Move} \;\middle|\; \vec{v} \in \overrightarrow{\textsc{Val}} \wedge h' \in \mathsf{a}\left((\!|P|\!)^{\rho[\vec{x} \mapsto \vec{v}]}_{\mathcal{A}}, (\!|Q|\!)^{\rho[\vec{x} \mapsto \vec{v}]}_{\mathcal{A}} \right)^{\mathcal{A}}_k (h) \right\} \\ \cup \left\{ (h, \lightning) \in \textsc{Heap}^{\lightning} \;\middle|\; \begin{array}{l} \vec{v} \in \overrightarrow{\textsc{Val}} \wedge \mathsf{a}\left((\!|P|\!)^{\rho[\vec{x} \mapsto \vec{v}]}_{\mathcal{A}}, (\!|Q|\!)^{\rho[\vec{x} \mapsto \vec{v}]}_{\mathcal{A}} \right)^{\mathcal{A}}_k (h) = \emptyset \\ \quad \wedge\, (\!|Q|\!)^{\rho[\vec{x} \mapsto \vec{v}]}_{\mathcal{A}} \neq \emptyset \end{array} \right\} \end{array} \right)^\dagger
\end{aligned}
$$

The semantics are relatively straightforward. Sequential composition is concatenation and parallel composition is non-deterministic interleaving (of closed traces). Angelic and demonic choice are union and intersection respectively. These correspond to the join and meet of the lattice of trace sets (the powerset lattice) respectively. Existential quantification is the standard set union over all values.

For recursive functions we use the Tarskian least fixed point. Note that functions from values to trace sets, $\textsc{Val} \to \mathcal{P}(\textsc{Trace})$, are ordered by the pointwise extension of the ordering on $\mathcal{P}(\textsc{Trace})$. Furthermore, the $\bigcap$ and $\subseteq$ in the fixed-point definition correspond to the meet and partial order of the lattice arising from the pointwise extension of $\mathcal{P}(\textsc{Trace})$ to the function space $\textsc{Val} \to \mathcal{P}(\textsc{Trace})$. Together with the following monotonicity lemma, this guarantees the existence of the fixed point.

Lemma 4. *Let $x_f \in \textsc{FuncVars} \uplus \textsc{RecVars}$, $\phi \in \mathcal{L}$ and $\rho \in \textsc{VarStore}_\mu$. The function $[\![\phi]\!]^{\rho[x_f \mapsto -]} : (\textsc{Val} \to \mathcal{P}(\textsc{Trace})) \to \mathcal{P}(\textsc{Trace})$ is monotonic.*

Proof. By straightforward induction over ϕ. Base cases A, f trivial. Inductive cases follow directly from the induction hypothesis. □

The denotational semantics are defined in terms of sets of finite traces. A finite trace is always terminated either by the implementation itself, or by a fault caused by the environment. Infinite traces are discarded. Consider the denotations of the specification program $(\mu A.\, \lambda x.\, Ax)\,()$. By the least fixpoint and rule (7.24) of the trace closure (definition 50), the only finite trace for this specification program is $(\lightning, \lightning)$. The denotational semantics of infinite recursion are finite traces that terminate with a fault caused by the environment.

With the denotational semantics defined, we can give a denotational version of refinement based on the partial order of trace sets.

Definition 52 (Denotational Refinement)**.** $\phi \sqsubseteq_{\text{den}} \psi$ *iff, for all closing* ρ, $[\![\phi]\!]^\rho \subseteq [\![\psi]\!]^\rho$.

Throughout this dissertation, unless explicitly stated, we use denotational refinement, writing $\phi \sqsubseteq \psi$ to mean $\phi \sqsubseteq_{\text{den}} \psi$. When $\phi \sqsubseteq \psi$ and $\psi \sqsubseteq \phi$, the specifications ϕ and ψ are equivalent: $\phi \equiv \psi$.

7.4.3. Adequacy

We relate the denotational and operational versions of refinement in two steps. First, we establish the following lemma, showing that denotational semantics produce the same closed trace sets as the operational semantics. The proof details are given appendix B.1.

Lemma 5. *If* ϕ *is closed, then* $[\![\phi]\!]^\emptyset = (\mathcal{O}[\![\phi]\!])^\dagger$.

Proof. From corollary 5 established in appendix B.1. □

Second, with the following theorem we establish that the denotational refinement is a sound approximation of contextual refinement. We are not interested in establishing completeness; all the refinement laws in the subsequent sections are justified by the denotational semantics.

Theorem 1 (Adequacy)**.** *If* $\phi \sqsubseteq_{\text{den}} \psi$*, then* $\phi \sqsubseteq_{\text{op}} \psi$.

Proof. Let $\phi \sqsubseteq_{\text{den}} \psi$. Let C be a context that closes both ϕ and ψ. We write $C[\phi]$ and $C[\psi]$ for the closed specifications under C. Then, by definition 52, $[\![C[\phi]]\!]^\emptyset \subseteq [\![C[\psi]]\!]^\emptyset$. By lemma 5, $(\mathcal{O}[\![C[\phi]]\!])^\dagger \subseteq (\mathcal{O}[\![C[\psi]]\!])^\dagger$.

- Let $C[\phi], h \rightsquigarrow^* \lightning$. By rule (7.17) (def. 48), $(h, \lightning) \in \mathcal{O}[\![C[\phi]]\!]$. By definition 50, $(h, \lightning) \in (\mathcal{O}[\![C[\phi]]\!])^\dagger$ and thus $(h, \lightning) \in (\mathcal{O}[\![C[\psi]]\!])^\dagger$. Then by definition 50, either $(h, \lightning) \in \mathcal{O}[\![C[\psi]]\!]$ or there exist h', h'', s such that $(h, h')s(h'', \lightning) \in \mathcal{O}[\![C[\psi]]\!]$. In both cases, by definition 48, we get that $C[\psi], h \rightsquigarrow^* \lightning$ and thus $C[\psi], h \lightning$.
- Let $C[\phi], h \rightsquigarrow^* h'$. By rule (7.17) (def. 48), $(h, h') \in \mathcal{O}[\![C[\phi]]\!]$. By definition 50, $(h, h') \in (\mathcal{O}[\![C[\phi]]\!])^\dagger$ and thus $(h, h') \in (\mathcal{O}[\![C[\psi]]\!])^\dagger$. Then, by definition 50, we have the following cases:

  i). $(h, h') \in \mathcal{O}[\![C[\psi]]\!]$

  ii). there exist h'', h''', s such that $(h, h'')s(h''', h') \in \mathcal{O}[\![C[\psi]]\!]$

  iii). $(h, \lightning) \in \mathcal{O}[\![C[\psi]]\!]$

  iv). there exist h', h'', s such that $(h, h')s(h'', \lightning) \in \mathcal{O}[\![C[\psi]]\!]$

  Cases i) and ii), by definition 48 give that $C[\psi], h \rightsquigarrow^* h'$.

  Cases iii) and iv), by definition 48 give that $C[\psi], h \rightsquigarrow^* \lightning$.

  Therefore, $C[\psi], h \rightsquigarrow^* h' \vee\ C[\psi], h \rightsquigarrow^* \lightning$.

□

7.5. Refinement Laws

Having defined the semantics of refinement in the previous section, we now state the refinement laws that comprise our refinement calculus for reasoning about concurrency. We distinguish the refinement laws into two broad groups: general refinement laws about our core specification language from definition 18, and refinement laws specific to primitive atomic specification statements.

First we define three primitive specifications statements that server as limits of our refinement calculus and behave as the unit of composition operators.

Definition 53 (Primitive specifications)**.**

$$\texttt{abort} \triangleq a(\mathsf{false},\ \mathsf{true})_k$$
$$\texttt{miracle} \triangleq a(\mathsf{true},\ \mathsf{false})_k$$
$$\texttt{skip} \triangleq a(\mathsf{true},\ \mathsf{true})_k$$

The `abort` statement always faults, since its precondition is never satisfied. It is the most permissive of specifications and serves as the top element in the partial order of refinement. Semantically, it is the set of all possible traces. On the other hand, the `miracle` statement never faults, as its precondition is always satisfied, but also never takes any steps as its postcondition is never satisfied. Semantically, `miracle` does nothing; modulo the closure of definition 50, it is the empty trace ϵ. It is a valid implementation of any specification and serves as the bottom element in the partial order of refinement. Finally, `skip` does not fault, but also does not modify the heap. The semantics of assertions is intuitionistic and therefore `true` denotes the empty heap resource. Thus, `skip` acts as the identity of sequential and parallel composition, as well as angelic and demonic choice. Note that `skip` is a specification statement which specifies that no state update is observed at the current level of abstraction. This does not necessarily mean that no update is performed. By contextual refinement we can refine `skip` to a statement that performs some update by, for example, strengthening the postcondition.

Definition 54 (General Refinement Laws)**.**

Refl
$$\phi \sqsubseteq \phi$$

Trans
$$\frac{\phi \sqsubseteq \psi' \qquad \psi' \sqsubseteq \psi}{\phi \sqsubseteq \psi}$$

AntiSymm
$$\frac{\phi \sqsubseteq \psi \qquad \psi \sqsubseteq \phi}{\phi \equiv \psi}$$

Skip
$$\texttt{skip}; \phi \equiv \phi \equiv \phi; \texttt{skip}$$

Assoc
$$\phi; (\psi_1; \psi_2) \equiv (\phi; \psi_1); \psi_2$$

MinMax
$$\texttt{miracle} \sqsubseteq \phi \sqsubseteq \texttt{abort}$$

EElim
$$\phi\,[e/x] \sqsubseteq \exists x.\, \phi$$

EIntro
$$\frac{x \notin free(\phi)}{\exists x.\, \phi \sqsubseteq \phi}$$

AChoiceEq
$$\phi \sqcup \phi \equiv \phi$$

AChoiceId
$$\phi \sqcup \texttt{miracle} \equiv \phi$$

AChoiceAssoc
$$\phi \sqcup (\psi_1 \sqcup \psi_2) \equiv (\phi \sqcup \psi_1) \sqcup \psi_2$$

AChoiceComm
$$\phi \sqcup \psi \equiv \psi \sqcup \phi$$

AChoiceElim
$$\phi \sqsubseteq \phi \sqcup \psi$$

AChoiceDstR
$$(\phi_1 \sqcup \phi_2); \psi \equiv (\phi_1; \psi) \sqcup (\phi_2; \psi)$$

AChoiceDstL
$$\psi; (\phi_1 \sqcup \phi_2) \equiv (\psi; \phi_1) \sqcup (\psi; \phi_2)$$

DChoiceEq
$$\phi \sqcap \phi \equiv \phi$$

DChoiceId
$\phi \sqcap \texttt{abort} \equiv \phi$

DChoiceAssoc
$\phi \sqcap (\psi_1 \sqcap \psi_2) \equiv (\phi \sqcap \psi_1) \sqcap \psi_2$

DChoiceComm
$\phi \sqcap \psi \equiv \psi \sqcap \phi$

DChoiceElim
$$\frac{\phi \sqsubseteq \psi_1 \qquad \phi \sqsubseteq \psi_2}{\phi \sqsubseteq \psi_1 \sqcap \psi_2}$$

DChoiceIntro
$\phi \sqcap \psi \sqsubseteq \phi$

DChoiceDstR
$(\phi_1 \sqcap \phi_2); \psi \equiv (\phi_1; \psi) \sqcap (\phi_2; \psi)$

DChoiceDstL
$\psi; (\phi_1 \sqcap \phi_2) \equiv (\psi; \phi_1) \sqcap (\psi; \phi_2)$

AChoiceDstD
$\phi \sqcup (\psi_1 \sqcap \psi_2) \equiv (\phi \sqcup \psi_1) \sqcap (\phi \sqcup \psi_2)$

DChoiceDstA
$\phi \sqcap (\psi_1 \sqcup \psi_2) \equiv (\phi \sqcap \psi_1) \sqcup (\phi \sqcap \psi_2)$

Absorb
$\phi \sqcup (\phi \sqcap \psi) \equiv \phi \equiv \phi \sqcap (\phi \sqcup \psi)$

Demonise
$\phi \sqcap \psi \sqsubseteq \phi \sqcup \psi$

ParSkip
$\phi \parallel \texttt{skip} \equiv \phi$

ParAssoc
$\phi \parallel (\psi_1 \parallel \psi_2) \equiv (\phi \parallel \psi_1) \parallel \psi_2$

ParComm
$\phi \parallel \psi \equiv \psi \parallel \phi$

Exchange
$(\phi \parallel \psi); (\phi' \parallel \psi') \sqsubseteq (\phi; \phi') \parallel (\psi; \psi')$

AChoiceExchange
$(\phi \parallel \psi) \sqcup (\phi' \parallel \psi') \sqsubseteq (\phi \sqcup \phi') \parallel (\psi \sqcup \psi')$

SeqPar
$\phi; \psi \sqsubseteq \phi \parallel \psi$

ParDstLR
$\phi; (\psi_1 \parallel \psi_2) \sqsubseteq (\phi; \psi_1) \parallel \psi_2$

ParDstLL
$\phi; (\psi_1 \parallel \psi_2) \sqsubseteq \psi_1 \parallel (\phi; \psi_2)$

ParDstRL
$(\phi \parallel \psi_1); \psi_2 \sqsubseteq \phi \parallel (\psi_1; \psi_2)$

ParDstRR
$(\phi \parallel \psi_1); \psi_2 \sqsubseteq (\phi; \psi_2) \parallel \psi_1$

EAChoiceEq
$\exists x.\, \phi \equiv \bigsqcup_{v \in \mathrm{Val}} \phi\,[v/x]$

ESeqExt
$$\frac{x \notin free(\phi)}{\exists x.\, \phi; \psi \equiv \phi; \exists x.\, \psi}$$

ESeqDst
$\exists x.\, \phi; \psi \sqsubseteq \exists x.\, \phi; \exists x.\, \psi$

EAChoiceDst
$\exists x.\, \phi \sqcup \psi \equiv (\exists x.\, \phi) \sqcup (\exists x.\, \psi)$

EDChoiceDst
$\exists x.\, \phi \sqcap \psi \equiv (\exists x.\, \phi) \sqcap (\exists x.\, \psi)$

EParDst
$\exists x.\, \phi \parallel \psi \sqsubseteq (\exists x.\, \phi) \parallel (\exists x.\, \psi)$

CMono
$$\frac{\phi \sqsubseteq \psi}{C[\phi] \sqsubseteq C[\psi]}$$

FApplyElim
$\phi\,[e/x] \equiv (\lambda x.\, \phi)\, e$

FApplyElimRec
$\phi\,[(\mu A.\, \lambda x.\, \phi)\,/A]\,[e/x] \equiv (\mu A.\, \lambda x.\, \phi)\, e$

FElim
$F_l \equiv \lambda x.\, F_l x$

FRename
$$\frac{\phi\,[e_1/x] \sqsubseteq \phi\,[e_2/x]}{(\lambda x.\, \phi)\, e_1 \sqsubseteq (\lambda x.\, \phi)\, e_2}$$

FRenameRec
$$\frac{\phi\,[(\mu A.\, \lambda x.\, \phi)\,/A]\,[e_1/x] \sqsubseteq \phi\,[(\mu A.\, \lambda x.\, \phi)\,/A]\,[e_2/x]}{(\mu A.\, \lambda x.\, \phi)\, e_1 \sqsubseteq (\mu A.\, \lambda x.\, \phi)\, e_2}$$

FuncIntro
$$\frac{x \notin free(\phi)}{(\lambda x.\, \phi)\,() \equiv \phi}$$

Inline
$\phi\,[F/f] \equiv \mathbf{let}\ f = F\ \mathbf{in}\ \phi$

Ind
$$\frac{\lambda x.\, \phi\,[\psi/A] \sqsubseteq \lambda x.\, \psi}{\mu A.\, \lambda x.\, \phi \sqsubseteq \lambda x.\, \psi}$$

UnrollR
$$\frac{A \notin free(\phi) \cup free(\psi)}{\left(\mu A.\, \lambda x.\, \psi \sqcup \phi; A e'\right) e \equiv \psi\,[e/x] \sqcup \phi\,[e/x]\,;\, \left(\mu A.\, \lambda x.\, \psi \sqcup \phi; A e'\right) e'\,[e/x]}$$

UnrollL
$$\frac{A \notin free(\phi) \cup free(\psi)}{\left(\mu A.\, \lambda x.\, \psi \sqcup A e'; \phi\right) e \equiv \psi\,[e/x] \sqcup \left(\mu A.\, \lambda x.\, \psi \sqcup A e'; \phi\right) e'\,[e/x]\,;\, \phi\,[e/x]}$$

RecSeq

$$\frac{A \notin free(\phi) \cup free(\psi_1) \cup free(\psi_2)}{\left(\mu A.\, \lambda x.\, \psi_1 \sqcup \phi; Ae'\right) e; \psi_2\,[e/x] \equiv \left(\mu A.\, \lambda x.\, \psi_1; \psi_2 \sqcup \phi; Ae'\right) e}$$

Typically, we name refinement laws in the order of refinement; that is, as if reading the law right to left. Many of the refinement laws in definition 54 are familiar from the literature.

From left to right, top to bottom, the laws Refl, Trans and AntiSymm reflect the fact that refinement is a partial order. Skip and Assoc state that `skip` is the unit of sequential composition and that sequential composition is associative respectively. The MinMax law defines `miracle` and `abort` as the top and bottom specifications in the partial order of refinement as discussed earlier.

The next two laws concern existential quantification. EElim allows elimination of the quantifier during refinement, by replacing the quantified variable with an expression. Conversely, the EIntro refinement law allows the introduction of an existentially quantified variable.

The next block of refinement laws are about angelic and demonic choice, most of which correspond to the laws of boolean algebra, with the join operator being angelic choice, the meet operator being demonic choice, the bottom element being `miracle` and the top element being `abort`. In fact, the partial order of refinement forms a complete, boolean and atomic lattice. The AChoiceElim refinement law captures the intuition behind the angelic non-deterministic choice: we can choose to refine the choice to one of the two components. On the other hand, the analogous law for demonic choice, DChoiceElim, states that the refinement of a demonic choice must be a refinement of both components. This law is analogous to the conjunction rule of Hoare logic.

Next is a set of laws regarding parallel composition. The most important refinement laws of this block are Exchange and AChoiceExchange. The former refines the parallel composition of two sequences into a sequential composition of two parallels, whereas the latter refines the parallel composition of two angelic choices to an angelic choice of two parallels. Both originate from Hoare's algebraic laws [57, 56]. The SeqPar law, as well as the subsequent distributivity laws for sequential and parallel composition can be derived from Exchange, ParComm and ParSkip.

In the next set of laws we return to existential quantification. The ESeqExt refinement law allows us to increase or decrease the scope of the existentially quantified variable. The rest of the laws for existential quantification concert its distributivity in sequential, non-deterministic choice and parallel composition.

The CMono refinement law is obvious, and the most pervasively used law in refinement derivations. It reflects the fact that denotational refinement is contextual refinement, as shown in theorem 1.

The next block of refinement laws is about functions. The laws follow directly from the semantics for functions in our specification language. FApplyElim allows the elimination of a function application by replacing the argument variable with the argument passed to the function. FElim allows us to eliminate indirect function applications. The FRename and FRenameRec allow the refinement of a function application to a different argument, for non-recursive and recursive functions respectively. The FuncIntro law allows the introduction of a function application and Inline allows function definitions to be inlined at the point of application.

The Ind refinement law is standard fixpoint induction.

The last block of refinement laws is useful in derivations of refinements between recursive specifications. The UnrollR and UnrollL laws allow us to do loop unrolling on recursive specification

programs. Finally, the RecSeq refinement law allows us to extract the final specification executed in a (tail) recursive specification outside the recursion.

We justify the soundness of our refinement laws by denotational refinement, which in turn is sound with respect to contextual refinement. Transitively, the refinement laws are also sound with respect to contextual refinement.

Theorem 2 (Soundness of General Refinement Laws)**.** *The general refinement laws of definition 54 are sound.*

Proof. See appendix B.2. □

Apart from the general refinement laws, we also define a set of laws for refinements between primitive atomic specification statements.

Definition 55 (Refinement Laws for Primitive Atomic Specification Statements.)**.**

UElim
$$a(\forall \vec{x}.\ P,\ Q)^{\mathcal{A}}_k \sqsubseteq a(\forall y, \vec{x}.\ P,\ Q)^{\mathcal{A}}_k$$

EPAtom
$$\frac{x \notin free(P)}{\exists x.\ a(\forall \vec{y}.\ P,\ Q)^{\mathcal{A}}_k \equiv a(\forall \vec{y}.\ P,\ \exists x.\ Q)^{\mathcal{A}}_k}$$

EPElim
$$a(\forall \vec{y}, x.\ P,\ Q)^{\mathcal{A}}_k \sqsubseteq a(\forall \vec{y}.\ \exists x.\ P,\ \exists x.\ Q)^{\mathcal{A}}_k$$

PDisjunction
$$a(\forall \vec{x}.\ P_1,\ Q_1)^{\mathcal{A}}_k \sqcup a(\forall \vec{x}.\ P_2,\ Q_2)^{\mathcal{A}}_k \sqsubseteq a(\forall \vec{x}.\ P_1 \vee P_2,\ Q_1 \vee Q_2)^{\mathcal{A}}_k$$

PConjunction
$$a(\forall \vec{x}.\ P_1,\ Q_1)^{\mathcal{A}}_k \sqcap a(\forall \vec{x}.\ P_2,\ Q_2)^{\mathcal{A}}_k \sqsubseteq a(\forall \vec{x}.\ P_1 \wedge P_2,\ Q_1 \wedge Q_2)^{\mathcal{A}}_k$$

Frame
$$a(\forall \vec{x}.\ P,\ Q)^{\mathcal{A}}_k \sqsubseteq a(\forall \vec{x}.\ P * R,\ Q * R)^{\mathcal{A}}_k$$

Stutter
$$a(\forall \vec{x}.\ P,\ P)^{\mathcal{A}}_k ; a(\forall \vec{x}.\ P,\ Q)^{\mathcal{A}}_k \sqsubseteq a(\forall \vec{x}.\ P,\ Q)^{\mathcal{A}}_k$$

Mumble
$$a(\forall \vec{x}.\ P,\ Q)^{\mathcal{A}}_k \sqsubseteq a\big(\forall \vec{x}.\ P,\ P'\big)^{\mathcal{A}}_k ; a\big(\forall \vec{x}.\ P',\ Q\big)^{\mathcal{A}}_k$$

Interleave
$$a(\forall \vec{x}.\ P_1,\ Q_1)^{\mathcal{A}}_k \parallel a(\forall \vec{x}.\ P_2,\ Q_2)^{\mathcal{A}}_k$$
$$\equiv \Big(a(\forall \vec{x}.\ P_1,\ Q_1)^{\mathcal{A}}_k ; a(\forall \vec{x}.\ P_2,\ Q_2)^{\mathcal{A}}_k\Big) \sqcup \Big(a(\forall \vec{x}.\ P_2,\ Q_2)^{\mathcal{A}}_k ; a(\forall \vec{x}.\ P_1,\ Q_1)^{\mathcal{A}}_k\Big)$$

PParallel
$$a(\forall \vec{x}.\ P_1,\ Q_1)^{\mathcal{A}}_k \parallel a(\forall \vec{x}.\ P_2,\ Q_2)^{\mathcal{A}}_k \sqsubseteq a(\forall \vec{x}.\ P_1 * P_2,\ Q_1 * Q_2)^{\mathcal{A}}_k$$

Cons
$$\frac{\forall \vec{x}.\ P \preccurlyeq P' \qquad \forall \vec{x}.\ Q' \preccurlyeq Q}{a\big(\forall \vec{x}.\ P',\ Q'\big)^{\mathcal{A}}_k \sqsubseteq a(\forall \vec{x}.\ P,\ Q)^{\mathcal{A}}_k}$$

RLevel
$$\frac{k_1 \leq k_2}{a(\forall \vec{x}.\ P,\ Q)^{\mathcal{A}}_{k_1} \sqsubseteq a(\forall \vec{x}.\ P,\ Q)^{\mathcal{A}}_{k_2}}$$

AContext
$$\frac{\alpha \notin \mathcal{A}}{a(\forall \vec{x}.\ P,\ Q)^{\mathcal{A}}_k \sqsubseteq a(\forall \vec{x}.\ P,\ Q)^{\alpha : x \in X \rightsquigarrow Y(x), \mathcal{A}}_k}$$

PAChoice
$$a\big(\forall \vec{x}.\ P,\ Q \vee Q'\big)^{\mathcal{A}}_k \sqsubseteq a(\forall \vec{x}.\ P,\ Q)^{\mathcal{A}}_k \sqcup a\big(\forall \vec{x}.\ P,\ Q'\big)^{\mathcal{A}}_k$$

RIEq

$$a\Big(\forall x \in X, \vec{x} \in \overrightarrow{X}.\, I_r(\mathbf{t}_\alpha^k(\vec{e}, x)) * P(\vec{x}),\ I_r(\mathbf{t}_\alpha^k(\vec{e}, x)) * Q(\vec{x})\Big)_k^{\mathcal{A}}$$
$$\equiv a\Big(\forall x \in X, \vec{x} \in \overrightarrow{X}.\, \mathbf{t}_\alpha^k(\vec{e}, x) * P(\vec{x}),\ \mathbf{t}_\alpha^k(\vec{e}, x) * Q(\vec{x})\Big)_{k+1}^{\mathcal{A}}$$

RUEq

$$\frac{\alpha \notin \mathcal{A} \qquad \forall x \in X.\, (x, f(x)) \in \mathcal{T}_{\mathbf{t}}(G)^*}{a\Big(\forall x \in X, \vec{x} \in \overrightarrow{X}.\, I_r(\mathbf{t}_\alpha^k(\vec{e}, x)) * P(\vec{x}) * [\mathsf{G}]_\alpha,\ I_r(\mathbf{t}_\alpha^k(\vec{e}, f(x))) * Q(\vec{x})\Big)_k^{\mathcal{A}}}$$
$$\equiv a\Big(\forall x \in X, \vec{x} \in \overrightarrow{X}.\, \mathbf{t}_\alpha^k(\vec{e}, x) * P(\vec{x}) * [\mathsf{G}]_\alpha,\ \mathbf{t}_\alpha^k(\vec{e}, f(x)) * Q(\vec{x})\Big)_{k+1}^{\mathcal{A}}$$

The refinement laws stated here have an implicit side condition that requires assertions on both sides of $\sqsubseteq$ are stable.

The UElim refinement law allows us to refine a primitive atomic specification statement in which a variable is explicitly universally quantified, to a primitive atomic specification in which the variable is free, and thus implicitly universally quantified in the context. The effect is that of turning a variable that is locally bound in the specification statement, to a global variable. The EPAtom states that late choice, in the existential quantification in the postcondition, is equivalent to early choice. EPAtom together with ESeqExt from definition 54, allow us to treat the existential quantifier similarly to the scope extrusion laws of π-calculus. With the EPElim law we can eliminate existential quantification analogously to the existential elimination rule of Hoare logic. The PDisjunction and PConjunction refinement laws are analogous to the conjunction and disjunction rules of Hoare logic. The Frame refinement law is directly analogous to the frame rule of separation logic [81].

The Stutter and Mumble refinement laws are due to the trace closure of definition 50, and originate from Brookes's trace semantics [21]. Stuttering reflects the fact that a specification is unable to observe steps of a refinement that do not modify the state. On the other hand, mumbling reflects the fact that a sequence of atomic steps can be implemented by a single atomic step. Note that by setting P' to be P in Mumble we obtain an equivalence for stuttering. The Interleave refinement law captures the intuition behind parallel composition: it causes interleaving between atomic steps. The PParallel refinement law is analogous to the parallel rule in disjoint concurrent separation logic [76] and can be derived from Interleave, Frame and AChoiceEq.

The Cons refinement law is directly analogous to the consequence rule of Views [35]. Recall that an implication between assertions is also a view-shift by lemma 3. Therefore, Cons is also used analogously to the consequence rule of Hoare logic. We can refine a primitive atomic specification statement by weakening its precondition and strengthening its postcondition. Conversely, we can abstract a primitive specification statement by strengthening the precondition and weakening the postcondition.

The refinement laws in definition 55 discussed so far, with the exception of EPAtom, are analogous to rules found in Hoare and separation logics. Analogous laws to EPAtom, EPElim, Frame, Stutter, Mumble and Cons exist in the calculus of Turon and Wand [94]. Our Cons law differs from Turon and Wand's in that it uses view-shifts instead of normal implications. Even though not present in the refinement calculus of Turon and Wand, we expect that analogous laws to UElim, PDisjunction, Interleave and PParallel can be justified for their atomic actions as well. This is not the case for PConjunction, as demonic choice is not supported in their specification language.

The next set of laws, following Cons, comes from our use of the TaDA model. RLevel reflects the fact that we can refine a specification statement of a higher region level to a specification statement of lower region level. The AContext law allows us to extend the atomicity context with an update to a region, for which an update is not already required. The RIEq and RUEq allows us to replace a shared region in a primitive atomic specification statement with its interpretation. In RIEq, where the state of the region is not modified, we can do this directly. In RUEq, the region is being updated and thus we require ownership of the guard by which the atomic update is allowed for that region in the state transition system.

Theorem 3 (Soundness of Primitive Atomic Refinement Laws)**.** *The primitive atomic refinement laws of definition 55 are sound.*

Proof. See appendix B.3. □

7.6. Abstract Atomicity

In the previous sections, we have defined our specification language and refinement calculus for concurrency. The basic statement of our language is the primitive atomic specification statement, $a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_{k}$, specifying a *physical* atomic update of the precondition state P to the postcondition state Q. The assertions P and Q are based on the assertion language of TaDA [30]. However, the whole purpose of combining refinement with TaDA is to use the concept of *abstract atomicity* specified in TaDA with atomic Hoare triples.

Atomic triples in TaDA have the general form:

$$k; \mathcal{A} \vdash \mathbb{V}\vec{x} \in \overrightarrow{X}.\, \langle P_p \mid P(\vec{x}) \rangle\, \mathbb{C}\ \ \mathbb{I}\vec{y} \in \overrightarrow{Y}.\, \langle Q_p(\vec{x}, \vec{y}) \mid Q(\vec{x}, \vec{y}) \rangle$$

The precondition and postcondition are split into two parts: the private part on the left of the vertical separator; and the public part on the right. Resources in the public part are updated atomically from $P(\vec{x})$ to $Q(\vec{x}, \vec{y})$. Interference by the environment is restricted by the pseudo universal quantifier, $\mathbb{V}\vec{x} \in \overrightarrow{X}$. Until the atomic update takes effect, the environment can change $\vec{x}$ as long as it is contained within $\overrightarrow{X}$, and as long as the precondition $P(\vec{x})$ is satisfied. Additionally the implementation must preserve $P(\vec{x})$ until the atomic update takes effect. After the atomic update takes effect, the environment can do anything with the public resources. Additionally, the implementation loses all access to resources in the public part. In the postcondition, $\vec{x}$ is bound to the values it obtains at the point the atomic update takes effect (the linearisation point). The update of the private resources from P_p to $Q_p(\vec{x}, \vec{y})$ is not atomic. Note that private does not mean local to the thread. Resources in the private part can still be shared, but they are not part of the atomic update. The private precondition does not depend on $\vec{x}$, since the environment can change it. The *pseudo existential* quantifier, $\mathbb{I}\vec{y} \in \overrightarrow{Y}$, links the private and public postcondition with $\vec{y}$, which is chosen arbitrarily by the implementation from $\overrightarrow{Y}$ at the point the atomic update takes effect.

The behaviour of a program satisfying the atomic Hoare triple is illustrated with the example trace in figure 7.1. In the example, we take the pseudo universal quantification to be over the values $\{0, 1\}$ and the pseudo existential quantification to be on the value 42. At the beginning of $\mathbb{C}$'s execution, the private part is P_p and x in the public part has value 0. The implementation initially updates

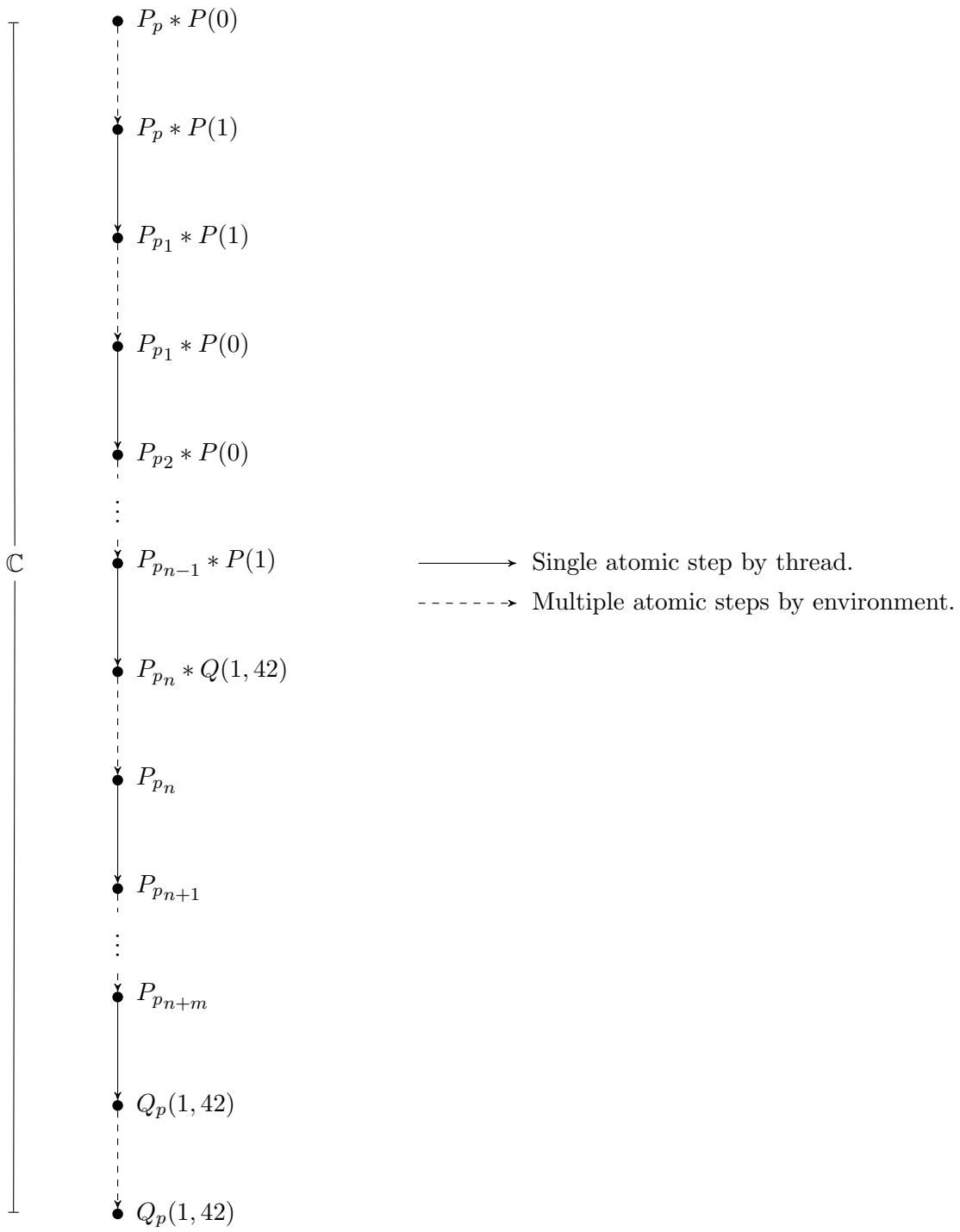

Figure 7.1.: Example behaviour of a program satisfying the general form of the atomic Hoare triple, where the pseudo universal quantification is $\mathbb{V}x \in \{0, 1\}$, the pseudo existential quantification is $\mathbb{I}y \in \{42\}$, and at the start of the execution the value of x is 0.

the private part to an intermediate state $P_{p_{n-1}}$ in a sequence of atomic steps, in between which the environment constantly changes x in the public part, from 0 to 1 and vice-versa, whilst maintaining $P(x)$. Then, the implementation takes one atomic step in which it updates the public precondition to the public postcondition as well as the intermediate private state. This is the step in which the abstract atomic update takes effect, thus forming the linearisation point. The implementation then relinquishes any information about the public part and proceeds with another sequence of atomic steps, updating the private part until reaching the private postcondition. Note that the environment does not modify the private part.

The example in figure 7.1 also illustrates that the behaviour of a program is abstractly atomic if the sequence of atomic steps it performs follows a particular pattern. Using our core specification language of definition 18, we can define this pattern of abstract atomicity as a specification program comprising as sequence of primitive atomic specification statements.

Notation

We often write specification programs that use inline functions: $(\lambda x.\, \phi)\, e$ or $(\mu A.\, \lambda x.\, \phi)\, e$. Several times, the function body, ϕ, may be relatively large. To increase readability in these cases, we use line breaks and whitespace instead of parenthesis to distinguish between the inline function and its application. For example, for a relatively large function body $\phi_{\text{large}}; \psi_{\text{huge}}$, we will write:

$$\begin{array}{l} \mu A.\, \lambda x.\ \phi_{\text{large}}; \\ \qquad\qquad \psi_{\text{huge}} \\ \cdot e \end{array}$$

to mean $(\mu A.\, \lambda x.\, \phi_{\text{large}}; \psi_{\text{huge}})\, e$.

Definition 56 (Atomic Specification Statement)**.** *The* atomic specification statement *is defined in terms of primitive atomic specification statements as follows:*

$$\begin{array}{l} \mathbb{V}\vec{x} \in \overrightarrow{X}.\, \Big\langle P_p \mid P(\vec{x}),\ \mathbb{E}\vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x},\vec{y}) \mid Q(\vec{x},\vec{y}) \Big\rangle_k^{\mathcal{A}} \triangleq \\ \quad \exists p_p.\, a\Big(\forall \vec{x} \in \overrightarrow{X}.\, P_p * P(\vec{x}),\ P_p \wedge p_p * P(\vec{x})\Big)_k^{\mathcal{A}}; \\ \quad \mu A.\, \lambda p_p.\ \exists p'_p.\, a\Big(\forall \vec{x} \in \overrightarrow{X}.\, p_p * P(\vec{x}),\ p'_p * P(\vec{x})\Big)_k^{\mathcal{A}}; A p'_p \\ \qquad \sqcup\ \exists \vec{x} \in \overrightarrow{X}, \vec{y} \in \overrightarrow{Y}.\, \exists p''_p.\ a\big(p_p * P(\vec{x}),\ p''_p * Q(\vec{x},\vec{y})\big)_k^{\mathcal{A}}; \\ \qquad\qquad \mu B.\, \lambda p''_p.\ \exists p'''_p.\, a\big(p''_p,\ p'''_p\big)_k^{\mathcal{A}}; B p'''_p \\ \qquad\qquad\qquad \sqcup\ a\big(p''_p,\ Q_p(\vec{x},\vec{y})\big)_k^{\mathcal{A}} \\ \qquad\qquad \cdot p''_p \\ \quad \cdot p_p \end{array}$$

The first primitive atomic statement solely serves to capture the states satisfied by the private precondition P_p into the variable p_p, so that it can be passed as an argument to the subsequent recursive function. This is a silent atomic step. Indeed, since it does not change the state before the step that immediately follows, by Stutter, the first primitive atomic statement is not observable. Furthermore, by Cons followed by Frame the first primitive atomic statement is refined into `skip`, and thus, by Skip, does not have to implemented at all.

The recursive function captures the essence of abstract atomicity. Initially, it performs a non-deterministic (angelically) number of atomic steps, including zero, that only update the private part. In between these steps, the environment can change the public part, $P(\vec{x})$. However, since the precondition of each primitive atomic specification statement requires $P(\vec{x})$ for all $\vec{x} \in \overrightarrow{X}$, each step in the environment can change $\vec{x}$, but only within $\overrightarrow{X}$, whilst maintaining $P(\vec{x})$, in order to avoid faulting. By the angelic choice, the implementation chooses when to finish updating the private part. When the updates to the private part complete, the subsequent primitive atomic specification statement updates the public part, from $P(\vec{x})$ to $Q(\vec{x}, \vec{y})$, as well as the private part. The values $\vec{x}$ and $\vec{y}$ are non-deterministically chosen by the implementation. This step is the linearisation point of the implementation. Finally, the inner recursive function follows, which performs a non-deterministic number of atomic updates to the private part, until it is updated to the private postcondition $Q_p(\vec{x}, \vec{y})$. The inner recursive function completely disregards the public parts, since after the linearisation point, the implementation is relieved of any responsibility for it.

Note the difference on the quantification on $\vec{x} \in \overrightarrow{X}$ between the sequence of primitive atomic steps that lead up to the linearisation point, and the primitive atomic step of the linearisation point. The former use universal quantification, whereas the latter uses existential. The universal quantification forces $P(\vec{x})$ to be maintained between steps, even if $\vec{x}$ changes within $\overrightarrow{X}$. The existential quantification on the step performing the linearisation point allows the implementation to choose for which $\vec{x} \in \overrightarrow{X}$ to perform the atomic update. If we had used universal quantification for this step as well, we would force implementations to commit the update for every $\vec{x} \in \overrightarrow{X}$, which would be wrong. For example, recall the abstract atomic lock specification discussed in chapter 6, section 6.2, where the unlocked state is denoted by 0 and the locked state is denoted by 1. According to the specification, the lock is locked only when the lock is previously unlocked: $x = 0$. However, universal quantification on $x \in \{0, 1\}$ at the linearisation point would force implementations to lock the lock all the time, even when it is still locked!

The pattern of definition 56, where we use a silent atomic read to capture the states satisfied by an assertion into a variable, which is then passed as an argument to a function, appears every time we prove various refinements for atomic specification statements, such as the refinement laws for abstract atomicity. The following lemma demonstrates that this step is indeed silent and is useful for several of the refinement proofs about atomic specification statements.

Lemma 6 (Assertions as Function Arguments)**.** *When Fe, with p free, does not occur within ϕ, then:*

$$\exists p.\, a(\forall \vec{x}.\, P,\ P \wedge p)^{\mathcal{A}}_k; \Big(\lambda p.\, a(\forall \vec{x}.\, p,\ Q)^{\mathcal{A}}_k; \phi\Big)\, p \sqsubseteq a(\forall \vec{x}.\, P,\ Q)^{\mathcal{A}}_k; \phi\,[P/p]$$

$$\begin{aligned}\exists p.\, a(\forall \vec{x}.\, P,\ P \wedge p)^{\mathcal{A}}_k; \Big(\lambda p.\, a(\forall \vec{x}.\, p,\ Q_1)^{\mathcal{A}}_k; \phi \sqcup a(\forall \vec{x}.\, p,\ Q_2)^{\mathcal{A}}_k; \psi\Big)\, p \\ \sqsubseteq a(\forall \vec{x}.\, P,\ Q_1)^{\mathcal{A}}_k; \phi\,[P/p] \sqcup a(\forall \vec{x}.\, P,\ Q_2)^{\mathcal{A}}_k; \psi\,[P/p]\end{aligned}$$

$$\exists p.\, a(\forall \vec{x}.\, P,\ P \wedge p)^{\mathcal{A}}_k; \Big(\mu A.\, \lambda p.\, a(\forall \vec{x}.\, p,\ Q)^{\mathcal{A}}_k; \phi\Big)\, p \sqsubseteq a(\forall \vec{x}.\, P,\ Q)^{\mathcal{A}}_k; \phi\,[P/p] \Big[(\mu A.\, \lambda p.\, a(\forall \vec{x}.\, p,\ Q)^{\mathcal{A}}_k; \phi)/A\Big]$$

$$\begin{aligned}\exists p.\, a(\forall \vec{x}.\, P,\ P \wedge p)^{\mathcal{A}}_k; \Big(\mu A.\, \lambda p.\, a(\forall \vec{x}.\, p,\ Q_1)^{\mathcal{A}}_k; \phi \sqcup a(\forall \vec{x}.\, p,\ Q_2)^{\mathcal{A}}_k; \psi\Big)\, p \\ \sqsubseteq \quad a(\forall \vec{x}.\, P,\ Q_1)^{\mathcal{A}}_k; \phi\,[P/p] \Big[(\mu A.\, \lambda p.\, a(\forall \vec{x}.\, p,\ Q_1)^{\mathcal{A}}_k; \phi \sqcup a(\forall \vec{x}.\, p,\ Q_2)^{\mathcal{A}}_k; \psi)/A\Big] \\ \sqcup\ a(\forall \vec{x}.\, P,\ Q_2)^{\mathcal{A}}_k; \psi\,[P/p] \Big[(\mu A.\, \lambda p.\, a(\forall \vec{x}.\, p,\ Q_1)^{\mathcal{A}}_k; \phi \sqcup a(\forall \vec{x}.\, p,\ Q_2)^{\mathcal{A}}_k; \psi)/A\Big]\end{aligned}$$

Proof. The first refinement is proven by application of FApplyElim, CMono, Stutter and finally EIntro. The second refinement is proven similarly, with AChoiceDstL before Stutter. The next two refinements are proven in the same way as the first two, except FApplyElimRec is used instead of FApplyElim. □

In most of chapter 6, we have used atomic specification statements of a simpler form that does not include the private part, defined as follows:

$$\mathbb{V}\vec{x} \in \overrightarrow{X}.\, \langle P(\vec{x}),\, Q(\vec{x})\rangle_k^{\mathcal{A}} \triangleq \mathbb{V}\vec{x} \in \overrightarrow{X}.\, \langle \mathsf{true} \mid P(\vec{x}),\, \mathbb{\exists} y \in \mathbf{1}.\, \mathsf{true} \mid Q(\vec{x})\rangle_k^{\mathcal{A}}$$

The general form of the atomic specification statement is a generalisation not only of an atomic update, but also of non-atomic updates. Other useful and important forms of specification statements, such as the Hoare specification statements we've informally discussed in chapter 6, are encoded in terms of the general atomic specification statement.

Definition 57 (Derived Specification Statements)**.** *The following specification statements are defined as special cases of the atomic specification statement.*

- $$\mathbb{\exists}\vec{x} \in \overrightarrow{X}.\, I(\vec{x}) \vdash \mathbb{V}\vec{x} \in \overrightarrow{X}.\, \left\langle P_p \mid P(\vec{x}),\, \mathbb{\exists}\vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x},\vec{y}) \mid Q(\vec{x},\vec{y}) \right\rangle_k^{\mathcal{A}} \triangleq \mathbb{V}\vec{x} \in \overrightarrow{X}.\, \left\langle P_p \mid P(\vec{x}) * I(\vec{x}),\, \mathbb{\exists}\vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x},\vec{y}) \mid Q(\vec{x},\vec{y}) * I(\vec{x}) \right\rangle_k^{\mathcal{A}}$$
- $\mathbb{\exists}\vec{x} \in \overrightarrow{X}.\, I(\vec{x}) \vdash \mathbb{V}\vec{x} \in \overrightarrow{X}.\, \langle P(\vec{x}),\, Q(\vec{x})\rangle_k^{\mathcal{A}} \triangleq \mathbb{V}\vec{x} \in \overrightarrow{X}.\, \langle P(\vec{x}) * I(\vec{x}),\, Q(\vec{x}) * I(\vec{x})\rangle_k^{\mathcal{A}}$
- $\mathbb{\exists}\vec{x} \in \overrightarrow{X}.\, I(\vec{x}) \vdash \{P,\, Q(\vec{x})\}_k^{\mathcal{A}} \triangleq \mathbb{V}\vec{x} \in \overrightarrow{X}.\, \langle P \mid I(\vec{x}),\, \mathbb{\exists} y \in \mathbf{1}.\, Q(\vec{x}) \mid I(\vec{x})\rangle_k^{\mathcal{A}}$
- $\{P,\, Q\}_k^{\mathcal{A}} \triangleq \mathbb{\exists} z \in \mathbf{1}.\, \mathsf{true} \vdash \{P,\, Q\}_k^{\mathcal{A}}$
- $[P]_k^{\mathcal{A}} \triangleq \{\mathsf{true},\, P\}_k^{\mathcal{A}}$
- $\mathbb{\exists}\vec{x} \in \overrightarrow{X}.\, I(\vec{x}) \vdash [P(\vec{x})]_k^{\mathcal{A}} \triangleq \mathbb{\exists}\vec{x} \in \overrightarrow{X}.\, I(\vec{x}) \vdash \{\mathsf{true},\, P(\vec{x})\}_k^{\mathcal{A}}$
- $\{P\}_k^{\mathcal{A}} \triangleq \{P,\, P\}_k^{\mathcal{A}}$
- $I \vdash \{P\}_k^{\mathcal{A}} \triangleq I \vdash \{P,\, P\}_k^{\mathcal{A}}$

The most important of the derived statements, is the *Hoare specification statement* of the form $\{P,\, Q\}$, which specifies an update from a state satisfying the precondition P, to a state satisfying the postcondition Q, without any atomicity guarantees. Intuitively, it stands for any program that satisfies the Hoare triple $\{P\} - \{Q\}$. Hoare specification statements also have a form using an invariant: $\mathbb{\exists}\vec{x} \in \overrightarrow{X}.\, I(\vec{x}) \vdash \{P,\, Q(\vec{x})\}$. This specifies a non-atomic update from a state satisfying P, to a state satisfying $Q(\vec{x})$, which maintains the invariant $I(\vec{x})$. The invariant is maintained at every step of the implementation that uses it. Note that the result of the update may depend on the invariant.

Both the general atomic specification statement and its simpler version that elides the private part are also given forms with invariants. These are useful in that resources that are only read during an atomic update do not have to be stated in both the precondition and postcondition, reducing verbosity.

The other two derived statements are the *assumption* statement of the form, $[P]$, and the *assertion* statement, $\{P\}$. The assumption statement specifies any non-atomic update that results in P, whereas

the assertion statement specifies non-atomic updates starting and ending in P. Both statements are also given forms with invariants.

Recall our use of context invariants in chapter 6 to specify modules implemented over the file system. We treat context invariants as normal invariants for specification statements. We write $I \vdash \phi \sqsubseteq \mathbb{V}x \in X.\, \langle P(x),\, Q(x)\rangle$ for a refinement of an atomic update conditional on the context invariant I, to mean $\phi \sqsubseteq I \vdash \mathbb{V}x \in X.\, \langle P(x),\, Q(x)\rangle$. From definition 57, this means that the atomic specification statement on the right maintains the context invariant I. The environment must also maintain I, at least up to the point the atomic update takes effect.

According to definitions 57 and 56 the Hoare specification statement, $\{P,\, Q\}$, is a sequence of atomic steps, where the first begins in state P and the last ends in state Q. However, the recursive function part of definition 56 is more complex that what is intuitively necessary for Hoare specification statements. Fortunately, with the following lemma we show that the encoding of Hoare specification statements according to definition 56 is equivalent to a much simpler pattern.

Lemma 7 (Hoare Specification Statement Refinement)**.**

$$\begin{aligned}\{P,\, Q\}_k^{\mathcal{A}} \equiv\ & \exists p.\, a(P,\, P \wedge p)_k^{\mathcal{A}};\\ & \mu A.\lambda p.\ \exists p'.\, a(p,\, p')_k^{\mathcal{A}}; Ap'\\ & \qquad \sqcup\, a(p,\, Q)_k^{\mathcal{A}}\\ & \cdot p\end{aligned}$$

Proof. We demonstrate a refinement between $\{P,\, Q\}_k^{\mathcal{A}}$, as given by definition 56, and the simpler form, in both directions. First, we show that:

$$\begin{aligned}\{P,\, Q\}_k^{\mathcal{A}} \sqsubseteq\ & \exists p.\, a(P,\, P \wedge p)_k^{\mathcal{A}};\\ & \mu A.\lambda p.\ \exists p'.\, a(p,\, p')_k^{\mathcal{A}}; Ap'\\ & \qquad \sqcup\, a(p,\, Q)_k^{\mathcal{A}}\\ & \cdot p\end{aligned}$$

$$\begin{aligned}\{P,\, Q\}_k^{\mathcal{A}} \equiv\ & \text{by definitions 57 and 56}\\ & \exists p.\, a(P,\, P \wedge p)_k^{\mathcal{A}};\\ & \mu A.\lambda p.\ \exists p'.\, a(p,\, p')_k^{\mathcal{A}}; Ap'\\ & \qquad \sqcup\, \exists p''.\, a(p,\, p'')_k^{\mathcal{A}};\\ & \qquad\quad \mu B.\lambda p''.\ \exists p'''.\, a(p'',\, p''')_k^{\mathcal{A}}; Bp'''\\ & \qquad\qquad \sqcup\, a(p'',\, Q)_k^{\mathcal{A}}\\ & \qquad\quad \cdot p''\\ & \cdot p\\ \sqsubseteq\ & \text{by IND and CMONO, where the following establishes the premiss}\\ & \text{begin with substitute } A \text{ and } \alpha\text{-convert}\end{aligned}$$

$$
\begin{array}{l}
\quad \exists p'.\, a(\forall \vec{x}.\, p,\ p')^{\mathcal{A}}_k; \mu A.\lambda p.\ \exists p''.\, a(p,\ p'')^{\mathcal{A}}_k; Ap'' \\
\qquad\qquad\qquad\qquad\qquad\qquad \sqcup\ a(p,\ Q)^{\mathcal{A}}_k \\
\qquad\qquad\qquad\qquad\qquad\qquad \cdot p' \\
\quad \sqcup\ \exists p'.\, a(\forall \vec{x}.\, p,\ p')^{\mathcal{A}}_k; \\
\qquad \mu A.\lambda p.\ \exists p''.\, a(p,\ p'')^{\mathcal{A}}_k; Ap'' \\
\qquad\qquad\qquad \sqcup\ a(p,\ Q)^{\mathcal{A}}_k \\
\qquad \cdot p' \\
\equiv \text{by } \textsc{AChoiceEq} \\
\qquad \exists p'.\, a(\forall \vec{x}.\, p,\ p')^{\mathcal{A}}_k; \mu A.\lambda p.\ \exists p''.\, a(p,\ p'')^{\mathcal{A}}_k; Ap'' \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \sqcup\ a(p,\ Q)^{\mathcal{A}}_k \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \cdot p' \\
\sqsubseteq \text{by } \textsc{AChoiceElim} \text{ and } \textsc{CMono} \\
\qquad a(\forall \vec{x}.\, p,\ Q)^{\mathcal{A}}_k \\
\qquad \sqcup\ \exists p'.\, a(\forall \vec{x}.\, p,\ p')^{\mathcal{A}}_k; \mu A.\lambda p.\ \exists p''.\, a(p,\ p'')^{\mathcal{A}}_k; Ap'' \\
\qquad\qquad\qquad\qquad\qquad\qquad \sqcup\ a(p,\ Q)^{\mathcal{A}}_k \\
\qquad\qquad\qquad\qquad\qquad\qquad \cdot p' \\
\equiv \text{by } \textsc{AChoiceComm} \text{ and } \textsc{UnrollR} \\
\qquad \mu A.\lambda p.\ \exists p'.\, a(p,\ p')^{\mathcal{A}}_k; Ap' \\
\qquad\qquad \sqcup\ a(p,\ Q)^{\mathcal{A}}_k \\
\qquad \cdot p \\
\sqsubseteq \exists p.\, a(P,\ P \wedge p)^{\mathcal{A}}_k; \\
\qquad \mu A.\lambda p.\ \exists p'.\, a(p,\ p')^{\mathcal{A}}_k; Ap' \\
\qquad\qquad \sqcup\ a(p,\ Q)^{\mathcal{A}}_k \\
\qquad \cdot p
\end{array}
$$

Now we show that:

$$
\begin{array}{l}
\exists p.\, a(P,\ P \wedge p)^{\mathcal{A}}_k; \\
\quad \mu A.\lambda p.\ \exists p'.\, a(p,\ p')^{\mathcal{A}}_k; Ap' \\
\qquad\quad \sqcup\ a(p,\ Q)^{\mathcal{A}}_k \\
\quad \cdot p
\end{array}
\quad \sqsubseteq \{P,\ Q\}^{\mathcal{A}}_k
$$

$$
\begin{array}{l}
\exists p.\, a(P,\ P \wedge p)^{\mathcal{A}}_k; \\
\quad \mu A.\lambda p.\ \exists p'.\, a(p,\ p')^{\mathcal{A}}_k; Ap' \\
\qquad\quad \sqcup\ a(p,\ Q)^{\mathcal{A}}_k \\
\quad \cdot p \\
\sqsubseteq \textsc{Mumble}, \textsc{EElim} \text{ and } \textsc{CMono} \\
\exists p.\, a(P,\ P \wedge p)^{\mathcal{A}}_k; \\
\quad \mu A.\lambda p.\ \exists p'.\, a(p,\ p')^{\mathcal{A}}_k; Ap' \\
\qquad\quad \sqcup\ \exists p''.\, a(p,\ p'')^{\mathcal{A}}_k; a(p'',\ Q)^{\mathcal{A}}_k \\
\quad \cdot p
\end{array}
$$

$\sqsubseteq$ by AChoiceElim and CMono

$\exists p.\, a(P,\ P \wedge p)^{\mathcal{A}}_k;$

$\mu A.\lambda p.\ \exists p'.\, a(p,\ p')^{\mathcal{A}}_k; Ap'$

$\sqcup\ \exists p''.\, a(p,\ p'')^{\mathcal{A}}_k; a(p'',\ Q)^{\mathcal{A}}_k \sqcup \exists p'''.\, a(p'',\ p''')^{\mathcal{A}}_k;\ \mu B.\lambda p''.\ \exists p'''.\, a(p'',\ p''')^{\mathcal{A}}_k; Bp'''$

$\sqcup\ a(p'',\ Q)^{\mathcal{A}}_k$

$\cdot p''$

$\cdot p$

$\sqsubseteq$ FApplyElimRec and CMono

$\exists p.\, a(P,\ P \wedge p)^{\mathcal{A}}_k;$

$\mu A.\lambda p.\ \exists p'.\, a(p,\ p')^{\mathcal{A}}_k; Ap'$

$\sqcup\ \exists p''.\, a(p,\ p'')^{\mathcal{A}}_k;$

$\mu B.\lambda p''.\ \exists p'''.\, a(p'',\ p''')^{\mathcal{A}}_k; Bp'''$

$\sqcup\ a(p'',\ Q)^{\mathcal{A}}_k$

$\cdot p''$

$\cdot p$

$\equiv$ by definitions 57 and 56

$\{P,\ Q\}^{\mathcal{A}}_k$

□

We define refinement laws for atomic specification statements, most of which directly correspond to rules in the TaDA program logic [30]. We have seen some of these laws, in their simpler form without the private part, in chapter 6.

Definition 58 (Abstract Atomicity Refinement Laws)**.**

MakeAtomic

$$\frac{\alpha \notin \mathcal{A} \qquad \{(x,y) \mid x \in X, y \in Y(x)\} \subseteq \mathcal{T}_{\mathbf{t}}(\mathsf{G})^*}{\begin{array}{l}\exists \vec{x} \in \overrightarrow{X}.\, I(\vec{x}) \vdash \left\{P_p * \exists x \in X.\, \mathbf{t}^k_\alpha(\vec{e}, x) * \alpha \Rrightarrow \blacklozenge,\ \exists x \in X, y \in Y(x).\, Q_p(x,y) * \alpha \Rrightarrow (x,y)\right\}^{\alpha : x \in X \rightsquigarrow Y(x), \mathcal{A}}_{k'} \\ \quad \sqsubseteq \forall x \in X, \vec{x} \in \overrightarrow{X}.\, \left\langle P_p \mid \mathbf{t}^k_\alpha(\vec{e}, x) * [\mathsf{G}(\vec{e}')]_\alpha * I(\vec{x}),\ \exists y \in Y(x).\, Q_p(x,y) \mid \mathbf{t}^k_\alpha(\vec{e}, y) * [\mathsf{G}(\vec{e}')]_\alpha * I(\vec{x}) \right\rangle^{\mathcal{A}}_{k'}\end{array}}$$

UpdateRegion

$$\forall x \in X, \vec{x} \in \overrightarrow{X}.\, \left\langle P_p \middle| I_r(\mathbf{t}^k_\alpha(\vec{e}, x)) * P(x,\vec{x}),\ \exists (y,z) \in Y(x) \times Z.\, Q_p(x,\vec{x},z,y) \middle| \begin{array}{l}\left(I_r(\mathbf{t}^k_\alpha(\vec{e}, y)) * Q_1(x,\vec{x},z,y)\right) \\ \vee \left(I_r(\mathbf{t}^k_\alpha(\vec{e}, x)) * Q_2(x,\vec{x},z)\right)\end{array} \right\rangle^{\mathcal{A}}_k$$

$$\sqsubseteq$$

$$\forall x \in X, \vec{x} \in \overrightarrow{X}.\, \left\langle \begin{array}{l} P_p \mid \mathbf{t}^k_\alpha(\vec{e}, x) * P(x,\vec{x}) * \alpha \Rrightarrow \blacklozenge, \\ \exists (y,z) \in Y(x) \times Z.\, Q_p(x,\vec{x},z,y) \mid \end{array} \begin{array}{l}\left(\mathbf{t}^k_\alpha(\vec{e}, y) * Q_1(x,\vec{x},z,y) * \alpha \Rrightarrow (x,y)\right) \\ \vee \left(\mathbf{t}^k_\alpha(\vec{e}, x) * Q_2(x,\vec{x},z) * \alpha \Rrightarrow \blacklozenge\right)\end{array} \right\rangle^{\alpha : x \in X \rightsquigarrow Y(x), \mathcal{A}}_{k+1}$$

OpenRegion

$$\begin{array}{l}\forall x \in X, \vec{x} \in \overrightarrow{X}.\, \left\langle P_p \mid I_r(\mathbf{t}^k_\alpha(\vec{e}, x)) * P(x,\vec{x}),\ \exists \vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x},\vec{y}) \mid I_r(\mathbf{t}^k_\alpha(\vec{e}, x)) * Q(x,\vec{x},\vec{y}) \right\rangle^{\mathcal{A}}_k \\ \quad \equiv \forall x \in X, \vec{x} \in \overrightarrow{X}.\, \left\langle P_p \mid \mathbf{t}^k_\alpha(\vec{e}, x) * P(x,\vec{x}),\ \exists \vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x},\vec{y}) \mid \mathbf{t}^k_\alpha(\vec{e}, x) * Q(x,\vec{x},\vec{y}) \right\rangle^{\mathcal{A}}_{k+1}\end{array}$$

UseAtomic

$$\frac{\alpha \notin \mathcal{A} \qquad \forall x \in X.\, (x, f(x)) \in \mathcal{T}_{\mathsf{t}}(\mathsf{G})^*}{\begin{array}{l}\mathbb{V}x \in X, \vec{x} \in \overrightarrow{X}.\, \Big\langle P_p \mid I_r(\mathbf{t}^k_\alpha(\vec{e}, x)) * P(x, \vec{x}) * [\mathsf{G}(\vec{e}')]_\alpha,\ \exists \vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x}, \vec{y}) \mid I_r(\mathbf{t}^k_\alpha(\vec{e}, f(x))) * Q(x, \vec{x}, \vec{y})\Big\rangle^{\mathcal{A}}_k \\ \quad \equiv \mathbb{V}x \in X, \vec{x} \in \overrightarrow{X}.\, \Big\langle P_p \mid \mathbf{t}^k_\alpha(\vec{e}, x) * P(\vec{x}) * [\mathsf{G}(\vec{e}')]_\alpha,\ \exists \vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x}, \vec{y}) \mid \mathbf{t}^k_\alpha(\vec{e}, f(x)) * Q(\vec{x}, \vec{y})\Big\rangle^{\mathcal{A}}_{k+1}\end{array}}$$

AFrame

$$\begin{array}{l}\mathbb{V}\vec{x} \in \overrightarrow{X}.\, \Big\langle P_p \mid P(\vec{x}),\ \exists \vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x}, \vec{y}) \mid Q(\vec{x}, \vec{y})\Big\rangle^{\mathcal{A}}_k \\ \sqsubseteq \mathbb{V}\vec{x} \in \overrightarrow{X}.\, \Big\langle P_p * R' \mid P(\vec{x}) * R(\vec{x}),\ \exists \vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x}, \vec{y}) * R' \mid Q(\vec{x}, \vec{y}) * R(\vec{x})\Big\rangle^{\mathcal{A}}_k\end{array}$$

AWeaken1

$$\begin{array}{l}\mathbb{V}\vec{x} \in \overrightarrow{X}.\, \Big\langle P_p \mid P' * P(\vec{x}),\ \exists \vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x}, \vec{y}) \mid Q'(\vec{x}, \vec{y}) * Q(\vec{x}, \vec{y})\Big\rangle^{\mathcal{A}}_k \\ \sqsubseteq \mathbb{V}\vec{x} \in \overrightarrow{X}.\, \Big\langle P_p * P' \mid P(\vec{x}),\ \exists \vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x}, \vec{y}) * Q'(\vec{x}, \vec{y}) \mid Q(\vec{x}, \vec{y})\Big\rangle^{\mathcal{A}}_k\end{array}$$

Primitive

$$\begin{array}{l}a\Big(\forall \vec{x} \in \overrightarrow{X}.\, P_p * P(\vec{x}),\ \exists \vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x}, \vec{y}) * Q(\vec{x}, \vec{y})\Big)^{\mathcal{A}}_k \\ \sqsubseteq \mathbb{V}\vec{x} \in \overrightarrow{X}.\, \Big\langle P_p \mid P(\vec{x}),\ \exists \vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x}, \vec{y}) \mid Q(\vec{x}, \vec{y})\Big\rangle^{\mathcal{A}}_k\end{array}$$

AEElim

$$\begin{array}{l}\mathbb{V}x \in X, \vec{x} \in \overrightarrow{X}.\, \Big\langle P_p \mid P(x, \vec{x}),\ \exists \vec{y} \in \overrightarrow{Y}.\, P_p(x, \vec{x}, \vec{y}) \mid Q(x, \vec{x}, \vec{y})\Big\rangle^{\mathcal{A}}_k \\ \sqsubseteq \mathbb{V}\vec{x} \in \overrightarrow{X}.\, \Big\langle P_p \mid \exists x \in X.\, P(x, \vec{x}),\ \exists \vec{y} \in \overrightarrow{Y}.\, \exists x \in X.\, P_p(x, \vec{x}, \vec{y}) \mid \exists x \in X.\, Q(x, \vec{x}, \vec{y})\Big\rangle^{\mathcal{A}}_k\end{array}$$

EAAtom

$$\frac{x \notin \mathtt{free}(P_p) \cup \mathtt{free}(P) \qquad y \notin \mathtt{free}(P_p) \cup \mathtt{free}(P)}{\begin{array}{l}\exists y.\, \exists x.\, \mathbb{V}\vec{x} \in \overrightarrow{X}.\, \Big\langle P_p \mid P(\vec{x}),\ \exists \vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x}, \vec{y}) \mid Q(\vec{x}, \vec{y})\Big\rangle^{\mathcal{A}}_k \\ \equiv \mathbb{V}\vec{x} \in \overrightarrow{X}.\, \Big\langle P_p \mid P(\vec{x}),\ \exists y, \vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x}, y, \vec{y}) \mid \exists x.\, Q(\vec{x}, y, \vec{y})\Big\rangle^{\mathcal{A}}_k\end{array}}$$

AStutter

$$\begin{array}{l}\mathbb{V}\vec{x} \in \overrightarrow{X}.\, \big\langle P_p \mid P(\vec{x}),\ P'_p \mid P(\vec{x})\big\rangle^{\mathcal{A}}_k; \mathbb{V}\vec{x} \in \overrightarrow{X}.\, \Big\langle P'_p \mid P(\vec{x}),\ \exists \vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x}, \vec{y}) \mid Q(\vec{x}, \vec{y})\Big\rangle^{\mathcal{A}}_k \\ \sqsubseteq \mathbb{V}\vec{x} \in \overrightarrow{X}.\, \Big\langle P_p \mid P(\vec{x}),\ \exists \vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x}, \vec{y}) \mid Q(\vec{x}, \vec{y})\Big\rangle^{\mathcal{A}}_k\end{array}$$

AMumble

$$\begin{array}{l}\mathbb{V}\vec{x} \in \overrightarrow{X}.\, \Big\langle P_p \mid P(\vec{x}),\ \exists \vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x}, \vec{y}) \mid Q(\vec{x}, \vec{y})\Big\rangle^{\mathcal{A}}_k \\ \sqsubseteq \mathbb{V}\vec{x} \in \overrightarrow{X}.\, \Big\langle P_p \mid P(\vec{x}),\ \exists \vec{y} \in \overrightarrow{Y}.\, P'_p \mid P'(\vec{x})\Big\rangle^{\mathcal{A}}_k; \mathbb{V}\vec{x} \in \overrightarrow{X}.\, \Big\langle P'_p \mid P'(\vec{x}),\ \exists \vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x}, \vec{y}) \mid Q(\vec{x}, \vec{y})\Big\rangle^{\mathcal{A}}_k\end{array}$$

ADisj

$$\begin{array}{l}\mathbb{V}\vec{x} \in \overrightarrow{X}.\, \Big\langle P_p \mid P(\vec{x}),\ \exists \vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x}, \vec{y}) \mid Q(\vec{x}, \vec{y})\Big\rangle^{\mathcal{A}}_k \sqcup \mathbb{V}\vec{x} \in \overrightarrow{X}.\, \Big\langle P'_p \mid P'(\vec{x}),\ \exists \vec{y} \in \overrightarrow{Y}.\, Q'_p(\vec{x}, \vec{y}) \mid Q'(\vec{x}, \vec{y})\Big\rangle^{\mathcal{A}}_k \\ \quad \sqsubseteq \mathbb{V}\vec{x} \in \overrightarrow{X}.\, \Big\langle P_p \vee P'_p \mid P(\vec{x}) \vee P'(\vec{x}),\ \exists \vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x}, \vec{y}) \vee Q'_p(\vec{x}, \vec{y}) \mid Q(\vec{x}, \vec{y}) \vee Q'(\vec{x}, \vec{y})\Big\rangle^{\mathcal{A}}_k\end{array}$$

AConj

$$\begin{array}{l}\mathbb{V}\vec{x} \in \overrightarrow{X}.\, \Big\langle P_p \mid P(\vec{x}),\ \exists \vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x}, \vec{y}) \mid Q(\vec{x}, \vec{y})\Big\rangle^{\mathcal{A}}_k \sqcap \mathbb{V}\vec{x} \in \overrightarrow{X}.\, \Big\langle P'_p \mid P'(\vec{x}),\ \exists \vec{y} \in \overrightarrow{Y}.\, Q'_p(\vec{x}, \vec{y}) \mid Q'(\vec{x}, \vec{y})\Big\rangle^{\mathcal{A}}_k \\ \quad \sqsubseteq \mathbb{V}\vec{x} \in \overrightarrow{X}.\, \Big\langle P_p \wedge P'_p \mid P(\vec{x}) \wedge P'(\vec{x}),\ \exists \vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x}, \vec{y}) \wedge Q'_p(\vec{x}, \vec{y}) \mid Q(\vec{x}, \vec{y}) \wedge Q'(\vec{x}, \vec{y})\Big\rangle^{\mathcal{A}}_k\end{array}$$

ACons

$$\frac{P_p \preccurlyeq P'_p \qquad \forall \vec{x} \in \overrightarrow{X}.\, P(\vec{x}) \preccurlyeq P'(\vec{x}) \qquad \forall \vec{x} \in \overrightarrow{X}, \vec{y} \in \overrightarrow{Y}.\, Q'_p(\vec{x},\vec{y}) \preccurlyeq Q_p(\vec{x},\vec{y}) \qquad \forall \vec{x} \in \overrightarrow{X}, \vec{y} \in \overrightarrow{Y}.\, Q'(\vec{x},\vec{y}) \preccurlyeq Q(\vec{x},\vec{y})}{\begin{array}{c}\forall \vec{x} \in \overrightarrow{X}.\, \left\langle P'_p \mid P'(\vec{x}),\ \exists \vec{y} \in \overrightarrow{Y}.\, Q'_p(\vec{x},\vec{y}) \mid Q'(\vec{x},\vec{y}) \right\rangle^{\mathcal{A}}_{k} \\ \sqsubseteq \forall \vec{x} \in \overrightarrow{X}.\, \left\langle P_p \mid P(\vec{x}),\ \exists \vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x},\vec{y}) \mid Q(\vec{x},\vec{y}) \right\rangle^{\mathcal{A}}_{k}\end{array}}$$

Subst1

$$\frac{f : X' \to X}{\begin{array}{c}\forall x \in X, \vec{x} \in \overrightarrow{X}.\, \left\langle P_p \mid P(x,\vec{x}),\ \exists y \in Y(x), \vec{y} \in \overrightarrow{Y}.\, Q_p(x,\vec{x},\vec{y}) \mid Q(x,\vec{x},\vec{y}) \right\rangle^{\mathcal{A}}_{k} \\ \sqsubseteq \forall x' \in X', \vec{x} \in \overrightarrow{X}.\, \left\langle P_p \mid P(f(x'),\vec{x}),\ \exists y \in Y(f(x')), \vec{y} \in \overrightarrow{Y}.\, Q_p(f(x'),y,\vec{y}) \mid Q(f(x'),y,\vec{y}) \right\rangle^{\mathcal{A}}_{k}\end{array}}$$

Subst2

$$\frac{f_x : Y'(x) \to Y(x)}{\begin{array}{c}\forall \vec{x} \in \overrightarrow{X}.\, \left\langle P_p \mid P(\vec{x}),\ \exists y' \in \overrightarrow{Y'(x)}, \vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x},f_x(y'),\vec{y}) \mid Q(\vec{x},f_x(y'),\vec{y}) \right\rangle^{\mathcal{A}}_{k} \\ \sqsubseteq \forall \vec{x} \in \overrightarrow{X}.\, \left\langle P_p \mid P(\vec{x}),\ \exists y \in Y(x), \vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x},y,\vec{y}) \mid Q(\vec{x},y,\vec{y}) \right\rangle^{\mathcal{A}}_{k}\end{array}}$$

ARLevel

$$\frac{k_1 \le k_2}{\begin{array}{c}\forall \vec{x} \in \overrightarrow{X}.\, \left\langle P_p \mid P(\vec{x}),\ \exists \vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x},\vec{y}) \mid Q(\vec{x},\vec{y}) \right\rangle^{\mathcal{A}}_{k_1} \\ \sqsubseteq \forall \vec{x} \in \overrightarrow{X}.\, \left\langle P_p \mid P(\vec{x}),\ \exists \vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x},\vec{y}) \mid Q(\vec{x},\vec{y}) \right\rangle^{\mathcal{A}}_{k_2}\end{array}}$$

AAContext

$$\frac{\alpha \notin \mathcal{A}}{\begin{array}{c}\forall \vec{x} \in \overrightarrow{X}.\, \left\langle P_p \mid P(\vec{x}),\ \exists \vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x},\vec{y}) \mid Q(\vec{x},\vec{y}) \right\rangle^{\mathcal{A}}_{k} \\ \sqsubseteq \forall \vec{x} \in \overrightarrow{X}.\, \left\langle P_p \mid P(\vec{x}),\ \exists \vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x},\vec{y}) \mid Q(\vec{x},\vec{y}) \right\rangle^{\alpha : x \in X \rightsquigarrow Y(x), \mathcal{A}}_{k}\end{array}}$$

The refinement laws stated here have an implicit side condition that requires assertions on both sides of $\sqsubseteq$ are stable.

The MakeAtomic refinement law allows atomicity abstraction. The atomic specification statement in the conclusion, atomically updates the region α from an abstract state $x \in X$, to the state $y \in Q(x)$. For that, it requires ownership of the guard G, for the same region, and by the premiss, the update must be allowed by the region's transition system for G. The atomic specification statement is refined by the Hoare specification statement, that uses the atomicity tracking resource, in place of the guard, to guarantee that the update on the region happens atomically. The atomicity context is extended to record the update that must take effect during the Hoare statement. Recall that assertions are stable under interference from the environment. Until the update takes effect, the region's state may be changed by the environment, but remains within X, and the atomicity tracking resource records that the update defined in the atomicity context has not taken effect. When the atomic update takes effect, the atomicity tracking resource is simultaneously updated to record the actual update on the region. Note that the region is not present in the postcondition of the Hoare specification statement. This is because after the atomic update takes effect, the environment is allowed to do anything, even destroy the region.

In MakeAtomic, the atomic specification statement is also atomically reading the resources in

$I(\vec{x})$. This allows the atomic update to the region to use additional resource, with the proviso that it does not modify $I(\vec{x})$. The pseudo universal quantification on $\vec{x} \in \overrightarrow{x}$ allows the environment to change $\vec{x}$ within $\overrightarrow{X}$ as long as $I(\vec{x})$ is maintained. Therefore, the Hoare specification statement is also required to be invariant under $I(\vec{x})$. This is a slight generalisation of the analogous rule in TaDA, which does not support such invariants.

The UpdateRegion refinement law allows us to discharge an atomicity tracking resource and thus commit the atomic update required in the atomicity context. Note that in the postcondition, either the update specified in the atomicity context occurs, or no update occurs. This is in order to allow working with conditional atomic operations, such as compare-and-swap. The OpenRegion law allows us to access the contents of a shared region, while maintaining its abstract state. The concrete resources that comprise the region's interpretation can be updated, as long as the abstract state is preserved. The UseAtomic law is used to justify an update to the abstract state of a region, by an atomic update to the region's interpretation. For that, we require ownership of a guard, for which the update to the abstract state of the region is permitted in its transition system. Note that even though UpdateRegion, OpenRegion and UseAtomic operate on a single region, they also allow other resources to be atomically updated. Therefore, it is possible by successive applications of these laws to refine atomic specification statements that atomically update multiple regions at the same time.

The AFrame law is the extension of Frame law for atomic specification statements. The AWeaken1 refinement law allows a non-atomic update to the private to be implemented as an atomic update. Primitive allows an abstractly atomic update to be implemented by a physical atomic update, including for the update to the private part. Note that this refinement law states an inequality; abstract atomicity is a generalisation of physical atomicity. The Primitive law comes as a consequence of Mumble.

The AEElim law allows us to eliminate existential quantification in the public part, by refining it to a pseudo universal quantifier. Note that elimination to a standard universal quantifier, as in EPElim would be unsound. Existential quantification allows the environment to change x, whereas universal quantification requires its value to be fixed during execution. The former allows more interference than the latter, thus the latter cannot be a refinement of the former. The EAAtom law extends the EPAtom law to atomic specification statements. Note that this law allows the early choice to be made for both the private and public parts of the postcondition.

The AStutter and AMumble refinement laws are extensions of Stutter and Mumble to atomic specification statements. AMumble is a direct consequence of Mumble. However, AStutter cannot be justified by Stutter because of private-part updates and requires more complex refinement reasoning. Similarly, the ADisj and AConj refinement laws are extensions of PDisjunction and PConjunction to atomic specification statements respectively.

The ACons refinement law is the analogous to Cons for atomic specification statements. Subst1 allows interference on x to be strengthened (when abstracting); an atomic update allowing less interference can be refined to an atomic update allowing more interference. On the other hand, Subst2 allows the analogous of weakening on the pseudo existentially quantified y.

Finally, ARLevel and AAContext are extensions of RLevel and AContext to atomic specification statements.

Since the atomic specification statement is just a program in out core specification language, most

of the refinement laws on abstract atomicity are simply proven as refinements between specification programs, by using the general laws of our refinement calculus (definition 54) and the laws for primitive atomic statements (definition 55). The only refinement laws where we need to appeal to the assertion model are the MakeAtomic and UpdateRegion rules. These are the most challenging refinement laws of our entire development. Fortunately, since these laws and the assertion language is based on TaDA, we can make the same case as in the TaDA semantics when appealing to the model.

Theorem 4 (Soundness of Abstract Atomicity Refinement Laws)**.** *The refinement laws for abstract atomicity in definition 58 are sound.*

Proof. See appendix B.4. □

We complete the development of refinement laws in this section, with laws for Hoare specification statements. At this point, none of these laws should be surprising.

Definition 59 (Hoare Specification Statement Refinement Laws)**.**

Seq
$$\frac{\phi \sqsubseteq I \vdash \{P,\, R\}^{\mathcal{A}}_k \qquad \psi \sqsubseteq I \vdash \{R,\, Q\}^{\mathcal{A}}_k}{\phi;\psi \sqsubseteq I \vdash \{P,\, Q\}^{\mathcal{A}}_k}$$

Disjunction
$$\frac{\phi \sqsubseteq I \vdash \{P_1,\, Q_1\}^{\mathcal{A}}_k \qquad \psi \sqsubseteq I \vdash \{P_2,\, Q_2\}^{\mathcal{A}}_k}{\phi \sqcup \psi \sqsubseteq I \vdash \{P_1 \vee P_2,\, Q_1 \vee Q_2\}^{\mathcal{A}}_k}$$

Conjunction
$$\frac{\phi \sqsubseteq I \vdash \{P_1,\, Q_1\}^{\mathcal{A}}_k \qquad \psi \sqsubseteq I \vdash \{P_2,\, Q_2\}^{\mathcal{A}}_k}{\phi \sqcap \psi \sqsubseteq I \vdash \{P_1 \wedge P_2,\, Q_1 \wedge Q_2\}^{\mathcal{A}}_k}$$

Parallel
$$\frac{\phi \sqsubseteq I \vdash \{P_1,\, Q_1\}^{\mathcal{A}}_k \qquad \psi \sqsubseteq I \vdash \{P_2,\, Q_2\}^{\mathcal{A}}_k}{\phi \parallel \psi \sqsubseteq I \vdash \{P_1 * P_2,\, Q_1 * Q_2\}^{\mathcal{A}}_k}$$

HFrame
$$I \vdash \{P,\, Q\}^{\mathcal{A}}_k \sqsubseteq I \vdash \{P * R,\, Q * R\}^{\mathcal{A}}_k$$

EElimHoare
$$I \vdash \{P,\, Q\}^{\mathcal{A}}_k \sqsubseteq I \vdash \{\exists y.\, P,\, \exists y.\, Q\}^{\mathcal{A}}_k$$

EHAtom
$$\frac{x \notin \mathtt{free}(P) \qquad x \notin \mathtt{free}(I)}{\exists x.\, I \vdash \{P,\, Q\}^{\mathcal{A}}_k \equiv I \vdash \{P,\, \exists x.\, Q\}^{\mathcal{A}}_k}$$

AWeaken2
$$\forall \vec{x} \in \overrightarrow{X}.\, \left\langle P_p \mid P(\vec{x}) * I(\vec{x}),\, \exists \vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x},\vec{y}) \mid Q(\vec{x},\vec{y}) * I(\vec{x}) \right\rangle^{\mathcal{A}}_k \sqsubseteq \exists \vec{x} \in \overrightarrow{X}.\, I(\vec{x}) \vdash \left\{ P_p * P(\vec{x}),\, \exists \vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x},\vec{y}) * Q(\vec{x},\vec{y}) \right\}^{\mathcal{A}}_k$$

HCons
$$\frac{P \preccurlyeq P' \qquad Q' \preccurlyeq Q}{I \vdash \{P',\, Q'\}^{\mathcal{A}}_k \sqsubseteq I \vdash \{P,\, Q\}^{\mathcal{A}}_k}$$

HRLevel
$$\frac{k_1 \leq k_2}{I \vdash \{P,\, Q\}^{\mathcal{A}}_{k_1} \sqsubseteq I \vdash \{P,\, Q\}^{\mathcal{A}}_{k_2}}$$

HAContext
$$\frac{\alpha \notin \mathcal{A}}{I \vdash \{P,\, Q\}^{\mathcal{A}}_k \sqsubseteq I \vdash \{P,\, Q\}^{\alpha:x \in X \rightsquigarrow Y(x),\mathcal{A}}_k}$$

The Seq, Disjunction and Conjunction refinement laws directly correspond to the sequence, disjunction and conjunction rules of Hoare logic. Parallel directly corresponds to the parallel rule of separation logic [76]. The HFrame law corresponds to the frame rule of separation logic [81], and is a direct consequence of AFrame. EElimHoare allows existential quantification elimination as the analogous rule in Hoare logic. The AWeaken2 refinement law allows a non-atomic update defined by the Hoare specification statement, to be implemented atomically by an atomic specification statement. It is a direct consequence of AWeaken1 when the entire public part is moved to the private

part. The HCons law, is the consequence rule of Hoare logic for Hoare specification statements. Finally, HRLevel and HAContext allows the same as ARLevel and AAContext, but for Hoare specification statements.

Theorem 5 (Soundness of Hoare Specification Statement Refinement Laws)**.** *The refinement laws for Hoare specification statements, in definition 59, are sound.*

Proof. See appendix B.5. □

7.7. Syntax Encodings

The specifications in chapter 6 use programming constructs such as let-bindings and control flow that are not present in the specification language formalised in this chapter. Fortunately, the specification language is expressive enough for these constructs to be encoded as syntactic sugar.

Definition 60 (Syntactic Sugar)**.**

$$\begin{aligned}
\textbf{if } P \textbf{ then } \phi \textbf{ else } \psi \textbf{ fi} &\triangleq ([P]; \phi) \sqcup ([\neg P]; \psi) \quad \textit{when } P \textit{ is pure} \\
\lambda x_1, x_2 \ldots, x_n.\, \phi &\triangleq \lambda x_1.\, \lambda x_2. \ldots \lambda x_n.\, \phi \\
f(x_1, x_2 \ldots x_n) &\triangleq ((f x_1) x_2) \ldots x_n \\
\textbf{let } f(\vec{x}) = \phi \textbf{ in } \psi &\triangleq \textbf{let } f = \lambda \vec{x}, \mathtt{ret}.\, \phi \textbf{ in } \psi \\
\textbf{letrec } f(\vec{x}) = \phi \textbf{ in } \psi &\triangleq \textbf{let } f = \mu A.\, \lambda \vec{x}, \mathtt{ret}.\, \phi\,[A/f] \textbf{ in } \psi \\
\mathtt{return}\ x &\triangleq [\mathtt{ret} = x] \\
\mathtt{return}\ f(\vec{x}) &\triangleq \textbf{let } y = f(\vec{x}) \textbf{ in } \mathtt{return}\ y \\
\mathtt{return}\ f_1(\vec{x_1}) \sqcap \ldots f_n(\vec{x_n}) &\triangleq \mathtt{return}\ f_1(\vec{x_1}) \sqcap \ldots \mathtt{return}\ f_n(\vec{x_n}) \\
\textbf{let } x = f(\vec{y}) \textbf{ in } \phi &\triangleq \exists x.\, f(\vec{y}, x); \phi \quad \textit{when } x \notin \vec{y} \\
\textbf{let } x = f(\vec{y}); \phi &\triangleq \textbf{let } x = f(\vec{y}) \textbf{ in } \phi \\
\textbf{let } w, z = f(\vec{x}) \parallel f(\vec{y}) \textbf{ in } \phi &\triangleq \exists w, z.\, f(\vec{x}, w) \parallel g(\vec{y}, z); \phi \\
\textbf{let } w, z = f(\vec{x}) \parallel f(\vec{y}); \phi &\triangleq \textbf{let } w, z = f(\vec{x}) \parallel f(\vec{y}) \textbf{ in } \phi
\end{aligned}$$

Our treatment of control flow and function return values are the most important aspects of the definitions above.

Control flow, via **if** P **then** ϕ **else** ψ **fi**, is encoded using angelic choice between the two branches, guarded by assumptions testing P. Recall that, $[P] \triangleq \{\mathsf{true}, P\}$. This means that one of the two assumptions guarding each branch in the angelic choice will fault. In that case, the faulting branch will be equivalent to `miracle` and by AChoiceId, only the behaviour of the non-faulting branch, in which the test for P succeeds, will be observed.

It is also possible to give the dual definition using demonic choice: $(\{P\}; \phi) \sqcap (\{\neg P\}; \psi)$. Recall that, $\{P\} \triangleq \{P, \mathsf{true}\}$. Here, the faulting branch will be equivalent to `abort`, and only the non-faulting branch is observed by DChoiceId. In fact, we can prove that the two encodings are equivalent:

$$\textbf{if } P \textbf{ then } \phi \textbf{ else } \psi \textbf{ fi} \equiv ([P]; \phi) \sqcup ([\neg P]; \psi) \equiv (\{P\}; \phi) \sqcap (\{\neg P\}; \psi)$$

Both encodings are standard in the refinement calculus [13].

To reason about if-then-else, we introduce the following refinement law, which corresponds the analogous rule in Hoare logic:

$$\frac{[P]; \phi_1 \sqsubseteq \psi \qquad [\neg P]; \phi_2 \sqsubseteq \psi}{\textbf{if } P \textbf{ then } \phi_1 \textbf{ else } \phi_2 \textbf{ fi} \sqsubseteq \psi} \textsc{IfThenElse}$$

In a demonic choice encoding of if-then-else, the premisses would need to use an assertion instead of an assumption.

To facilitate function return values, we add the distinguished variable `ret` at the end of the argument list of each function definition. In a function body, `return` x, simply binds the value of x to the variable `ret`. When invoking a function, the let-binding of the return value is encoded as existential quantification of the additional last argument that binds to `ret` within the function.

7.8. Conclusions

We have presented the technical details of our specification language and associated refinement calculus. Even though the specifications discussed in chapter 6 where built around atomic specification statements of the form $\mathbb{V}x \in X.\, \langle P(x),\, Q(x)\rangle$ and Hoare specification statements of the form $\{P,\, Q\}$, we have opted to a simpler core specification language based on the primitive atomic statement of the form $a(\forall \vec{x}.\, P,\, Q)$. A primitive atomic specification statement specifies a physically atomic operation. In the setting of contextual refinement it intuitively corresponds to the specification of the linearisation point of a linearisable operation.

Our core specification language is inspired by the earlier approach of Turon and Wand [94] and therefore the specification languages are similar. However, the most fundamental differences between them are the choice of assertions and their semantics as well as the semantics of the primitive atomic specification statement. Whereas Turon and Wand are using plain classical separation logic assertions and their semantics we are using the intuitionistic assertion language and semantics of the TaDA program logic [30]. This allows us to dispense with the fenced refinement approach of Turon and Wand and use more recent features of concurrent separation logics for fine-grained concurrency such as abstract predicates and client-defined separation algebras for defining protocols of concurrent interactions. Even though these differences are fundamental, we are still able to retain the overall structure of the main adequacy theorem (theorem 1).

Even though the core specification language is simple, it is also flexible as several of the constructs used in chapter 6 are actually encoded into the core language. The most important of these encoded constructs is the atomic specification statement which is captures the concept of abstract atomicity from the atomic Hoare triples of TaDA. In direct contrast to TaDA abstract atomicity is not a primitive construct. Instead, we derive abstract atomicity directly from the primitive atomicity of our core specification language. This means that primitive atomicity and abstract atomicity can coexist within the same language and reasoning system. TaDA, being a program logic, introduced a series inference rules for deriving atomic Hoare triples. These rules are now recast as refinement laws for atomic specification statements of our calculus. As atomic specification statements are encoded in terms of primitive atomic statements the soundness of most of the refinement laws for abstract atomicity is justified within the refinement calculus itself, the only exceptions being the MakeAtomic and Up-

dateRegion laws for which we also appeal to the properties of the model. In contrast the soundness of all the TaDA inference rules is justified by appealing to the underlying operational semantics. Additionally to the refinement laws corresponding to TaDA inference rules we also introduce additional laws for abstract atomicity, namely the laws for mumbling and stuttering. These allow Lipton reduction [66] style proofs for abstract atomicity, something that is not possible with the TaDA program logic.

8. Client Examples

In this chapter we demonstrate the scalability of our refinement calculus for client reasoning by considering further client examples. In section 8.1 we revisit lock files and derive a CAP-style specification that utilises ownership transfer of a resource invariant. In section 8.2 we revisit the example of the concurrent email client-server interaction from chapter 2, section 2.3.2, which highlights the importance of client reasoning based on the fine-grained concurrent specification of POSIX operations. With our formal POSIX specification and reasoning in place, we study this example formally, demonstrating that reasoning with simplified coarse-grained specifications is unreliable. In section 8.3 we present our case-study example of named pipes. We develop a specification for named pipes based on their informal specification in the POSIX standard. We then proceed to develop an implementation of named pipes within our specified file-system fragment by using regular files, and lock files. Finally, we verify the implementation utilising the CAP-style specification of lock files developed in section 8.1

8.1. CAP Style Locks

In chapter 6, section 6.2, we have introduced atomic specifications for the operations of a lock-file module that implements the behaviour of an abstract lock. The atomicity guarantee in the specification captures the essence of a lock as a synchronisation primitive, regardless of its particular use by clients. On the other hand, client applications most typically use locks to enforce mutual exclusion over a critical section of code, that non-atomically updates some shared resource. In essence, the lock, acting a mutual exclusion mechanism, protects access to the shared resource.

In separation logic based program logics, reasoning about mutual exclusion employs *ownership transfer*. If we do not assume mutual exclusion as a language primitive, as in concurrent separation logic [76], the ownership-transfer pattern is used within the specification of operations used to enforce mutual exclusion. This is the case of the lock specification using CAP by Dinsdale-Young et al. [36], which enforces exclusive ownership over a resource invariant.

In the case of our lock-file module, we can derive the following CAP-style specification for the lock-file operations:

$$
\begin{array}{l}
\mathsf{LFCtx}(lf) \vdash \\
\quad \mathtt{lock}(lf) \sqsubseteq \{\mathsf{isLock}(s, lf)\,,\ \mathsf{isLock}(s, lf) * \mathsf{Locked}(s, lf) * \mathsf{Inv}\} \\
\quad \mathtt{unlock}(lf) \sqsubseteq \{\mathsf{Locked}(s, lf) * \mathsf{Inv},\ \mathsf{true}\}
\end{array}
$$

The specification is parameterised by an abstract predicate Inv, representing the resource invariant protected by the lock. The client can choose how to instantiate the predicate as long as it is invariant under interference from the environment. The specification employs two abstract predicates itself: $\mathsf{isLock}(s, lf)$ and $\mathsf{Locked}(s, lf)$. The implementation of these abstract predicates is required to satisfy

the following axioms:

$$\mathsf{isLock}(s, lf) \iff \mathsf{isLock}(s, lf) * \mathsf{isLock}(s, lf) \qquad \mathsf{Locked}(s, lf) * \mathsf{Locked}(s, lf) \Rightarrow \mathsf{false}$$

The first axiom states that ownership of isLock can be freely duplicated. Any number of threads can own isLock. By the specification of `lock`, ownership of isLock bestows the capability to lock the lock at path lf. The second axiom states that Locked is an exclusive resource. Locked depicts that the lock is locked. Furthermore, by the specification of `unlock`, only one thread can unlock the lock. The abstract parameter $s \in \mathbb{T}_2$ captures implementation specific invariant information.

When the lock is unlocked, the resource invariant is owned by the implementation of the module, and thus is inaccessible by any thread. When the lock is locked, ownership of the resource invariant is transferred to thread that performed the lock. When the lock is unlocked, ownership of the resource invariant is returned back to the module's implementation.

In order to implement the specification we introduce the region type **CapLock**. The guard separation algebra comprises an indivisible guard named K as well as the empty guard $\mathbf{0}$. We define the guard composition operator such that $\forall x \in \{\mathbf{0}, \mathsf{K}\} \,.\, \mathbf{0} \bullet x = x$ and $\mathsf{K} \bullet \mathsf{K}$ is undefined.

The transition system for the region has two states: 0 and 1, stating that the lock is unlocked and locked respectively. We allow any thread to acquire the lock if it is unlocked, but only the thread holding the guard K is able to unlock it. This is enforced with the following transition system:

$$\mathbf{0} : 0 \rightsquigarrow 1 \qquad \mathsf{K} : 1 \rightsquigarrow 0$$

We give the region interpretation for each abstract state as follows:

$$I_r(\mathbf{CapLock}_\alpha(lf, s, 0)) \triangleq \mathsf{Lock}(s, lf, 0) * [\mathsf{K}]_\alpha * \mathsf{Inv}$$
$$I_r(\mathbf{CapLock}_\alpha(lf, s, 1)) \triangleq \mathsf{Lock}(s, lf, 1)$$

With this interpretation we guarantee that when the lock is unlocked, i.e. at abstract state 0, the guards K and Inv are in the region. When a thread acquires the lock by transitioning to abstract state 1, it removes the guard and invariant from the region. We can now give the interpretation to the abstract predicates and $\mathbb{T}_2$ as follows:

$$\mathbb{T}_2 \triangleq \textsc{RId} \times \mathbb{T}_1$$
$$\mathsf{isLock}((\alpha, s), lf) \triangleq \exists v \in \{0, 1\} \,.\, \mathbf{CapLock}_\alpha(lf, s, v)$$
$$\mathsf{Locked}((\alpha, s), lf) \triangleq \mathbf{CapLock}_\alpha(lf, s, 1) * [\mathsf{K}]_\alpha$$

The first argument of both predicates is instantiated as a pair containing the region identifier α and the logical identifier s of the underlying module. The isLock predicate states that there is a lock at path lf. The Locked predicate asserts the region must be at state 1, meaning it is locked, as guaranteed by holding the guard K.

The proofs that `lock` and `unlock` satisfy their CAP-style specification are given in figure 8.1 and figure 8.2 respectively. Throughout the proofs we assume that the context invariant LFCtx holds.

The CAP-style specification and the proofs presented here are analogous to those using the TaDA

$$
\begin{array}{l}
\texttt{lock}(lf) \\
\sqsubseteq \forall v \in \{0,1\} \,.\, \langle \mathsf{Lock}(s', lf, v)\,,\ \mathsf{Lock}(s', lf, 1) * v = 0 \rangle \\
\text{——————} \ \textsc{AFrame} : \ v = 0 \Rightarrow [\mathsf{K}]_\alpha * \mathsf{Inv},\ \textsc{ACons} \ \text{——————} \\
\sqsubseteq \forall v \in \{0,1\} \,.\, \langle \mathsf{Lock}(s', lf, v) * (v = 0 * [\mathsf{K}]_\alpha * \mathsf{Inv}) \vee v = 1,\ \mathsf{Lock}(s', lf, 1) * [\mathsf{K}]_\alpha * \mathsf{Inv} \rangle \\
\text{——————} \ \textsc{UseAtomic},\ \textsc{AEElim},\ \textsc{AWeaken2} \ \text{——————} \\
\sqsubseteq \{\exists v \in \{0,1\} \,.\, \mathbf{CapLock}_\alpha(lf, s', v),\ \mathbf{CapLock}_\alpha(lf, s', 1) * [\mathsf{K}]_\alpha * \mathsf{Inv}\} \\
\text{——————} \ \textsc{Abstract},\ s = (\alpha, s') \ \text{——————} \\
\sqsubseteq \{\mathsf{isLock}(s, lf)\,,\ \mathsf{isLock}(s, lf) * \mathsf{Locked}(s, lf) * \mathsf{Inv}\}
\end{array}
$$

Figure 8.1.: Proof that `lock` satisfies the CAP-style specification.

$$
\begin{array}{l}
\texttt{unlock}(lf) \sqsubseteq \langle \mathsf{Lock}(s', lf, 1)\,,\ \mathsf{Lock}(s', lf, 0) \rangle \\
\qquad \text{——————} \ \textsc{AFrame} \ \text{——————} \\
\qquad \langle \mathsf{Lock}(s', lf, 1) * [\mathsf{K}]_\alpha * \mathsf{Inv},\ \mathsf{Lock}(s', lf, 0) * [\mathsf{K}]_\alpha * \mathsf{Inv} \rangle \\
\qquad \text{— } \textsc{UseAtomic},\ \textsc{AWeaken2}, \text{ weaken postcondition to stabilise —} \\
\qquad \{\mathbf{CapLock}_\alpha(lf, s', 1) * [\mathsf{K}]_\alpha * \mathsf{Inv},\ \exists v \in \{0,1\} \,.\, \mathbf{CapLock}_\alpha(lf, s', v)\} \\
\qquad \text{——————} \ \textsc{Abstract}, \text{ weaken postcondition} \ \text{——————} \\
\qquad \sqsubseteq \{\mathsf{Locked}(s, lf) * \mathsf{Inv},\ \mathsf{true}\}
\end{array}
$$

Figure 8.2.: Proof that `unlock` satisfies the CAP-style specification.

program logic [30]. The difference here is that they are conditional upon the context invariant. The difference with the original CAP specification [36] is that the implementation of the lock operations does not require the use of atomic blocks.

8.2. Coarse-grained vs Fine-grained Specifications for POSIX

In chapter 2, section 2.3.2, we have informally demonstrated the importance of following the POSIX standard in specifying file-system operations in terms of multiple atomic steps by examining an example of a concurrent email client-server interaction. Having developed our formal specification and refinement calculus we now revisit the same example, given in in figure 8.3, and formally demonstrate that assuming a coarse-grained, singe-step, atomic specification for POSIX operations leads to unsafe client reasoning.

Figure 8.3 consists of an email server responsible for the delivery of an email message with identifier 42, on the right, in parallel with a simple email client testing if that message has been delivered, on the left. Initially, the undelivered email is stored at the path /mail/tmp/42.msg. The email server is responsible for delivering the message by moving it to the path /mail/42/msg.eml, but only if it has been previously deemed to be virus-free by an anti-virus scanner. To scan the message before the delivery, the email server first moves the message to the quarantine directory /mail/quarantine by sequence of renames. If the anti-virus scanner determines the message to not contain a virus, the server performs a final `rename` delivering the message.

The safety property we wish to prove is that the email client only ever reports the email message to be delivered if it is virus free. To verify this property it suffices to establish that at the end of the

```
let delivered =                  ||  mkdir(/mail/42);
  stat(/mail/42/msg.eml);        ||  rename(/mail/tmp/42.msg, /mail/42/unsafe.eml);
                                 ||  rename(/mail/42, /mail/quarantine);
                                 ||  rename(/mail/quarantine/unsafe.eml, /mail/quarantine/msg.eml);
                                 ||  let is_not_virus = run_av_scan();
                                 ||  if is_not_virus then
                                 ||    rename(/mail/quarantine, /mail/42);
                                 ||  fi
```

Figure 8.3.: Unsafe email client-server interaction.

parallel execution $\mathit{delivered} \notin \textsc{Errs} \Rightarrow \mathit{is\_not\_virus}$ holds in the postcondition. We attempt to verify this property first by assuming a simple coarse-grained atomic specification for the POSIX operations involved in section 8.2.1, and then by using the fine-grained POSIX specification developed in this dissertation in section 8.2.2. We demonstrate that using the coarse-grained file-system specification we can verify the email client-server interaction is safe, and that the same is not possible with the fine-grained POSIX semantics. In section 8.2.3 we adapt figure 8.3 such that we can prove it safe with the formal POSIX specification.

In all cases we assume the following specification for the anti-virus scanner:

$$\texttt{run\_av\_scan}() \sqsubseteq \mathbb{V}FS.\left\langle \mathsf{fs}(FS) \wedge \text{/mail/quarantine/msg.eml} \overset{FS}{\longmapsto} -,\ \mathsf{fs}(FS) * \texttt{ret} \in \mathbb{B} \right\rangle$$

The precondition requires the path /mail/quarantine/msg.eml to exist in the file system. The post-condition specifies that the operation non-deterministically returns a boolean without modifying the file system. For the purposes of this example we are not interested in the details of virus scanning.

In order to keep the reasoning simple in sections 8.2.1 and 8.2.2, and focus on the safety of the email client-server interaction, we assume that the file system is only accessed by the client and server threads. We lift this restriction in section 8.2.3, to demonstrate reasoning about this example in contexts where the file system is shared with the environment.

8.2.1. Behaviour with Coarse-grained Specifications

We assume the coarse-grained atomic specification in figure 8.4 for the file-system operations used in the example. For simplicity, the specifications of `stat` and `mkdir` are given for the concrete path arguments that we use. In the specification of `stat` we abstract the error cases of resolving the given path into a single case. In `mkdir` and `rename` we ignore the error cases altogether, since they always succeed in this example.

Assuming the coarse-grained specification, we can prove that the parallel email client-server interaction of figure 8.3 is a refinement of the following specification statement:

$$\left\{\begin{array}{l} \mathsf{fs}(FS) \wedge \text{/mail/tmp/42.msg} \overset{FS}{\longmapsto} \iota \wedge \mathsf{isfile}(FS(\iota)) \wedge \neg\text{/mail/42} \overset{FS}{\longmapsto} - \wedge \neg\text{/mail/quarantine} \overset{FS}{\longmapsto} -, \\ \mathsf{fs}(-) * \mathit{delivered} \notin \textsc{Errs} \Rightarrow \mathit{is\_not\_virus} \end{array}\right\}$$

The precondition describes the initial state of the file-system where the email message of interest is

$$\texttt{stat}(/\text{mail}/42/\text{msg.eml}) \sqsubseteq$$
$$\quad \forall FS, \iota. \left\langle \begin{array}{l} \mathsf{fs}(FS) \wedge /\text{mail}/42/\text{msg.eml} \xmapsto{FS} \iota, \\ \mathsf{fs}(FS) \wedge /\text{mail}/42/\text{msg.eml} \xmapsto{FS} \iota * \texttt{ret} = \mathsf{ftype}(FS(\iota)) \end{array} \right\rangle$$
$$\sqcap\ \forall FS. \left\langle \mathsf{fs}(FS) \wedge \neg/\text{mail}/42/\text{msg.eml} \xmapsto{FS} -,\ \mathsf{fs}(FS) \wedge \neg/\text{mail}/42/\text{msg.eml} \xmapsto{FS} - * \texttt{ret} \in \textsc{Errs} \right\rangle$$

$$\texttt{mkdir}(/\text{mail}/42) \sqsubseteq$$
$$\quad \forall FS, \iota. \left\langle \mathsf{fs}(FS) \wedge /\text{mail} \xmapsto{FS} \iota \wedge 42 \notin FS(\iota),\ \exists \iota'. \mathsf{fs}(FS[\iota \mapsto FS(\iota)[42 \mapsto \iota']][\iota' \mapsto \emptyset]) \wedge /\text{mail} \xmapsto{FS} \iota \right\rangle$$

$$\texttt{rename}(p/a, p'/b) \sqsubseteq$$
$$\quad \forall FS, \iota. \left\langle \begin{array}{l} \mathsf{fs}(FS) \wedge p \xmapsto{FS} \iota \wedge \mathsf{isfile}(FS(\iota)(a)) \wedge p' \xmapsto{FS} \iota' \wedge b \notin FS(\iota'), \\ \mathsf{fs}(FS[\iota \mapsto FS(\iota) \setminus \{a\}][\iota' \mapsto FS(\iota')[b \mapsto FS(\iota)(a)]]) * \texttt{ret} = 0 \end{array} \right\rangle$$
$$\sqcap\ \forall FS, \iota. \left\langle \begin{array}{l} \mathsf{fs}(FS) \wedge p \xmapsto{FS} \iota \wedge \mathsf{isdir}(FS(\iota)(a)) \wedge p' \xmapsto{FS} \iota' \wedge \iota' \notin \mathsf{descendants}(\iota) \wedge b \notin FS(\iota'), \\ \mathsf{fs}(FS[\iota \mapsto FS(\iota) \setminus \{a\}][\iota' \mapsto FS(\iota')[b \mapsto FS(\iota)(a)]]) * \texttt{ret} = 0 \end{array} \right\rangle$$

Figure 8.4.: Coarse-grained specification for the operations used in figure 8.3.

undelivered. The postcondition states that if the email client reports the message to be delivered (*delivered* $\notin$ Errs), then the server has declared to be virus free (*is_not_virus* is true).

In order to verify that the parallel client-server interaction is indeed a refinement of the specification above, we introduce the region type **Eml**. Regions of this type are parameterised by the inode of the file storing the email message of interest. We use integers between 0 and 6 (inclusive) as abstract states of **Eml** regions, with the following interpretation:

$$\begin{array}{l}
I_r(\mathbf{Eml}_\alpha(\iota, 1)) \triangleq \exists FS.\, \mathsf{fs}(FS) \wedge /\text{mail}/\text{tmp}/42.\text{msg} \xmapsto{FS} \iota \\
\qquad\qquad \wedge\, \mathsf{isfile}(FS(\iota)) \wedge \neg/\text{mail}/42 \xmapsto{FS} - \wedge \neg/\text{mail}/\text{quarantine} \xmapsto{FS} - \\
I_r(\mathbf{Eml}_\alpha(\iota, 2)) \triangleq \exists FS.\, \mathsf{fs}(FS) \wedge /\text{mail}/\text{tmp}/42.\text{msg} \xmapsto{FS} \iota \\
\qquad\qquad \wedge\, \mathsf{isfile}(FS(\iota)) \wedge \exists j.\, /\text{mail}/42 \xmapsto{FS} j \wedge \mathsf{isdir}(FS(j)) \wedge \neg/\text{mail}/\text{quarantine} \xmapsto{FS} - \\
I_r(\mathbf{Eml}_\alpha(\iota, 3)) \triangleq \exists FS.\, \mathsf{fs}(FS) \wedge \neg/\text{mail}/\text{tmp}/42.\text{msg} \xmapsto{FS} - \\
\qquad\qquad \wedge\, /\text{mail}/42/\text{unsafe.eml} \xmapsto{FS} \iota \wedge \mathsf{isfile}(FS(\iota)) \wedge \neg/\text{mail}/\text{quarantine} \xmapsto{FS} - \\
I_r(\mathbf{Eml}_\alpha(\iota, 4)) \triangleq \exists FS.\, \mathsf{fs}(FS) \wedge /\text{mail}/\text{quarantine}/\text{unsafe.eml} \xmapsto{FS} \iota \wedge \mathsf{isfile}(FS(\iota)) \wedge \neg/\text{mail}/42 \xmapsto{FS} - \\
I_r(\mathbf{Eml}_\alpha(\iota, 5)) \triangleq \exists FS.\, \mathsf{fs}(FS) \wedge /\text{mail}/\text{quarantine}/\text{msg.eml} \xmapsto{FS} \iota \wedge \mathsf{isfile}(FS(\iota)) \wedge \neg/\text{mail}/42 \xmapsto{FS} - \\
I_r(\mathbf{Eml}_\alpha(\iota, 6)) \triangleq \exists FS.\, \mathsf{fs}(FS) \wedge /\text{mail}/42/\text{msg.eml} \xmapsto{FS} \iota \wedge \mathsf{isfile}(FS(\iota)) \wedge \neg/\text{mail}/\text{quarantine} \xmapsto{FS} - \\
I_r(\mathbf{Eml}_\alpha(\iota, 0)) \triangleq \mathsf{true}
\end{array}$$

We use abstract states between 1 and 6 to denote the different states in which the email server can be in. State 1 corresponds to the file-system state at the beginning. State 2 corresponds to the file-system state after the `mkdir` performed by the email server thread and so forth. State 6 corresponds to the file-system state where the email message has been delivered. The abstract state 0 is used to mark the end of both the email client and email server. Once in the abstract state 0, the file system is no longer shared between the client and server threads. We interpret this state as true, so that the file

system can be removed from the shared region.

We define the guards G, R, W and F for **Eml** regions. We define the guard algebra such that it satisfies the equation:

$$\mathsf{F} = \mathsf{R} \bullet \mathsf{W}$$

Lastly, we define the following labelled transition system for **Eml** regions:

$$\mathsf{G} : 1 \rightsquigarrow 2 \qquad \mathsf{G} : 2 \rightsquigarrow 3 \qquad \mathsf{G} : 3 \rightsquigarrow 4 \qquad \mathsf{G} : 4 \rightsquigarrow 5 \qquad \mathsf{G} : 5 \rightsquigarrow 6 \qquad \mathsf{F} : 5 \rightsquigarrow 0 \qquad \mathsf{F} : 6 \rightsquigarrow 0$$

Ownership of the G grants the capability to perform the updates of the email server. Transitioning to the final state 0 requires ownership of the F guard.

$$\begin{array}{l|\!|l}
\textbf{let } \mathit{delivered} = & \texttt{mkdir}(/\text{mail}/42); \\
\quad \texttt{stat}(/\text{mail}/42/\text{msg.eml}); & \texttt{rename}(/\text{mail}/\text{tmp}/42.\text{msg}, /\text{mail}/42/\text{unsafe.eml}); \\
& \texttt{rename}(/\text{mail}/42, /\text{mail}/\text{quarantine}); \\
& \texttt{rename}(/\text{mail}/\text{quarantine}/\text{unsafe.eml}, /\text{mail}/\text{quarantine}/\text{msg.eml}); \\
& \textbf{let } \mathit{is_not_virus} = \texttt{run_av_scan}(); \\
& \textbf{if } \mathit{is_not_virus} \textbf{ then} \\
& \quad \texttt{rename}(/\text{mail}/\text{quarantine}, /\text{mail}/42); \\
& \textbf{fi}
\end{array}$$

$\sqsubseteq$ by figure 8.6, figure 8.7, Parallel and HCons

$$\left\{ \begin{array}{l} \left(\exists n \in \mathbb{N}_1^6.\, \mathbf{Eml}_\alpha(\iota, n) * [\mathsf{R}]_\alpha\right) * \left(\mathbf{Eml}_\alpha(\iota, 1) * [\mathsf{G}]_\alpha * [\mathsf{W}]_\alpha\right), \\ \left(\exists n \in \mathbb{N}_1^6.\, \mathbf{Eml}_\alpha(\iota, n) * [\mathsf{R}]_\alpha * \mathit{delivered} \notin \textsc{Errs} \Rightarrow n = 6\right) \\ * \left(\exists n \in \mathbb{N}_5^6.\, \mathbf{Eml}_\alpha(\iota, n) * [\mathsf{G}]_\alpha * [\mathsf{W}]_\alpha * n = 6 \Rightarrow \mathit{is\_not\_virus}\right) \end{array} \right\}$$

$\sqsubseteq$ by HCons

$$\left\{\mathbf{Eml}_\alpha(\iota, 1) * [\mathsf{G}]_\alpha * [\mathsf{F}]_\alpha\,,\ \exists n \in \mathbb{N}_5^6.\, \mathbf{Eml}_\alpha(\iota, n) * [\mathsf{G}]_\alpha * [\mathsf{F}]_\alpha * \mathit{delivered} \notin \textsc{Errs} \Rightarrow \mathit{is\_not\_virus}\right\}$$

$\sqsubseteq$ by Skip

$$\left\{\mathbf{Eml}_\alpha(\iota, 1) * [\mathsf{G}]_\alpha * [\mathsf{F}]_\alpha\,,\ \exists n \in \mathbb{N}_5^6.\, \mathbf{Eml}_\alpha(\iota, n) * [\mathsf{G}]_\alpha * [\mathsf{F}]_\alpha * \mathit{delivered} \notin \textsc{Errs} \Rightarrow \mathit{is\_not\_virus}\right\};$$

Seq

$\texttt{skip} \sqsubseteq$ by Primitive, AWeaken2 and HFrame

$$\left\{ \begin{array}{l} \exists n \in \mathbb{N}_5^6.\, I_r(\mathbf{Eml}_\alpha(\iota, n)) * [\mathsf{G}]_\alpha * [\mathsf{F}]_\alpha * \mathit{delivered} \notin \textsc{Errs} \Rightarrow \mathit{is\_not\_virus}, \\ \exists n \in \mathbb{N}_5^6.\, I_r(\mathbf{Eml}_\alpha(\iota, n)) * [\mathsf{G}]_\alpha * [\mathsf{F}]_\alpha * \mathit{delivered} \notin \textsc{Errs} \Rightarrow \mathit{is\_not\_virus} \end{array} \right\}$$

$\sqsubseteq$ by HCons

$$\left\{ \begin{array}{l} \exists n \in \mathbb{N}_5^6.\, I_r(\mathbf{Eml}_\alpha(\iota, n)) * [\mathsf{G}]_\alpha * [\mathsf{F}]_\alpha * \mathit{delivered} \notin \textsc{Errs} \Rightarrow \mathit{is\_not\_virus}, \\ I_r(\mathbf{Eml}_\alpha(\iota, 0)) * \mathsf{fs}(-) * [\mathsf{G}]_\alpha * [\mathsf{F}]_\alpha * \mathit{delivered} \notin \textsc{Errs} \Rightarrow \mathit{is\_not\_virus} \end{array} \right\}$$

$\sqsubseteq$ by UseAtomic

$$\left\{ \begin{array}{l} \exists n \in \mathbb{N}_5^6.\, \mathbf{Eml}_\alpha(\iota, n) * [\mathsf{G}]_\alpha * [\mathsf{F}]_\alpha * \mathit{delivered} \notin \textsc{Errs} \Rightarrow \mathit{is\_not\_virus}, \\ \mathbf{Eml}_\alpha(\iota, 0) * [\mathsf{G}]_\alpha * [\mathsf{F}]_\alpha * \mathsf{fs}(-) * \mathit{delivered} \notin \textsc{Errs} \Rightarrow \mathit{is\_not\_virus} \end{array} \right\}$$

$\sqsubseteq \{\mathbf{Eml}_\alpha(\iota, 1) * [\mathsf{G}]_\alpha * [\mathsf{F}]_\alpha\,,\ \mathbf{Eml}_\alpha(\iota, 0) * [\mathsf{G}]_\alpha * [\mathsf{F}]_\alpha * \mathsf{fs}(-) * \mathit{delivered} \notin \textsc{Errs} \Rightarrow \mathit{is\_not\_virus}\}$

$\sqsubseteq$ by EElimHoare

$$\{\exists \alpha.\, \mathbf{Eml}_\alpha(\iota, 1) * [\mathsf{G}]_\alpha * [\mathsf{F}]_\alpha\,,\ \exists \alpha.\, \mathbf{Eml}_\alpha(\iota, 0) * [\mathsf{G}]_\alpha * [\mathsf{F}]_\alpha * \mathsf{fs}(-) * \mathit{delivered} \notin \textsc{Errs} \Rightarrow \mathit{is\_not\_virus}\}$$

$\sqsubseteq$ by CreateRegion and HCons

$$\left\{ \begin{array}{l} \mathsf{fs}(FS) \wedge /\text{mail}/\text{tmp}/42.\text{msg} \xmapsto{FS} \iota \wedge \mathsf{isfile}(FS(\iota)) \wedge \neg /\text{mail}/42 \xmapsto{FS} - \wedge \neg /\text{mail}/\text{quarantine} \xmapsto{FS} -, \\ \mathsf{fs}(-) * \mathit{delivered} \notin \textsc{Errs} \Rightarrow \mathit{is\_not\_virus} \end{array} \right\}$$

Figure 8.5.: Proof of email client-server interaction using a coarse-grained file-system specification.

In figure 8.5 we prove that the parallel composition of the email client and server is a refinement of the specification we have given at the beginning of the current subsection. We explain the refinement

steps by starting with the specification at the bottom and going up to the implementation at the top.

The first refinement step involves creating a new **Eml** shared region to hold the file system such that shared access to the file system is performed through this region. This refinement is justified by HCons, where in the precondition we use the CreateRegion view-shift to create a new **Eml** shared region in the initial abstract state 1 together with the G and F guards. At the same time we strengthen the postcondition by introducing the region and its guards in the distinguished final abstract state 0. Thus we have refined the specification to an update of an **Eml** shared region from abstract state 1 to abstract state 0.

In the second refinement step we simply apply EElimHoare to eliminate the existential quantification over the region identifier α. Thus in the rest of the derivation the region identifier is fixed.

The third refinement step involves refining the update to **Eml** into an update from abstract state 1 to abstract states 5 or 6. The latter states are the final abstract states of the server thread. This refinement is justified by a subderivation, in which we use Seq to split the update into two steps: the first updates 1 to either 5 or 6, and the second updates 5 and 6 to 0. This second step is refined to `skip` in a further subderivation. In this subderivation, we first apply UseAtomic to refine the update on the **Eml** to an update on its interpretation. Then we apply HCons strengthening the postcondition to the interpretation of abstract states 5 or 6. We conclude the subderivation by framing everything off with HFrame after which AWeaken2 followed by Primitive refine directly to `skip`.

In the next refinement step we take advantage of the second sequent refining to `skip` and apply Skip to drop it. At this point we have refined the original specification to an update on the **Eml** shared region from abstract state 1 to abstract states 5 or 6. Furthermore, up to this point we were able to thread the implication $\mathit{delivered} \notin \textsc{Errs} \Rightarrow \mathit{is\_not\_virus}$ through the refinement in the postcondition.

We proceed by applying HCons to partition the precondition and postcondition into two disjoint parts, such that ownership of the two parts can be split between the client and server in accordance to the Parallel refinement law. In particular, we duplicate the **Eml** region, weakening the abstract state of the client's copy in the precondition such that it is stable with respect to the server's updates. Additionally, we split the F guard into the R guard, given to the client, and the W guard, given to the server. This prevents any thread to update the region to the final 0 state by using the F guard. Note that we also split the original implication into two parts: $\mathit{delivered} \notin \textsc{Errs} \Rightarrow n = 6$ established by the client, and $n = 6 \Rightarrow \mathit{is\_not\_virus}$ established by the server.

We then proceed by applying Parallel, where the refinements for the client and server threads are given in figures 8.6 and 8.7.

$$
\begin{array}{l}
\texttt{stat}(/\text{mail}/42/\text{msg.eml}) \\
\quad \sqsubseteq \text{ by figure 8.4, DChoiceIntro and ACons} \\
\quad\quad \forall FS.\, \left\langle \mathsf{fs}(FS)\,,\ \mathsf{fs}(FS) * \texttt{ret} \notin \textsc{Errs} \Rightarrow /\text{mail}/42/\text{msg.eml} \xmapsto{FS} - \right\rangle \\
\quad \sqsubseteq \text{ by AEElim, ACons, AFrame and OpenRegion} \\
\quad\quad \forall n \in \mathbb{N}_1^6.\, \langle \mathbf{Eml}_\alpha(n) * [\mathsf{R}]_\alpha\,,\ \mathbf{Eml}_\alpha(n) * [\mathsf{R}]_\alpha * \texttt{ret} \notin \textsc{Errs} \Rightarrow n = 6 \rangle \\
\quad \sqsubseteq \text{ by AEElim and AWeaken2} \\
\quad\quad \left\{ \exists n \in \mathbb{N}_1^6.\, \mathbf{Eml}_\alpha(n) * [\mathsf{R}]_\alpha\,,\ \exists n \in \mathbb{N}_1^6.\, \mathbf{Eml}_\alpha(n) * [\mathsf{R}]_\alpha * \texttt{ret} \notin \textsc{Errs} \Rightarrow n = 6 \right\}
\end{array}
$$

Figure 8.6.: Email client refinement using a coarse-grained file-system specification.

Consider the refinement for the email client in figure 8.6. We begin by using AWeaken2 to refine the Hoare specification statement to an atomic specification statement, and AEElim to turn the existential quantification on the region's abstract state to a pseudo-universal quantification. In the next step we use OpenRegion to open the **Eml** region, AFrame to frame-off the R guard, ACons to simplify the file-system state assertions, and finally AEElim to eliminate the existential quantification over the file-system graphs that we obtain from the **Eml** region's interpretation. In the final refinement step we use ACons to reach the first demonic case of `stat`'s specification in figure 8.4 and then DChoiceIntro to introduce the second case.

Seq

`mkdir`(/mail/42);

$\sqsubseteq$ by figure 8.4, ACons and AEElim

$\left\langle \exists FS, j.\, \mathsf{fs}(FS) \wedge /\text{mail} \xmapsto{FS} j \wedge \neg /\text{mail/42} \xmapsto{FS} -,\ \exists FS, j.\, \mathsf{fs}(FS) \wedge /\text{mail/42} \xmapsto{FS} j \right\rangle$

$\sqsubseteq$ by ACons, AFrame and UseAtomic

$\langle \mathbf{Eml}_\alpha(\iota, 1) * [\mathsf{G}]_\alpha * [\mathsf{W}]_\alpha\,,\ \mathbf{Eml}_\alpha(\iota, 2) * [\mathsf{G}]_\alpha * [\mathsf{W}]_\alpha \rangle$

$\sqsubseteq$ by AWeaken2

$\{\mathbf{Eml}_\alpha(\iota, 1) * [\mathsf{G}]_\alpha * [\mathsf{W}]_\alpha\,,\ \mathbf{Eml}_\alpha(\iota, 2) * [\mathsf{G}]_\alpha * [\mathsf{W}]_\alpha\}$

`rename`(/mail/tmp/42.msg, /mail/42/unsafe.eml);

$\sqsubseteq$ by figure 8.4, ACons, AEElim, AFrame, UseAtomic and AWeaken2

$\{\mathbf{Eml}_\alpha(\iota, 2) * [\mathsf{G}]_\alpha * [\mathsf{W}]_\alpha\,,\ \mathbf{Eml}_\alpha(\iota, 3) * [\mathsf{G}]_\alpha * [\mathsf{W}]_\alpha\}$

`rename`(/mail/42, /mail/quarantine);

$\sqsubseteq$ by figure 8.4, ACons, AEElim, AFrame, UseAtomic and AWeaken2

$\{\mathbf{Eml}_\alpha(\iota, 3) * [\mathsf{G}]_\alpha * [\mathsf{W}]_\alpha\,,\ \mathbf{Eml}_\alpha(\iota, 4) * [\mathsf{G}]_\alpha * [\mathsf{W}]_\alpha\}$

`rename`(/mail/quarantine/unsafe.eml, /mail/quarantine/msg.eml);

$\sqsubseteq$ by figure 8.4, ACons, AEElim, AFrame, UseAtomic and AWeaken2

$\{\mathbf{Eml}_\alpha(\iota, 4) * [\mathsf{G}]_\alpha * [\mathsf{W}]_\alpha\,,\ \mathbf{Eml}_\alpha(\iota, 5) * [\mathsf{G}]_\alpha * [\mathsf{W}]_\alpha\}$

let *is_not_virus* = `run_av_scan`();

$\sqsubseteq$ by specification, AEElim, ACons, AFrame, OpenRegion and AWeaken2

$\{\mathbf{Eml}_\alpha(\iota, 5) * [\mathsf{G}]_\alpha * [\mathsf{W}]_\alpha\,,\ \mathbf{Eml}_\alpha(\iota, 5) * [\mathsf{G}]_\alpha * [\mathsf{W}]_\alpha * \mathit{is\_not\_virus} \in \mathbb{B}\}$

IfThenElse

if *is_not_virus* **then**

`rename`(/mail/quarantine, /mail/42);

$\sqsubseteq$ by figure 8.4, AEElim, ACons, AFrame, UseAtomic and AWeaken2

$\{\mathbf{Eml}_\alpha(\iota, 5) * [\mathsf{G}]_\alpha * [\mathsf{W}]_\alpha * \mathit{is\_not\_virus},\ \mathbf{Eml}_\alpha(\iota, 6) * [\mathsf{G}]_\alpha * [\mathsf{W}]_\alpha * \mathit{is\_not\_virus}\}$

fi

$\sqsubseteq \left\{ \begin{array}{l} \mathbf{Eml}_\alpha(\iota, 5) * [\mathsf{G}]_\alpha * [\mathsf{W}]_\alpha * \mathit{is\_not\_virus} \in \mathbb{B}, \\ \exists n \in \mathbb{N}^6_5.\, \mathbf{Eml}_\alpha(\iota, n) * [\mathsf{G}]_\alpha * [\mathsf{W}]_\alpha * n = 6 \Rightarrow \mathit{is\_not\_virus} \end{array} \right\}$

$\sqsubseteq \{\mathbf{Eml}_\alpha(\iota, 1) * [\mathsf{G}]_\alpha * [\mathsf{W}]_\alpha\,,\ \exists n \in \mathbb{N}^6_5.\, \mathbf{Eml}_\alpha(\iota, n) * [\mathsf{G}]_\alpha * [\mathsf{W}]_\alpha * n = 6 \Rightarrow \mathit{is\_not\_virus}\}$

Figure 8.7.: Proof of email server assuming a coarse-grained specification.

Now consider the refinement for the email server in figure 8.7. Here we apply Seq over the sequence of operations comprising the server. For the first operation, `mkdir`, we show that it refines an update to the **Eml** from state 1 to state 2. First, we use AWeaken2 to refine the Hoare specification statement into an atomic specification statement. Next, we use UseAtomic to refine the update on the **Eml** region to an update on the file-system state, followed by AFrame to frame-off the G and W guards and elements of the region's interpretation we do not need for the update. Last, we use AEElim to eliminate the existential quantification on the file-system graphs updated, and ACons to refine into

the coarse-grained `mkdir` specification in figure 8.4. The refinements for the subsequent operations follows the same pattern, and thus are given in less detail.

8.2.2. Behaviour with POSIX Specifications

In chapter 6, section 6.1, we have given a formal specification to the POSIX `mkdir` and `rename` operations. The specification of POSIX `stat`, given in figure 8.8. follows the same pattern as the specifications examined in chapter 6.

$$
\begin{array}{l}
\texttt{stat}(\mathit{path}) \\
\quad \sqsubseteq \textbf{let } p = \texttt{dirname}(\mathit{path}); \\
\qquad \textbf{let } a = \texttt{basename}(\mathit{path}); \\
\qquad \textbf{let } r = \texttt{resolve}(p, \iota_0); \\
\qquad \textbf{if } \neg \mathsf{iserr}(r) \textbf{ then} \\
\qquad\quad \texttt{return link\_stat}(r, a) \\
\qquad \textbf{else return } r \textbf{ fi}
\end{array}
$$

$$
\begin{array}{l}
\textbf{let } \texttt{link\_stat}(\iota, a) \triangleq \forall FS.\, \langle \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota))\,,\ a \in FS(\iota) \Rightarrow \mathsf{fs}(FS) * \texttt{ret} = \mathsf{ftype}(FS(\iota)(a)) \rangle \\
\qquad \sqcap \texttt{return enotdir}(\iota) \\
\qquad \sqcap \texttt{return enoent}(\iota, a)
\end{array}
$$

Figure 8.8.: POSIX specification of `stat`.

Following the actual POSIX specification, we are unable to show that the email client-server interaction satisfies the same specification as when we assume the simpler coarse-grained specification. In the coarse-grained specification of figure 8.4 paths are resolved atomically and path resolution is not subject to interference; the file-system graph does not change during path resolution. This has allowed us to prove in the previous section that the client's `stat` only succeeds after the server's `rename`(/mail/quarantine, /mail/42). However, in the actual POSIX specification path resolution is a sequence of atomic lookups, and thus it is subject to interference; the file-system graph may change during path resolution. As a result, we can prove that the client's `stat` may also succeed after the earlier `rename`(/mail/quarantine/unsafe.eml, /mail/quarantine/msg.eml).

In fact, the actual specification that the email client-server interaction satisfies in POSIX is the following:

$$
\left\{
\begin{array}{l}
\mathsf{fs}(FS) \wedge /\mathrm{mail/tmp/42.msg} \xmapsto{FS} \iota \wedge \mathsf{isfile}(FS(\iota)) \wedge \neg /\mathrm{mail/42} \xmapsto{FS} - \wedge \neg /\mathrm{mail/quarantine} \xmapsto{FS} -, \\
\mathsf{fs}(-) * \mathit{delivered} \notin \textsc{Errs} \Rightarrow \mathit{is\_not\_a\_virus} \in \mathbb{B}
\end{array}
\right\}
$$

The precondition is the same as in the coarse-grained setting. However, the postcondition is weaker. Knowledge that the email client observes the message of interest to have been delivered ($\mathit{delivered} \notin$ Errs), does not have any bearing on whether it has been deemed virus free or not. In fact, the assertion $\mathit{delivered} \notin \textsc{Errs} \Rightarrow \mathit{is\_not\_a\_virus} \in \mathbb{B}$ is a tautology since the consequent is always true in this example. We use this assertion in the postcondition nonetheless to highlight the difference.

In order to show that the parallel email client-server interaction is a refinement of the specification above, we introduce a new region type: **UnsafeEml**. Due to the fine-grained nature of the POSIX

specification, this region type is necessarily more complex than **Eml**. In the coarse-grained setting path resolution is not interfered with and thus we can base our reasoning on the existence or non-existence of paths. In contrast, in the fine-grained setting path resolution is subject to interference. Therefore, to reason about path resolution we need to track information that is not interfered with: inodes.

Regions of type **UnsafeEml** are parameterised by three inodes: the inode ι of the file storing the email of interest, the inode j of the /mail directory, and the inode k of the /mail/tmp directory. Abstract states of **UnsafeEml** are pairs taken from the set $(\mathbb{N}_2^6 \times \textsc{Inodes}) \cup (\{0\} \times \textsc{Inodes}) \cup \{(1, \bot)\}$. The first element of the pair is a natural number from 0 to 6 as in **Eml**, where number 1 denotes the initial file-system state, states 2 to 6 denote the file-system states resulting from the email server's actions, and 0 denotes the final state where the region is no longer needed. The second element is used to record the inode of the directory that is created with `mkdir`(/mail/42), where $\bot$ denotes the fact that the directory has yet to be created. The path to this directory is subject to change by the server, and thus we use the inode to refer to it during the resolution of the path /mail/42/msg.eml. The abstract states of **UnsafeEml** are given the following interpretation:

$$
\begin{aligned}
I_r(\mathbf{UnsafeEml}_\alpha(\iota, j, k, (1, \bot))) \triangleq\ & \exists FS.\, \mathsf{fs}(FS) \land FS(\iota_0)(\text{mail}) = j \land FS(j)(\text{tmp}) = k \\
& \land FS(k)(42.\text{msg}) = \iota \land \mathsf{isfile}(FS(\iota)) \land 42 \notin FS(j) \\
& \land \text{quarantine} \notin FS(j) \\
I_r(\mathbf{UnsafeEml}_\alpha(\iota, j, k, (2, l))) \triangleq\ & \exists FS.\, \mathsf{fs}(FS) \land FS(\iota_0)(\text{mail}) = j \land FS(j)(\text{tmp}) = k \\
& \land FS(k)(42.\text{msg}) = \iota \land \mathsf{isfile}(FS(\iota)) \land FS(j)(42) = l \\
& \land \text{unsafe.eml} \notin FS(l) \land \text{msg.eml} \notin FS(l) \land \text{quarantine} \notin FS(j) \\
I_r(\mathbf{UnsafeEml}_\alpha(\iota, j, k, (3, l))) \triangleq\ & \exists FS.\, \mathsf{fs}(FS) \land FS(\iota_0)(\text{mail}) = j \land FS(j)(\text{tmp}) = k \\
& \land 42.\text{msg} \notin FS(k) \land FS(j)(42) = l \land FS(l)(\text{unsafe.eml}) = \iota \\
& \land \mathsf{isfile}(FS(\iota)) \land \text{msg.eml} \notin FS(l) \land \text{quarantine} \notin FS(j) \\
I_r(\mathbf{UnsafeEml}_\alpha(\iota, j, k, (4, l))) \triangleq\ & \exists FS.\, \mathsf{fs}(FS) \land FS(\iota_0)(\text{mail}) = j \land FS(j)(\text{tmp}) = k \\
& \land FS(j)(\text{quarantine}) = l \land FS(l)(\text{unsafe.eml}) = \iota \\
& \land \mathsf{isfile}(FS(\iota)) \land \text{msg.eml} \notin FS(l) \land 42 \notin FS(j) \\
I_r(\mathbf{UnsafeEml}_\alpha(\iota, j, k, (5, l))) \triangleq\ & \exists FS.\, \mathsf{fs}(FS) \land FS(\iota_0)(\text{mail}) = j \land FS(j)(\text{tmp}) = k \\
& \land FS(j)(\text{quarantine}) = l \land FS(l)(\text{msg.eml}) = \iota \\
& \land \mathsf{isfile}(FS(\iota)) \land 42 \notin FS(j) \\
I_r(\mathbf{UnsafeEml}_\alpha(\iota, j, k, (6, l))) \triangleq\ & \exists FS.\, \mathsf{fs}(FS) \land FS(\iota_0)(\text{mail}) = j \land FS(j)(\text{tmp}) = k \\
& \land FS(j)(42) = l \land FS(l)(\text{msg.eml}) = \iota \\
& \land \mathsf{isfile}(FS(\iota)) \land \text{quarantine} \notin FS(j) \\
I_r(\mathbf{UnsafeEml}_\alpha(\iota, j, k, (0, l))) \triangleq\ & \mathsf{true}
\end{aligned}
$$

We define the guards G, R, W and F for **UnsafeEml**, similarly to **Eml** except that the guard G is parameterised by a permission $\pi \in (0, 1]$. The guard algebra is defined as follows:

$$
\mathsf{F} = \mathsf{R} \bullet \mathsf{W} \qquad\qquad \forall \pi_1, \pi_2.\, \pi_1 + \pi_2 \leq 1 \land \mathsf{G}\,(\pi_1 + \pi_2) = \mathsf{G}\,(\pi_1) \bullet \mathsf{G}\,(\pi_2)
$$

The labelled transition system for **UnsafeEml** is defined analogously to that of **Eml**:

$$\mathsf{G}\,(1) : (1, \bot) \rightsquigarrow \exists \iota.(2, \iota) \qquad \mathsf{G}\,(1) : \forall \iota \in \textsc{Inodes}.\,(2, \iota) \rightsquigarrow (3, \iota) \qquad \mathsf{G}\,(1) : \forall \iota \in \textsc{Inodes}.\,(3, \iota) \rightsquigarrow (4, \iota)$$

$$\mathsf{G}\,(1) : \forall \iota \in \textsc{Inodes}.\,(4, \iota) \rightsquigarrow (5, \iota) \qquad \mathsf{G}\,(1) : \forall \iota \in \textsc{Inodes}.\,(5, \iota) \rightsquigarrow (6, \iota)$$

$$\mathsf{F} : \forall \iota \in \textsc{Inodes}.\,(5, \iota) \rightsquigarrow (0, \iota) \qquad \mathsf{F} : \forall \iota \in \textsc{Inodes}.\,(6, \iota) \rightsquigarrow (0, \iota)$$

We prove that according to the POSIX file-system specification the concurrent email client-server interaction satisfies its specification in figure 8.9. The proof for the parallel composition follows a similar pattern to that of figure 8.5 which assumes the coarse-grained file-system specification. Reading the derivation from bottom to top, in the first refinement step we use HCons to split the path structures in the precondition into individual links mapping filenames to inodes. In the next step we apply EElimHoare to eliminate the existential quantification on the inodes j and k, and then apply HCons and CreateRegion in the same manner as in figure 8.5 to refine the update on the file system to an update on the **UnsafeEml** shared region. We simplify the justification of this refinement step by omitting the subderivation used in figure 8.5. We then proceed by applying

Let $\mathcal{D} \triangleq \mathbb{N}_1^6 \times (\{\bot\} \cup \textsc{Inodes})$.

$$\begin{array}{l}
\textbf{let } \mathit{delivered} = \\
\quad \texttt{stat}(/\text{mail}/42/\text{msg.eml}); \\
\end{array}
\;\Big\|\;
\begin{array}{l}
\texttt{mkdir}(/\text{mail}/42); \\
\texttt{rename}(/\text{mail}/\text{tmp}/42.\text{msg}, /\text{mail}/42/\text{unsafe.eml}); \\
\texttt{rename}(/\text{mail}/42, /\text{mail}/\text{quarantine}); \\
\texttt{rename}(/\text{mail}/\text{quarantine}/\text{unsafe.eml}, /\text{mail}/\text{quarantine}/\text{msg.eml}); \\
\textbf{let } \mathit{is_not_virus} = \texttt{run_av_scan}(); \\
\textbf{if } \mathit{is_not_virus} \textbf{ then} \\
\quad \texttt{rename}(/\text{mail}/\text{quarantine}, /\text{mail}/42); \\
\textbf{fi}
\end{array}$$

$\sqsubseteq$ by figure 8.10, figure 8.12, Parallel and HCons

$$\left\{\begin{array}{l}
\left(\exists (n, l) \in \mathbb{N}_1^6 \times \mathcal{D}.\, \mathbf{UnsafeEml}_\alpha(\iota, j, k, (n, l)) * [\mathsf{R}]_\alpha\right) \\
* \left(\mathbf{UnsafeEml}_\alpha(\iota, j, k, (1, \bot)) * [\mathsf{G}(1)]_\alpha * [\mathsf{W}]_\alpha\right), \\
\left(\exists (n, l) \in \mathbb{N}_1^6 \times \mathcal{D}.\, \mathbf{UnsafeEml}_\alpha(\iota, j, (n, l)) * [\mathsf{R}]_\alpha * \mathit{delivered} \notin \textsc{Errs} \Rightarrow n \geq 5\right) \\
* \left(\exists (n, l) \in \mathbb{N}_5^6 \times \mathcal{D}.\, \mathbf{UnsafeEml}_\alpha(\iota, j, k, (n, l)) * [\mathsf{G}(1)]_\alpha * [\mathsf{W}]_\alpha * n = 6 \Rightarrow \mathit{is_not_virus}\right)
\end{array}\right\}$$

$\sqsubseteq$ by HCons

$$\left\{\begin{array}{l}
\mathbf{UnsafeEml}_\alpha(\iota, j, k, (1, \bot)) * [\mathsf{G}(1)]_\alpha * [\mathsf{F}]_\alpha, \\
\exists (n, l) \in \mathbb{N}_5^6 \times \mathcal{D}.\, \mathbf{UnsafeEml}_\alpha(\iota, j, k, (n, l)) * [\mathsf{G}(1)]_\alpha * [\mathsf{F}]_\alpha \\
* \mathit{delivered} \notin \textsc{Errs} \Rightarrow \mathit{is_not_virus} \in \mathbb{B}
\end{array}\right\}$$

$\sqsubseteq$ by EElimHoare on α, CreateRegion and HCons similarly to figure 8.5 and EElimHoare on j, k

$$\left\{\begin{array}{l}
\mathsf{fs}(FS) \wedge \exists j, k.\, FS(\iota_0)(\text{mail}) = j \wedge FS(j)(\text{tmp}) = k \wedge FS(k)(42.\text{msg}) = \iota \wedge \mathsf{isfile}(FS(\iota)) \\
\wedge\, 42 \notin FS(j) \wedge \text{quarantine} \notin FS(j), \\
\mathsf{fs}(-) * \mathit{delivered} \notin \textsc{Errs} \Rightarrow \mathit{is_not_virus} \in \mathbb{B}
\end{array}\right\}$$

$\sqsubseteq$ by HCons

$$\left\{\begin{array}{l}
\mathsf{fs}(FS) \wedge /\text{mail}/\text{tmp}/42.\text{msg} \xmapsto{FS} \iota \wedge \mathsf{isfile}(FS(\iota)) \wedge \neg /\text{mail}/42 \xmapsto{FS} - \wedge \neg /\text{mail}/\text{quarantine} \xmapsto{FS} -, \\
\mathsf{fs}(-) * \mathit{delivered} \notin \textsc{Errs} \Rightarrow \mathit{is_not_virus} \in \mathbb{B}
\end{array}\right\}$$

Figure 8.9.: Proof of email client-server interaction using the POSIX file-system specification.

HCons to split the state into two parts which are subsequently distributed between the client and server threads with Parallel. We then continue with refinements of the client and server threads in

figures 8.10 and 8.12 respectively.

First, consider the refinement for the email client in figure 8.10. It is more complicated in contrast to the refinement using the simplified coarse-grained specification in figure 8.6. Since `stat` is not atomic in POSIX, we must to reason about each individual step comprising the operation, since each step is affected by the server's actions. Since this proof is about a specific invocation of `stat`, before

Let $\mathcal{D} \triangleq \mathbb{N}_1^6 \times (\{\bot\} \cup \textsc{Inodes})$.

let *delivered* = `stat`(/mail/42/msg.eml)

$\sqsubseteq$ by FApplyElim

let $r = \texttt{resolve}(/\text{mail}/42, \iota_0)$;

$\sqsubseteq$ by figure 8.11

$$\left\{\begin{array}{l}\exists (n,l) \in \mathbb{N}_1^6 \times \mathcal{D}.\, \mathbf{UnsafeEml}_\alpha(\iota, j, k, (n,l)) * [\mathsf{R}]_\alpha\,, \\ \exists (n,l) \in \mathbb{N}_1^6 \times \mathcal{D}.\, \mathbf{UnsafeEml}_\alpha(\iota, j, k, (n,l)) * [\mathsf{R}]_\alpha * r \in \textsc{Inodes} \Rightarrow n \geq 2 \wedge l = r\end{array}\right\}$$

Seq / IfThenElse

if $\neg\mathsf{iserr}(r)$ **then** `return link_stat`$(r, \text{msg.eml})$

$\sqsubseteq$ by figure 8.8, FApplyElim and DChoiceIntro

$$\forall FS.\, \left\langle \begin{array}{l}\mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(r))\,, \\ \text{msg.eml} \in FS(r) \Rightarrow \mathsf{fs}(FS) * \mathit{delivered} = \mathsf{ftype}(FS(r)(\text{msg.eml}))\end{array} \right\rangle$$

$\sqsubseteq$ by AEElim, ACons, AFrame and OpenRegion

$$\forall n \in \mathbb{N}_2^6.\, \left\langle \begin{array}{l}\mathbf{UnsafeEml}_\alpha(\iota, j, k, (n,r)) * [\mathsf{R}]_\alpha\,, \\ \mathbf{UnsafeEml}_\alpha(\iota, j, k, (n,r)) * [\mathsf{R}]_\alpha * \mathit{delivered} \notin \textsc{Errs} \Rightarrow n \geq 5\end{array} \right\rangle$$

$\sqsubseteq$ by AEElim and AWeaken2

$$\left\{\begin{array}{l}\exists n \in \mathbb{N}_2^6.\, \mathbf{UnsafeEml}_\alpha(\iota, j, k, (n,r)) * [\mathsf{R}]_\alpha\,, \\ \exists n \in \mathbb{N}_2^6.\, \mathbf{UnsafeEml}_\alpha(\iota, j, k, (n,r)) * [\mathsf{R}]_\alpha * \mathit{delivered} \notin \textsc{Errs} \Rightarrow n \geq 5\end{array}\right\}$$

else `return` r **fi**

$$\sqsubseteq \left\{\begin{array}{l}\exists (n,l) \in \mathbb{N}_1^6 \times \mathcal{D}.\, \mathbf{UnsafeEml}_\alpha(\iota, j, k, (n,l)) * [\mathsf{R}]_\alpha * r \in \textsc{Inodes} \Rightarrow n \geq 2 \wedge l = r, \\ \exists (n,l) \in \mathbb{N}_1^6 \times \mathcal{D}.\, \mathbf{UnsafeEml}_\alpha(\iota, j, k, (n,l)) * [\mathsf{R}]_\alpha * \mathit{delivered} \notin \textsc{Errs} \Rightarrow n \geq 5 \wedge l = r\end{array}\right\}$$

$$\sqsubseteq \left\{\begin{array}{l}\exists (n,l) \in \mathbb{N}_1^6 \times \mathcal{D}.\, \mathbf{UnsafeEml}_\alpha(\iota, j, k, (n,l)) * [\mathsf{R}]_\alpha\,, \\ \exists (n,l) \in \mathbb{N}_1^6 \times \mathcal{D}.\, \mathbf{UnsafeEml}_\alpha(\iota, j, k, (n,l)) * [\mathsf{R}]_\alpha * \mathit{delivered} \notin \textsc{Errs} \Rightarrow n \geq 5 \wedge l = r\end{array}\right\}$$

Figure 8.10.: Derivation of the email client's specification using the POSIX file-system specification.

we proceed with the refinement proof we use FApplyElim (at the top of the derivation) to apply the argument of `stat` to its POSIX specification. Thus we get a POSIX specification for the particular invocation of `stat`(/mail/42/msg.eml). We then proceed with the derivation as usual, starting with the specification we wish to refine to the POSIX specification of `stat`(/mail/42/msg.eml) at the bottom.

In the first refinement step we apply Seq over the sequence of abstract operations comprising the specification of `stat`(/mail/42/msg.eml). The first operation is `resolve`$(/\text{mail}/42, \iota_0)$. Its specification states that if the operation succeeds ($r \in \textsc{Inodes}$), then the **UnsafeEml** is in an abstract state (n, l), where the directory with initial path /mail/42 had been created with inode l ($n \geq 2$) which also is the returned inode ($l = r$). The refinement of this specification to `resolve`$(/\text{mail}/42, \iota_0)$ is given in figure 8.11 and will be discussed shortly. In the subsequent **if-then-else**, we use IfThenElse focusing on the success branch, where we implicitly use the assumption $[\neg\mathsf{iserr}(r)]$. In the first refinement within the success branch we use AWeaken2 to refine to an atomic update and AEElim to eliminate the existential quantification on the region's state. Then, we apply OpenRegion to open the region's interpretation, AFrame to frame-off the R guard, ACons to simplify and then AEElim to eliminate

the existential quantification on file-system graphs. At this point we have refined the **if**-branch to the success case of `link_stat`$(r, \text{msg.eml})$. Thus we apply DChoiceIntro to introduce the other demonic cases and refine to the invocation `link_stat`$(r, \text{msg.eml})$, which by FApplyElim is refined to `link_stat` given in figure 8.8.

`resolve`$(/\text{mail}/42, \iota_0)$

$\sqsubseteq$ by successive FApplyElimRec

HCons and Seq

let $r = $ `link_lookup`(ι_0, mail);

$\sqsubseteq$ by figure 6.2, FApplyElim, DChoiceIntro and ACons

$$\forall FS.\, \left\langle \begin{array}{l} \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota_0)) \wedge FS(\iota_0)(\text{mail}) = j, \\ \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota_0)) \wedge FS(\iota_0)(\text{mail}) = j * r = j \end{array} \right\rangle$$

$\sqsubseteq$ by AEElim, ACons, AFrame and OpenRegion

$$\forall (n,l) \in \mathbb{N}_1^6 \times \mathcal{D}.\, \left\langle \begin{array}{l} \mathbf{UnsafeEml}_\alpha(\iota, j, k, (n,l)) * [\mathsf{R}]_\alpha, \\ \mathbf{UnsafeEml}_\alpha(\iota, j, k, (n,l)) * [\mathsf{R}]_\alpha * r = j \end{array} \right\rangle$$

$\sqsubseteq$ by AEElim and AWeaken2

$$\left\{ \begin{array}{l} \exists (n,l) \in \mathbb{N}_1^6 \times \mathcal{D}.\, \mathbf{UnsafeEml}_\alpha(\iota, j, k, (n,l)) * [\mathsf{R}]_\alpha, \\ \exists (n,l) \in \mathbb{N}_1^6 \times \mathcal{D}.\, \mathbf{UnsafeEml}_\alpha(\iota, j, k, (n,l)) * [\mathsf{R}]_\alpha * r = j \end{array} \right\}$$

IfThenElse and HFrame

if $\mathsf{iserr}(r)$ **then** `return` r

else `return` `link_lookup`$(r, 42)$

$\sqsubseteq$ by figure 6.2, FApplyElim, DChoiceIntro and ACons

$$\forall FS.\, \langle \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(r)),\ 42 \in FS(r) \Rightarrow \mathsf{fs}(FS) * \texttt{ret} = FS(r)(42) \rangle$$

$\sqsubseteq$ by AEElim, ACons, AFrame and OpenRegion

$$\forall (n,l) \in \mathbb{N}_1^6 \times \mathcal{D}.\, \left\langle \begin{array}{l} \mathbf{UnsafeEml}_\alpha(\iota, r, k, (n,l)) * [\mathsf{R}]_\alpha, \\ \mathbf{UnsafeEml}_\alpha(\iota, r, k, (n,l)) * [\mathsf{R}]_\alpha * \texttt{ret} \in \textsc{Inodes} \Rightarrow n \geq 2 \wedge l = \texttt{ret} \end{array} \right\rangle$$

$\sqsubseteq$ by AEElim and AWeaken2

$$\left\{ \begin{array}{l} \exists (n,l) \in \mathbb{N}_1^6 \times \mathcal{D}.\, \mathbf{UnsafeEml}_\alpha(\iota, r, k, (n,l)) * [\mathsf{R}]_\alpha, \\ \exists (n,k) \in \mathbb{N}_1^6 \times \mathcal{D}.\, \mathbf{UnsafeEml}_\alpha(\iota, r, k, (n,l)) * [\mathsf{R}]_\alpha * \texttt{ret} \in \textsc{Inodes} \Rightarrow n \geq 2 \wedge l = \texttt{ret} \end{array} \right\}$$

fi

$$\sqsubseteq \left\{ \begin{array}{l} \exists (n,l) \in \mathbb{N}_1^6 \times \mathcal{D}.\, \mathbf{UnsafeEml}_\alpha(\iota, r, k, (n,l)) * [\mathsf{R}]_\alpha * r = j, \\ \exists (n,l) \in \mathbb{N}_1^6 \times \mathcal{D}.\, \mathbf{UnsafeEml}_\alpha(\iota, r, k, (n,l)) * [\mathsf{R}]_\alpha \\ * \texttt{ret} \in \textsc{Inodes} \Rightarrow n \geq 2 \wedge l = \texttt{ret} * r = j \end{array} \right\}$$

$$\sqsubseteq \left\{ \begin{array}{l} \exists (n,l) \in \mathbb{N}_1^6 \times \mathcal{D}.\, \mathbf{UnsafeEml}_\alpha(\iota, j, k, (n,l)) * [\mathsf{R}]_\alpha, \\ \exists (n,k) \in \mathbb{N}_1^6 \times \mathcal{D}.\, \mathbf{UnsafeEml}_\alpha(\iota, j, k, (n,l)) * [\mathsf{R}]_\alpha * \texttt{ret} \in \textsc{Inodes} \Rightarrow n \geq 2 \wedge l = \texttt{ret} \end{array} \right\}$$

Figure 8.11.: Derivation of specification for client's `resolve`$(/\text{mail}/42, \iota_0)$.

The refinement of the specification for `resolve`$(/\text{mail}/42, \iota_0)$, which we used in figure 8.10, is given in figure 8.11 and follows a similar pattern. Recall from chapter 6, section 6.1.1, that `resolve` is defined recursively. Since the path argument to `resolve` is a known value, we successively apply FApplyElimRec until we effectively unroll the recursion into a sequence of `link_lookup` operations. From there we proceed similarly to figure 8.10, applying Seq on the first `link_lookup` and the **if-then-else** containing the second `link_lookup` that follows, on which we use IfThenElse, except that this time we focus on the **else**-branch. The additional HCons and HFrame steps are taken such that the postcondition of the first `link_lookup` matches the precondition of the **if-then-else** statement containing the second.

We now turn our attention to the refinement proof for the server thread given in figure 8.12. This also follows a similar pattern to the proof using the coarse-grained file-system specification in figure 8.7,

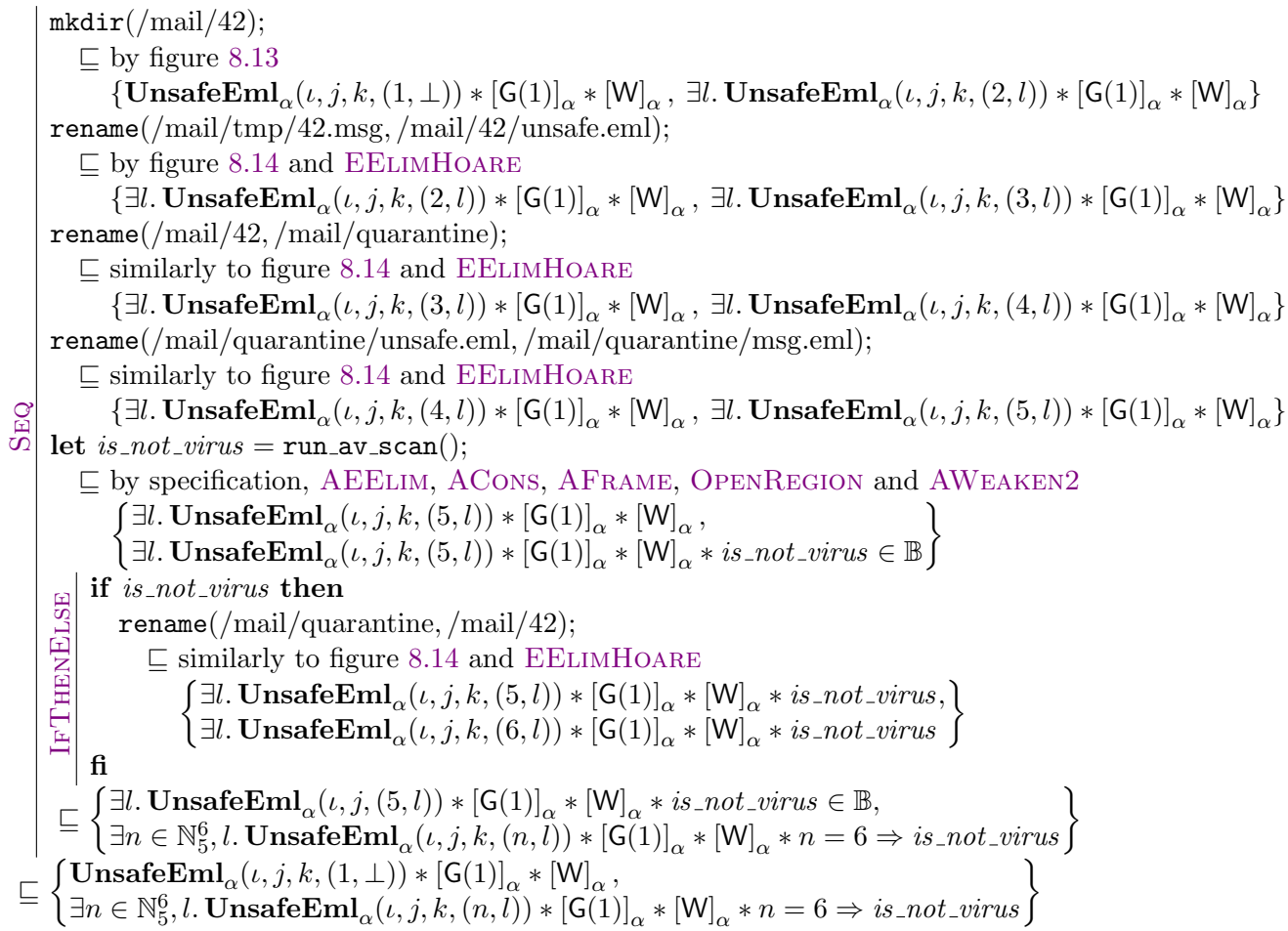

Figure 8.12.: Derivation of email server's specification using the POSIX file-system specification.

except that the refinement for each operation in the sequence is more elaborate due to the non-atomic nature of `mkdir` and `rename` in POSIX.

The refinement for the server's `mkdir`(/mail/42) is given in figure 8.13. First we use FAPPLYELIM to apply the argument to `mkdir`'s specification, which we discussed in chapter 6, section 6.1.2. We then proceed with refining the specification at the bottom of the derivation. In the first step of this refinement we apply SEQ over the sequence of abstract operations comprising `mkdir`(/mail/42), where we use HCONS to match the postcondition of `resolve`(/mail, ι_0) with the precondition of the **if-then-else** statement that follows. In the refinement to `resolve`(/mail, ι_0) we successively apply FAPPLYELIMREC to unroll the recursion; effectively, to a single `link_lookup`. We then proceed similarly to the `link_lookup` refinements in figure 8.11. Next, in the refinement to the **if-then-else** statement we first use HFRAME to frame-off $r = j$, as it is not needed further in the derivation, and then we apply IFTHENELSE focusing on the **if**-branch. Within the **if**-branch we apply DCHOICEINTRO to introduce the demonic error cases composed with `link_new_dir`, with the rest of the derivation refining to `link_new_dir`$(r, 42)$. For simplicity we assume the atomic variant of `link_new_dir`.

The refinement to the server's `rename`(/mail/tmp/42.msg, /mail/42/unsafe.eml) is given in figure 8.14. Here, we apply FAPPLYELIM to `rename`'s POSIX specification from chapter 6, section 6.1.1, to obtain the sequence of statements comprising `rename`(/mail/tmp/42.msg, /mail/42/unsafe.eml).

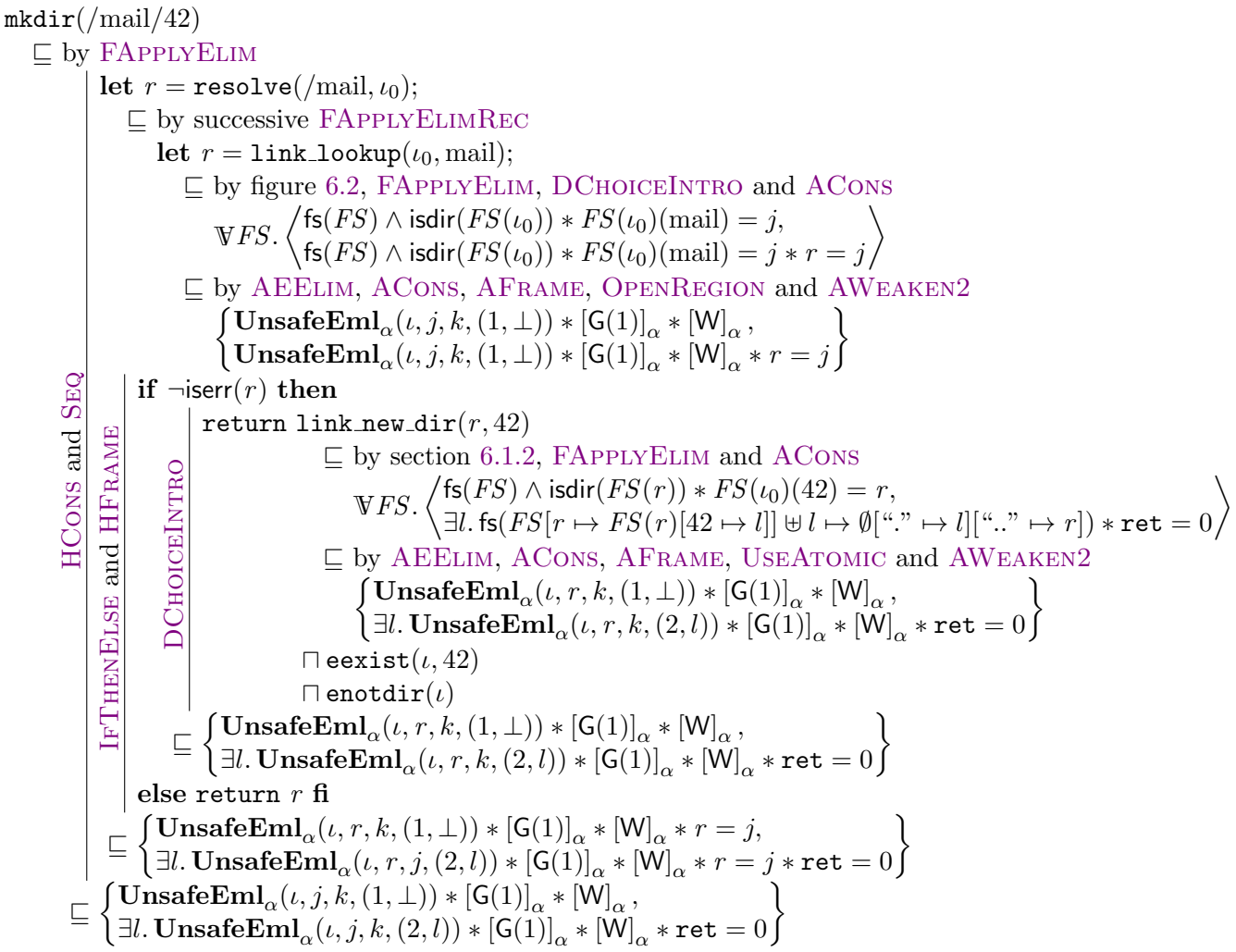

Figure 8.13.: Derivation of server's `mkdir`(/mail/42) using the POSIX file-system specification.

Note that in this invocation of `rename` the source path identifies and existing regular file whereas the basename of the target path does not exist. Therefore, the refinement can focus on the success case `link_move_file_target_not_exists`, and discard all others, by applying DChoiceIntro. The aspect of this proof that differs from those we have seen so far is the parallel composition of the two path resolutions. Thus we use HCons to partition the state between the two `resolve`s and then apply Parallel. The refinements to `resolve`(/mail/tmp, ι_0) and `resolve`(/mail/42/unsafe.eml, ι_0) are given in figures 8.15 and 8.16 respectively and follow a similar pattern to figure 8.11. Note that each `resolve` requires ownership of the guard G with permission 0.5. Ownership of this guard in each `resolve` is needed to maintain the stability of the region's state. The absence of this guard would indicate that it may be owned by the environment, allowing the environment to concurrently update the region's state. Ownership of this guard guarantees that the **UnsafeEml** region is not updated concurrently.

$\texttt{rename}(/\text{mail/tmp/42.msg}, /\text{mail/42/unsafe.eml})$

$\sqsubseteq$ by FApplyElim and DChoiceIntro

HCons and Seq

IfThenElse and HFrame

$\quad$ **let** $r_s, r_t = \texttt{resolve}(/\text{mail/tmp}, \iota_0) \parallel \texttt{resolve}(/\text{mail/42}, \iota_0)$;

$\quad\sqsubseteq$ by figures 8.15 and 8.16 and Parallel

$$\left\{\begin{array}{l}(\mathbf{UnsafeEml}_\alpha(\iota,j,k,(2,l)) * [\mathsf{G}(0.5)]_\alpha) * (\mathbf{UnsafeEml}_\alpha(\iota,j,k,(2,l)) * [\mathsf{G}(0.5)]_\alpha)\,, \\ (\mathbf{UnsafeEml}_\alpha(\iota,j,k,(2,l)) * [\mathsf{G}(0.5)]_\alpha * r_s = k) \\ \quad * (\mathbf{UnsafeEml}_\alpha(\iota,j,k,(2,l)) * [\mathsf{G}(0.5)]_\alpha * r_t = l)\end{array}\right\}$$

$\quad\sqsubseteq$ by HCons

$$\{\mathbf{UnsafeEml}_\alpha(\iota,j,k,(2,l)) * [\mathsf{G}(1)]_\alpha\,,\ \mathbf{UnsafeEml}_\alpha(\iota,j,k,(2,l)) * [\mathsf{G}(1)]_\alpha * r_s = k * r_t = l\}$$

$\quad$ **if** $\neg\mathsf{iserr}(r_s) \wedge \neg\mathsf{iserr}(r_t)$ **then**

$\quad\quad \texttt{return link\_move\_file\_target\_not\_exists}(r_s, 42.\text{msg}, r_t, \text{unsafe.eml})$

$\quad\quad\sqsubseteq$ by figure 6.5, FApplyElim and ACons

$$\forall FS.\ \left\langle\begin{array}{l}\mathsf{fs}(FS) \wedge FS(\iota_0)(\text{mail}) = j \wedge FS(j)(\text{tmp}) = r_s \wedge FS(r_s)(42.\text{msg}) = \iota \\ \quad \wedge\ \mathsf{isfile}(FS(\iota)) \wedge FS(j)(42) = r_t \wedge \text{unsafe.eml} \notin FS(k), \\ \mathsf{fs}(FS[r_s \mapsto FS(r_s) \setminus \{42.\text{msg}\}][r_t \mapsto FS(r_t)[\text{unsafe.eml} \mapsto FS(r_s)(42.\text{msg})]]) * \texttt{ret} = 0\end{array}\right\rangle$$

$\quad\quad\sqsubseteq$ by AEElim, ACons, AFrame and UseAtomic

$$\left\langle\begin{array}{l}\mathbf{UnsafeEml}_\alpha(\iota,j,r_s,(2,r_t)) * [\mathsf{G}(1)]_\alpha * [\mathsf{W}]_\alpha\,, \\ \mathbf{UnsafeEml}_\alpha(\iota,j,r_s,(3,r_t)) * [\mathsf{G}(1)]_\alpha * [\mathsf{W}]_\alpha * \texttt{ret} = 0\end{array}\right\rangle$$

$\quad\quad\sqsubseteq$ by AWeaken2

$$\left\{\begin{array}{l}\mathbf{UnsafeEml}_\alpha(\iota,j,r_s,(2,r_t)) * [\mathsf{G}(1)]_\alpha * [\mathsf{W}]_\alpha\,, \\ \mathbf{UnsafeEml}_\alpha(\iota,j,r_s,(3,r_t)) * [\mathsf{G}(1)]_\alpha * [\mathsf{W}]_\alpha * \texttt{ret} = 0\end{array}\right\}$$

$\quad$ **else if** $\mathsf{iserr}(r_s) \wedge \neg\mathsf{iserr}(r_t)$ **then return** r_s

$\quad$ **else if** $\neg\mathsf{iserr}(r_s) \wedge \mathsf{iserr}(r_t)$ **then return** r_t

$\quad$ **else if** $\mathsf{iserr}(r_s) \wedge \mathsf{iserr}(r_t)$ **then return** $r_s \sqcup$ **return** r_t **fi**

$$\sqsubseteq \left\{\begin{array}{l}\mathbf{UnsafeEml}_\alpha(\iota,j,r_s,(2,r_t)) * [\mathsf{G}(1)]_\alpha * [\mathsf{W}]_\alpha * r_s = k * r_t = l, \\ \mathbf{UnsafeEml}_\alpha(\iota,j,r_s,(3,r_t)) * [\mathsf{G}(1)]_\alpha * [\mathsf{W}]_\alpha * r_s = k * r_t = l * \texttt{ret} = 0\end{array}\right\}$$

$$\sqsubseteq \left\{\begin{array}{l}\mathbf{UnsafeEml}_\alpha(\iota,j,k,(2,l)) * [\mathsf{G}(1)]_\alpha * [\mathsf{W}]_\alpha\,, \\ \mathbf{UnsafeEml}_\alpha(\iota,j,k,(3,l)) * [\mathsf{G}(1)]_\alpha * [\mathsf{W}]_\alpha * \texttt{ret} = 0\end{array}\right\}$$

Figure 8.14.: Derivation of server's $\texttt{rename}(/\text{mail/tmp/42.msg}, /\text{mail/42/unsafe.eml})$ using the POSIX file-system specification

Remarks

We have studied the behaviour of an example concurrent email client-server interaction using two different approaches to the concurrent specification of the POSIX file-system operations: a coarse grained approach where file-system operations are atomic, and the fine-grained concurrent specification we develop in this dissertation. Even though fictitious, the example serves as an apt comparison between the two approaches. Arguably, a coarse-grained specification for file-system operations is simpler than our POSIX specification, leading to simpler client reasoning with the tradeoff of not capturing accurately the POSIX standard. However, in reasoning about the example we demonstrate a fatal flaw of the coarse-grained approach: it is ill-suited for client reasoning. Specifically, using the coarse-grained approach we are able to prove a strong functional property — if the email client observes a message delivered, then the server has declared it virus-free — that is impossible to prove in the fine-grained approach that follows POSIX. Reasoning with a coarse-grained specification of POSIX file-system concurrency fails to account for all the possible behaviours clients may observe,

$\texttt{resolve}(/\text{mail/tmp}, \iota_0)$

$\sqsubseteq$ by successive FApplyElimRec

HCons and Seq

$\quad$ **let** $r = \texttt{link\_lookup}(\iota_0, \text{mail})$;

$\quad\quad \sqsubseteq$ by figure 6.2, DChoiceIntro, ACons, AEElim, AFrame, OpenRegion and AWeaken2

$\quad\quad \{\mathbf{UnsafeEml}_\alpha(\iota, j, k, (2, l)) * [\mathsf{G}(0.5)]_\alpha\,,\ \mathbf{UnsafeEml}_\alpha(\iota, j, k, (2, l)) * [\mathsf{G}(0.5)]_\alpha * r = j\}$

IfThenElse and HFrame

$\quad$ **if** $\neg\mathsf{iserr}(r)$ **then** $\texttt{return}\ r$

$\quad$ **else** $\texttt{return link\_lookup}(r, \text{tmp})$

$\quad\quad \sqsubseteq$ by figure 6.2, DChoiceIntro and ACons

$\quad\quad \forall FS.\, \langle \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(r)) \wedge FS(r)(\text{tmp}) = k,\ \mathsf{fs}(FS) * \texttt{ret} = FS(r)(\text{tmp}) \rangle$

$\quad\quad \sqsubseteq$ by AEElim, ACons, AFrame and OpenRegion

$\quad\quad \left\langle \begin{array}{l} \mathbf{UnsafeEml}_\alpha(\iota, r, k, (2, l)) * [\mathsf{G}(0.5)]_\alpha\,, \\ \mathbf{UnsafeEml}_\alpha(\iota, r, k, (2, l)) * [\mathsf{G}(0.5)]_\alpha * \texttt{ret} = k \end{array} \right\rangle$

$\quad\quad \sqsubseteq$ by AWeaken2

$\quad\quad \left\{ \begin{array}{l} \mathbf{UnsafeEml}_\alpha(\iota, r, k, (2, l)) * [\mathsf{G}(0.5)]_\alpha\,, \\ \mathbf{UnsafeEml}_\alpha(\iota, r, k, (2, l)) * [\mathsf{G}(0.5)]_\alpha * \texttt{ret} = k \end{array} \right\}$

$\quad$ **fi**

$\quad \sqsubseteq \left\{ \begin{array}{l} \mathbf{UnsafeEml}_\alpha(\iota, r, k, (2, l)) * [\mathsf{G}(0.5)]_\alpha * r = j, \\ \mathbf{UnsafeEml}_\alpha(\iota, r, k, (2, l)) * [\mathsf{G}(0.5)]_\alpha * r = j * \texttt{ret} = k \end{array} \right\}$

$\sqsubseteq \{\mathbf{UnsafeEml}_\alpha(\iota, j, k, (2, l)) * [\mathsf{G}(0.5)]_\alpha\,,\ \mathbf{UnsafeEml}_\alpha(\iota, j, k, (2, l)) * [\mathsf{G}(0.5)]_\alpha * \texttt{ret} = k\}$

Figure 8.15.: Derivation of specification statement for $\texttt{resolve}(/\text{mail/tmp/42.msg}, \iota_0)$ used in email server's $\texttt{rename}(/\text{mail/tmp/42.msg}, /\text{mail/42/unsafe.eml})$.

$\texttt{resolve}(/\text{mail/42}, \iota_0)$

$\sqsubseteq$ by successive FApplyElimRec

HCons and Seq

$\quad$ **let** $r = \texttt{link\_lookup}(\iota_0, \text{mail})$;

$\quad\quad \sqsubseteq$ by figure 6.2, DChoiceIntro, ACons, AEElim, AFrame, OpenRegion and AWeaken2

$\quad\quad \{\mathbf{UnsafeEml}_\alpha(\iota, j, k, (2, l)) * [\mathsf{G}(0.5)]_\alpha\,,\ \mathbf{UnsafeEml}_\alpha(\iota, j, k, (2, l)) * [\mathsf{G}(0.5)]_\alpha * r = j\}$

IfThenElse and HFrame

$\quad$ **if** $\neg\mathsf{iserr}(r)$ **then** $\texttt{return}\ r$

$\quad$ **else** $\texttt{return link\_lookup}(r, 42)$

$\quad\quad \sqsubseteq$ by figure 6.2, DChoiceIntro and ACons

$\quad\quad \forall FS.\, \langle \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(r)) \wedge FS(r)(42) = l,\ \mathsf{fs}(FS) * \texttt{ret} = FS(r)(42) \rangle$

$\quad\quad \sqsubseteq$ by AEElim, ACons, AFrame and OpenRegion

$\quad\quad \left\langle \begin{array}{l} \mathbf{UnsafeEml}_\alpha(\iota, r, k, (2, l)) * [\mathsf{G}(0.5)]_\alpha\,, \\ \mathbf{UnsafeEml}_\alpha(\iota, r, k, (2, l)) * [\mathsf{G}(0.5)]_\alpha * \texttt{ret} = l \end{array} \right\rangle$

$\quad\quad \sqsubseteq$ by AWeaken2

$\quad\quad \left\{ \begin{array}{l} \mathbf{UnsafeEml}_\alpha(\iota, r, k, (2, l)) * [\mathsf{G}(0.5)]_\alpha\,, \\ \mathbf{UnsafeEml}_\alpha(\iota, r, k, (2, l)) * [\mathsf{G}(0.5)]_\alpha * \texttt{ret} = l \end{array} \right\}$

$\quad$ **fi**

$\quad \sqsubseteq \left\{ \begin{array}{l} \mathbf{UnsafeEml}_\alpha(\iota, r, k, (2, l)) * [\mathsf{G}(0.5)]_\alpha * r = j, \\ \mathbf{UnsafeEml}_\alpha(\iota, r, k, (2, l)) * [\mathsf{G}(0.5)]_\alpha * r = j * \texttt{ret} = l \end{array} \right\}$

$\sqsubseteq \{\mathbf{UnsafeEml}_\alpha(\iota, j, k, (2, l)) * [\mathsf{G}(0.5)]_\alpha\,,\ \mathbf{UnsafeEml}_\alpha(\iota, j, k, (2, l)) * [\mathsf{G}(0.5)]_\alpha * \texttt{ret} = l\}$

Figure 8.16.: Derivation of specification statement for $\texttt{resolve}(/\text{mail/42}, \iota_0)$ used in email server's $\texttt{rename}(/\text{mail/tmp/42.msg}, /\text{mail/42/unsafe.eml})$.

leading to unsound conclusions.

8.2.3. Correcting the Email Client-Server Interaction

The flaw of the coarse-grained file-system specification is that it assumes additional synchronisation between file-system operations that is not specified in the POSIX standard. We can thus adapt the email client-server interaction in figure 8.3 to make it safe under the POSIX semantics by explicitly introducing the additional synchronisation. One possible way to achieve this is by creating a coarse-grained version of each file-system operation by wrapping each operation in a lock. However, it is not

```
lock(/mail/.lock);                    ||  mkdir(/mail/42);
let delivered =                       ||  rename(/mail/tmp/42.msg, /mail/42/unsafe.eml);
  stat(/mail/42/msg.eml);             ||  rename(/mail/42, /mail/quarantine);
unlock(/mail/.lock)                   ||  lock(/mail/.lock);
                                      ||  rename(/mail/quarantine/unsafe.eml, /mail/quarantine/msg.eml);
                                      ||  let is_not_virus = run_av_scan();
                                      ||  if is_not_virus then
                                      ||    rename(/mail/quarantine, /mail/42);
                                      ||  fi
                                      ||  unlock(/mail/.lock)
```

Figure 8.17.: A safe email client-server interaction.

necessary to synchronise on every operation. We can simply use a lock such that the client's `stat` is never performed in between the server's last two `rename`s, as in figure 8.17.

This time we will reason about this example assuming the file system is shared with the rest of the environment via the global file-system region **GFS** restricted according to a context invariant. In figure 8.17 we are using a lock file at path /mail/.lock to synchronise the client and server. Therefore, the context invariant for this example must include $\mathsf{LFCtx}(/\text{mail}/.\text{lock})$ so that the lock file behaves as intended.

To aid in the definition of the context invariant we define the following auxiliary predicate describing the file-system graphs of the client-server interaction:

$$\begin{aligned}
\mathcal{EFS}(FS,\iota,j,k,(1,\bot)) \triangleq\ & FS(\iota_0)(\text{mail}) = j \wedge FS(j)(\text{tmp}) = k \wedge FS(k)(42.\text{msg}) = \iota \\
& \wedge \mathsf{isfile}(FS(\iota)) \wedge 42 \notin FS(j) \wedge \text{quarantine} \notin FS(j) \\
\mathcal{EFS}(FS,\iota,j,k,(2,l)) \triangleq\ & FS(\iota_0)(\text{mail}) = j \wedge FS(j)(\text{tmp}) = k \\
& \wedge FS(k)(42.\text{msg}) = \iota \wedge \mathsf{isfile}(FS(\iota)) \wedge FS(j)(42) = l \\
& \wedge \text{unsafe.eml} \notin FS(l) \wedge \text{msg.eml} \notin FS(l) \wedge \text{quarantine} \notin FS(j) \\
\mathcal{EFS}(FS,\iota,j,k,(3,l)) \triangleq\ & FS(\iota_0)(\text{mail}) = j \wedge FS(j)(\text{tmp}) = k \\
& \wedge 42.\text{msg} \notin FS(k) \wedge FS(j)(42) = l \wedge FS(l)(\text{unsafe.eml}) = \iota \\
& \wedge \mathsf{isfile}(FS(\iota)) \wedge \text{msg.eml} \notin FS(l) \wedge \text{quarantine} \notin FS(j) \\
\mathcal{EFS}(FS,\iota,j,k,(4,l)) \triangleq\ & FS(\iota_0)(\text{mail}) = j \wedge FS(j)(\text{tmp}) = k \\
& \wedge FS(j)(\text{quarantine}) = l \wedge FS(l)(\text{unsafe.eml}) = \iota \\
& \wedge \mathsf{isfile}(FS(\iota)) \wedge \text{msg.eml} \notin FS(l) \wedge 42 \notin FS(j)
\end{aligned}$$

$$\begin{aligned}\mathcal{EFS}(FS,\iota,j,k,(5,l)) \triangleq\ & FS(\iota_0)(\text{mail}) = j \wedge FS(j)(\text{tmp}) = k \\ & \wedge FS(j)(\text{quarantine}) = l \wedge FS(l)(\text{msg.eml}) = \iota \\ & \wedge \mathsf{isfile}(FS(\iota)) \wedge 42 \notin FS(j)\end{aligned}$$

$$\begin{aligned}\mathcal{EFS}(FS,\iota,j,k,(6,l)) \triangleq\ & FS(\iota_0)(\text{mail}) = j \wedge FS(j)(\text{tmp}) = k \\ & \wedge FS(j)(42) = l \wedge FS(l)(\text{msg.eml}) = \iota \\ & \wedge \mathsf{isfile}(FS(\iota)) \wedge \text{quarantine} \notin FS(j)\end{aligned}$$

Note that the definition is similar to the interpretation of the **UnsafeEml** in section 8.2.2. The arguments ι, j and k are the inodes of the /mail/tmp/42.msg email file, the /mail directory and the /mail/tmp directory respectively. The argument l, when $l \neq \bot$, captures the inode of the directory created with the path /mail/42 by the server's `mkdir`. Additionally, we define the following set of possible file-system graphs:

$$\mathcal{EML}(\iota,j,k,l) \triangleq \left\{FS \;\middle|\; \mathcal{EFS}(FS,\iota,j,k,(1,\bot))\right\} \cup \left\{FS \;\middle|\; n \in \mathbb{N}_2^6.\, \mathcal{EFS}(FS,\iota,j,k,(n,l))\right\}$$

We define the context invariant as follows:

$$\begin{aligned}&\mathsf{EmlCtx}(\iota,j,k) \triangleq \\ &\quad \mathsf{LFCtx}(\text{/mail/.lock}) \\ &\quad \wedge \exists l, FS \in \mathcal{EML}(\iota,j,k,l).\, \mathbf{GFS}(FS) \\ &\quad \wedge !\,[\mathsf{E}] \in \mathcal{G}_{\mathbf{GFS}} \\ &\quad \wedge \forall FS, l.\, \mathcal{EML}(FS,\iota,j,k,(1,\bot)) \wedge (FS, FS[j \mapsto FS(j)[42 \mapsto l]])\ \dagger_{\mathbf{GFS}}\ [\mathsf{E}] \\ &\quad \wedge \forall FS, l.\, \mathcal{EML}(FS,\iota,j,k,(2,l)) \wedge (FS, FS[k \mapsto FS(k) \setminus \{42.\text{msg}\}][l \mapsto FS(l)[\text{unsafe.eml} \mapsto \iota]])\ \dagger_{\mathbf{GFS}}\ [\mathsf{E}] \\ &\quad \wedge \forall FS, l.\, \mathcal{EML}(FS,\iota,j,k,(3,l)) \wedge (FS, FS[j \mapsto FS(j) \setminus \{42\}][j \mapsto FS(j)[\text{quarantine} \mapsto l]])\ \dagger_{\mathbf{GFS}}\ [\mathsf{E}] \\ &\quad \wedge \forall FS, l.\, \mathcal{EML}(FS,\iota,j,k,(4,l)) \wedge (FS, FS[l \mapsto FS(l)[\text{msg.eml} \mapsto \iota]])\ \dagger_{\mathbf{GFS}}\ [\mathsf{E}] \\ &\quad \wedge \forall FS, l.\, \mathcal{EML}(FS,\iota,j,k,(5,l)) \wedge (FS, FS[j \mapsto FS(j) \setminus \{\text{quarantine}\}][j \mapsto FS(j)[42 \mapsto l]])\ \dagger_{\mathbf{GFS}}\ [\mathsf{E}] \\ &\quad \wedge \forall \mathsf{G} \in \mathcal{G}_{\mathbf{GFS}}.\, \mathsf{G}\#\mathsf{E} \wedge \forall FS, FS' \in \mathcal{EML}(\iota,j,k). \\ &\qquad (FS, FS') \in \mathcal{T}_{\mathbf{GFS}}(\mathsf{G})^* \wedge \text{/mail/42} \xmapsto{FS} - \Rightarrow FS\restriction_{\text{/mail/42}} = FS'\restriction_{\text{/mail/42}} \\ &\qquad (FS, FS') \in \mathcal{T}_{\mathbf{GFS}}(\mathsf{G})^* \wedge \text{/mail/quarantine} \xmapsto{FS} - \Rightarrow FS\restriction_{\text{/mail/quarantine}} = FS'\restriction_{\text{/mail/quarantine}}\end{aligned}$$

The arguments ι, j and k are the inodes of the /mail/tmp/42.msg email file, the /mail directory and the /mail/tmp directory respectively. The first line in the definition states that lock-file context invariant LFCtx must hold for the path to the lock file we are using. Recall the definition of LFCtx in chapter 6, section 6.2. LFCtx(/mail/.lock) restricts the shared file-system state such that it always contains the /mail which in turn may contain a file named .lock: file-system graphs must be within the set $\mathcal{LF}$(/mail/.lock). The second line in the definition of EmlCtx restricts these states even further to those within the set $\mathcal{EML}(\iota,j,k,l)$ for some inode l. The conjunction between the two effectively restricts the shared file-system state to the intersection $\mathcal{LF}(\text{/mail/.lock}) \cap \mathcal{EML}(\iota,j,k,l)$. It is easy to see that this intersection is non-empty. The third line in the definition of EmlCtx requires that the indivisible guard E is defined for the global file-system region **GFS**. The next five lines state that updating the email client-server state is exclusively defined for the E guard. The final part in the definition of EmlCtx restricts all other guards defined for **GFS** to preserve the file-system sub-graphs identified by the paths /mail/42 and /mail/quarantine. This guarantees that the context does not concurrently divert these paths to a different location.

Assuming the context invariant, we verify that the email client-server interaction in figure 8.17 satisfies the following specification:

$$\left\{\begin{array}{l}\exists FS.\, \mathbf{GFS}(FS) \wedge /\text{mail/tmp/42.msg} \overset{FS}{\longmapsto} \iota \wedge \mathsf{isfile}(FS(\iota)) \wedge \neg /\text{mail/42} \overset{FS}{\longmapsto} - \\ \wedge \neg /\text{mail/quarantine} \overset{FS}{\longmapsto} - \wedge \neg /\text{mail/.lock} \overset{FS}{\longmapsto} - * [\mathsf{E}] * [\mathsf{LF}(/\text{mail/.lock})]\,, \\ \mathbf{GFS}(-) * [\mathsf{E}] * [\mathsf{LF}(/\text{mail/.lock})] * \mathit{delivered} \notin \textsc{Errs} \Rightarrow \mathit{is\_not\_virus}\end{array}\right\}$$

Note that this differs from the specification in section 8.2.1 only in the use of the global file-system region, the requirement that /mail.lock does not initially exist and the guards E and $[\mathsf{LF}(/\text{mail/.lock})]$. Crucially, the postcondition establishes the desired safety property: $\mathit{delivered} \notin \textsc{Errs} \Rightarrow \mathit{is\_not\_virus}$.

In order to verify that the email client-server interaction in figure 8.17 is a refinement of the specification given above, we introduce a new region type **SafeEml**, defined similarly to **UnsafeEml** from section 8.2.2. We take the abstract states of this region from the set $(\mathbb{N}_2^6 \times \textsc{Inodes}) \cup (\{0\} \times \textsc{Inodes}) \cup \{(1, \bot)\}$, as in **UnsafeEml**. The abstract states of **SafeEml** are given the following interpretation:

$$\begin{aligned}
I_r(\mathbf{SafeEml}_\alpha(\iota, j, k, (1, \bot))) &\triangleq \exists FS.\, \mathbf{GFS}(FS) \wedge \mathcal{EFS}(FS, \iota, j, k, (1, \bot)) * [\mathsf{E}] \\
I_r(\mathbf{SafeEml}_\alpha(\iota, j, k, (2, l))) &\triangleq \exists FS.\, \mathbf{GFS}(FS) \wedge \mathcal{EFS}(FS, \iota, j, k, (2, l)) * [\mathsf{E}] \\
I_r(\mathbf{SafeEml}_\alpha(\iota, j, k, (3, l))) &\triangleq \exists FS.\, \mathbf{GFS}(FS) \wedge \mathcal{EFS}(FS, \iota, j, k, (3, l)) * [\mathsf{E}] \\
I_r(\mathbf{SafeEml}_\alpha(\iota, j, k, (4, l))) &\triangleq \exists FS.\, \mathbf{GFS}(FS) \wedge \mathcal{EFS}(FS, \iota, j, k, (4, l)) * [\mathsf{E}] \\
I_r(\mathbf{SafeEml}_\alpha(\iota, j, k, (5, l))) &\triangleq \exists FS.\, \mathbf{GFS}(FS) \wedge \mathcal{EFS}(FS, \iota, j, k, (5, l)) * [\mathsf{E}] \\
I_r(\mathbf{SafeEml}_\alpha(\iota, j, k, (6, l))) &\triangleq \exists FS.\, \mathbf{GFS}(FS) \wedge \mathcal{EFS}(FS, \iota, j, k, (6, l)) * [\mathsf{E}] \\
I_r(\mathbf{SafeEml}_\alpha(\iota, j, k, (0, l))) &\triangleq \mathsf{true}
\end{aligned}$$

This differs from the interpretation of **UnsafeEml** in the use of the global file-system region **GFS** and the additional guard G.

We define the guards G, R, W, F, L and U for **SafeEml**, where the guards G and U are parameterised by a permission $\pi \in (0, 1]$. The guard algebra is defined as follows:

$$\mathsf{F} = \mathsf{R} \bullet \mathsf{W} \qquad \forall \pi_1, \pi_2.\, \pi_1 + \pi_2 \leq 1 \wedge \mathsf{G}(\pi_1 + \pi_2) = \mathsf{G}(\pi_1) \bullet \mathsf{G}(\pi_2) \qquad \mathsf{G}(1) \bullet \mathsf{L} = \mathsf{U}(1)$$

The labelled transition system for **SafeEml** is defined as follows:

$$\mathsf{G}(1) : (1, \bot) \rightsquigarrow \exists \iota.(2, \iota) \qquad \mathsf{G}(1) : \forall \iota \in \textsc{Inodes}.\, (2, \iota) \rightsquigarrow (3, \iota) \qquad \mathsf{G}(1) : \forall \iota \in \textsc{Inodes}.\, (3, \iota) \rightsquigarrow (4, \iota)$$

$$\mathsf{U}(1) : \forall \iota \in \textsc{Inodes}.\, (4, \iota) \rightsquigarrow (5, \iota) \qquad \mathsf{U}(1) : \forall \iota \in \textsc{Inodes}.\, (5, \iota) \rightsquigarrow (6, \iota)$$

$$\mathsf{F} : \forall \iota \in \textsc{Inodes}.\, (5, \iota) \rightsquigarrow (0, \iota) \qquad \mathsf{F} : \forall \iota \in \textsc{Inodes}.\, (6, \iota) \rightsquigarrow (0, \iota)$$

The capability to update the region's state from $(1, \bot)$ to $(4, \iota)$ for some inode ι is given by the guard G as for **UnsafeEml**. However, the capability to update the region's state from $(4, \iota)$ to $(6, \iota)$ is given by the U guard. By the guard algebra defined previously, this guard can be obtained by composing G with L. Thus in order to perform these updates the email server must own both guards. We will use the CAP-style specification for lock files from section 8.1 with the guard L as the resource invariant protected by the lock. Thus the updates from state $(4, \iota)$ to $(6, \iota)$ can only be performed when the

lock is locked and the guard G is owned.

Let $\mathcal{D} \triangleq \mathbb{N}_1^6 \times (\{\bot\} \cup \textsc{Inodes})$.

$\mathsf{EmlCtx}(\iota, j, k) \vdash$

	$\Vert$	`mkdir`(/mail/42);
	$\Vert$	`rename`(/mail/tmp/42.msg, /mail/42/unsafe.eml);
	$\Vert$	`rename`(/mail/42, /mail/quarantine);
`lock`(/mail/.lock);	$\Vert$	`lock`(/mail/.lock);
let *delivered* =	$\Vert$	`rename`(/mail/quarantine/unsafe.eml, /mail/quarantine/msg.eml);
`stat`(/mail/42/msg.eml);	$\Vert$	**let** *is_not_virus* = `run_av_scan`();
`unlock`(/mail/.lock)	$\Vert$	**if** *is_not_virus* **then**
	$\Vert$	`rename`(/mail/quarantine, /mail/42);
	$\Vert$	**fi**
	$\Vert$	`unlock`(/mail/.lock)

$\sqsubseteq$ by figure 8.10, figure 8.12, Parallel and HCons

$$\left\{\begin{array}{l}\big(\exists(n,l) \in \mathbb{N}_1^6 \times \mathcal{D}.\, \mathbf{SafeEml}_\alpha(\iota, j, k, (n,l)) * [\mathsf{R}]_\alpha * \mathsf{isLock}(s, /\text{mail}/.\text{lock})\big) \\ * \left(\mathbf{SafeEml}_\alpha(\iota, j, k, (1, \bot)) * [\mathsf{G}(1)]_\alpha * [\mathsf{W}]_\alpha * \mathsf{isLock}(s, /\text{mail}/.\text{lock})\right), \\ \big(\exists(n,l) \in \mathbb{N}_1^6 \times \mathcal{D}.\, \mathbf{SafeEml}_\alpha(\iota, j, (n,l)) * [\mathsf{R}]_\alpha * \mathit{delivered} \notin \textsc{Errs} \Rightarrow n = 6\big) \\ * \big(\exists(n,l) \in \mathbb{N}_5^6 \times \mathcal{D}.\, \mathbf{SafeEml}_\alpha(\iota, j, k, (n,l)) * [\mathsf{G}(1)]_\alpha * [\mathsf{W}]_\alpha * n = 6 \Rightarrow \mathit{is\_not\_virus}\big)\end{array}\right\}$$

$\sqsubseteq$ by HCons

$$\left\{\begin{array}{l}\mathbf{SafeEml}_\alpha(\iota, j, k, (1, \bot)) * [\mathsf{G}(1)]_\alpha * [\mathsf{F}]_\alpha * \mathsf{isLock}(s, /\text{mail}/.\text{lock}), \\ \exists(n,l) \in \mathbb{N}_5^6 \times \mathcal{D}.\, \mathbf{SafeEml}_\alpha(\iota, j, k, (n,l)) * [\mathsf{G}(1)]_\alpha * [\mathsf{F}]_\alpha \\ * \mathit{delivered} \notin \textsc{Errs} \Rightarrow \mathit{is\_not\_virus}\end{array}\right\}$$

$\sqsubseteq$ by EElimHoare on α and s, CreateRegion and HCons similarly to figure 8.5 and EElimHoare on j, k

$$\left\{\begin{array}{l}\mathbf{GFS}(FS) \wedge \exists j, k.\, FS(\iota_0)(\text{mail}) = j \wedge FS(j)(\text{tmp}) = k \wedge FS(k)(42.\text{msg}) = \iota \wedge \mathsf{isfile}(FS(\iota)) \\ \wedge\, 42 \notin FS(j) \wedge \text{quarantine} \notin FS(j) \wedge .\text{lock} \notin FS(j) * [\mathsf{E}] * [\mathsf{LF}(/\text{mail}/.\text{lock})], \\ \mathbf{GFS}(-) * [\mathsf{E}] * [\mathsf{LF}(/\text{mail}/.\text{lock})] * \mathit{delivered} \notin \textsc{Errs} \Rightarrow \mathit{is\_not\_virus}\end{array}\right\}$$

$\sqsubseteq$ by HCons

$$\left\{\begin{array}{l}\exists FS.\, \mathbf{GFS}(FS) \wedge /\text{mail/tmp/42.msg} \xmapsto{FS} \iota \wedge \mathsf{isfile}(FS(\iota)) \wedge \neg /\text{mail}/42 \xmapsto{FS} - \\ \wedge\, \neg /\text{mail/quarantine} \xmapsto{FS} - \wedge \neg /\text{mail}/.\text{lock} \xmapsto{FS} - * [\mathsf{E}] * [\mathsf{LF}(/\text{mail}/.\text{lock})], \\ \mathbf{GFS}(-) * [\mathsf{E}] * [\mathsf{LF}(/\text{mail}/.\text{lock})] * \mathit{delivered} \notin \textsc{Errs} \Rightarrow \mathit{is\_not\_virus}\end{array}\right\}$$

Figure 8.18.: Proof of safe email client-server interaction assuming $\mathsf{EmlCtx}(\iota, j, k)$.

We prove that the email client-server interaction is a refinement of the specification given previously in figure 8.18. The proof follows a similar pattern to the unsafe email client-server proof in figure 8.9, with the main difference being the use of the lock file.

The refinement to the email client thread is given in figure 8.19. Note that after performing `lock`(/mail.lock) we obtain ownership of the L guard. Ownership of this guard together with the absence of the G, which owned by the server, guarantees that once the lock is locked the **SafeEml** cannot be in state 5: the server requires ownership of L in order to perform this update. Recall from section 8.2.2, figure 8.10, that `stat` could succeed if the **UnsafeEml** region was in state 5. Now, `stat` only succeeds if the region is in state 6; that is, only if the path /mail/42/msg.eml actually exists.

Finally, the refinement to the server thread is given in figure 8.20. Here, after the lock is locked we obtain ownership of the L which we subsequently compose with the G guard to obtain U. Then, ownership of U grants the capability to perform the `rename`s between `lock` and `unlock`.

Let $\mathcal{D} \triangleq \mathbb{N}_1^6 \times (\{\bot\} \cup \textsc{Inodes})$.

$\mathsf{EmlCtx}(\iota, j, k) \vdash$

SEQ

`lock`(/mail/.lock);

$\sqsubseteq$ by section 8.1 with $\mathsf{Inv}= [\mathsf{L}]_\alpha$ and HFRAME

$$\left\{\begin{array}{l}\mathsf{isLock}(s, /\text{mail}/.\text{lock}) * \exists (n,l) \in \mathbb{N}_1^6 \times \mathcal{D}.\, \mathbf{SafeEml}_\alpha(\iota, j, k, (n,l)) * [\mathsf{R}]_\alpha\,, \\ \mathsf{isLock}(s, /\text{mail}/.\text{lock}) * \mathsf{Locked}(s, /\text{mail}/.\text{lock}) * [\mathsf{L}]_\alpha \\ * \exists (n,l) \in \mathbb{N}_1^6 \times \mathcal{D}.\, \mathbf{SafeEml}_\alpha(\iota, j, k, (n,l)) * [\mathsf{R}]_\alpha \end{array}\right\}$$

$\sqsubseteq$ by HCONS

$$\left\{\begin{array}{l}\mathsf{isLock}(s, /\text{mail}/.\text{lock}) * \exists (n,l) \in \mathbb{N}_1^6 \times \mathcal{D}.\, \mathbf{SafeEml}_\alpha(\iota, j, k, (n,l)) * [\mathsf{R}]_\alpha\,, \\ \mathsf{isLock}(s, /\text{mail}/.\text{lock}) * \mathsf{Locked}(s, /\text{mail}/.\text{lock}) * [\mathsf{L}]_\alpha \\ * \exists (n,l) \in \mathbb{N}_1^4 \cup \{6\} \times \mathcal{D}.\, \mathbf{SafeEml}_\alpha(\iota, j, k, (n,l)) * [\mathsf{R}]_\alpha \end{array}\right\}$$

let *delivered* =

`stat`(/mail/42/msg.eml);

$\sqsubseteq$ similarly to figure 8.10 and HFRAME

$$\left\{\begin{array}{l}[\mathsf{L}]_\alpha * \exists (n,l) \in \mathbb{N}_1^4 \cup \{6\} \times \mathcal{D}.\, \mathbf{SafeEml}_\alpha(\iota, j, k, (n,l)) * [\mathsf{R}]_\alpha\,, \\ [\mathsf{L}]_\alpha * \exists (n,l) \in \mathbb{N}_1^4 \cup \{6\} \times \mathcal{D}.\, \mathbf{SafeEml}_\alpha(\iota, j, k, (n,l)) * [\mathsf{R}]_\alpha * \mathit{delivered} \notin \textsc{Errs} \Rightarrow n = 6 \end{array}\right\}$$

`unlock`(/mail/.lock);

$\sqsubseteq$ by section 8.1 with $\mathsf{Inv}= [\mathsf{L}]_\alpha$, HCONS and HFRAME

$$\left\{\begin{array}{l}\mathsf{isLock}(s, /\text{mail}/.\text{lock}) * \mathsf{Locked}(s, /\text{mail}/.\text{lock}) * [\mathsf{L}]_\alpha \\ * \exists (n,l) \in \mathbb{N}_1^4 \cup \{6\} \times \mathcal{D}.\, \mathbf{SafeEml}_\alpha(\iota, j, k, (n,l)) * [\mathsf{R}]_\alpha * \mathit{delivered} \notin \textsc{Errs} \Rightarrow n = 6, \\ \mathsf{isLock}(s, /\text{mail}/.\text{lock}) * \exists (n,l) \in \mathbb{N}_1^6 \times \mathcal{D}.\, \mathbf{SafeEml}_\alpha(\iota, j, k, (n,l)) \\ * [\mathsf{R}]_\alpha * \mathit{delivered} \notin \textsc{Errs} \Rightarrow n = 6 \end{array}\right\}$$

$\sqsubseteq$ $\left\{\begin{array}{l}\mathsf{isLock}(s, /\text{mail}/.\text{lock}) * \exists (n,l) \in \mathbb{N}_1^6 \times \mathcal{D}.\, \mathbf{SafeEml}_\alpha(\iota, j, k, (n,l)) * [\mathsf{R}]_\alpha\,, \\ \mathsf{isLock}(s, /\text{mail}/.\text{lock}) * \exists (n,l) \in \mathbb{N}_1^6 \times \mathcal{D}.\, \mathbf{SafeEml}_\alpha(\iota, j, k, (n,l)) * [\mathsf{R}]_\alpha * \mathit{delivered} \notin \textsc{Errs} \Rightarrow n = 6 \end{array}\right\}$

Figure 8.19.: Refinement to the safe email client assuming $\mathsf{EmlCtx}(\iota, j, k)$.

The specification we have verified for the safe email client-server interaction in figure 8.18 holds for all contexts that maintain the EmlCtx context invariant. Consequently, it holds for the trivial context where the email client and server are the only threads accessing the file system. In fact, we can show that it is a refinement of a specification that owns the entire file system as follows:

$$\mathsf{EmlCtx}(\iota, j, k) \vdash \left\{\begin{array}{l}\exists FS.\, \mathbf{GFS}(FS) \wedge /\text{mail/tmp/42.msg} \overset{FS}{\longmapsto} \iota \wedge \mathsf{isfile}(FS(\iota)) \wedge \neg /\text{mail}/42 \overset{FS}{\longmapsto} - \\ \wedge \neg /\text{mail/quarantine} \overset{FS}{\longmapsto} - \wedge \neg /\text{mail}/.\text{lock} \overset{FS}{\longmapsto} - * [\mathsf{E}] * [\mathsf{LF}(/\text{mail}/.\text{lock})]\,, \\ \mathbf{GFS}(-) * [\mathsf{E}] * [\mathsf{LF}(/\text{mail}/.\text{lock})] * \mathit{delivered} \notin \textsc{Errs} \Rightarrow \mathit{is\_not\_virus} \end{array}\right\}$$

$\sqsubseteq$ by definition 57 and AWEAKEN1

$$\left\{\begin{array}{l}\exists FS.\, \mathbf{GFS}(FS) \wedge /\text{mail/tmp/42.msg} \overset{FS}{\longmapsto} \iota \wedge \mathsf{isfile}(FS(\iota)) \wedge \neg /\text{mail}/42 \overset{FS}{\longmapsto} - \\ \wedge \neg /\text{mail/quarantine} \overset{FS}{\longmapsto} - \wedge \neg /\text{mail}/.\text{lock} \overset{FS}{\longmapsto} - * [\mathsf{E}] * [\mathsf{LF}(/\text{mail}/.\text{lock})] * \mathsf{EmlCtx}(\iota, j, k)\,, \\ \mathbf{GFS}(-) * [\mathsf{E}] * [\mathsf{LF}(/\text{mail}/.\text{lock})] * \mathsf{EmlCtx}(\iota, j, k) * \mathit{delivered} \notin \textsc{Errs} \Rightarrow \mathit{is\_not\_virus} \end{array}\right\}$$

$\sqsubseteq$ by HCONS

$$\left\{\begin{array}{l}\exists FS.\, \mathbf{GFS}(FS) \wedge /\text{mail/tmp/42.msg} \overset{FS}{\longmapsto} \iota \wedge \mathsf{isfile}(FS(\iota)) \wedge \neg /\text{mail}/42 \overset{FS}{\longmapsto} - \\ \wedge \neg /\text{mail/quarantine} \overset{FS}{\longmapsto} - \wedge \neg /\text{mail}/.\text{lock} \overset{FS}{\longmapsto} - * [\mathsf{E}] * [\mathsf{LF}(/\text{mail}/.\text{lock})]\,, \\ \mathbf{GFS}(-) * [\mathsf{E}] * [\mathsf{LF}(/\text{mail}/.\text{lock})] * \mathit{delivered} \notin \textsc{Errs} \Rightarrow \mathit{is\_not\_virus} \end{array}\right\}$$

$\sqsubseteq$ by CREATEREGION and HCONS

$$\left\{\begin{array}{l}\mathsf{fs}(FS) \wedge /\text{mail/tmp/42.msg} \xmapsto{FS} \iota \wedge \mathsf{isfile}(FS(\iota)) \wedge \neg/\text{mail/42} \xmapsto{FS} - \\ \wedge \neg/\text{mail/quarantine} \xmapsto{FS} - \wedge \neg/\text{mail/.lock} \xmapsto{FS} -, \\ \mathsf{fs}(-) * \mathit{delivered} \notin \textsc{Errs} \Rightarrow \mathit{is\_not\_virus}\end{array}\right\}$$

Reading the derivation from the specification at the bottom up to the top specification, in the first refinement step we use HCons with the CreateRegion view-shift to create the global file-system region. Additional with HCons we weaken the precondition by existentially quantifying the file-system graph FS. At this point the precondition satisfies the EmlCtx context invariant. Thus, in the next step we apply HCons introducing $\mathsf{EmlCtx}(\iota, j, k)$ in the precondition and postcondition (by strengthening). Recall that from definition 57 the Hoare specification statement is just a special case of the atomic specification statement. In the last refinement step we use the definition of the Hoare specification statement in terms of the atomic statement and apply AWeaken1 to bring EmlCtx to the public part. This, by definition 57 results in the Hoare specification statement invariant under $\mathsf{EmlCtx}(\iota, j, k)$ at the top of the derivation, which is the specification we have proven in figure 8.18. This refinement proof serves as an example of how a context invariant can be introduced.

$\mathsf{EmlCtx}(\iota, j, k) \vdash$

HFrame, HCons and Seq

$\texttt{mkdir}(/\text{mail}/42);$

$\quad \sqsubseteq$ similarly to figure 8.13

$\quad\quad \{\mathbf{SafeEml}_\alpha(\iota, j, k, (1, \bot)) * [\mathsf{G}(1)]_\alpha * [\mathsf{W}]_\alpha\,,\ \exists l.\, \mathbf{SafeEml}_\alpha(\iota, j, k, (2, l)) * [\mathsf{G}(1)]_\alpha * [\mathsf{W}]_\alpha\}$

$\texttt{rename}(/\text{mail}/\text{tmp}/42.\text{msg}, /\text{mail}/42/\text{unsafe.eml});$

$\quad \sqsubseteq$ similarly to figure 8.14 and EElimHoare

$\quad\quad \{\exists l.\, \mathbf{SafeEml}_\alpha(\iota, j, k, (2, l)) * [\mathsf{G}(1)]_\alpha * [\mathsf{W}]_\alpha\,,\ \exists l.\, \mathbf{SafeEml}_\alpha(\iota, j, k, (3, l)) * [\mathsf{G}(1)]_\alpha * [\mathsf{W}]_\alpha\}$

$\texttt{rename}(/\text{mail}/42, /\text{mail}/\text{quarantine});$

$\quad \sqsubseteq$ similarly to figure 8.14 and EElimHoare

$\quad\quad \{\exists l.\, \mathbf{SafeEml}_\alpha(\iota, j, k, (3, l)) * [\mathsf{G}(1)]_\alpha * [\mathsf{W}]_\alpha\,,\ \exists l.\, \mathbf{SafeEml}_\alpha(\iota, j, k, (4, l)) * [\mathsf{G}(1)]_\alpha * [\mathsf{W}]_\alpha\}$

$\texttt{lock}(/\text{mail}/.\text{lock});$

$\quad \sqsubseteq$ by section 8.1 and HFrame

$$\left\{\begin{array}{l}\mathsf{isLock}(s, /\text{mail}/.\text{lock}) * \exists l.\, \mathbf{SafeEml}_\alpha(\iota, j, k, (4, l)) * [\mathsf{G}(1)]_\alpha * [\mathsf{W}]_\alpha\,,\\ \mathsf{isLock}(s, /\text{mail}/.\text{lock}) * \mathsf{Locked}(s, /\text{mail}/.\text{lock}) * [\mathsf{L}]_\alpha\\ \quad * \exists l.\, \mathbf{SafeEml}_\alpha(\iota, j, k, (4, l)) * [\mathsf{G}(1)]_\alpha * [\mathsf{W}]_\alpha\end{array}\right\}$$

$\texttt{rename}(/\text{mail}/\text{quarantine}/\text{unsafe.eml}, /\text{mail}/\text{quarantine}/\text{msg.eml});$

$\sqsubseteq$ similarly to figure 8.14 and EElimHoare

$\quad \{\exists l.\, \mathbf{SafeEml}_\alpha(\iota, j, k, (4, l)) * [\mathsf{U}(1)]_\alpha * [\mathsf{W}]_\alpha\,,\ \exists l.\, \mathbf{SafeEml}_\alpha(\iota, j, k, (5, l)) * [\mathsf{U}(1)]_\alpha * [\mathsf{W}]_\alpha\}$

$\mathbf{let}\ \mathit{is\_not\_virus} = \texttt{run\_av\_scan}();$

$\quad \sqsubseteq$ by specification, AEElim, ACons, AFrame, OpenRegion and AWeaken2

$$\left\{\begin{array}{l}\exists l.\, \mathbf{SafeEml}_\alpha(\iota, j, k, (5, l)) * [\mathsf{G}(1)]_\alpha * [\mathsf{W}]_\alpha\,,\\ \exists l.\, \mathbf{SafeEml}_\alpha(\iota, j, k, (5, l)) * [\mathsf{G}(1)]_\alpha * [\mathsf{W}]_\alpha * \mathit{is\_not\_virus} \in \mathbb{B}\end{array}\right\}$$

IfThenElse

$\mathbf{if}\ \mathit{is\_not\_virus}\ \mathbf{then}$

$\quad \texttt{rename}(/\text{mail}/\text{quarantine}, /\text{mail}/42);$

$\quad\quad \sqsubseteq$ similarly to figure 8.14 and EElimHoare

$$\left\{\begin{array}{l}\exists l.\, \mathbf{SafeEml}_\alpha(\iota, j, k, (5, l)) * [\mathsf{U}(1)]_\alpha * [\mathsf{W}]_\alpha * \mathit{is\_not\_virus},\\ \exists l.\, \mathbf{SafeEml}_\alpha(\iota, j, k, (6, l)) * [\mathsf{U}(1)]_\alpha * [\mathsf{W}]_\alpha * \mathit{is\_not\_virus}\end{array}\right\}$$

$\mathbf{fi}$

$$\sqsubseteq \left\{\begin{array}{l}\exists l.\, \mathbf{SafeEml}_\alpha(\iota, j, k, (5, l)) * [\mathsf{U}(1)]_\alpha * [\mathsf{W}]_\alpha * \mathit{is\_not\_virus} \in \mathbb{B},\\ \exists n \in \mathbb{N}^6_5, l.\, \mathbf{SafeEml}_\alpha(\iota, j, k, (n, l)) * [\mathsf{U}(1)]_\alpha * [\mathsf{W}]_\alpha * n = 6 \Rightarrow \mathit{is\_not\_virus}\end{array}\right\}$$

$\texttt{unlock}(/\text{mail}/.\text{lock})$

$\quad \sqsubseteq$ by section 8.1 with $\mathsf{Inv} = [\mathsf{L}]_\alpha$ and HFrame

$$\left\{\begin{array}{l}\mathsf{isLock}(s, /\text{mail}/.\text{lock}) * \mathsf{Locked}(s, /\text{mail}/.\text{lock})\\ \quad * \exists n \in \mathbb{N}^6_5, l.\, \mathbf{SafeEml}_\alpha(\iota, j, k, (n, l)) * [\mathsf{L}]_\alpha * [\mathsf{G}(1)]_\alpha * [\mathsf{W}]_\alpha * n = 6 \Rightarrow \mathit{is\_not\_virus},\\ \mathsf{isLock}(s, /\text{mail}/.\text{lock}) * \exists n \in \mathbb{N}^6_5, l.\, \mathbf{SafeEml}_\alpha(\iota, j, k, (n, l))\\ \quad * [\mathsf{G}(1)]_\alpha * [\mathsf{W}]_\alpha * n = 6 \Rightarrow \mathit{is\_not\_virus}\end{array}\right\}$$

$$\sqsubseteq \left\{\begin{array}{l}\mathsf{isLock}(s, /\text{mail}/.\text{lock}) * \mathbf{SafeEml}_\alpha(\iota, j, k, (1, \bot)),\\ \mathsf{isLock}(s, /\text{mail}/.\text{lock}) * \exists n \in \mathbb{N}^6_5, l.\, \mathbf{SafeEml}_\alpha(\iota, j, k, (n, l)) * [\mathsf{G}(1)]_\alpha * [\mathsf{W}]_\alpha\\ * n = 6 \Rightarrow \mathit{is\_not\_virus}\end{array}\right\}$$

Figure 8.20.: Refinement to the safe email server assuming $\mathsf{EmlCtx}(\iota, j, k)$.

8.3. Case Study: Named Pipes

We now apply our reasoning and file-system specification to study named pipes. Named pipes are specified as part of the POSIX standard and provide a widely used mechanism for inter-process communication.

A named pipe is a special type of file. Like a regular file, a named-pipe file stores an arbitrary sequence of bytes. However, in contrast to a regular file, the behaviour of I/O operations acting on a named-pipe file is restricted. First, named pipes do not support random access. The current file offset associated with a file descriptor opened on a named pipe is irrelevant, and it cannot be modified by `lseek`. Second, writing and reading from a named-pipe file behaves on a *first in, first out* (FIFO) basis. A `write` appends the user supplied sequence of bytes to the end of the named-pipe file. A `read` reads the user defined number of bytes from the beginning of the named-pipe file, while simultaneously removing them. Therefore, named pipes behave similarly to queues and are also referred to as FIFOs.

Imagine an otherwise POSIX compliant system which does not support named pipes. The named-pipe functionality will then have to be implemented as a client module. This presents an opportunity to demonstrate the scalability of our client reasoning, to client modules that implement a concurrent data structure on top of the file system – in this case, a concurrent queue-like structure – and define both atomic and non-atomic operations to access it. Instead of extending our POSIX fragment specification to include named pipes, we develop a module that implements the named-pipe functionality via regular file I/O. We then apply our reasoning and show that the named-pipe module implementation satisfies its specification.

8.3.1. Specification

There can be multiple readers and writers for the named pipe, reading and writing arbitrary numbers of bytes. We use the abstract predicate $\mathsf{fifo}(s, \iota, wr, rd, \overline{y})$ to assert the existence of named-pipe file with inode ι, that is opened for writing wr times and opened for reading rd times, and with contents given by the byte sequence $\overline{y}$. The first argument, $s \in \mathbb{T}_3$, ranges over an abstract type that captures implementation defined invariant information.

Named pipes are constructed with the `mkfifo` operation, that works similarly to `open`, except that it creates a new empty named pipe, instead of a regular file, with the following specification:

$$
\begin{array}{l}
\texttt{mkfifo}(\mathit{path}) \\
\quad \sqsubseteq \textbf{let } p = \texttt{dirname}(\mathit{path}); \\
\qquad \textbf{let } a = \texttt{basename}(\mathit{path}); \\
\qquad \textbf{let } r = \texttt{resolve}(p, \iota_0); \\
\qquad \textbf{if } \neg\mathsf{iserr}(r) \textbf{ then } \texttt{return } \texttt{new\_fifo}(r, a) \\
\qquad \textbf{else } \texttt{return } r \textbf{ fi}
\end{array}
$$

where `new_fifo` is defined in figure 8.21, along with other lower level abstract named-pipe operations. According to the specification, `mkfifo` first resolves the path prefix p. If the resolution succeeds, it creates a new named-pipe file and adds a link to it, named a, in the resolved directory. If a link named a already exists, then `ENOENT` is returned. If p does not resolve to a directory, then `ENOTDIR` is returned.

$$\textbf{let } \texttt{new\_fifo}(\iota, a) \triangleq$$
$$\exists \iota'. \begin{pmatrix} \forall FS. \langle \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota)) \,,\ a \notin FS(\iota) \Rightarrow \mathsf{fs}(FS[\iota \mapsto FS(\iota)[a \mapsto \iota']] \uplus \iota' \mapsto \epsilon) * \texttt{ret} = 0 \rangle; \\ I \vdash [\texttt{ret} = 0 \Rightarrow \exists s. \mathsf{fifo}(s, \iota', 0, 0, \epsilon)] \end{pmatrix}$$
$$\sqcap\ \texttt{return eexist}(\iota, a)$$
$$\sqcap\ \texttt{return enotdir}(\iota)$$

$$\textbf{let } \texttt{fifo\_open\_read}(\iota) \triangleq$$
$$\forall wr, rd, \overline{y}. \langle \mathsf{fifo}(s, \iota, wr, rd, \overline{y}) \,,\ \mathsf{fifo}(s, \iota, wr, rd+1, \overline{y}) * \mathsf{ffd}(\texttt{ret}, r, \texttt{O\_RDONLY}) \rangle$$
$$\sqcap\ \texttt{return eisdir}(\iota)$$

$$\textbf{let } \texttt{fifo\_open\_write}(\iota) \triangleq$$
$$\forall wr, rd, \overline{y}. \langle \mathsf{fifo}(s, \iota, wr, rd, \overline{y}) \,,\ \mathsf{fifo}(s, \iota, wr+1, rd, \overline{y}) * \mathsf{ffd}(\texttt{ret}, r, \texttt{O\_WRONLY}) \rangle$$
$$\sqcap\ \texttt{return eisdir}(\iota)$$

$$\textbf{let } \texttt{fifo\_write\_readers}(fd, ptr, sz) \triangleq$$
$$\forall wr, rd, \overline{y_s}. \left\langle \begin{array}{l} \mathsf{ffd}(fd, \iota, \texttt{O\_WRONLY}) * \mathsf{fifo}(s, \iota, wr, rd, \overline{y_s}) * \mathsf{buf}(ptr, \overline{y_t}) \wedge \mathsf{len}(\overline{y_t}) = sz, \\ rd > 0 \Rightarrow \mathsf{ffd}(fd, \iota, \texttt{O\_WRONLY}) * \mathsf{fifo}(s, \iota, wr, rd, \overline{y_s} :: \overline{y_t}) * \mathsf{buf}(ptr, \overline{y_t}) * \texttt{ret} = sz \end{array} \right\rangle$$

$$\textbf{let } \texttt{fifo\_read\_nowriters}(fd) \triangleq$$
$$\forall wr, rd, \overline{y_s}. \left\langle \begin{array}{l} \mathsf{ffd}(fd, \iota, \texttt{O\_RDONLY}) * \mathsf{fifo}(s, \iota, wr, rd, \overline{y_s}) \,, \\ w = 0 \wedge \overline{y_s} = \epsilon \Rightarrow \mathsf{ffd}(fd, \iota, \texttt{O\_RDONLY}) * \mathsf{fifo}(s, \iota, wr, rd, \overline{y_s}) * \texttt{ret} = 0 \end{array} \right\rangle$$

$$\textbf{let } \texttt{fifo\_read\_partial}(fd, ptr, sz) \triangleq$$
$$\forall wr, rd, \overline{y_s}. \left\langle \begin{array}{l} \mathsf{ffd}(fd, \iota, \texttt{O\_RDONLY}) * \mathsf{fifo}(s, \iota, wr, rd, \overline{y_s}) * \mathsf{buf}(ptr, \overline{y_t}) \wedge \mathsf{len}(\overline{y_t}) = sz, \\ w > 0 \wedge \mathsf{len}(\overline{y_s}) < sz \Rightarrow \mathsf{ffd}(fd, \iota, \texttt{O\_RDONLY}) * \mathsf{fifo}(s, \iota, wr, rd, \epsilon) \\ \qquad\qquad * \mathsf{buf}(ptr, \overline{y_t} \upharpoonleft \overline{y_s}) * \texttt{ret} = \mathsf{len}(\overline{y_s}) \end{array} \right\rangle$$

$$\textbf{let } \texttt{fifo\_read\_complete}(fd, ptr, sz) \triangleq$$
$$\forall wr, rd, \overline{y_s}. \left\langle \begin{array}{l} \mathsf{ffd}(fd, \iota, \texttt{O\_RDONLY}) * \mathsf{fifo}(s, \iota, wr, rd, \overline{y_s}) * \mathsf{buf}(ptr, \overline{y_t}) \wedge \mathsf{len}(\overline{y_t}) = sz, \\ w > 0 \wedge \mathsf{len}(\overline{y_s}) >= sz \Rightarrow \mathsf{ffd}(fd, \iota, \texttt{O\_RDONLY}) * \mathsf{fifo}(s, \iota, wr, rd, \mathsf{skipseq}(sz, \overline{y_s})) \\ \qquad\qquad * \mathsf{buf}(ptr, \mathsf{subseq}(0, sz, \overline{y_s})) * \texttt{ret} = sz \end{array} \right\rangle$$

Figure 8.21.: Specification of atomic named-pipe operations.

$$\textbf{let } \texttt{fifo\_eaccess}(fd, \mathit{flag}) \triangleq$$
$$\langle \mathsf{ffd}(fd, \iota, \mathit{flag}) \,,\ \mathsf{ffd}(fd, \iota, \mathit{flag}) * \texttt{ret} = \texttt{EACCESS} \rangle$$

$$\textbf{let } \texttt{fifo\_epipe}(fd) \triangleq$$
$$\forall wr, rd, \overline{y_s}. \left\langle \begin{array}{l} \mathsf{ffd}(fd, \iota, \texttt{O\_WRONLY}) * \mathsf{fifo}(s, \iota, wr, rd, \overline{y_s}) \,, \\ rd = 0 \Rightarrow \mathsf{ffd}(fd, \iota, \texttt{O\_WRONLY}) * \mathsf{fifo}(s, \iota, wr, rd, \overline{y_s}) * \texttt{ret} = \texttt{EPIPE} \end{array} \right\rangle$$

Figure 8.22.: Specification of named-pipe specific error cases.

Consider the definition of `new_fifo` in figure 8.21. The success case consists of two steps. The atomic statement in the first step states that if a link named a does not exist within the directory ι, then a link named a to a new empty regular file with inode ι' is created atomically, within that directory, and the return variable `ret` is set to 0 to indicate success. The assumption statement in the second step states that if the first step was successful (`ret` $= 0$), then at the end of this step the file with inode ι' is an empty named pipe.

Recall from chapter 7, section 7.6, that assumption statements are not atomic. This means that clients of `mkfifo` can rely on the link to the named-pipe file to be created atomically, but not on the creation of the named pipe itself. This reflects the fact that implementations can create a named pipes in a sequence of steps, first creating the file for storing the named pipe and then initialising that file to an empty named pipe. Even though the file is created atomically, the subsequent initialisation is implementation defined and thus the specification does not impose any requirements other than the fact that at the end we obtain an empty named pipe.

Note that the assumption statement holds under the invariant I. This invariant denotes the context invariant of the particular implementation of the named-pipe module. The invariant comes in effect only after the link to the named pipe has been created.

Like any other file, a named pipe must first be opened for I/O in order to read or write using a file descriptor. For presentation simplicity, we consider separate operations for opening, reading, writing and closing opened named pipes than the I/O operations for regular files. A named pipe can be either opened for reading or writing, but not both, using `fifo_open` with the following specification:

```
fifo_open(path, mode)
  ⊑ let r = resolve(path, ι0);
    if ¬iserr(r) then
      if mode = O_RDONLY then return fifo_open_read(r)
      else if mode = O_WRONLY then return fifo_open_write(r)
      else return EACCESS fi
    else return r fi
```

The operation takes as arguments the path to the named pipe and the I/O mode *mode*, with possible values `O_RDONLY` and `O_WRONLY`, indicating opening for reading or writing respectively. It first resolves the path to the named pipe and, if successful, opens it for I/O according to the value of *mode*, allocating and returning a file descriptor for the named pipe. We use the abstract predicate $\mathsf{ffd}(s, fd, \iota, mode)$ to assert a named-pipe file descriptor fd, for the named pipe with inode ι, opened with mode *mode*. The first argument $s \in \mathbb{T}_3$, ranges over the abstract type $\mathbb{T}_3$ capturing the same implementation defined invariant information as the first argument to the fifo predicate. We use a different predicate for named-pipe file descriptors from regular file descriptors because the notions of offset and random access are undefined for named pipes.

A named-pipe file descriptor allows us to read an arbitrary number of bytes from the named pipe to a heap buffer, using the `fifo_read` operation. Any bytes read are also removed from the named

pipe. The specification of `fifo_read` is given as follows:

$$
\begin{aligned}
&\texttt{fifo\_read}(fd, ptr, sz)\\
&\quad \sqsubseteq \quad \texttt{return fifo\_read\_partial}(fd, ptr, sz)\\
&\qquad \sqcap \texttt{return fifo\_read\_complete}(fd, ptr, sz)\\
&\qquad \sqcap \texttt{return fifo\_read\_nowriters}(fd)\\
&\qquad \sqcap \texttt{return fifo\_eaccess}(fd, \texttt{O\_WRONLY})
\end{aligned}
$$

In contrast to the previous operations, `fifo_read` is atomic. We use the predicate $\mathsf{buf}(ptr, \overline{y})$ to describe the heap buffer at address ptr storing the sequence of bytes $\overline{y}$. We denote the length of a sequence $\overline{y}$ with $\mathsf{len}(\overline{y})$, and $\overline{y_s} :: \overline{y_t}$ denotes the concatenation of sequences $\overline{y_s}$ and $\overline{y_t}$. There are four demonic cases. The case of `fifo_read_partial` specifies an attempt to read a larger number of bytes than what is stored in the named pipe. In this case, all the bytes in the named pipe are removed from the named pipe and placed into the heap buffer with address ptr. Note that the number of bytes read will be smaller than sz. In contrast, `fifo_read_complete` specifies the case of being able read exactly sz bytes. The case of `fifo_read_nowriters` specifies that if there are no writers for the named pipe, or the named pipe is empty, `fifo_read` immediately returns 0. Finally, `fifo_eaccess`, defined in figure 8.22 along with other named-pipe specific error cases, specifies that if we attempt to read from a named pipe with a named-pipe file descriptor opened for writing, then the `EACCESS` error code is returned.

A named-pipe file descriptor allows us to write an arbitrary number of bytes to the named pipe from a heap buffer, using the `fifo_write` operation specified as follows:

$$
\begin{aligned}
&\texttt{fifo\_write}(fd, ptr, sz)\\
&\quad \sqsubseteq \quad \texttt{return fifo\_write\_readers}(fd, ptr, sz)\\
&\qquad \sqcap \texttt{return fifo\_epipe}(fd)\\
&\qquad \sqcap \texttt{return fifo\_eaccess}(fd, \texttt{O\_RDONLY})
\end{aligned}
$$

There are three demonic cases. The case of `fifo_write_readers` specifies that the contents of the heap buffer with address ptr are appended to the named pipe, with the proviso that the named pipe is opened for reading $rd > 0$. In contrast, `fifo_epipe`, defined in figure 8.22, specifies that if the named pipe is not opened for reading ($rd = 0$), then the `EPIPE` error code is returned. Finally, `fifo_eaccess` specifies that if we use `fifo_write` with a file descriptor for reading, then the `EACCESS` error code is returned.

8.3.2. Implementation

We study an implementation of the named-pipe specification given previously, purely in terms of regular file I/O and lock files (section 8.1). The implementation stores metadata and other information in a header block at the file's beginning, following the format given below:

Header Block

writers	readers	data_offset	name_size	name

The header block begins at file-offset 0, and stores in order: the number of writers, the number of readers, the file-offset at which the contents of the named pipe begin, the size of the filename of the

pipe, and the name of the pipe itself. The header block is followed by the byte contents stored in the named pipe beginning at the data offset stored in the header. If the pipe is empty, there are no available bytes to read and the data offset in the header has value -1.

Implementation of `mkfifo`

We implement the `mkfifo` operation as follows:

```
let mkfifo(path) ≜
  let a = basename(path);
  let fd = open(path, O_EXCL|O_CREAT|O_RDWR);
  if ¬iserr(fd) then
    write_fifo_header(fd, a);
    close(fd);
    return 0
  else return fd fi
```

Initially, we attempt to create and open a new regular file at the end of *path* for reading and writing with `open`. If `open` is unsuccessful, `mkfifo` will either return the `EEXIST` error if the named pipe already exists at the end of *path*, or the `ENOTDIR` error if the path-prefix does not resolve to a directory. Otherwise `open` creates a new regular file and opens it for I/O returning a file descriptor. To turn this new file into a named-pipe, we proceed to write the initial header block of the named-pipe with `write_fifo_header`, after which we close the file descriptor and return 0.

We implement `write_fifo_header` as follows:

```
let write_fifo_header(fd, a) ≜
  let h = malloc(4 * sizeof(int) + len(a));
  memwrite(h, i2b(0) :: i2b(0) :: i2b(−1) :: len(a) :: a);
  write(fd, h, 4 * sizeof(int) + len(a))
```

The argument *fd* is the file descriptor to the newly created file and *a* is the file's name. We use `malloc` to allocate a heap buffer that will store the bytes of the header block. Note that the size of the header block, and thus also the heap buffer allocation size, is $4 * \texttt{sizeof}(\texttt{int}) + \mathsf{len}(a)$: 5 integers (writers,readers,data_offset,name_size) and the number of bytes comprising the name *a*. Then, we use the `memcpy` operation to set the contents of the heap buffer to be the bytes comprising the header block. Since we are creating a new named-pipe, the number of writers and readers is set to 0, and the data offset is set to -1. We write the contents of the heap buffer to the file using `write`, the specification of which we discussed in chapter 6, section 6.1.3.

Implementation of `fifo_open`

We implement the `fifo_open` operation in figure 8.23. Depending on the value of *mode*, the implementation increments the number of readers or writers with `fifo_incr_readers` and `fifo_incr_writers` respectively, before constructing and returning the named-pipe file descriptor with `fifo_mk_desc`. Note that the increment is protected by a lock file constructed from the filename of the named-pipe

```
fifo_open(path, mode) ≜
  let a = basename(path);
  let lf = /"tmp"/(a :: ".lock");
  let fd = open(path, O_RDWR);
  if ¬iserr(fd) then
    if mode = O_RDONLY then
      lock(lf);
      fifo_incr_readers(fd);
      unlock(lf);
      return fifo_mk_desc(fd, O_RDONLY)
    else if mode = O_WRONLY then
      lock(lf);
      fifo_incr_writers(fd);
      unlock(lf);
      return fifo_mk_desc(fd, O_WRONLY)
    else
      close(fd);
      return EACCESS
    fi
  else return fd fi
```

Figure 8.23.: Implementation of `fifo_open`.

file. This is because the increment is not atomic: we must first read the file, bringing the value of the counter to (heap) memory, and then write the incremented value back. The `fifo_incr_readers` and `fifo_incr_writers` operations are implemented as follows:

```
let fifo_incr_readers(fd) ≜
  let h = malloc(sizeof(int));
  pread(fd, h, sizeof(int), sizeof(int));
  let rd = memread(h, int);
  memwrite(h, i2b(rd + 1));
  pwrite(fd, h, sizeof(int), sizeof(int))

let fifo_incr_writers(fd) ≜
  let h = malloc(sizeof(int));
  pread(fd, h, sizeof(int), 0);
  let wr = memread(h, int);
  memwrite(h, i2b(wr + 1));
  pwrite(fd, h, sizeof(int), 0)
```

Both operations first allocate a heap buffer for a single integer, then read the appropriate counter stored within the header block, increment its value in memory, and write its new value back to the header block.

Note that here we use the `pwrite` and `pread` operations. These behave similarly to `write` and `read`,

except that file offset at which the operations take effect is not taken from the file descriptor, but given explicitly as an argument instead. The specifications of `pread` and `pwrite` are given in figure 8.24. The advantage of these operations is that they allow atomic random access to the file associated with the file descriptor. In contrast, `write` and `read` require the use of the `lseek` operation to change the file offset associated with the file descriptor.

We use these operations merely for convenience: we do not have to perform an extra `lseek` before each read and write. At the point where `fifo_incr_readers` and `fifo_incr_writers` are used the file descriptor to the named-pipe file is private to the thread performing the operations. Therefore, the current file offset associated with the file descriptor is not subject to interference from the environment. Thus, using `read` and `write` instead of `pread` and `pwrite`, with an additional `lseek` before each, would be valid, albeit more verbose.

$$\begin{array}{l}\texttt{pwrite}(fd, ptr, sz, off)\\ \quad \sqsubseteq \quad \texttt{pwrite\_off}(fd, ptr, sz, off)\\ \qquad \sqcap\ \texttt{write\_badf}(fd)\end{array}$$

$$\begin{array}{l}\texttt{pread}(fd, ptr, sz, off)\\ \quad \sqsubseteq \quad \texttt{pread\_norm}(fd, ptr, sz, off)\\ \qquad \sqcap\ \texttt{read\_badf}(fd)\end{array}$$

$$\textbf{let}\ \texttt{pwrite\_off}(fd, ptr, sz, off) \triangleq$$
$$\mathbb{V}FS. \left\langle \begin{array}{l}\mathsf{fs}(FS) \wedge \mathsf{isfile}(FS(\iota)) * \mathsf{fd}(fd, \iota, -, fl) \wedge \mathsf{iswrfd}(fl) * \mathsf{buf}(ptr, \overline{y}) \wedge \mathsf{len}(\overline{y}) = sz,\\ \mathsf{fs}(FS[\iota \mapsto FS(\iota)[off \leftarrow \overline{y}]]) * \mathsf{fd}(fd, \iota, -, fl) \wedge \mathsf{iswrfd}(fl) * \mathsf{buf}(ptr, \overline{y}) \wedge \mathsf{len}(\overline{y}) = sz * \texttt{ret} = sz\end{array} \right\rangle$$

$$\textbf{let}\ \texttt{pread\_norm}(fd, ptr, sz, off) \triangleq$$
$$\mathbb{V}FS. \left\langle \begin{array}{l}\mathsf{fs}(FS) \wedge \mathsf{isfile}(FS(\iota)) * \mathsf{fd}(fd, \iota, -, fl) \wedge \mathsf{isrdfd}(fl) * \mathsf{buf}(ptr, \overline{y}) \wedge \mathsf{len}(\overline{y}) = sz,\\ \exists \overline{y_t}.\, \mathsf{fs}(FS) * \mathsf{fd}(fd, \iota, -, fl) * \mathsf{buf}(ptr, \overline{y} \upharpoonleft \overline{y_t}) \wedge \overline{y_t} = FS(\iota)[off, sz] * \texttt{ret} = \mathsf{len}(\overline{y_t})\end{array} \right\rangle$$

Figure 8.24.: Specification of POSIX `pwrite` and `pread`

Finally, `fifo_mk_desc` is implemented as follows:

$$\begin{array}{l}\textbf{let}\ \texttt{fifo\_mk\_desc}(fd, mode) \triangleq\\ \quad \textbf{let}\ h = \texttt{malloc}(2 * \texttt{sizeof}(\texttt{int}));\\ \quad \texttt{memcpy}(h, \mathsf{i2b}(fd) :: \mathsf{i2b}(mode));\\ \quad \texttt{return}\ h\end{array}$$

This simply allocates a heap buffer to store the real file descriptor to the file, which has an integer value, and the mode under which the named pipe is opened. The address of this heap buffer then serves the role of a "named-pipe file descriptor", effectively hiding the real file descriptor used for I/O on the regular file that stores the named pipe.

Implementation of `fifo_read`

We implement `fifo_read` in figure 8.25. The *ffd* argument is a pointer to the heap buffer that has

```
fifo_read(ffd, ptr, sz) ≜
  let mode = fifo_get_mode(ffd);
  let fd = fifo_get_fd(ffd);
  if mode = O_RDONLY then
    let name = fifo_get_name(fd);
    let lockfile = /"tmp"/(name + ".lock");
    lock(lockfile);
    let writers = fifo_get_writers(fd);
    let empty = fifo_is_empty(fd);
    if writers = 0 ∧ empty then
      unlock(lockfile);
      return 0
    else if empty then
      unlock(lockfile);
      return fifo_read(fd, ptr, sz)
    else
      let sz' = fifo_read_contents(fd, ptr, sz);
      unlock(lockfile);
      return sz'
    fi
  else return EACCESS fi
```

Figure 8.25.: Implementation of `fifo_read`.

been previously created by `fifo_mk_desc`. From this buffer we read the mode with which the named pipe was opened for I/O, with `fifo_get_mode`, and the file descriptor opened for the named pipe file, with `fifo_get_fd`. If the named pipe has not been opened for reading, then we directly return `EACCESS`. Otherwise, we read the name of the named pipe file with `fifo_get_name` and construct the path of the lock file that we use in the implementation. `fifo_get_name` reads the name from the header block. Therefore, even if the name of the named pipe file is changed by the environment, or even if it is unlinked, we still use the same lock file path throughout the lifetime of the named pipe. We then proceed to lock the lock file, thus preventing other named-pipe operations from modifying its internal structure. Next, we read the number of writers and whether the named-pipe is empty from the header block with `fifo_get_writers` and `fifo_is_empty`. If there are no writers and the named pipe is empty, we unlock the lock and return 0 to indicate that there are no data to read. If there are writers but the named pipe is empty, we unlock and retry. Otherwise, either there are writers or the named pipe is not empty and we read the contents named pipe into the heap buffer *ptr* with `fifo_read_contents`, unlock and return the number of bytes actually read.

The implementations of `fifo_get_mode`, `fifo_get_fd` and `fifo_get_name` are as follows:

```
let fifo_get_mode(ffd) ≜
  return memread(ffd + sizeof(int), int)
```

```
let fifo_get_fd(ffd) ≜
  return memread(ffd, int)
```

```
let fifo_get_name(fd) ≜
  let hsz = malloc(sizeof(int));
  pread(fd, hsz, sizeof(int), 4 * sizeof(int));
  let size = memread(hsz, int);
  let hn = malloc(size);
  pread(fd, hn, size, 5 * sizeof(int));
  return memread(hn, STR(size))
```

`fifo_get_mode` and `fifo_get_fd` simply read the from the heap buffer *ffd* the integer corresponding to the I/O mode and file descriptor accordingly. In `fifo_get_name` we first read the size of the name stored in the header block, allocate a heap buffer of that size, read the name stored in header block to the buffer, and finally return its contents.

The implementations of `fifo_get_writers`, `fifo_get_readers` and `fifo_is_empty` are as follows:

```
let fifo_get_writers(fd) ≜
  let h = malloc(sizeof(int));
  pread(fd, h, sizeof(int), 0);
  return memread(h, int)
```

```
let fifo_get_readers(fd) ≜
  let h = malloc(sizeof(int));
  pread(fd, h, sizeof(int), sizeof(int));
  return memread(h, int)
```

```
let fifo_is_empty(fd) ≜
  let h = malloc(sizeof(int));
  pread(fd, h, sizeof(int), 2 * sizeof(int));
  let next = memread(h, int);
  return next = −1
```

The operations simply allocate a buffer for storing an integer, read the appropriate field from the header block and return the read contents. In the case of `fifo_is_empty` if the data offset is -1, we return true, otherwise we return false.

The implementation of `fifo_read_contents` is given as follows:

```
let fifo_read_contents(fd, ptr, sz) ≜
  let hoff = malloc(sizeof(int));
  pread(fd, hoff, sizeof(int), 2 * sizeof(int));
  let off = memread(hoff, int);
  let sz′ = pread(fd, off, ptr, sz);
  if sz′ < sz then memwrite(hoff, i2b(−1))
  else memwrite(hoff, i2b(off + sz)) fi
  pwrite(fd, hoff, sizeof(int), 2 * sizeof(int));
  return sz′
```

The arguments *ptr* and *sz* are the same arguments given to `fifo_read`: *ptr* is a pointer to the heap

buffer into which we will read the contents of the named pipe, and *sz* is the size of the buffer. We first allocate a buffer for a single integer into which we read the data offset from the header block using `pread`. Then, we proceed to read *sz* number of bytes from the file starting at that the data offset. We now need to update the header block to a new offset. The named pipe may store less bytes than the number *sz* requested. If this is the case, we have read everything stored in the named pipe, and thus we write -1 to the header field storing the data offset. Otherwise, we increment the data offset stored in the header by the number of bytes we have read. Finally, we return the number of bytes we have actually read from the named pipe.

Implementation of `fifo_write`

We implement `fifo_write` in figure 8.26. The implementation follows a similar structure to `fifo_read`.

```
fifo_write(ffd, ptr, sz) ≜
  let fd = fifo_get_fd(ffd);
  let mode = fifo_get_mode(ffd);
  if mode = O_WRONLY then
    let name = fifo_get_name(fd);
    let lockfile = /"tmp"/(name + ".lock");
    let readers = fifo_get_readers(fd);
    if readers = 0 then return EPIPE
    else
      lock(lockfile);
      let sz' = fifo_write_contents(fd, ptr, sz);
      unlock(lockfile);
      return sz'
    fi
  else return EACCESS fi
```

Figure 8.26.: Implementation of `fifo_write`.

An attempt to write to the named pipe is only performed if the named pipe has been opened in `O_WRONLY` mode. In that case, the lock file associated with the named pipe is locked before any modification occurs. The actual write is performed by the auxiliary operation `fifo_write_contents` after which the lock is unlocked. If there are no readers, the `EPIPE` error is returned.

```
let fifo_write_contents(fd, ptr, sz) ≜
  let empty = fifo_is_empty(fd);
  let off = lseek_end(fd, 0, SEEK_END});
  pwrite(fd, h, sz, off);
  if empty then
    malloc(hoff, sizeof(int));
    memwrite(hoff, i2b(off));
    pwrite(fd, hoff, sizeof(int), 2 * sizeof(int));
  fi
  return sz
```

$\mathsf{hdr}(\overline{y}, a, wr, rd, fbo) \triangleq \overline{y} = \mathsf{i2b}(wr) :: \mathsf{i2b}(rd) :: \mathsf{i2b}(fbo) :: \mathsf{i2b}(\mathsf{len}(a)) :: a$

$\mathsf{fhr}(\overline{y}, a, rd) \triangleq \mathsf{hdr}(\overline{y}, a, -, rd, -)$

$\mathsf{fhw}(\overline{y}, a, wr) \triangleq \mathsf{hdr}(\overline{y}, a, wr, -, -)$

$\mathsf{fhn}(\overline{y}, a) \triangleq \mathsf{hdr}(\overline{y}, a, -, -, -)$

$\mathsf{fhns}(\overline{y}, a, sz) \triangleq \mathsf{hdr}(\overline{y}, a, -, -, -) \wedge sz = \mathsf{len}(a)$

$\mathsf{fhfbo}(\overline{y}, a, fbo) \triangleq \mathsf{hdr}(\overline{y}, a, -, -, fbo)$

$\mathsf{empfifo}(\overline{y}, a) \triangleq \exists \overline{y}', \overline{y}''. \overline{y} = \overline{y}' :: \overline{y}'' \wedge \mathsf{hdr}(\overline{y}', a, 0, 0, -1)$

$\mathsf{newfifo}(\overline{y}, a) \triangleq \mathsf{hdr}(\overline{y}, a, 0, 0, -1, -1)$

$\mathsf{fimp}(\overline{y}, a, wr, rd, \overline{y_s}) \triangleq \exists fbo, \overline{y_h}, \overline{y_d}. \overline{y} = \overline{y_h} :: \overline{y_d} :: \overline{y_s} \wedge \mathsf{hdr}(\overline{y_h}, a, wr, rd, fbo) \wedge \mathsf{len}(\overline{y_h} :: \overline{y_d}) = fbo$

$\mathsf{fbs}(\overline{y}, a, \overline{y_s}) \triangleq \exists wr, rd. \mathsf{fimp}(\overline{y}, a, wr, rd, \overline{y_s})$

$\mathsf{ffby}(\overline{y}, a, fbo, \overline{y_s}) \triangleq \exists \overline{y_h}, \overline{y_d}. \overline{y} = \overline{y_h} :: \overline{y_d} :: \overline{y_s} \wedge \mathsf{hdr}(\overline{y_h}, a, -, -, fbo) \wedge \mathsf{len}(\overline{y_h} :: \overline{y_d}) = fbo$

$\mathsf{isfifo}(\iota, a, FS) \triangleq \mathsf{fimp}(FS(\iota), a, -, -, -)$

Figure 8.27.: Naped-pipe header and data predicates.

In the `fifo_write_contents` operations we first read if the named pipe is empty from the header block and then we use `lseek` to get the offset of end of the file. Then we write the contents of the user supplied buffer *ptr* at that offset, thus extending the file. If the named pipe was previously empty, we write the offset as the new data offset to the header block. Finally, we return the number of bytes written.

8.3.3. Verification

The correctness of our named-pipe implementation is far from obvious. We now proceed to verify that our implementation of the named-pipe module meets its specification.

In figure 8.27 we define predicates that describe the structure of the file contents. The predicate $\mathsf{hdr}(\overline{y}, a, wr, rd, fbo)$ describes the header block stored in the byte sequence $\overline{y}$, with the named-pipe filename a, number of writers wr, number of readers rd and the first block offset fbo. The predicate $\mathsf{fhr}(\overline{y}, a, rd)$ states that the number of readers in the named pipe header $\overline{y}$ with name a is rd. The predicate $\mathsf{fhw}(\overline{y}, a, wr)$ states that the number of writers in the named pipe header $\overline{y}$ with name a is rd. The predicate $\mathsf{fhn}(\overline{y}, a)$ states that the name of the named pipe in the header $\overline{y}$ is a. The predicate $\mathsf{fhns}(\overline{y}, a, sz)$ states the name of the named pipe a in the header $\overline{y}$ is of length sz. The predicate $\mathsf{fhfbo}(\overline{y}, a, fbo)$ states that the data offset in the header $\overline{y}$ of the named pipe with name a is fbo. The predicate $\mathsf{empfifo}(\overline{y}, a)$ states that the byte sequence $\overline{y}$ stores the header of an empty and

unopened named pipe with name a. The predicate $\mathsf{fimp}(\overline{y}, a, wr, rd, \overline{y_s})$ states that the byte sequence $\overline{y}$ stores a named pipe with the name a, number of writers wr, number of readers rd and contents $\overline{y_s}$. The predicate $\mathsf{fbs}(\overline{y}, a, \overline{y_s})$ states that the byte sequence $\overline{y}$ stores a named pipe named a with contents $\overline{y_s}$. The predicate $\mathsf{ffby}(\overline{y}, a, fbo, \overline{y_s})$ states that the byte sequence $\overline{y}$ stores a named pipe named a with contents $\overline{y_s}$ that start at data offset fbo. Finally, the predicate $\mathsf{isfifo}(\iota, a, FS)$ states that within the file-system graph FS, the file with inode ι stores the named-pipe structure of our implementation, associated with the name a.

In order to account for the fact that the named pipe is shared according to a specific protocol, we introduce the shared region type **Fifo**. Regions of this type are parameterised by the inode and name of the named-pipe file. Its abstract states are triples, $(wr, rd, \overline{y})$, consisting of the number of writers wr, number of readers rd, and byte sequence stored in the named pipe $\overline{y}$. We associate this region type with four guards: W, R, WR and RD. Additionally we will make use of the empty guard $\mathbf{0}$. The labelled state transition system for this region type is defined as follows:

$$\begin{aligned}
\mathbf{0} &: \forall wr, rd, \overline{y}.\, (wr, rd, \overline{y}) \rightsquigarrow (wr + 1, rd, \overline{y}) \\
\mathbf{0} &: \forall wr, rd, \overline{y}.\, (wr, rd, \overline{y}) \rightsquigarrow (wr, rd + 1, \overline{y}) \\
\mathsf{W} &: \forall wr, rd, \overline{y}, \overline{y}'.\, (wr, rd, \overline{y}) \rightsquigarrow (wr, rd, \overline{y} :: \overline{y}') \\
\mathsf{W} &: \forall wr, rd, \overline{y}.\, (wr, rd, \overline{y}) \rightsquigarrow (wr - 1, rd, \overline{y}) \\
\mathsf{R} &: \forall wr, rd, \overline{y}, \overline{y}'.\, (wr, rd, \overline{y} :: \overline{y}') \rightsquigarrow (wr, rd, \overline{y}') \\
\mathsf{R} &: \forall wr, rd, \overline{y}.\, (wr, rd, \overline{y}) \rightsquigarrow (wr, rd - 1, \overline{y})
\end{aligned}$$

The empty guard $\mathbf{0}$ allows opening the named pipe for writing and reading, incrementing the number of writers and readers respectively. The W guard allows writing to the named pipe, as well as closing it by decrementing the number of writers. Similarly, the R guard allows reading from the named pipe and closing it. Finally, the guards WR and RD are used to count the number of W and R instances respectively, with the following equivalence:

$$\mathsf{WR}\,(n) = \mathsf{WR}\,(n+1) \bullet \mathsf{W} \qquad\qquad \mathsf{RD}\,(n) = \mathsf{RD}\,(n+1) \bullet \mathsf{R}$$

We define the following interpretation for the region:

$$I_r(\mathbf{Fifo}_\alpha(\iota, a, wr, rd, \overline{y})) \triangleq \exists FS.\, \mathbf{GFS}(FS) \wedge \mathsf{fimp}(FS(\iota), wr, rd, \overline{y})$$

The interpretation requires the file system to contain a regular file with inode ι, the contents of which store the named pipe according to the fimp predicate.

We are using the lock-file module introduced in chapter 6 in the implementation of operations that manipulate a named pipe, and in particular the CAP-style specification of the module we derived in section 8.1. This means that we need the context invariant LFCtx to hold. In addition, we require a further restriction on the context to ensure that the contents of the named-pipe file may only be modified through the operations of the named-pipe module. Therefore, we require a new context

invariant, FifoCtx, that combines LFCtx with the additional restrictions over the named-pipe file.

$$\begin{aligned}\mathsf{FifoCtx}(\iota, a) \triangleq\ & \exists FS.\, \mathbf{GFS}(FS) \wedge \mathsf{LFCtx}(/\text{“tmp”}/a) \\ & \wedge\, !\mathsf{F}\,(\iota) \in \mathcal{G}_{\mathbf{GFS}} \wedge \forall FS, FS'.\, \mathsf{isfifo}(\iota, a, FS) \wedge \mathsf{isfifo}(\iota, a, FS') \wedge (FS, FS') \in \dagger_{\mathbf{GFS}}\mathsf{F}\,(\iota) \\ & \wedge\, \forall FS, FS'.\, FS(\iota) \wedge \mathsf{empfifo}(FS'(\iota), a) \wedge (FS, FS') \in \dagger_{\mathbf{GFS}}\mathbf{0}\end{aligned}$$

$\mathsf{FifoCtx}(\iota)$ requires the indivisible guard $\mathsf{F}\,(\iota)$ to be defined for global file-system region, and that atomic updates to the named-pipe file are only allowed by this guard. Furthermore, only the zero guard grants the capability to create a new named-pipe file. We are using the guard F of the global file system region **GFS** as the resource invariant protected by the lock-file module. Thus, when the lock is locked, we get ownership of F, thus gaining the capability to update the named-pipe file. When the lock is unlocked that capability is returned. This ensures that under the context invariant only one thread at a time can update the named pipe.

Assuming the context restriction FifoCtx holds, we implement the abstract named-pipe predicates as follows:

$$\mathbb{T}_3 \triangleq \textsc{RId} \times \mathbb{T}_2$$

$$\mathsf{fifo}((\alpha, s), \iota, wr, rd, \overline{y}) \triangleq \exists a.\, \mathbf{Fifo}_{\alpha}(\iota, a, wr, rd, \overline{y}) * [\mathsf{WR}(wr)]_{\alpha} * [\mathsf{RD}(rd)]_{\alpha} * \mathsf{isLock}(s, /\text{“tmp”}/a)$$

$$\mathsf{ffd}((\alpha, s), \mathit{ffd}, \iota, \texttt{O\_WRONLY}) \triangleq \exists \mathit{fd}'.\, \mathsf{buf}\big(\mathit{ffd}, \mathsf{i2b}(\texttt{O\_WRONLY}) :: \mathsf{i2b}\big(\mathit{fd}'\big)\big) * \mathsf{fd}\big(\mathit{fd}', \iota, -, \texttt{O\_RDWR}\big) * [\mathsf{W}]_{\alpha}$$

$$\mathsf{ffd}((\alpha, s), \mathit{ffd}, \iota, \texttt{O\_RDONLY}) \triangleq \exists \mathit{fd}'.\, \mathsf{buf}\big(\mathit{ffd}, \mathsf{i2b}(\texttt{O\_RDONLY}) :: \mathsf{i2b}\big(\mathit{fd}'\big)\big) * \mathsf{fd}\big(\mathit{fd}', \iota, -, \texttt{O\_RDWR}\big) * [\mathsf{R}]_{\alpha}$$

The abstract type $\mathbb{T}_3$ is instantiated to the region identifier for the **Fifo** region type, and the abstract type $\mathbb{T}_1$ used by the lock-file specification. The abstract predicate fifo encapsulates the **Fifo** region, the capability to open the named-pipe for reading and writing with the WR and RD guards, as well as the capability to lock the lock-file used in the named-pipe implementation with the isLock predicate. The abstract predicate ffd encapsulates the heap buffer containing the mode with which a named pipe is opened with `fifo_open`, the file descriptor used by the implementation for I/O on the named-pipe file, as well as the guard granting the capability to perform a write or a read on the named pipe.

Verification of `mkfifo`

First, we derive the refinement of figure 8.28 for the implementation of `mkfifo`. This new specification program abstraction is obtained by replacing the invocation of `open` with its specification, and using the encoding of **if-then-else** and distributivity of sequential composition over angelic choice.

In figure 8.29, we show the proof that `mkfifo` satisfies the specification given in section 8.3.1. We read the proof in the direction of refinement: that is, from bottom to top. In our first refinement step we we apply HAbstractR on the assumption to open the abstract predicate fifo, and immediately apply HCons with the CreateRegion view-shift to open the **Fifo** region. We also use definition 57 to open the assumption into a Hoare specification statement. In the second refinement step we apply ACons on the atomic statement, strengthening the postcondition to introduce the file descriptor obtained by creating the named-pipe file. Additionally, in the Hoare statement we use the context

```
mkfifo(path)
  ⊑ let p = dirname(path);
    let a = basename(path);
    let r = resolve(p, ι0);
    if ¬iserr(r) then
      let fd = link_new_file(r, a, O_EXCL|O_CREAT|O_RDWR)
                ⊓ eexist(r, a)
                ⊓ enotdir(r);
      if ¬iserr(fd) then
        write_fifo_header(fd);
        close(fd);
        return 0
      else return fd fi
    else return r fi
```

Figure 8.28.: Intermediate refinement of `mkfifo`, replacing `open` with its specifcation.

invariant and ACons to introduce the global file system region into the precondition. We then refine to the sequence of the `link_new_file` and the **if-then-else** statement.

We refine to `link_new_file` by ACons and FApplyElim to the specification given in figure 6.7. For `write_fifo_header` we use the refinement given in figure 8.30. In the first refinement step (starting from the bottom) we apply AEElim to eliminate the existential quantification on file-system graphs, followed by UseAtomic using the zero guard of **GFS** and ACons to update the contents of **GFS**. We then refine the specification to the sequence of steps comprising `write_fifo_header`. In this refinement, we use the specifications in figure 6.9 for `malloc` and `memwrite`, and AWeaken2 to abstract them to non-atomic operations that we coalesce with Seq. Then, by the definition of Hoare specification statements as a special form of atomic specification statements (definition 57), we turn the heap update of `malloc` and `memwrite` to an update on the private part of general-form atomic specification statement. Treating the heap resource private in this manner, allows us to consider the heap operations as stuttering steps for the `write` to the file. Recall that AStutter coalesces stuttering steps on the public part of an atomic specification statement, but acts as Seq for the private part.

The `write_fifo_header` proof in figure 8.30 demonstrates a reasoning pattern for I/O operations. Typically, before or after an I/O operation it is necessary to manipulate the heap to setup the buffer used by a read or write. Such operations are stuttering steps for I/O since they do not modify the contents of a file. Furthermore, the heap buffer used in an I/O operation is typically allocated and owned by the thread performing the operation. In such cases, these heap resources are manipulated non-atomically in the private part of the atomic specification statement that atomically reads or writes the file contents.

Verification of `fifo_open`

Most of the proof effort for `fifo_open` is in the branching on the *mode* argument. Again, throughout the proof of `fifo_open` we assume that the context invariant FifoCtx holds.

$$
\begin{array}{l}
\textbf{let } p = \texttt{dirname}(path);\\
\textbf{let } a = \texttt{basename}(path);\\
\textbf{let } r = \texttt{resolve}(p, \iota_0);\\
\textbf{if } \neg\mathsf{iserr}(r) \textbf{ then}\\
\quad \textbf{let } fd = \texttt{link\_new\_file}(r, a, \texttt{O\_EXCL}|\texttt{O\_CREAT}|\texttt{O\_RDWR})\\
\qquad\qquad \sqsubseteq \text{ by figure 6.7, FApplyElim and ACons}\\
\qquad\qquad\quad \forall FS.\left\langle \begin{array}{l}\mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(r)),\\ a \notin FS(\iota) \Rightarrow \mathsf{fs}(FS[\iota \mapsto FS(\iota)[a \mapsto \iota']] \uplus \iota' \mapsto \epsilon) * \mathsf{fd}(fd, \iota', 0, \texttt{O\_RDWR})\end{array}\right\rangle\\
\qquad\qquad \sqcap\, \texttt{eexist}(r, a) \sqcap \texttt{enotdir}(r);\\
\quad \textbf{if } \neg\mathsf{iserr}(fd) \textbf{ then}\\
\quad\quad \mathsf{FifoCtx}(\iota', a) \vdash\\
\quad\quad\quad \texttt{write\_fifo\_header}(fd);\\
\quad\quad\quad\quad \sqsubseteq \text{ by figure 8.30}\\
\quad\quad\quad\quad\quad \left\{\begin{array}{l}\exists FS.\, \mathbf{GFS}(FS) \wedge FS(\iota') = \epsilon * \mathsf{fd}(fd, \iota', 0, \texttt{O\_RDWR}),\\ \exists FS.\, \mathbf{GFS}(FS) \wedge \mathsf{empfifo}(FS(\iota'), a) * \mathsf{fd}(fd, \iota', -, \texttt{O\_RDWR})\end{array}\right\}\\
\quad\quad\quad \texttt{close}(fd);\\
\quad\quad\quad\quad \sqsubseteq \text{ by specification, AWeaken2 and HFrame}\\
\quad\quad\quad\quad\quad \left\{\begin{array}{l}\exists FS.\, \mathbf{GFS}(FS) \wedge \mathsf{empfifo}(FS(\iota'), a) * \mathsf{fd}(\iota', -, \texttt{O\_RDWR}),\\ \exists FS.\, \mathbf{GFS}(FS) \wedge \mathsf{empfifo}(FS(\iota'), a)\end{array}\right\}\\
\quad\quad\quad \texttt{return } 0\\
\quad\quad \sqsubseteq \left\{\begin{array}{l}\exists FS.\, \mathbf{GFS}(FS) \wedge FS(\iota') = \epsilon * \mathsf{fd}(fd, \iota', 0, \texttt{O\_RDWR}),\\ \exists FS.\, \mathbf{GFS}(FS) \wedge \mathsf{empfifo}(FS(\iota'), a) * \texttt{ret} = 0\end{array}\right\}\\
\quad \textbf{else } \texttt{return } fd \textbf{ fi}\\
\quad \sqsubseteq \left(\begin{array}{l}\forall FS.\left\langle \begin{array}{l}\mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(r)),\\ a \notin FS(\iota) \Rightarrow \mathsf{fs}(FS[\iota \mapsto FS(\iota)[a \mapsto \iota']] \uplus \iota' \mapsto \epsilon) * \mathsf{fd}(fd, \iota', 0, \texttt{O\_RDWR})\end{array}\right\rangle;\\ \mathsf{FifoCtx}(\iota', a) \vdash \left\{\begin{array}{l}\exists FS.\, \mathbf{GFS}(FS),\\ fd \notin \textsc{Errs} \Rightarrow \exists FS.\, \mathbf{GFS}(FS) \wedge \mathsf{empfifo}(FS(\iota'), a) * \texttt{ret} = 0\end{array}\right\}\end{array}\right)\\
\quad\quad \sqcap\, \texttt{eexist}(r, a) \sqcap \texttt{enotdir}(r);\\
\quad\quad \sqsubseteq \text{ by ACons}\\
\quad\quad\quad \left(\begin{array}{l}\forall FS.\left\langle \begin{array}{l}\mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(r)),\\ a \notin FS(\iota) \Rightarrow \mathsf{fs}(FS[\iota \mapsto FS(\iota)[a \mapsto \iota']] \uplus \iota' \mapsto \epsilon) * \texttt{ret} = 0\end{array}\right\rangle;\\ \mathsf{FifoCtx}(\iota', a) \vdash \left\{\begin{array}{l}\mathsf{true},\\ \texttt{ret} = 0 \Rightarrow \exists FS.\, \mathbf{GFS}(FS) \wedge \mathsf{empfifo}(FS(\iota'), a)\end{array}\right\}\end{array}\right)\\
\quad\quad\quad \sqcap\, \texttt{eexist}(r, a) \sqcap \texttt{enotdir}(r);\\
\quad\quad\quad \sqsubseteq \text{ by CreateRegion, HCons, HAbstractR and definition 57}\\
\quad\quad\quad\quad \left(\begin{array}{l}\forall FS.\left\langle \begin{array}{l}\mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(r)),\\ a \notin FS(\iota) \Rightarrow \mathsf{fs}(FS[\iota \mapsto FS(\iota)[a \mapsto \iota']] \uplus \iota' \mapsto \epsilon) * \texttt{ret} = 0\end{array}\right\rangle;\\ \mathsf{FifoCtx}(\iota', a) \vdash [\texttt{ret} = 0 \Rightarrow \exists s.\, \mathsf{fifo}(s, \iota', 0, 0, \epsilon)]\end{array}\right)\\
\quad\quad\quad \sqcap\, \texttt{eexist}(r, a) \sqcap \texttt{enotdir}(r);\\
\textbf{else } \texttt{return } r \textbf{ fi}
\end{array}
$$

HCons, IfThenElse and Seq

Seq

Figure 8.29.: Proof of `mkfifo` satisfying its specification.

Let $\mathsf{fehb}(a) \triangleq \mathsf{i2b}(0) :: \mathsf{i2b}(0) :: \mathsf{i2b}(-1) :: \mathsf{i2b}(-1) :: \mathsf{len}(a) :: a$

ASTUTTER

SEQ

let $h = \texttt{malloc}(5 * \texttt{sizeof}(\texttt{int}) + \mathsf{len}(a))$;

$\sqsubseteq$ by figure 6.9, FAPPLYELIM

$\langle \mathsf{true},\ \exists \overline{y}.\ \mathsf{buf}(h, \overline{y}) \wedge \mathsf{len}(\overline{y}) = 5 * \texttt{sizeof}(\texttt{int}) + \mathsf{len}(a) \rangle$

$\sqsubseteq$ by AWEAKEN2

$\{\mathsf{true},\ \exists \overline{y}.\ \mathsf{buf}(h, \overline{y}) \wedge \mathsf{len}(\overline{y}) = 5 * \texttt{sizeof}(\texttt{int}) + \mathsf{len}(a)\}$

$\texttt{memwrite}(h, \mathsf{i2b}(0) :: \mathsf{i2b}(0) :: \mathsf{i2b}(-1) :: \mathsf{i2b}(-1) :: \mathsf{len}(a) :: a)$;

$\sqsubseteq$ by figure 6.9, FAPPLYELIM and ACONS

$\langle \mathsf{buf}(h, \overline{y}) \wedge \mathsf{len}(\overline{y}) = 5 * \texttt{sizeof}(\texttt{int}) + \mathsf{len}(a)\ ,\ \mathsf{buf}(h, \mathsf{fehb}(a)) \rangle$

$\sqsubseteq$ by AWEAKEN2, EELIMHOARE and HCONS

$\{\exists \overline{y}.\ \mathsf{buf}(h, \overline{y}) \wedge \mathsf{len}(\overline{y}) = 5 * \texttt{sizeof}(\texttt{int}) + \mathsf{len}(a)\ ,\ \mathsf{buf}(h, \mathsf{fehb}(a))\}$

$\sqsubseteq \{\mathsf{true},\ \mathsf{buf}(h, \mathsf{fehb}(a))\}$

$\equiv$ by definition 57

$\langle \mathsf{true} \mid \mathsf{true},\ \mathsf{buf}(h, \mathsf{fehb}(a)) \mid \mathsf{true} \rangle$

$\sqsubseteq$ by ACONS, SUBST1 and AFRAME

$$\forall FS.\ \left\langle \begin{array}{l|l} \mathsf{true} & \mathsf{fs}(FS) \wedge \mathsf{isfile}(FS(\iota)) * \mathsf{fd}(fd, \iota, 0, \texttt{O\_RDWR})\ , \\ \mathsf{buf}(h, \mathsf{fehb}(a)) & \mathsf{fs}(FS) \wedge \mathsf{isfile}(FS(\iota)) * \mathsf{fd}(fd, \iota, 0, \texttt{O\_RDWR}) \end{array} \right\rangle$$

$\texttt{write}(fd, h, 4 * \texttt{sizeof}(\texttt{int}) + \mathsf{len}(a))$

$\sqsubseteq$ by section 6.1.3, FAPPLYELIM, ACONS and DCHOICEINTRO

$$\forall FS.\ \left\langle \begin{array}{l} \mathsf{fs}(FS) \wedge \mathsf{isfile}(FS(\iota)) * \mathsf{fd}(fd, \iota, 0, \texttt{O\_RDWR}) * \mathsf{buf}(h, \mathsf{fehb}(a))\ , \\ \mathsf{fs}(FS[\iota \mapsto FS(\iota)[0 \leftarrow \mathsf{fehb}(a)]]) * \mathsf{fd}(fd, \iota, 0, \texttt{O\_RDWR}) * \mathsf{buf}(h, \mathsf{fehb}(a)) \end{array} \right\rangle$$

$\sqsubseteq$ by ACONS

$$\forall FS.\ \left\langle \begin{array}{l} \mathsf{fs}(FS) \wedge \mathsf{isfile}(FS(\iota)) * \mathsf{fd}(fd, \iota, 0, \texttt{O\_RDWR}) * \mathsf{buf}(h, \mathsf{fehb}(a))\ , \\ \exists \overline{y}.\ \mathsf{fs}(FS[\iota \mapsto \overline{y}]) \wedge \mathsf{empfifo}(\overline{y}, a) * \mathsf{fd}(fd, \iota, 0, \texttt{O\_RDWR}) * \mathsf{buf}(h, \mathsf{fehb}(a)) \end{array} \right\rangle$$

$\sqsubseteq$ by AWEAKEN1

$$\forall FS.\ \left\langle \begin{array}{l|l} \mathsf{buf}(h, \mathsf{fehb}(a)) & \mathsf{fs}(FS) \wedge \mathsf{isfile}(FS(\iota)) * \mathsf{fd}(fd, \iota, 0, \texttt{O\_RDWR})\ , \\ \mathsf{buf}(h, \mathsf{fehb}(a)) & \exists \overline{y}.\ \mathsf{fs}(FS[\iota \mapsto \overline{y}]) \wedge \mathsf{empfifo}(\overline{y}, a) * \mathsf{fd}(fd, \iota, 0, \texttt{O\_RDWR}) \end{array} \right\rangle$$

$$\sqsubseteq \forall FS.\ \left\langle \begin{array}{l|l} \mathsf{true} & \mathsf{fs}(FS) \wedge \mathsf{isfile}(FS(\iota)) * \mathsf{fd}(fd, \iota, 0, \texttt{O\_RDWR})\ , \\ \mathsf{true} & \exists \overline{y}.\ \mathsf{fs}(FS[\iota \mapsto \overline{y}]) \wedge \mathsf{empfifo}(\overline{y}, a) * \mathsf{fd}(fd, \iota, 0, \texttt{O\_RDWR}) \end{array} \right\rangle$$

$\sqsubseteq$ by USEATOMIC, ACONS and AEELIM

$$\left\langle \begin{array}{l} \exists FS.\ \mathbf{GFS}(FS) \wedge \mathsf{isfile}(FS(\iota)) * \mathsf{fd}(fd, \iota, 0, \texttt{O\_RDWR})\ , \\ \exists FS.\ \mathbf{GFS}(FS) \wedge \mathsf{empfifo}(FS(\iota), a) * \mathsf{fd}(fd, \iota, 0, \texttt{O\_RDWR}) \end{array} \right\rangle$$

Figure 8.30.: Refinement to `write_fifo_header`.

$\texttt{fifo\_open}(path, mode) \sqsubseteq$

let $p = \texttt{dirname}(path)$; **let** $a = \texttt{basename}(path)$; **let** $lf = /\text{“tmp”}/(a :: \text{“.lock”})$;
let $r = \texttt{resolve}(p, \iota_0)$;
if $\neg\mathsf{iserr}(r)$ **then**

$\exists fd.\, \texttt{open\_file}(r, a, \texttt{O\_RDWR}, fd)$

$\sqsubseteq$ by figure 6.7, ACons, AWeaken1, OpenRegion, Abstract and DChoiceElim

$\forall wr, rd, \overline{y}.\, \langle \mathsf{true} \mid \mathsf{fifo}(s, r, wr, rd, \overline{y}) \,,\ \mathsf{fd}(fd, r, -, \texttt{O\_RDWR}) \mid \mathsf{fifo}(s, r, wr, rd, \overline{y}) \rangle$
$\sqcap\, \exists \iota.\, \forall FS.\, \langle \mathsf{fs}(FS) \wedge \mathsf{isdir}(r) \,,\ a \in FS(r) \Rightarrow \mathsf{fs}(FS) * \iota = FS(r)(a) \rangle$
$\sqcap\, \texttt{enoent}(r, a, fd)$
$\sqcap\, \texttt{enotdir}(r)$;
if $\neg\mathsf{iserr}(fd)$ **then**
if $mode = \texttt{O\_RDONLY}$ **then**

figure 8.32:
$\texttt{lock}(lf)$;
$\texttt{fifo\_incr\_readers}(fd)$;
$\texttt{unlock}(lf)$;
return $\texttt{fifo\_mk\_desc}(fd, \texttt{O\_RDONLY})$

$$\forall wr, rd, \overline{y}.\, \left\langle \begin{array}{l|l} \mathsf{fd}(fd, r, -, \texttt{O\_RDWR}) & \mathsf{fifo}(s, r, wr, rd, \overline{y}) \,, \\ \mathsf{true} & \mathsf{ffd}(s, \texttt{ret}, r, \texttt{O\_RDONLY}) * \mathsf{fifo}(s, r, wr, rd + 1, \overline{y}) \end{array} \right\rangle$$

else if $mode = \texttt{O\_WRONLY}$ **then**

sim. to 8.32:
$\texttt{lock}(lf)$;
$\texttt{fifo\_incr\_writers}(fd)$;
$\texttt{unlock}(lf)$;
return $\texttt{fifo\_mk\_desc}(fd, \texttt{O\_WRONLY})$

$$\forall wr, rd, \overline{y}.\, \left\langle \begin{array}{l|l} \mathsf{fd}(fd, r, -, \texttt{O\_RDWR}) & \mathsf{fifo}(s, r, wr, rd, \overline{y}) \,, \\ \mathsf{true} & \mathsf{ffd}(s, \texttt{ret}, r, \texttt{O\_RDONLY}) * \mathsf{fifo}(s, r, wr + 1, rd, \overline{y}) \end{array} \right\rangle$$

else
$\texttt{close}(fd)$; **return** EACCESS

$\sqsubseteq$ by specification, AFrame, AWeaken1

$$\forall wr, rd, \overline{y}.\, \left\langle \begin{array}{l|l} \mathsf{fd}(fd, r, -, \texttt{O\_RDWR}) & \mathsf{fifo}(s, r, wr, rd, \overline{y}) \,, \\ \mathsf{true} & \mathsf{fifo}(s, r, wr, rd, \overline{y}) * \texttt{ret} = \texttt{EACCESS} \end{array} \right\rangle$$

fi
else return fd **fi**

$\sqsubseteq \exists r'.\, \texttt{resolve}(a, r, r')$;
if $\neg\mathsf{iserr}(r)$ **then**
if $mode = \texttt{O\_RDONLY}$ **then return** $\texttt{fifo\_open\_read}(r')$
else if $mode = \texttt{O\_WRONLY}$ **then return** $\texttt{fifo\_open\_write}(r')$
else return EACCESS **fi**
else return r' **fi**
else return r **fi**

resolve, **if-then-else** encoding, AChoiceDstL and AStutter

$\sqsubseteq$ **let** $r = \texttt{resolve}(path, \iota_0)$;
if $\neg\mathsf{iserr}(r)$ **then**
if $mode = \texttt{O\_RDONLY}$ **then return** $\texttt{fifo\_open\_read}(r)$
else if $mode = \texttt{O\_WRONLY}$ **then return** $\texttt{fifo\_open\_write}(r)$
else return EACCESS **fi**
else return r **fi**

Figure 8.31.: Proof of `fifo_open` satisfying its specification.

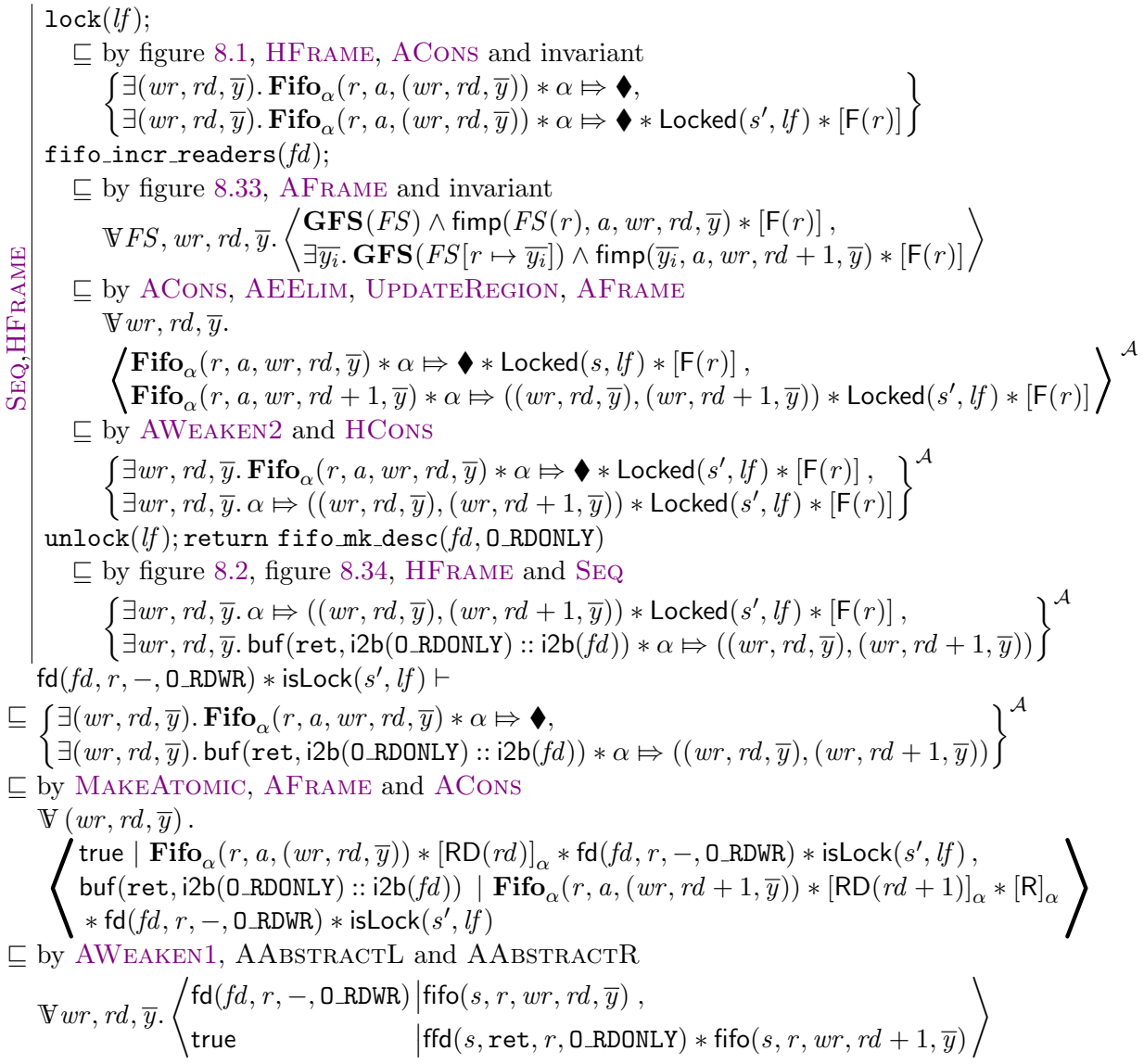

Figure 8.32.: Proof sketch of the `O_RDONLY` branch of `fifo_open`.

In figure 8.31 we show that our implementation of `fifo_open` refines its specification. Similarly to the proof of `mkfifo` our first step is to replace the invocation of `open` with its specification, as pertaining to the `O_RDWR` argument. Note that we derive a demonic specification for the invocation of `open_file` with two cases. This first specifies `open_file` to allocate the file descriptor for the named-pipe file. The second specifies `open_file` to perform a lookup of the link named a – the name of the link to the named-pipe file – with the directory resolved by the path prefix p. We use the first specification as a stuttering step for each branch on *mode*. Both branches are refined similarly to their implementation. The only difference is that the number of writers is incremented instated of the number of readers. In the subsequent discussion we will focus only on the `O_RDONLY` branch.

In figure 8.32 we abstract the implementation of the `O_RDONLY` branch into a single atomic specification statement, which allocates a new named-pipe file descriptor and increments the number of readers for the named pipe that is being opened for I/O. Note that the atomic specification statement

we derive exhibits ownership transfer from the private precondition to the public postcondition. The private part of the precondition requires ownership of the file descriptor for the named-file that we previously obtain by `open`. Ownership of this file descriptor is transferred to the implementation of the abstract named-pipe file descriptor predicate ffd in the public part of the postcondition. During the proof in figure 8.32 we are relying on the CAP-style lock-file module specification derived in section 8.1. We take the F guard of the global file system region to be the resource invariant that the lock-file specification protects. Therefore, when the lock is locked, we obtain the exclusive capability to update the named-pipe file.

In the first refinement step (at the bottom) in figure 8.32 we apply AAbstractL and AAbstractR to open the ffd and fifo abstract predicates and then use AWeaken1 to move the file descriptor from the private part of the precondition to the public part. In the next step we use ACons at the postcondition to coalesce the R with RD, which we then frame-off using AFrame. We then apply MakeAtomic. Note that the file descriptor and the isLock predicate are placed in the invariant of the refined Hoare specification statement: every step of the implementation maintains this invariant. In the next refinement step we use Seq to refine the Hoare specification statement to the sequence of operations comprising the `O_RDONLY`branch of `fifo_open`. Note that since every operation maintains the invariant of the Hoare specification statement, we do not restate the invariant in the refinement to each individual operation to save space. We use this invariant whenever necessary in subsequent refinements. For example, in the refinement to `lock` we apply ACons such that we can use the isLock predicate from the invariant. The main steps in the refinement to `fifo_incr_readers` involve using AWeaken2 to refine the Hoare statement to an atomic statement and then using UpdateRegion to justify the update to the **Fifo** by an update to its interpretation that uses the global file-system region **GFS**. From there we use the refinement shown in figure 8.33. Subsequently, in the refinement to `unlock` we relinquish ownership of the F guard and for `fifo_mk_desc` we use the refinement established in figure 8.34.

The refinement to `fifo_incr_readers` in figure 8.33 is a straightforward use of MakeAtomic. The actual step that commits the atomic increment to the number of reads in the header of the named-pipe file is the last `pwrite` which we justify by using UpdateRegion. All other steps either operate on resource that is private to the thread, such the heap buffer allocated by the `malloc` at the beginning, or does not modify the named-pipe file such as the `pread`.

The refinement to `fifo_mk_desc` shown in figure 8.34 is straightforward as the operation only operates on non-shared private resources. The refinement involves simple sequential reasoning using the refinement laws for Hoare specification statements.

AWeaken2 and Seq

$\mathbf{let}\ h = \texttt{malloc}(\texttt{sizeof}(\texttt{int}));$

$\sqsubseteq$ by figure 6.9 and FApplyElim

$\langle \mathsf{true},\ \exists \overline{y}'.\, \mathsf{buf}(h, \overline{y}') \wedge \mathsf{len}(\overline{y}') = \texttt{sizeof}(\texttt{int}) \rangle^{\mathcal{A}}$

$\sqsubseteq$ by AWeaken1 and AFrame

$\exists FS, wr, rd, \overline{y}.\, \mathbf{GFS}(FS) \wedge \mathsf{fimp}(FS(\iota), a, wr, rd, \overline{y}) * 0 \Rightarrow \blacklozenge \vdash$

$\{\mathsf{true},\ \exists \overline{y}'.\, \mathsf{buf}(h, \overline{y}') \wedge \mathsf{len}(\overline{y}') = \texttt{sizeof}(\texttt{int})\}^{\mathcal{A}}$

$\texttt{pread}(fd, h, \texttt{sizeof}(\texttt{int}), \texttt{sizeof}(\texttt{int}));$

$\sqsubseteq$ by figure 8.24, FApplyElim, Subst1, ACons and DChoiceIntro

$$\forall FS, wr, rd, \overline{y}.\ \left\langle \begin{array}{l} \exists \overline{y}'.\, \mathsf{fs}(FS) \wedge \mathsf{fimp}(FS(\iota), a, wr, rd, \overline{y}) * \mathsf{buf}(h, \overline{y}') \wedge \mathsf{len}(\overline{y}') = \texttt{sizeof}(\texttt{int}), \\ \exists \overline{y_i}.\, \mathsf{fs}(FS[\iota \mapsto \overline{y_i}]) \wedge \mathsf{fimp}(FS(\iota), a, wr, rd, \overline{y}) * \mathsf{buf}(h, \mathsf{i2b}(rd)) \end{array} \right\rangle^{\mathcal{A}}$$

$\sqsubseteq$ by OpenRegion, AEElim, AWeaken1, AFrame and invariant

$\exists FS, wr, rd, \overline{y}.\, \mathbf{GFS}(FS) \wedge \mathsf{fimp}(FS(\iota), a, wr, rd, \overline{y}) * 0 \Rightarrow \blacklozenge \vdash$

$\{\exists \overline{y}'.\, \mathsf{buf}(h, \overline{y}') \wedge \mathsf{len}(\overline{y}') = \texttt{sizeof}(\texttt{int}),\ \mathsf{buf}(h, \mathsf{i2b}(rd))\}^{\mathcal{A}}$

$\mathbf{let}\ rd = \texttt{memread}(h, \texttt{int});$

$\sqsubseteq$ by figure 6.9, FApplyElim and ACons

$\langle \mathsf{buf}(h, \mathsf{i2b}(rd)),\ \mathsf{buf}(h, \mathsf{i2b}(rd)) \rangle$

$\sqsubseteq$ by AWeaken1 and AFrame

$\exists FS, wr, rd, \overline{y}.\, \mathbf{GFS}(FS) \wedge \mathsf{fimp}(FS(\iota), a, wr, rd, \overline{y}) * 0 \Rightarrow \blacklozenge \vdash \{\mathsf{buf}(h, \mathsf{i2b}(rd)),\ \mathsf{buf}(h, \mathsf{i2b}(rd))\}^{\mathcal{A}}$

$\texttt{memwrite}(h, \mathsf{i2b}(rd + 1));$

$\sqsubseteq$ by figure 6.9, FApplyElim and ACons

$\langle \mathsf{buf}(h, \mathsf{i2b}(rd)),\ \mathsf{buf}(h, \mathsf{i2b}(rd + 1)) \rangle^{\mathcal{A}}$

$\sqsubseteq$ by AWeaken1 and AFrame

$\exists FS, wr, rd, \overline{y}.\, \mathbf{GFS}(FS) \wedge \mathsf{fimp}(FS(\iota), a, wr, rd, \overline{y}) * 0 \Rightarrow \blacklozenge \vdash$

$\{\mathsf{buf}(h, \mathsf{i2b}(rd)),\ \mathsf{buf}(h, \mathsf{i2b}(rd + 1))\}^{\mathcal{A}}$

$\texttt{pwrite}(fd, h, \texttt{sizeof}(\texttt{int}), \texttt{sizeof}(\texttt{int}))$

$\sqsubseteq$ by figure 8.24, FApplyElim, Subst1, ACons and DChoiceIntro

$$\forall FS, wr, rd, \overline{y}.\ \left\langle \begin{array}{l} \mathsf{fs}(FS) \wedge \mathsf{fimp}(FS(\iota), a, wr, rd, \overline{y}) * \mathsf{buf}(h, \mathsf{i2b}(rd + 1)), \\ \exists \overline{y_i}.\, \mathsf{fs}(FS[\iota \mapsto \overline{y_i}]) \wedge \mathsf{fimp}(\overline{y_i}, a, wr, rd + 1, \overline{y}) * \mathsf{buf}(h, \mathsf{i2b}(rd + 1)) \end{array} \right\rangle$$

$\sqsubseteq$ by UpdateRegion, AWeaken1 and ACons

$$\left\{ \begin{array}{l} \exists FS, wr, rd, \overline{y}.\, \mathbf{GFS}(FS) \wedge \mathsf{fimp}(FS(\iota), a, wr, rd, \overline{y}) * 0 \Rightarrow \blacklozenge * \mathsf{buf}(h, \mathsf{i2b}(rd + 1)), \\ \exists FS, wr, rd, \overline{y}, \overline{y_i}.\, 0 \Rightarrow (FS, FS[\iota \mapsto \overline{y_i}]) \wedge \mathsf{fimp}(FS(\iota), a, wr, rd, \overline{y}) \wedge \mathsf{fimp}(\overline{y_i}, a, wr, rd + 1, \overline{y}) \end{array} \right\}^{\mathcal{A}}$$

$\mathsf{fd}(fd, \iota, -, \texttt{O\_RDWR}) \vdash$

$$\sqsubseteq \left\{ \begin{array}{l} \exists FS, wr, rd, \overline{y}.\, \mathbf{GFS}(FS) \wedge \mathsf{fimp}(FS(\iota), a, wr, rd, \overline{y}) * 0 \Rightarrow \blacklozenge, \\ \exists FS, wr, rd, \overline{y}, \overline{y_i}.\, 0 \Rightarrow (FS, FS[\iota \mapsto \overline{y_i}]) \wedge \mathsf{fimp}(FS(\iota), a, wr, rd, \overline{y}) \wedge \mathsf{fimp}(\overline{y_i}, a, wr, rd + 1, \overline{y}) \end{array} \right\}^{\mathcal{A}}$$

$\sqsubseteq$ by MakeAtomic

$\mathsf{fd}(fd, \iota, -, \texttt{O\_RDWR}) * [\mathsf{F}(\iota)] \vdash \forall FS, wr, rd, \overline{y}.$

$\langle \mathbf{GFS}(FS) \wedge \mathsf{fimp}(FS(\iota), a, wr, rd, \overline{y})\, n,\ \exists \overline{y_i}.\, \mathbf{GFS}(FS[\iota \mapsto \overline{y_i}]) \wedge \mathsf{fimp}(FS(\iota), a, wr, rd + 1, \overline{y}) \rangle$

Figure 8.33.: Proof of `fifo_incr_readers` abstraction.

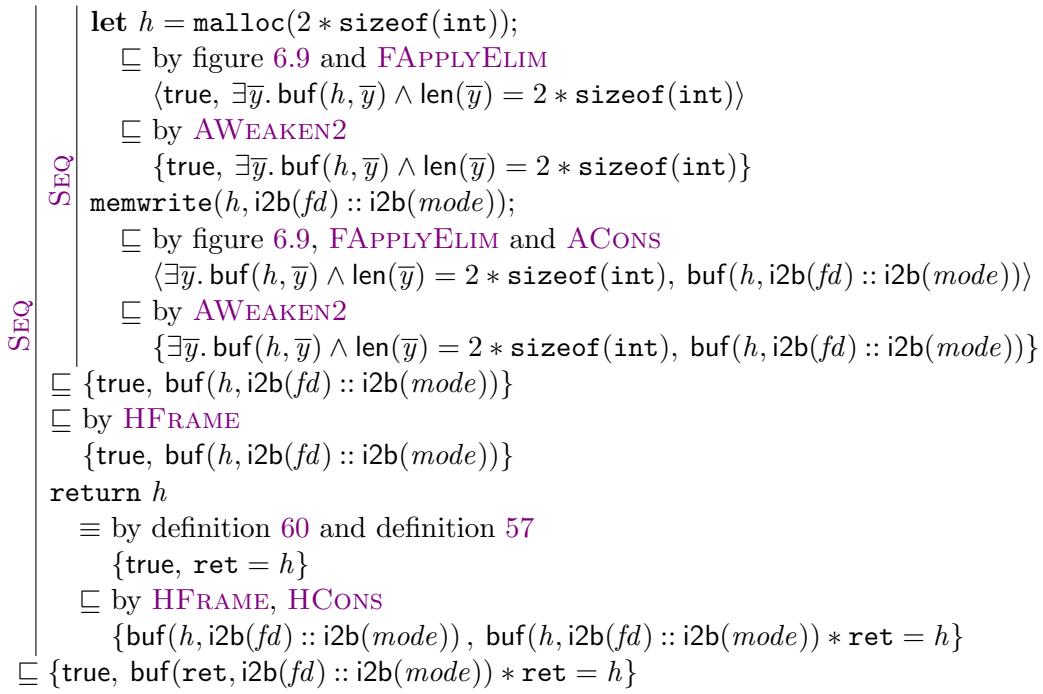

Figure 8.34.: Proof of `fifo_mk_desc` abstraction.

Verification of `fifo_read`

The specification of `fifo_read` defined in section 8.3.1 consists of five demonic cases. This makes `fifo_read` the most complex named-pipe operation we verify. Fortunately, we do not have show that `fifo_read` is a refinement of all the demonic cases simultaneously. By the DChoiceElim refinement law it is sufficient to prove each individual demonic case separately.

First, in figure 8.35 we show that `fifo_read`$(ffd, ptr, sz) \sqsubseteq$ `fifo_read_nowriters`(ffd). Note that this case applies when in the implementation the condition $writers = 0 \wedge empty$ is true. The implementation uses several lower level operations of the named-pipe implementation. For the current case we are verifying, we only need to consider `fifo_get_mode`, `fifo_get_fd`, `fifo_get_name`, `fifo_get_writers` and `fifo_is_empty`. We derive a specification for each, in terms of single atomic specification statement, in figures 8.36, 8.37, 8.38 and 8.39 respectively, which we then use during the proof in figure 8.35.

The first refinement step (at the bottom) in figure 8.35 uses AAbstractL and AAbstractR to open the abstract predicates and then AFrame to frame-off resources that are not required in this proof. Note that we have written the refined specification using an invariant. This is so that we do not have to repeat the assertions in both the precondition and postcondition. This invariant holds at every step of the implementation, but we do not restate it during the refinement of individual operations in the implementation. The next refinement step uses AStutter on the sequence of operations that comprise the implementation of `fifo_read`. Since we are considering the case of `fifo_read_nowriters` the named pipe is not modified in any step. For `fifo_get_mode` and `fifo_get_fd` we use the specifications derived in figure 8.36 to extract the file descriptor and mode for the named-pipe file respectively. Since the two steps only read the *ffd* buffer, we use AStutter and coalesce them into a single atomic specification statement. The precondition of this statement corresponds to the implementation of the ffd abstract predicate pertaining to the current case. Note that since each step operates on different section of the buffer *ffd*, we use ACons to split the buffer into two sub-buffers, buf$(ffd, \mathsf{i2b}(fd))$ used by `fifo_get_fd` and buf$(ffd +$ `sizeof`$(\texttt{int}), \mathsf{i2b}($`O_RDONLY`$))$ used by `fifo_get_mode`. Furthermore, we apply AFrame to frame off the sub-buffer and the RD guard, since they are not used.

In the next step, we use the specification of `fifo_get_name` to obtain the name of the named pipe used to construct the lock-file path. We refine to the specification given in figure 8.37 by applying OpenRegion twice: the first time to open the **Fifo** into its interpretation in terms of the global file-system region **GFS**, and the second time to open **GFS** into its interpretation in terms of the file system. Along the way we use AFrame to frame-off unused resources, and AEElim to eliminate the existentially quantified variables in the interpretation of **Fifo** to pseudo universally quantified variables.

We proceed to `lock`, where we use the CAP-style specification to obtain the guard F that allows us to modify the contents of the named-pipe file. To refine the atomic specification statement for `lock` which includes the named-pipe implementation, we first apply AFrame to frame off the **Fifo** region. Then, we use definition 57 to turn the atomic statement to a Hoare statement. Updating the lock file resources in the private part, allow us to coalesce this step with that of `unlock`, effectively eliding the updates on the lock altogether. By the context invariant, the F guard is non-duplicable and it is the only guard that allows an update of the named-pipe file contents. Therefore, for the duration the lock is locked we are guaranteed that the named-pipe file does not change.

AFrame,AWeaken1 and AStutter

let $mode = \texttt{fifo\_get\_mode}(ffd)$;
let $fd = \texttt{fifo\_get\_fd}(ffd)$;
$\sqsubseteq$ by figure 8.36, AFrame, ACons, AStutter and invariant
$\langle [\mathsf{R}]_\alpha\,,\ mode = \texttt{O\_RDONLY} * [\mathsf{R}]_\alpha \rangle$
if $mode = \texttt{O\_RDONLY}$ **then**
let $a = \texttt{fifo\_get\_name}(fd)$;
$\sqsubseteq$ by figure 8.37, ACons, AFrame, 2×OpenRegion, AEElim and invariant
$\forall wr, rd, \overline{y_s}.\ \langle \mathbf{Fifo}_\alpha(\iota, a, (wr, rd, \overline{y_s})) * [\mathsf{R}]_\alpha\,,\ \mathbf{Fifo}_\alpha(\iota, a, (wr, rd, \overline{y_s})) * [\mathsf{R}]_\alpha \rangle$
let $lf = /\text{“tmp”}/(a + \text{“.lock”})$;
$\texttt{lock}(lf)$;
$\sqsubseteq$ by figure 8.1 with $\mathsf{Inv} = [\mathsf{F}(\iota)]$, ACons and invariant
$\{\mathsf{true},\ \mathsf{Locked}(s, lf) * [\mathsf{F}(\iota)]\}$
$\sqsubseteq$ by definition 57 and AFrame
$$\forall wr, rd, \overline{y_s}.\ \left\langle \begin{array}{l|l} \mathsf{true} & \mathbf{Fifo}_\alpha(\iota, a, (wr, rd, \overline{y_s})) * [\mathsf{R}]_\alpha\,, \\ \mathsf{Locked}(s, lf) * [\mathsf{F}(\iota)] & \mathbf{Fifo}_\alpha(\iota, a, (wr, rd, \overline{y_s})) * [\mathsf{R}]_\alpha \end{array} \right\rangle$$
let $writers = \texttt{fifo\_get\_writers}(fd)$;
$\sqsubseteq$ by figure 8.38, ACons, AFrame, 2 × OpenRegion, AEElim and invariant
$$\forall wr, rd, \overline{y_s}.\ \left\langle \begin{array}{l} \mathbf{Fifo}_\alpha(\iota, a, (wr, rd, \overline{y_s})) * [\mathsf{R}]_\alpha * [\mathsf{F}(\iota)]\,, \\ \mathbf{Fifo}_\alpha(\iota, a, (wr, rd, \overline{y_s})) * [\mathsf{R}]_\alpha * [\mathsf{F}(\iota)] * writers = wr \end{array} \right\rangle$$
let $empty = \texttt{fifo\_is\_empty}(fd)$;
$\sqsubseteq$ similarly to figure 8.38, AFrame, 2 × OpenRegion, AEElim and invariant
$$\forall wr, rd, \overline{y_s}.\ \left\langle \begin{array}{l} \mathbf{Fifo}_\alpha(\iota, a, (wr, rd, \overline{y_s})) * [\mathsf{R}]_\alpha * [\mathsf{F}(\iota)]\,, \\ \mathbf{Fifo}_\alpha(\iota, a, (wr, rd, \overline{y_s})) * [\mathsf{R}]_\alpha * [\mathsf{F}(\iota)] * writers = wr * empty = (\overline{y_s} = \epsilon) \end{array} \right\rangle$$
if $writers = 0 \wedge empty$ **then**
$\texttt{unlock}(lockfile); \texttt{return}\ 0$
$\sqsubseteq$ by figure 8.2, $\mathsf{Inv} = [\mathsf{F}(\iota)]$ and Seq
$\{\mathsf{Locked}(s, lf) * [\mathsf{F}(\iota)]\,,\ \texttt{ret} = 0\}$
$\sqsubseteq$ by definition 57 and AFrame
$\langle \mathsf{Locked}(s, lf) * [\mathsf{F}(\iota)] \mid writers = 0 \wedge empty,\ \mathsf{true} \mid writers = 0 \wedge empty * \texttt{ret} = 0 \rangle$
else if $empty$ **then**
$\texttt{unlock}(lockfile); \texttt{return}\ \texttt{fifo\_read}(fd, ptr, sz)$
else
let $sz' = \texttt{fifo\_read\_contents}(fd, ptr, sz); \texttt{unlock}(lockfile); \texttt{return}\ sz'$
fi
else return EACCESS **fi**

$\sqsubseteq$ $\mathsf{isLock}(s', /\text{tmp}/(a + .\text{lock})) * \mathsf{buf}(ffd, \mathsf{i2b}(\texttt{O\_RDONLY}) :: \mathsf{i2b}(fd)) * \mathsf{fd}(fd, \iota, -, \texttt{O\_RDWR}) \vdash \forall wr, rd, \overline{y_s}.$
$\langle \mathbf{Fifo}_\alpha(\iota, a, (wr, rd, \overline{y_s})) * [\mathsf{R}]_\alpha\,,\ wr = 0 \wedge \overline{y_s} = \epsilon \Rightarrow (\mathbf{Fifo}_\alpha(\iota, a, (wr, rd, \overline{y_s})) * [\mathsf{R}]_\alpha * \texttt{ret} = 0) \rangle$
$\sqsubseteq$ by AFrame, AAbstractL, AAbstractR with $s = (\alpha, s')$
$$\forall wr, rd, \overline{y_s}.\ \left\langle \begin{array}{l} \mathsf{ffd}(s, fd, \iota, \texttt{O\_RDONLY}) * \mathsf{fifo}(s, \iota, wr, rd, \overline{y_s})\,, \\ wr = 0 \wedge \overline{y_s} = \epsilon \Rightarrow \mathsf{ffd}(s, fd, \iota, \texttt{O\_RDONLY}) * \mathsf{fifo}(s, \iota, wr, rd, \overline{y_s}) * \texttt{ret} = 0 \end{array} \right\rangle$$

Figure 8.35.: Proof of `fifo_read` in the case of `fifo_read_nowriters`.

$\texttt{return memread}(\mathit{ffd} + \texttt{sizeof(int)}, \texttt{int})$
$\sqsubseteq$ by figure 6.9, FApplyElim, DChoiceIntro and ACons
$\langle \mathsf{buf}(\mathit{ffd} + \texttt{sizeof(int)}, \mathsf{i2b}(\mathit{mode}))\,,\ \mathsf{buf}(\mathit{ffd} + \texttt{sizeof(int)}, \mathsf{i2b}(\mathit{mode})) * \texttt{ret} = \mathit{mode} \rangle$

$\texttt{return memread}(\mathit{ffd}), \texttt{int}$
$\sqsubseteq$ by figure 6.9, FApplyElim, DChoiceIntro and ACons
$\langle \mathsf{buf}(\mathit{ffd}, \mathsf{i2b}(\mathit{fd}))\,,\ \mathsf{buf}(\mathit{ffd}, \mathsf{i2b}(\mathit{fd})) * \texttt{ret} = \mathit{fd} \rangle$
$\sqsubseteq$ by AFrame and ACons
$\langle \mathsf{buf}(\mathit{ffd}, \mathsf{i2b}(\mathit{fd})) * \mathsf{fd}(\mathit{fd}, \iota, o, \texttt{O\_RDWR})\,,\ \mathsf{buf}(\mathit{ffd}, \mathsf{i2b}(\mathit{fd})) * \mathsf{fd}(\texttt{ret}, \iota, o, \texttt{O\_RDWR}) * \texttt{ret} = \mathit{fd} \rangle$

Figure 8.36.: Proof of `fifo_get_mode` and `fifo_get_fd` abstractions.

$\textbf{let}\ \mathit{hsz} = \texttt{malloc(sizeof(int))};$
$\sqsubseteq$ by figure 6.9, FApplyElim and DChoiceIntro
$\langle \mathsf{true},\ \exists \overline{y}.\, \mathsf{buf}(\mathit{hsz}, \overline{y}) \wedge \mathsf{len}(\overline{y}) = \texttt{sizeof(int)} \rangle$
$\sqsubseteq$ by AFrame and AWeaken1
$$\forall FS. \left\langle \begin{array}{ll} \mathsf{true} & \left| \exists \overline{y_h}.\, \mathsf{fs}(FS[\iota \mapsto \overline{y_h} :: -]) \wedge \mathsf{fhn}(\overline{y_h}, a)\,, \right. \\ \exists \overline{y}.\, \mathsf{buf}(\mathit{hsz}, \overline{y}) \wedge \mathsf{len}(\overline{y}) = \texttt{sizeof(int)} & \left| \exists \overline{y_h}.\, \mathsf{fs}(FS[\iota \mapsto \overline{y_h} :: -]) \wedge \mathsf{fhn}(\overline{y_h}, a) \right. \end{array} \right\rangle$$
$\texttt{pread}(\mathit{fd}, \mathit{hsz}, \texttt{sizeof(int)}, 4 * \texttt{sizeof(int)});$
$\sqsubseteq$ by definition, FApplyElim, ACons and DChoiceIntro
$$\forall FS. \left\langle \begin{array}{l} \exists \overline{y_h}.\, \mathsf{fs}(FS[\iota \mapsto \overline{y_h} :: -]) \wedge \mathsf{fhns}(\overline{y_h}, a, \mathsf{len}(a)) * \mathsf{buf}(\mathit{hsz}, \overline{y}) \wedge \mathsf{len}(\overline{y}) = \texttt{sizeof(int)}, \\ \exists \overline{y_h}.\, \mathsf{fs}(FS[\iota \mapsto \overline{y_h} :: -]) \wedge \mathsf{fhns}(\overline{y_h}, a, \mathsf{len}(a)) * \mathsf{buf}(\mathit{hsz}, \mathsf{i2b}(\mathsf{len}(a))) \end{array} \right\rangle$$
$\sqsubseteq$ by AWeaken1
$$\forall FS. \left\langle \begin{array}{ll} \mathsf{buf}(\mathit{hsz}, \overline{y}) \wedge \mathsf{len}(\overline{y}) = \texttt{sizeof(int)} & \left| \exists \overline{y_h}.\, \mathsf{fs}(FS[\iota \mapsto \overline{y_h} :: -]) \wedge \mathsf{fhns}(\overline{y_h}, a, \mathsf{len}(a))\,, \right. \\ \exists a.\, \mathsf{buf}(\mathit{hsz}, \mathsf{i2b}(\mathsf{len}(a))) & \left| \exists \overline{y_h}.\, \mathsf{fs}(FS[\iota \mapsto \overline{y_h} :: -]) \wedge \mathsf{fhns}(\overline{y_h}, a, \mathsf{len}(a)) \right. \end{array} \right\rangle$$
$\textbf{let}\ \mathit{size} = \texttt{memread}(\mathit{hsz}, \texttt{int});$
$\sqsubseteq$ by figure 6.9, FApplyElim, DChoiceIntro and ACons
$\langle \mathsf{buf}(\mathit{hsz}, \mathsf{i2b}(\mathsf{len}(a)))\,,\ \mathsf{buf}(\mathit{hsz}, \mathsf{i2b}(\mathsf{len}(a))) * \mathit{size} = \mathsf{len}(a) \rangle$
$\sqsubseteq$ by AFrame and AWeaken1
$$\forall FS. \left\langle \begin{array}{ll} \mathsf{buf}(\mathit{hsz}, \mathsf{i2b}(\mathsf{len}(a))) & \left| \exists \overline{y_h}.\, \mathsf{fs}(FS[\iota \mapsto \overline{y_h} :: -]) \wedge \mathsf{fhns}(\overline{y_h}, a, \mathsf{len}(a))\,, \right. \\ \mathsf{true} & \left| \exists \overline{y_h}.\, \mathsf{fs}(FS[\iota \mapsto \overline{y_h} :: -]) \wedge \mathsf{fhns}(\overline{y_h}, a, \mathsf{len}(a)) * \mathit{size} = \mathsf{len}(a) \right. \end{array} \right\rangle$$
$\textbf{let}\ \mathit{hn} = \texttt{malloc}(\mathit{size});$
$\sqsubseteq$ by figure 6.9, FApplyElim, AFrame and AWeaken1
$$\forall FS. \left\langle \begin{array}{ll} \mathsf{true} & \left| \exists \overline{y_h}.\, \mathsf{fs}(FS[\iota \mapsto \overline{y_h} :: -]) \wedge \mathsf{fhn}(\overline{y_h}, a) * \mathit{size} = \mathsf{len}(a)\,, \right. \\ \exists \overline{y}.\, \mathsf{buf}(\mathit{hsz}, \overline{y}) \wedge \mathsf{len}(\overline{y}) = \mathit{size} & \left| \exists \overline{y_h}.\, \mathsf{fs}(FS[\iota \mapsto \overline{y_h} :: -]) \wedge \mathsf{fhn}(\overline{y_h}, a) * \mathit{size} = \mathsf{len}(a) \right. \end{array} \right\rangle$$
$\texttt{pread}(\mathit{fd}, \mathit{hn}, \mathit{size}, 4 * \texttt{sizeof(int)});$
$\sqsubseteq$ by definition, FApplyElim, ACons, DChoiceIntro and AWeaken1
$$\forall FS. \left\langle \begin{array}{ll} \exists \overline{y}.\, \mathsf{buf}(\mathit{hn}, \overline{y}) \wedge \mathsf{len}(\overline{y}) = \mathit{size} & \left| \exists \overline{y_h}.\, \mathsf{fs}(FS[\iota \mapsto \overline{y_h} :: -]) \wedge \mathsf{fhn}(\overline{y_h}, a) * \mathit{size} = \mathsf{len}(a)\,, \right. \\ \exists a.\, \mathsf{buf}(\mathit{hn}, a) & \left| \exists \overline{y_h}.\, \mathsf{fs}(FS[\iota \mapsto \overline{y_h} :: -]) \wedge \mathsf{fhn}(\overline{y_h}, a) * \mathit{size} = \mathsf{len}(a) \right. \end{array} \right\rangle$$
$\texttt{return memread}(\mathit{hn}, \texttt{STR}(\mathit{size}))$
AStutter
$\sqsubseteq$ $\mathsf{fd}(\mathit{fd}, \iota, -, \texttt{O\_RDWR}) \vdash \forall FS.$
$\langle \exists \overline{y_h}.\, \mathsf{fs}(FS[\iota \mapsto \overline{y_h} :: -]) \wedge \mathsf{fhn}(\overline{y_h}, a)\,,\ \exists \overline{y_h}.\, \mathsf{fs}(FS[\iota \mapsto \overline{y_h} :: -]) \wedge \mathsf{fhn}(\overline{y_h}, a) * \texttt{ret} = a \rangle$

Figure 8.37.: Proof of `fifo_get_name` abstraction.

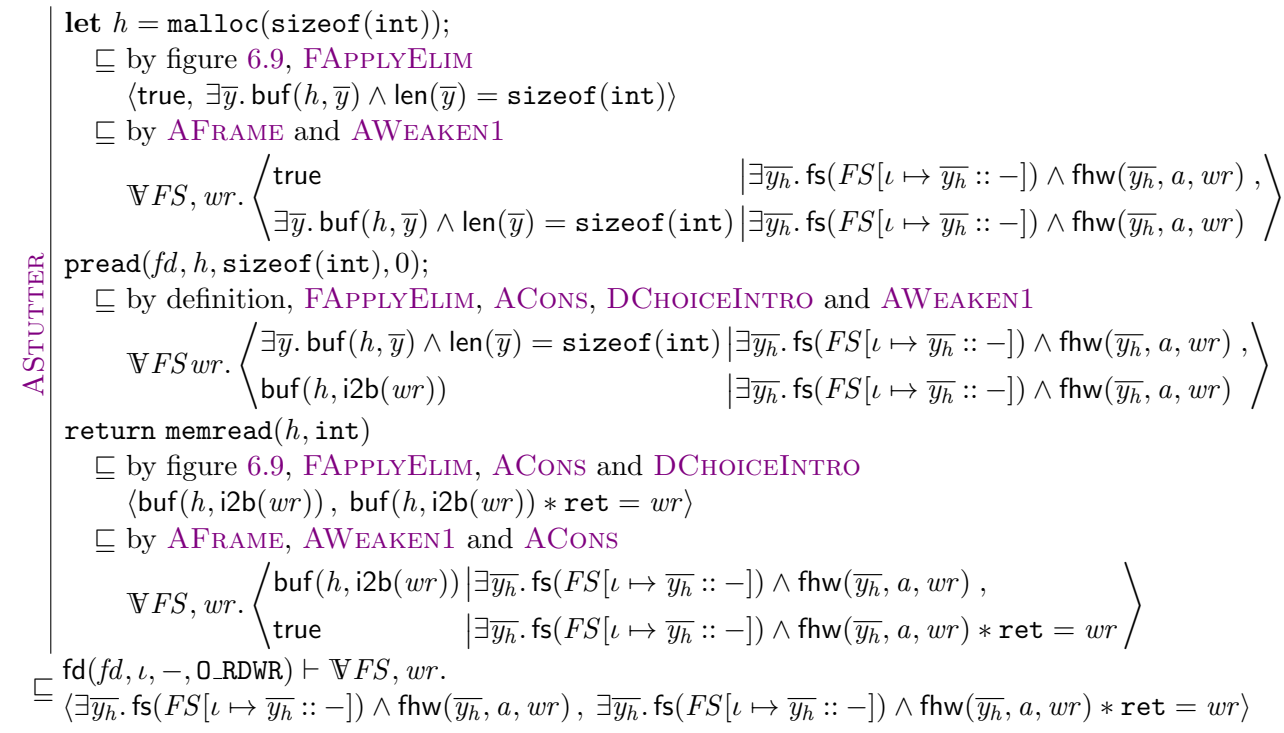

Figure 8.38.: Proof of `fifo_get_writers` abstraction.

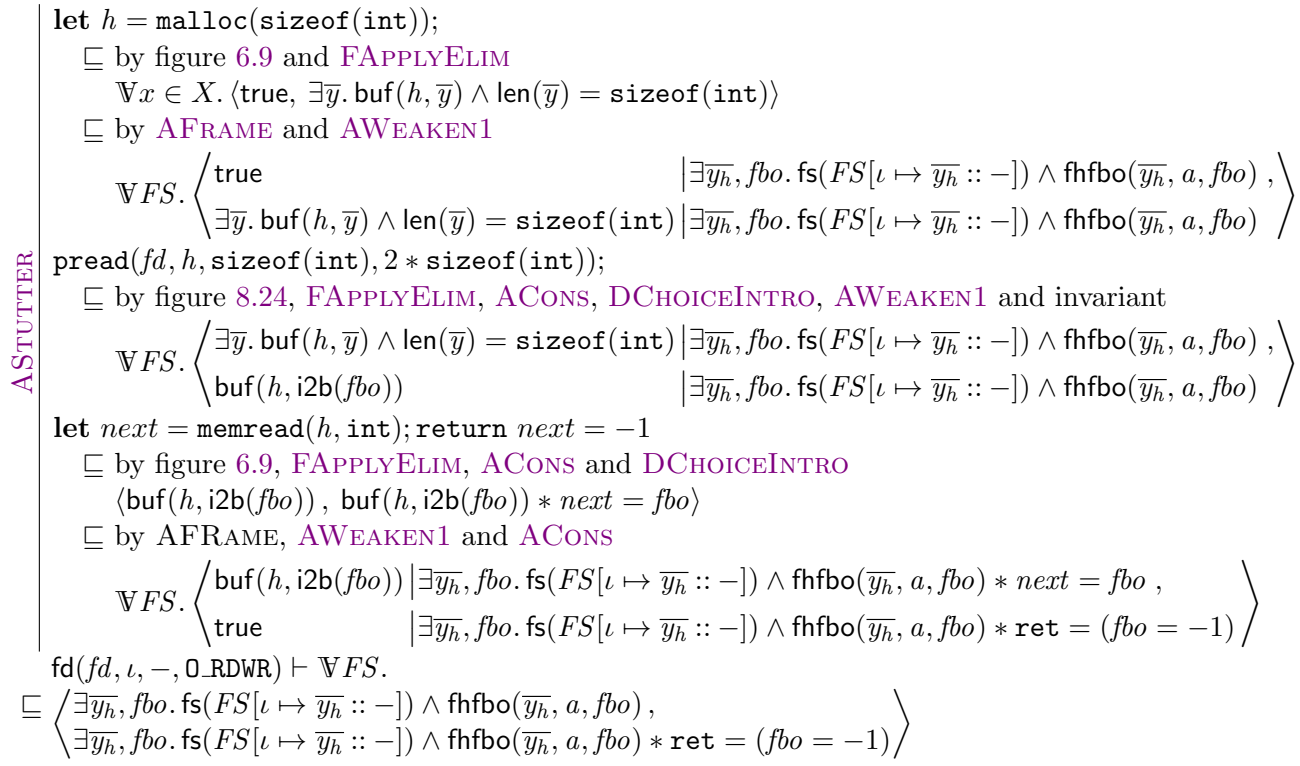

Figure 8.39.: Derivation of specification for `fifo_is_empty`.

We then proceed to `fifo_get_writers` and `fifo_is_empty`. To refine to the specifications in figures 8.38 and 8.39 respectively, we apply OpenRegion twice as in the case of `fifo_get_name` previously.

Finally, we perform the `unlock` in the $writers = 0 \wedge empty$ branch where, similarly to `lock` earlier, we derive a specification in which the lock resource is updated in the private part of an atomic statement. To conclude the proof we remove the branches not applicable to the `fifo_read_nowriters` case. For this we use the demonic interpretation of **if-then-else**, abstract each unused branch to `abort` via MinMax and then retain only the relevant branches by applying DChoiceId. This results in a sequence of atomic specification statements none of which update the named pipe. Then, it is a simple matter of using AFrame and AWeaken1 to frame on resource and the private part not used in some steps and then applying AStutter to coalesce all the steps into the atomic specification at the bottom.

Now let us consider the `fifo_read_partial` case. In contrast to `fifo_read_nowriters`, the named pipe is updated this time. We give a proof sketch of `fifo_read`$(ffd, ptr, sz) \sqsubseteq$ `fifo_read_partial`(ffd, ptr, sz) in figure 8.40. The proof is similar to the `fifo_read_nowriters` case in figure 8.35, except that the relevant branch in this case is the one using `fifo_read_contents`. This is the only operation that performs an update to the named pipe. All other operations are stuttering steps, and thus by using UseAtomic, to abstract `fifo_read_contents` to an update on the **Fifo** region, and AFrame to frame on the additional resources for all the stuttering steps, and then applying AStutter, we coalesce all the steps into a single atomic specification statement from which we derive `fifo_read_partial`.

We refine the specification of `fifo_read_contents` in figure 8.41. The proof is straightforward. The only operation that updates the named pipe is the `pwrite` that updates the data offset in the header of the named-pipe file. All other operations are stuttering steps.

Next, consider `fifo_read`$(ffd, ptr, sz) \sqsubseteq$ `fifo_read_complete`(ffd, ptr, sz). The proof of this case follows exactly the same refinement steps to that of `fifo_read_partial` in figure 8.40. The only difference is in the established postcondition.

Finally, consider the case of `fifo_read`$(ffd, ptr, sz) \sqsubseteq$ `fifo_eaccess`$(fd,$ `O_WRONLY`$)$. This case is trivial to prove: there is no access to the named pipe at all.

AFrame, AStutter and Ind

let $mode = \texttt{fifo\_get\_mode}(\mathit{ffd})$;
let $fd = \texttt{fifo\_get\_fd}(\mathit{ffd})$;
if $mode = \texttt{O\_RDONLY}$ **then**
  let $a = \texttt{fifo\_get\_name}(fd)$;
  let $lf = /\text{“tmp”}/(a + \text{“.lock”})$;
  $\texttt{lock}(lf)$;
    $\sqsubseteq$ by figure 8.1 and $\mathsf{Inv} = [\mathsf{F}(\iota)]$
      $\{\mathsf{isLock}(s, lf)\ ,\ \mathsf{isLock}(s, lf) * \mathsf{Locked}(s, lf) * [\mathsf{F}(\iota)]\}$
    $\sqsubseteq$ by definition 57
      $\sqsubseteq \langle \mathsf{isLock}(s, lf)\ \mid \mathsf{true},\ \mathsf{isLock}(s, lf) * \mathsf{Locked}(s, lf) * [\mathsf{F}(\iota)]\ \mid \mathsf{true} \rangle$
  let $writers = \texttt{fifo\_get\_writers}(fd)$;
  let $empty = \texttt{fifo\_is\_empty}(fd)$;
  if $writers = 0 \wedge empty$ **then**
    $\texttt{unlock}(lf)$;
    $\texttt{return}\ 0$
  else if $empty$ **then**
    $\texttt{unlock}(lf)$;
    $\texttt{return fifo\_read}(fd, ptr, sz)$
  else
    let $sz' = \texttt{fifo\_read\_contents}(fd, ptr, sz)$;
      $\sqsubseteq$ by figure 8.41
        $\mathsf{fd}(fd, \iota, -, \texttt{O\_RDWR}) * [\mathsf{F}(\iota)] * [\mathsf{R}]_\alpha \vdash \mathbb{V} wr, rd, \overline{y}.$
        $\left\langle \begin{array}{l} \mathbf{Fifo}_\alpha(\iota, a, (wr, rd, \overline{y})) * \mathsf{buf}(ptr, \overline{y_t}) \wedge \mathsf{len}(\overline{y_t}) = sz, \\ \mathsf{len}(\overline{y}) < sz \Rightarrow \mathbf{Fifo}_\alpha(\iota, a, (wr, rd, \epsilon)) * \mathsf{buf}(ptr, \overline{y_t} \upharpoonleft \overline{y}) * sz' = \mathsf{len}(\overline{y}) \end{array} \right\rangle$
    $\texttt{unlock}(lf)$;
      $\sqsubseteq$ by figure 8.2 and $\mathsf{Inv} = [\mathsf{F}(\iota)]$
        $\{\mathsf{Locked}(s, lf) * [\mathsf{F}(\iota)]\ ,\ \mathsf{true}\}$
      $\sqsubseteq$ by definition 57 and AFrame
        $\langle \mathsf{isLock}(s, lf) * \mathsf{Locked}(s, lf) * [\mathsf{F}(\iota)]\ \mid \mathsf{true},\ \mathsf{isLock}(s, lf)\ \mid \mathsf{true} \rangle$
    $\texttt{return}\ sz'$
  fi
else $\texttt{return EACCESS}$ **fi**

$\mathsf{isLock}(s', lf)\ \mathsf{buf}(\mathit{ffd}, \mathsf{i2b}(\texttt{O\_RDONLY}) :: \mathsf{i2b}(fd)) * \mathsf{fd}(fd, \iota, -, \texttt{O\_RDWR}) * [\mathsf{R}]_\alpha \vdash \mathbb{V} wr, rd, \overline{y}.$
$\sqsubseteq \left\langle \begin{array}{l} \mathbf{Fifo}_\alpha(\iota, a, (wr, rd, \overline{y})) * \mathsf{buf}(ptr, \overline{y_t}) \wedge \mathsf{len}(\overline{y_t}) = sz, \\ wr > 0 \wedge \mathsf{len}(\overline{y}) < sz \Rightarrow *\mathbf{Fifo}_\alpha(\iota, a, (wr, rd, \epsilon)) * \mathsf{buf}(ptr, \overline{y_t} \upharpoonleft \overline{y_s}) * \texttt{ret} = \mathsf{len}(\overline{y}) \end{array} \right\rangle$
$\sqsubseteq$ by AFrame and Abstract and $s = (\alpha, s')$
$\mathbb{V} wr, rd, \overline{y_s}. \left\langle \begin{array}{l} \mathsf{ffd}(s, \mathit{ffd}, \iota, \texttt{O\_RDONLY}) * \mathsf{fifo}(s, \iota, wr, rd, \overline{y}) * \mathsf{buf}(ptr, \overline{y_t}) \wedge \mathsf{len}(\overline{y_t}) = sz, \\ wr > 0 \wedge \mathsf{len}(\overline{y_s}) < sz \Rightarrow \mathsf{ffd}(s, \mathit{ffd}, \iota, \texttt{O\_RDONLY}) * \mathsf{fifo}(s, \iota, wr, rd, \epsilon) \\ \qquad\qquad * \mathsf{buf}(ptr, \overline{y_t} \upharpoonleft \overline{y}) * \texttt{ret} = \mathsf{len}(\overline{y}) \end{array} \right\rangle$

Figure 8.40.: Proof of `fifo_read` in the case of `fifo_read_partial`.

Let $\mathsf{ffby}(\overline{y}, a, \mathit{fbo}, \overline{y}') \triangleq \mathsf{fhfbo}(\overline{y}, a, \mathit{fbo}) \wedge \mathsf{fbs}(\overline{y}, a, \overline{y}')$

AFrame, ACons and AStutter

let $h = \texttt{malloc}(\mathit{sz})$;
$\sqsubseteq$ by figure 6.9, FApplyElim, AWeaken2
$\{\mathsf{true},\ \exists \overline{y}'.\, \mathsf{buf}(h, \overline{y}') \wedge \mathsf{len}(\overline{y}') = \mathit{sz}\}$
let $\mathit{hoff} = \texttt{malloc}(\texttt{sizeof}(\texttt{int}))$;
$\sqsubseteq$ by figure 6.9, FApplyElim, AWeaken2
$\{\mathsf{true},\ \exists \overline{y}''.\, \mathsf{buf}(\mathit{hoff}, \overline{y}'') \wedge \mathsf{len}(\overline{y}'') = \texttt{sizeof}(\texttt{int})\}$
$\texttt{pread}(\mathit{fd}, \mathit{hoff}, \texttt{sizeof}(\texttt{int}), 2 * \texttt{sizeof}(\texttt{int}))$;
$\sqsubseteq$ by specification, FApplyElim, ACons, AWeaken1 and invariant
$\exists \mathit{FS}, \overline{y}, \mathit{fbo}.\, \mathbf{GFS}(\mathit{FS}) \wedge \mathsf{ffby}(\mathit{FS}(\iota), a, \mathit{fbo}, \overline{y}) * \mathsf{buf}(\mathit{ptr}, \overline{y_t}) \wedge \mathsf{len}(\overline{y_t}) = \mathit{sz} \vdash$
$\{\exists \overline{y}''.\, \mathsf{buf}(\mathit{hoff}, \overline{y}'') \wedge \mathsf{len}(\overline{y}'') = \texttt{sizeof}(\texttt{int}),\ \mathsf{buf}(\mathit{hoff}, \mathsf{i2b}(\mathit{fbo}))\}$
let $\mathit{off} = \texttt{memread}(\mathit{hoff}, \texttt{int})$;
$\sqsubseteq$ by figure 6.9, FApplyElim, AWeaken2 and HCons
$\{\mathsf{buf}(\mathit{hoff}, \mathsf{i2b}(\mathit{fbo}))\,,\ \mathit{off} = \mathit{fbo}\}$
let $\mathit{sz}' = \texttt{pread}(\mathit{fd}, h, \mathit{sz}, \mathit{off})$;
$\sqsubseteq$ by specification, FApplyElim, ACons, AEElim, OpenRegion and invariant
$\exists \mathit{FS}, \overline{y}.\, \mathbf{GFS}(\mathit{FS}) \wedge \mathsf{ffby}(\mathit{FS}(\iota), a, \mathit{off}, \overline{y}) * \mathsf{buf}(\mathit{ptr}, \overline{y_t}) \wedge \mathsf{len}(\overline{y_t}) = \mathit{sz} \vdash$
$\{\exists \overline{y}'.\, \mathsf{buf}(h, \overline{y}') \wedge \mathsf{len}(\overline{y}') = \mathit{sz},\ \mathsf{len}(\overline{y}) < \mathit{sz} \Rightarrow \mathsf{buf}(h, \overline{y}' \upharpoonleft \overline{y}) * \mathit{sz}' = \mathsf{len}(\overline{y})\}$
if $\mathit{sz}' < \mathit{sz}$ **then**
$\texttt{memwrite}(\mathit{hoff}, \mathsf{i2b}(-1))$;
else $\texttt{memwrite}(\mathit{hoff}, \mathsf{i2b}(\mathit{off} + \mathit{sz}))$ **fi**
$\sqsubseteq$ by figure 6.9, FApplyElim, ACons, AWeaken1 and IfThenElse
$\exists \mathit{FS}, \overline{y}.\, \mathbf{GFS}(\mathit{FS}) \wedge \mathsf{ffby}(\mathit{FS}(\iota), a, \mathit{off}, \overline{y}) * \mathsf{buf}(\mathit{ptr}, \overline{y_t}) \wedge \mathsf{len}(\overline{y_t}) = \mathit{sz} \vdash$
$\left\{\begin{array}{l}\mathsf{buf}(\mathit{hoff}, \mathsf{i2b}(\mathit{off})) * \mathsf{len}(\overline{y}) < \mathit{sz} \Rightarrow \mathsf{buf}(h, \overline{y}' \upharpoonleft \overline{y}) * \mathit{sz}' = \mathsf{len}(\overline{y})\,, \\ \mathsf{len}(\overline{y}) < \mathit{sz} \Rightarrow \mathsf{buf}(h, \overline{y}' \upharpoonleft \overline{y}) * \mathsf{buf}(\mathit{hoff}, \mathsf{i2b}(-1)) * \mathit{sz}' = \mathsf{len}(\overline{y})\end{array}\right\}$
$\texttt{pwrite}(\mathit{fd}, \mathit{hoff}, \texttt{sizeof}(\texttt{int}), 2 * \texttt{sizeof}(\texttt{int}))$;
$\sqsubseteq$ by specification, FApplyElim, ACons, AEElim, AWeaken1, UseAtomic and invariant
$\mathsf{buf}(\mathit{ptr}, \overline{y_t}) \wedge \mathsf{len}(\overline{y_t}) = \mathit{sz} \vdash \forall \overline{y}.$
$\left\langle \begin{array}{ll|l} \mathsf{buf}(\mathit{hoff}, \mathsf{i2b}(-)) & & \exists \mathit{FS}.\, \mathbf{GFS}(\mathit{FS}) \wedge \mathsf{ffby}(\mathit{FS}(\iota), a, \mathit{off}, \overline{y})\,, \\ \mathsf{len}(\overline{y}) < \mathit{sz} \Rightarrow \mathsf{buf}(\mathit{hoff}, \mathsf{i2b}(-1)) & & \mathsf{len}(\overline{y}) < \mathit{sz} \Rightarrow \exists \mathit{FS}.\, \mathbf{GFS}(\mathit{FS}) \wedge \mathsf{empfifo}(\mathit{FS}(\iota), a) \end{array} \right\rangle$
$\texttt{memcpy}(\mathit{ptr}, h, \mathit{sz} - \mathit{sz}')$;
$\sqsubseteq$ by figure 6.9, FApplyElim, AFrame, AWeaken1 and ACons
$\exists \mathit{FS}.\, \mathbf{GFS}(\mathit{FS}) \wedge \mathsf{empfifo}(\mathit{FS}(\iota), a) \vdash$
$\left\langle \begin{array}{ll|l} \mathsf{len}(\overline{y}) < \mathit{sz} \Rightarrow \mathsf{buf}(h, \overline{y}' \upharpoonleft \overline{y}) \wedge \mathsf{len}(\overline{y}) = \mathit{sz}' & & \mathsf{buf}(\mathit{ptr}, \overline{y_t}) \wedge \mathsf{len}(\overline{y_t}) = \mathit{sz}\,, \\ \mathsf{true} & & \mathsf{buf}(\mathit{ptr}, \overline{y_t} \upharpoonleft \overline{y}) \end{array} \right\rangle$
$\texttt{return}\ \mathit{sz}'$

$\sqsubseteq \forall \overline{y}. \left\langle \begin{array}{l} \exists \mathit{FS}.\, \mathbf{GFS}(\mathit{FS}) \wedge \mathsf{fimp}(\mathit{FS}(\iota), a, \mathit{wr}, \mathit{rd}, \overline{y}) * \mathsf{buf}(\mathit{ptr}, \overline{y_t}) \wedge \mathsf{len}(\overline{y_t}) = \mathit{sz} \wedge \mathsf{len}(\overline{y}) < \mathit{sz}, \\ \mathsf{len}(\overline{y}) < \mathit{sz} \Rightarrow \exists \mathit{FS}.\, \mathbf{GFS}(\mathit{FS}) \wedge \mathsf{empfifo}(\mathit{FS}(\iota), a) * \mathsf{buf}(\mathit{ptr}, \overline{y_t} \upharpoonleft \overline{y}) * \texttt{ret} = \mathsf{len}(\overline{y}) \end{array} \right\rangle$

$\sqsubseteq$ by ACons and UseAtomic
$\mathsf{fd}(\mathit{fd}, \iota, -, \texttt{O\_RDWR}) * [\mathsf{F}(\iota)] * [\mathsf{R}]_\alpha \vdash \forall \mathit{wr}, \mathit{rd}, \overline{y}.$
$\left\langle \begin{array}{l} \mathbf{Fifo}_\alpha(\iota, a, (\mathit{wr}, \mathit{rd}, \overline{y})) * \mathsf{buf}(\mathit{ptr}, \overline{y_t}) \wedge \mathsf{len}(\overline{y_t}) = \mathit{sz}, \\ \mathsf{len}(\overline{y}) < \mathit{sz} \Rightarrow \mathbf{Fifo}_\alpha(\iota, a, (\mathit{wr}, \mathit{rd}, \epsilon)) * \mathsf{buf}(\mathit{ptr}, \overline{y_t} \upharpoonleft \overline{y}) * \texttt{ret} = \mathsf{len}(\overline{y}) \end{array} \right\rangle$

Figure 8.41.: Proof of `fifo_read_contents` abstraction in the case of a partial read.

Verification of `fifo_write`

The specification of `fifo_write` given in section 8.3.1 is slightly simpler than that of `fifo_read` as it has only three demonic cases: `fifo_write_readers`, `fifo_epipe` and `fifo_eaccess`. Out of these `fifo_epipe` is proven similarly to the `fifo_read_nowriters` case of `fifo_read`, the only difference being returning the `EPIPE` error code if the number of readers is 0, instead of returning 0 if there are no writers and the named pipe is empty. The case of `fifo_eaccess` is trivial to prove. Therefore, here we focus only on proving that $\texttt{fifo\_write}(fd, ptr, sz) \sqsubseteq \texttt{fifo\_write\_readers}(fd, ptr, sz)$.

We establish this refinement in figure 8.42. Again, we assume that the context invariant FifoCtx holds at every step. The refinement proofs follows a similar pattern to that of `fifo_read_partial` in figure 8.40. The only operation that updates the named pipe is `fifo_write_contents`. All other operations are stuttering steps.

In the first refinement step (at the bottom) in figure 8.42 we open the abstract predicates ffd and ffd and frame off the resources not required, such as the RD and WR guards. Note that the refined specification uses an invariant for the resources used, but not modified in the implementation. In the next step we apply AStutter to refine the specification to the sequence of operations comprising the implementation of `fifo_write`. The refinements to `lock` and `unlock` are established in the same manner as in the case of `fifo_read_partial`, discussed previously. For the refinement to `fifo_write_contents` we use the specification derived in figure 8.43.

The first refinement step in figure 8.43 (at the bottom) uses UseAtomic to refine the update on the **Fifo** region to an update on its interpretation in terms of the global file-system region **GFS**. From there we apply AStutter to refine the specification to the sequence of operations comprising `fifo_write_contents`. Which of the two `pwrite`s updates the named pipe depends on whether the named pipe is initially empty. If the named pipe is empty then the first `pwrite` is a stuttering step for the second `pwrite`, within the **if**-branch, which updates the data offset field in the header of the named-pipe file. Otherwise, the first `pwrite` updates the named pipe with the second `pwrite` not occurring at all.

IfThenElse and AStutter

let $fd = \texttt{fifo\_get\_fd}(\mathit{ffd})$;
let $\mathit{mode} = \texttt{fifo\_get\_mode}(\mathit{ffd})$;
if $\mathit{mode} = \texttt{O\_WRONLY}$ **then**
  let $\mathit{name} = \texttt{fifo\_get\_name}(\mathit{fd})$;
  let $\mathit{lf} = /\text{“tmp”}/(a + \text{“.lock”})$;
  $\texttt{lock}(\mathit{lf})$;
    $\sqsubseteq$ by figure 8.1 and $\mathsf{Inv} = [\mathsf{F}(\iota)]$
      $\{\mathsf{isLock}(s', \mathit{lf})\ ,\ \mathsf{isLock}(s', \mathit{lf}) * \mathsf{Locked}(s', \mathit{lf}) * [\mathsf{F}(\iota)]\}$
    $\sqsubseteq$ by AFrame and definition 57
      $\exists wr, rd, \overline{y_s}.\ \mathbf{Fifo}_\alpha(\iota, a, (wr, rd, \overline{y_s})) \vdash \{\mathsf{isLock}(s', \mathit{lf})\ ,\ \mathsf{isLock}(s', \mathit{lf}) * \mathsf{Locked}(s', \mathit{lf}) * [\mathsf{F}(\iota)]\}$
  let $\mathit{readers} = \texttt{fifo\_get\_readers}(\mathit{fd})$;
    $\sqsubseteq$ by figure 8.38 (similarly), ACons, AFrame, AEElim, 2 × OpenRegion and invariant
      $\exists wr, rd, \overline{y_s}.\ \mathbf{Fifo}_\alpha(\iota, a, (wr, rd, \overline{y_s})) * [\mathsf{F}(\iota)] \vdash \langle \mathsf{true},\ \mathit{readers} = rd \rangle$
  if $\mathit{readers} = 0$ **then** `return EPIPE`
  else
    let $sz' = \texttt{fifo\_write\_contents}(\mathit{fd}, \mathit{ptr}, \mathit{sz})$;
      $\sqsubseteq$ by figure 8.43, AFrame, AWeaken1 and invariant
        $$\forall FS, wr, rd, \overline{y_s}.\ \left\langle \begin{array}{l|l} \mathsf{Locked}(s', \mathit{lf}) * [\mathsf{F}(\iota)] & \mathbf{Fifo}_\alpha(\iota, a, (wr, rd, \overline{y_s}))\ , \\ \mathsf{Locked}(s', \mathit{lf}) * [\mathsf{F}(\iota)] & \mathbf{Fifo}_\alpha(\iota, a, (wr, rd, \overline{y_s} :: \overline{y_t})) * sz' = \mathsf{len}(\overline{y_t}) \end{array} \right\rangle$$
    $\texttt{unlock}(\mathit{lockfile})$;
      $\sqsubseteq$ by figure 8.2 and $\mathsf{Inv} = [\mathsf{F}(\iota)]$
        $\{\mathsf{Locked}(s', \mathit{lf}) * [\mathsf{F}(\iota)]\ ,\ \mathsf{true}\}$
      $\sqsubseteq$ by definition 57 and AFrame
        $\exists wr, rd, \overline{y_s}.\ \mathbf{Fifo}_\alpha(\iota, a, (wr, rd, \overline{y_s} :: \overline{y_t})) \vdash \{\mathsf{isLock}(s', \mathit{lf}) * \mathsf{Locked}(s', \mathit{lf}) * [\mathsf{F}(\iota)]\ ,\ \mathsf{isLock}(s', \mathit{lf})\}$
    `return` sz'
  fi
else `return EACCESS` **fi**

$\mathsf{buf}(\mathit{ffd}, \mathsf{i2b}(\texttt{O\_WRONLY}) :: \mathsf{i2b}(\mathit{fd})) * \mathsf{fd}(\mathit{fd}, \iota, -, \texttt{O\_RDWR}) * [\mathsf{W}]_\alpha * \mathsf{isLock}(s, \mathit{lf})$
$\sqsubseteq$ $* \mathsf{buf}(\mathit{ptr}, \overline{y_t}) \wedge \mathsf{len}(\overline{y_t}) = \mathit{sz} \vdash \forall wr, rd, \overline{y_s}.$
$\langle \mathbf{Fifo}_\alpha(\iota, a, (wr, rd, \overline{y_s})),\ rd > 0 \Rightarrow \mathbf{Fifo}_\alpha(\iota, a, wr, rd, (\overline{y_s} :: \overline{y_t})) \rangle$
$\sqsubseteq$ by AFrame, Abstract and $s = (\alpha, s')$
$$\forall wr, rd, \overline{y_s}.\ \left\langle \begin{array}{l} \mathsf{ffd}(s, \mathit{fd}, \iota, \texttt{O\_WRONLY}) * \mathsf{fifo}(s, \iota, wr, rd, \overline{y_s}) * \mathsf{buf}(\mathit{ptr}, \overline{y_t}) \wedge \mathsf{len}(\overline{y_t}) = \mathit{sz}, \\ rd > 0 \Rightarrow \mathsf{ffd}(s, \mathit{fd}, \iota, \texttt{O\_WRONLY}) * \mathsf{fifo}(s, \iota, wr, rd, \overline{y_s} :: \overline{y_t}) * \mathsf{buf}(\mathit{ptr}, \overline{y_t}) * \texttt{ret} = \mathit{sz} \end{array} \right\rangle$$

Figure 8.42.: Proof of `fifo_write` in the case of available readers.

$\sqsubseteq$ IfThenElse, ACons, AFrame and AStutter

let $empty = \texttt{fifo\_is\_empty}(fd)$;
$\sqsubseteq$ similarly to figure 8.37, ACons, AEElim, OpenRegion and invariant
$\exists FS, \overline{y_s}.\ \mathbf{GFS}(FS) \wedge \mathsf{fbs}(FS(\iota), a, \overline{y_s}) \wedge \mathsf{fhfbo}(FS(\iota), a, fbo) * \mathsf{buf}(ptr, \overline{y_t}) \wedge \mathsf{len}(\overline{y_t}) = sz \vdash$
$\langle \mathsf{true},\ empty \Rightarrow fbo = -1 \wedge \overline{y_s} = \epsilon \rangle$
let $off = \texttt{lseek\_end}(fd, 0, \texttt{SEEK\_END}\})$;
$\sqsubseteq$ by specification, FApplyElim, ACons, AEElim, DChoiceIntro, OpenRegion and invariant
$\exists FS, \overline{y_s}.\ \mathbf{GFS}(FS) \wedge \mathsf{fbs}(FS(\iota), a, \overline{y_s}) \wedge \mathsf{fhfbo}(FS(\iota), a, fbo) * \mathsf{buf}(ptr, \overline{y_t}) \wedge \mathsf{len}(\overline{y_t}) = sz \vdash$
$\langle \mathsf{true},\ off = \mathsf{len}(FS(\iota)) \rangle$
$\texttt{pwrite}(fd, h, sz, off)$;
$\sqsubseteq$ by specification, FApplyElim, ACons, AEElim, DChoiceIntro, UpdateRegion and invariant
$\mathsf{buf}(ptr, \overline{y_t}) \wedge \mathsf{len}(\overline{y_t}) = sz \vdash \forall FS, \overline{y_i}, \overline{y_s}.$
$\left\langle \begin{array}{l} \mathbf{GFS}(FS[\iota \mapsto \overline{y_i}]) \wedge \mathsf{fbs}(\overline{y_i}, a, \overline{y_s}) * off = \overline{y_i}, \\ \mathbf{GFS}(FS[\iota \mapsto \overline{y_i} :: \overline{y_t}]) \wedge (\neg empty \Rightarrow \mathsf{fbs}(\overline{y_i} :: \overline{y_t}, a, \overline{y_s} :: \overline{y_t})) \wedge (empty \Rightarrow \mathsf{fbs}(\overline{y_i} :: \overline{y_t}, a, \epsilon)) \end{array} \right\rangle$
if $empty$ **then**
$\texttt{malloc}(hoff, \texttt{sizeof}(\texttt{int}))$;
$\texttt{memwrite}(hoff, \mathsf{i2b}(off))$;
$\sqsubseteq$ by figure 6.9, FApplyElim, AWeaken2 and Seq
$\{\mathsf{true},\ \mathsf{buf}(hoff, \mathsf{i2b}(off))\}$
$\texttt{pwrite}(fd, hoff, \texttt{sizeof}(\texttt{int}), 2 * \texttt{sizeof}(\texttt{int}))$; **return** sz
$\sqsubseteq$ by specification, FApplyElim, ACons, AEElim, UseAtomic and invariant
$\mathsf{buf}(ptr, \overline{y_t}) \wedge \mathsf{len}(\overline{y_t}) = sz \vdash \forall FS, \overline{y_i}.$
$\left\langle \begin{array}{ll} \mathsf{true} & \left| \mathbf{GFS}(FS[\iota \mapsto \overline{y_i} :: \overline{y_t}]) \wedge \mathsf{fbs}(\overline{y_i} :: \overline{y_t}, a, \epsilon) \right. , \\ \mathsf{buf}(hoff, \mathsf{i2b}(off)) & \left| \mathbf{GFS}(FS[\iota \mapsto \overline{y_i} :: \overline{y_t}]) \wedge \mathsf{fbs}(\overline{y_i} :: \overline{y_t}, a, \overline{y_t}) \right. \end{array} \right\rangle$
fi
$\sqsubseteq$ $\mathsf{buf}(ptr, \overline{y_t}) \wedge sz = \mathsf{len}(\overline{y_t}) * \mathsf{fd}(fd, \iota, -, \texttt{O\_RDWR}) * [\mathsf{F}(\iota)] \vdash \forall FS, \overline{y_s}.$
$\langle \mathbf{GFS}(FS) \wedge \mathsf{fbs}(FS(\iota), a, \overline{y_s}),\ \mathbf{GFS}(FS) \wedge \mathsf{fbs}(FS(\iota), a, \overline{y_s} :: \overline{y_t}) \rangle$
$\sqsubseteq$ by ACons, AEElim, AFrame and UseAtomic
$\mathsf{buf}(ptr, \overline{y_t}) \wedge sz = \mathsf{len}(\overline{y_t}) * \mathsf{fd}(fd, \iota, -, \texttt{O\_RDWR}) * [\mathsf{W}]_\alpha * [\mathsf{F}(\iota)] \vdash \forall wr, rd, \overline{y_s}.$
$\langle \mathbf{Fifo}_\alpha(\iota, a, (wr, rd, \overline{y_s})),\ \mathbf{Fifo}_\alpha(\iota, a, (wr, rd, \overline{y_s} :: \overline{y_t})) \rangle$

Figure 8.43.: Proof of `fifo_write_contents` abstraction.

8.4. Conclusions

Formal specifications of POSIX file-system operations are valuable only if they are useful. In this dissertation we demonstrate their usefulness in reasoning about client applications, something less explored in the literature than reasoning about implementations. We first discussed client reasoning with our specifications in chapter 6. In this chapter we have further explored the merits of our approach by examining further examples of client programs. We revisited the lock-file module deriving a CAP-style specification from the atomic specification first seen in chapter 6, giving a first demonstration of building up layers of abstractions over module specifications. We reused the CAP-style specification in our case-study of named pipes demonstrating the modularity and compositionality of our approach. This case study also demonstrates the scalability of our approach to non-trivial clients that implement data structures over the file systems. Additionally, this example demonstrates that we can derive formal specifications for parts of the POSIX standard, such as named pipes, from smaller formally specified fragments of POSIX. Finally, by formally examining an example of a concurrent email client and server interaction we have demonstrated the significance of staying faithful to the complexity of the POSIX standard regarding concurrency. Assuming simpler behaviours in a formal specification, for example by ignoring the non-atomicity of path resolution, leads to proving false facts about client programs.

9. Fault Tolerance

The primary purpose of file systems is the persistence of data. File systems play such a crucial role within operating systems that they generally strive to behave sensibly, avoiding data corruptions, even if all else goes wrong. It is then no surprise that POSIX specifies file-systems operations to maintain a valid file system under normal circumstances. Even though POSIX remains silent on what implementations should do in the exceptional case where a host failure, such as power failure or a system wide crash, interrupts the execution of file-system operations, most implementations provide the same guarantee; they maintain a valid file system. Many file-system implementations also provide stronger guarantees where not only a valid file system is maintained, but no data loss occurs as well.

The file system specifications and reasoning developed in chapters 6 and 7 ignores such faults or crashes. Our specification language and refinement calculus for atomicity assumes that such events never happen. This is reasonable for formally specifying the POSIX file-system interface, since POSIX remains silent on the issue of fault-tolerance. Yet reasoning about faults and fault-recovery is clearly desirable for file-system implementations as well as client applications that provide fault-tolerance guarantees such as database systems. The fundamental problem is that separation-logic based reasoning, on which this dissertation builds upon, is inherently fault-avoiding. Therefore, we go back to the basics of resource reasoning, as introduced by separation logic [81, 76], and extend it to programs that experience faults and potentially recover from such events.

We begin in section 9.1 by giving an overview of faults and how we extend resource reasoning to reason about them. In section 9.2 we present the fundamental concepts of approach through the "hello world" example of fault tolerant systems: the bank account transfer. We then discuss how our approach can be applied to file-system specifications in section 9.3. As our focus now is on reasoning about fault tolerance, we keep this discussion in a simpler setting, not aiming at POSIX faithfulness as we did earlier in chapter 6. Our fault-tolerant resource reasoning is a generalised extension of separation-logic based reasoning and in chapter 9.4 we present FTCSL, a particular instance where we extend concurrent separation logic [76] with faults. In section 9.5 we discuss faults in the presence of concurrency by revisiting the bank account transfer example in the concurrent setting. We then proceed to discuss the semantics and soundness of our approach in section 9.6 the bedrock of which is fault-tolerant extension of the Views framework [35]. In section 9.7 we apply our reasoning to the ARIES recovery algorithm as a case-study. Finally, in section 9.8 we discuss related work on fault tolerance and program logics.

9.1. Faults and Resource Reasoning

There are many ways that software can fail: either software itself can be the cause of the failure (e.g. memory overflow or null pointer dereferencing); or the failure can arise independently of the software. These unpredictable failures are either *transient faults*, such as when a bit is flipped by

cosmic radiation, or *host failures* (also referred to as crashes). Host failures can be classified into *soft*, such as those arising from power loss which can be fixed by rebooting the host, and *hard*, such as permanent hardware failure.

Consider a simple transfer operation that moves money between bank accounts. Assuming that bank accounts can have overdrafts, the transfer can be regarded as a sequence of two steps: first, subtract the money from one bank account; and then add the money to the other account. In the absence of host failures, the operation should succeed. However, if a host failure occurs in the middle of the transfer, money is lost. Programmers employ various techniques to recover some consistency after a crash, such as write-ahead logging (WAL) and associated recovery code. In this dissertation, we develop the reasoning to verify programs that can recover from host failures, assuming hard failures do not happen.

Resource reasoning, as introduced by separation logic [81], is a method for verifying that programs do not fail. A triple $\{P\}\,\mathbb{C}\,\{Q\}$ is given a *fault avoiding*, partial correctness interpretation. This means that, assuming the precondition P holds then, if program $\mathbb{C}$ terminates, it must be the case that P does not fail and has all the resource necessary to yield a result which satisfies postcondition Q. Such reasoning guarantees the correct behaviour of the program, ensuring that the software does not crash itself due to bugs, e.g. invalid memory access. However, it assumes that there are no other failures of any form. To reason about programs that can recover from host failures, we must change the underlying assumptions of resource reasoning.

We swap the traditional resource models with one that distinguishes between *volatile* and *durable* resource: the volatile resource (e.g. in RAM) does not survive crashes; whereas the durable resource (e.g. on the hard drive) does. Recovery operations use the durable state to repair any corruptions caused by the host failure. We introduce *fault-tolerant resource reasoning* to reason about programs in the presence of host failures and their associated recovery operations. We introduce a new fault-tolerant Hoare triple judgement of the form:

$$S \vdash \{P_V \mid P_D\}\,\mathbb{C}\,\{Q_V \mid Q_D\}$$

which has a partial-correctness, *resource fault-avoiding* and *host failing* interpretation. From the standard resource fault avoiding interpretation: assuming the precondition $P_V \mid P_D$ holds, where the volatile state satisfies P_V and the durable P_D, then if $\mathbb{C}$ terminates and there is no host failure, the volatile and durable resource will satisfy Q_V and Q_D respectively. From the host-failing interpretation: when there is a host failure, the volatile state is lost and after potential recovery operations, the remaining durable state will satisfy the *fault-condition* S.

We extend the Views framework [35], which provides a general account of concurrent resource reasoning, with these fault-tolerant triples to provide a general framework for fault-tolerant resource reasoning. We instantiate our framework to give a fault-tolerant extension of concurrent separation logic [76] as an illustrative example. We use this instantiation to verify the correctness of programs that make use of recovery protocols to guarantee different levels of fault tolerance. In particular, we study a simple bank transaction using write-ahead logging and a simplified ARIES recovery algorithm [69], widely used in database systems.

9.2. Motivating Examples

We introduce fault-tolerant resource reasoning by showing how a simple bank transfer can be implemented and verified to be robust against host failures.

9.2.1. Naive Bank Transfer

Consider a simple transfer operation that moves money between bank accounts. Using a separation logic [81] triple, we can specify the transfer operation as:

$$\left\{ \mathsf{Account}(\texttt{from}, v) * \mathsf{Account}(\texttt{to}, w) \right\}$$
$$\texttt{transfer}(\texttt{from}, \texttt{to}, \texttt{amount})$$
$$\left\{ \mathsf{Account}(\texttt{from}, v - \texttt{amount}) * \mathsf{Account}(\texttt{to}, w + \texttt{amount}) \right\}$$

The internal structure of the account is abstracted using the abstract predicate [77], $\mathsf{Account}(x, v)$, which states that there is an account x with balance v. The specification says that, with access to the accounts `from` and `to`, the `transfer` will not fault. It will decrease the balance of account `from` by `amount` and increase the balance of account `to` by the same value. We can implement the transfer operation as follows:

```
function transfer(from, to, amount) {
  widthdraw(from, amount);
  deposit(to, amount);
}
```

Using separation logic, it is possible to prove that this implementation satisfies the specification, assuming no host failures. This implementation gives no guarantees in the presence of host failures. However, for this example, it is clearly desirable for the implementation to be aware that host failures occur. In addition, the implementation should guarantee that in the event of a host failure the operation is *atomic*: either it happened as a whole, or nothing happened. Note that the word atomic is also used in concurrency literature to describe an operation that takes effect at a single, discrete instant in time. In section 9.4 we combine concurrency atomicity of concurrent separation logic with host failure atomicity: if an operation is concurrently atomic then it is also host-failure atomic.

9.2.2. Fault-tolerant Bank Transfer: Implementation

We want an implementation of `transfer` to be robust against host failures and guarantee atomicity. One way to achieve this is to use write-ahead logging (WAL) combined with a recovery operation. We assume a simplified file-system module which provides standard operations to atomically create and delete files, test their existence, and write to and read from files. Since file systems are critical, their operations have associated internal recovery operations in the event of a host failure.

Given an arbitrary program $\mathbb{C}$, we use $[\mathbb{C}]$ to identify that the program is associated with a recovery. We can now rewrite the `transfer` operation, making use of the file-system operations to implement

a stylised WAL protocol as follows:

```
function transfer(from, to, amount) {
  fromAmount := getAmount(from);
  toAmount := getAmount(to);
  [create(log)];
  [write(log, (from, to, fromAmount, toAmount))];
  setAmount(from, fromAmount − amount);
  setAmount(to, toAmount + amount);
  [delete(log)];
}
```

The operation works by first reading the amounts stored in each account. It then creates a log file, `log`, where it stores the amounts for each account. It then updates each account, and finally deletes the log file. If a host failure occurs the log provides enough information to implement a recovery operation. In particular, its absence from the durable state means the transfer either happened or not, while its presence indicates the operation has not completed. In the latter case, we restore the initial balance by reading the log. An example of a recovery operation is the following:

```
function transferRecovery() {
  b := [exists(log)];
  if (b) {
    (from, to, fromAmount, toAmount) := [read(log)];
    if (from ≠ nil && to ≠ nil) {
      setAmount(from, fromAmount); setAmount(to, toAmount);
    }
    [delete(log)];
  }
}
```

The operation tests if the log file exists. If it does not, the recovery completes immediately since the balance is already consistent. Otherwise, the values of the accounts are reset to those stored in the log file which correspond to the initial balance. While the recovery operation is running, a host failure may occur, which means that upon reboot the recovery operation will run again. Eventually the recovery operation completes, at which point the transfer either occurred or did not. This guarantees that `transfer` is atomic with respect to host failures.

9.2.3. Fault-tolerant Bank Transfer: Verification

We introduce the following new Hoare triple for specifying programs that run in a machine where host failures can occur:

$$S \vdash \{P_V \mid P_D\}\, \mathbb{C}\, \{Q_V \mid Q_D\}$$

where P_V, P_D, Q_V, Q_D and S are assertions in the style of intuitionistic separation logic and $\mathbb{C}$ is a program. P_V and Q_V describe the volatile resource, and P_D and Q_D describe the durable resource.

$$\mathsf{nofile}(\mathtt{name}) \vee \mathsf{file}(\mathtt{name}, []) \vdash \{\mathsf{true} \mid \mathsf{nofile}(\mathtt{name})\}\,[\mathtt{create}(\mathtt{name})]\,\{\mathsf{true} \mid \mathsf{file}(\mathtt{name}, [])\}$$
$$\mathsf{nofile}(\mathtt{name}) \vee \mathsf{file}(\mathtt{name}, xs) \vdash \{\mathsf{true} \mid \mathsf{file}(\mathtt{name}, xs)\}\,[\mathtt{delete}(\mathtt{name})]\,\{\mathsf{true} \mid \mathsf{nofile}(\mathtt{name})\}$$
$$\mathsf{nofile}(\mathtt{name}) \vdash \{\mathsf{true} \mid \mathsf{nofile}(\mathtt{name})\}\,[\mathtt{exists}(\mathtt{name})]\,\{\mathtt{ret} = \mathsf{false} \mid \mathsf{nofile}(\mathtt{name})\}$$
$$\mathsf{file}(\mathtt{name}, xs) \vdash \{\mathsf{true} \mid \mathsf{file}(\mathtt{name}, xs)\}\,[\mathtt{exists}(\mathtt{name})]\,\{\mathtt{ret} = \mathsf{true} \mid \mathsf{file}(\mathtt{name}, xs)\}$$
$$\mathsf{file}(\mathtt{name}, xs) \vee \mathsf{file}(\mathtt{name}, xs +\!\!+ [\mathtt{x}]) \vdash \begin{array}{c} \{\ \mathsf{true} \mid \mathsf{file}(\mathtt{name}, xs)\ \} \\ [\mathtt{write}(\mathtt{name}, \mathtt{x})] \\ \{\ \mathsf{true} \mid \mathsf{file}(\mathtt{name}, xs +\!\!+ [\mathtt{x}])\ \} \end{array}$$
$$\mathsf{file}(\mathtt{name}, []) \vdash \{\mathsf{true} \mid \mathsf{file}(\mathtt{name}, [])\}\,[\mathtt{read}(\mathtt{name})]\,\{\mathtt{ret} = \mathtt{null} \mid \mathsf{file}(\mathtt{name}, [])\}$$
$$\mathsf{file}(\mathtt{name}, [x] +\!\!+ xs) \vdash \{\mathsf{true} \mid \mathsf{file}(\mathtt{name}, [x] +\!\!+ xs)\}\,[\mathtt{read}(\mathtt{name})]\,\{\mathtt{ret} = x \mid \mathsf{file}(\mathtt{name}, [x] +\!\!+ xs)\}$$

Figure 9.1.: Specification of a simplified journaling file system.

The judgement is read as a normal Hoare triple when there are no host failures. The interpretation of the triples is partial *resource fault avoiding* and *host failing*. Given an initial $P_V \mid P_D$, it is safe to execute $\mathbb{C}$ without causing a resource fault. If no host failure occurs, and $\mathbb{C}$ terminates, the resulting state will satisfy $Q_V \mid Q_D$. On the other hand if a host failure occurs, then the durable state will satisfy the *fault-condition* S.

Given the new judgement, we can describe the resulting state after a host failure. Protocols designed to make programs robust against host failures make use of the durable resource to return to a consistent state after reboot. We must be able to describe programs that have a recovery operation running after reboot. We introduce the following triple:

$$R \vdash \{P_V \mid P_D\}\,[\mathbb{C}]\,\{Q_V \mid Q_D\}$$

The notation $[\mathbb{C}]$ is used to identify a program with an associated recovery. The assertion R describes the durable resource after the recovery takes place.

We can now use the new judgements to verify the write-ahead logging `transfer` and its recovery. We use a simplified journaling file system as the durable resource in their implementation with the operations specified in figure 9.1. In this setting we specify the write-ahead logging `transfer` with the following triple:

$$S \vdash \begin{array}{c} \left\{ \dfrac{\mathtt{from} = f \wedge \mathtt{to} = t \wedge \mathtt{amount} = a}{\mathsf{Account}(f, v) * \mathsf{Account}(t, w) * \mathsf{nofile}(\mathtt{log})} \right\} \\ \mathtt{transfer}(\mathtt{from}, \mathtt{to}, \mathtt{amount}) \\ \left\{ \dfrac{\mathtt{from} = f \wedge \mathtt{to} = t \wedge \mathtt{amount} = a}{\mathsf{Account}(f, v - a) * \mathsf{Account}(t, w + a) * \mathsf{nofile}(\mathtt{log})} \right\} \end{array}$$

where the fault-condition S describes all the possible durable states if a host failure occurs:

$$\begin{aligned} S = {} & (\mathsf{Account}(f, v) * \mathsf{Account}(t, w) * \mathsf{nofile}(\mathtt{log})) \\ \vee {} & (\mathsf{Account}(f, v) * \mathsf{Account}(t, w) * \mathsf{file}(\mathtt{log}, [])) \\ \vee {} & (\mathsf{Account}(f, v) * \mathsf{Account}(t, w) * \mathsf{file}(\mathtt{log}, [(f, t, v, w)])) \\ \vee {} & (\mathsf{Account}(f, v - a) * \mathsf{Account}(t, w) * \mathsf{file}(\mathtt{log}, [(f, t, v, w)])) \\ \vee {} & (\mathsf{Account}(f, v - a) * \mathsf{Account}(t, w + a) * \mathsf{file}(\mathtt{log}, [(f, t, v, w)])) \\ \vee {} & (\mathsf{Account}(f, v - a) * \mathsf{Account}(t, w + a) * \mathsf{nofile}(\mathtt{log})) \end{aligned}$$

$S \vdash$

$$\left\{ \frac{\texttt{from} = f \land \texttt{to} = t \land \texttt{amount} = a}{\mathsf{Account}(f, v) * \mathsf{Account}(t, w) * \mathsf{nofile}(\texttt{log})} \right\}$$

$\texttt{fromAmount} := \texttt{getAmount}(\texttt{from});$

$\texttt{toAmount} := \texttt{getAmount}(\texttt{to});$

$$\left\{ \frac{\texttt{from} = f \land \texttt{to} = t \land \texttt{amount} = a \land \texttt{fromAmount} = v \land \texttt{toAmount} = w}{\mathsf{Account}(f, v) * \mathsf{Account}(t, w) * \mathsf{nofile}(\texttt{log})} \right\}$$

$[\texttt{create}(\texttt{log})];$

$$\left\{ \frac{\texttt{from} = f \land \texttt{to} = t \land \texttt{amount} = a \land \texttt{fromAmount} = v \land \texttt{toAmount} = w}{\mathsf{Account}(f, v) * \mathsf{Account}(t, w) * \mathsf{file}(\texttt{log}, [])} \right\}$$

$[\texttt{write}(\texttt{log}, (\texttt{from}, \texttt{to}, \texttt{fromAmount}, \texttt{toAmount}))];$

$$\left\{ \frac{\texttt{from} = f \land \texttt{to} = t \land \texttt{amount} = a \land \texttt{fromAmount} = v \land \texttt{toAmount} = w}{\mathsf{Account}(f, v) * \mathsf{Account}(t, w) * \mathsf{file}(\texttt{log}, [(f, t, v, w)])} \right\}$$

$\texttt{setAmount}(\texttt{from}, \texttt{fromAmount} - \texttt{amount});$

$$\left\{ \frac{\texttt{from} = f \land \texttt{to} = t \land \texttt{amount} = a \land \texttt{fromAmount} = v \land \texttt{toAmount} = w}{\mathsf{Account}(f, v - a) * \mathsf{Account}(t, w) * \mathsf{file}(\texttt{log}, [(f, t, v, w)])} \right\}$$

$\texttt{setAmount}(\texttt{to}, \texttt{toAmount} + \texttt{amount});$

$$\left\{ \frac{\texttt{from} = f \land \texttt{to} = t \land \texttt{amount} = a \land \texttt{fromAmount} = v \land \texttt{toAmount} = w}{\mathsf{Account}(f, v - a) * \mathsf{Account}(t, w - a) * \mathsf{file}(\texttt{log}, [(f, t, v, w)])} \right\}$$

$[\texttt{delete}(\texttt{log})];$

$$\left\{ \frac{\texttt{from} = f \land \texttt{to} = t \land \texttt{amount} = a}{\mathsf{Account}(f, v - a) * \mathsf{Account}(t, w + a) * \mathsf{nofile}(\texttt{log})} \right\}$$

Figure 9.2.: Proof of transfer operation using write-ahead logging.

The proof that the implementation satisfies the specification is shown in figure 9.2. If there is a host failure, the current specification of `transfer` only guarantees that the durable resource satisfies S. This includes the case where money is lost. This is undesirable. What we want is a guarantee that the operation is atomic. In order to add this guarantee, we must combine reasoning about the operation with reasoning about its recovery to establish that undesirable states are fixed after recovery. We formalise the combination of an operation and its recovery in order to provide robustness guarantees against host failures in the *recovery abstraction* rule:

$$\frac{\mathbb{C}_R \textbf{ recovers } \mathbb{C} \qquad S \vdash \{P_V \mid P_D\}\,\mathbb{C}\,\{Q_V \mid Q_D\} \qquad S \vdash \{\mathsf{true} \mid S\}\,\mathbb{C}_R\,\{\mathsf{true} \mid R\}}{R \vdash \{P_V \mid P_D\}\,[\mathbb{C}]\,\{Q_V \mid Q_D\}}$$

When implementing a new operation, we use the *recovery abstraction* rule to establish the fault-condition R we wish to expose to the client. In the second premiss, we must first derive what the durable resource S will be immediately after a host-failure. In the third premiss, we establish that given S, the associated recovery operation will change the durable resource to the desired R. Note that because the recovery $\mathbb{C}_R$ runs immediately after the host failure, there is no volatile resource in the precondition. Furthermore, we require the fault-condition of the recovery to be the same as the resource that is being recovered, since the recovery operation itself may fail due to a host-failure; i.e. recovery operations must be able to recover themselves.

We allow recovery abstraction to derive any fault-condition that is established by the recovery operation. If that fault-condition is a disjunction between the durable pre- and postconditions, $P_D \lor$

Q_D, then the operation $[\mathbb{C}]$ appears to be atomic with respect to host failures. Either the operation's (durable) resource updates completely, or not at all. No intermediate states are visible to the client.

In order for `transfer` to be atomic, according to the recovery abstraction rule, `transferRecovery` must satisfy the following specification:

$$S \vdash \left\{\frac{\mathsf{true}}{S}\right\} \texttt{transferRecovery}() \left\{\frac{\mathsf{true}}{(\mathsf{Account}(f, v) * \mathsf{Account}(t, w)) \vee (\mathsf{Account}(f, v - a) * \mathsf{Account}(t, w + a)) * \mathsf{nofile}(\texttt{log})}\right\}$$

The proof that the implementation satisfies this specification is given in figure 9.3. By applying the abstraction recovery rule we get the following specification for `transfer` which guarantees atomicity in case of a host-failure:

$$R \vdash \left\{\frac{\texttt{from} = f \wedge \texttt{to} = t \wedge \texttt{amount} = a}{\mathsf{Account}(f, v) * \mathsf{Account}(t, w) * \mathsf{nofile}(\texttt{log})}\right\} [\texttt{transfer}(\texttt{from}, \texttt{to}, \texttt{amount})] \left\{\frac{\texttt{from} = f \wedge \texttt{to} = t \wedge \texttt{amount} = a}{\mathsf{Account}(f, v - a) * \mathsf{Account}(t, w + a) * \mathsf{nofile}(\texttt{log})}\right\}$$

where the fault-condition R describes the recovered durable state:

$$R = (\mathsf{Account}(f, v) * \mathsf{Account}(t, w)) \vee (\mathsf{Account}(f, v - a) * \mathsf{Account}(t, w + a)) * \mathsf{nofile}(\texttt{log})$$

With this example, we have seen how to guarantee atomicity by logging the information required to undo operations. Advanced WAL protocols also store information allowing to redo operations and use concurrency control. We do not go into depth on how to enforce concurrency control in our examples other than the example shown in section 9.5. It follows the common techniques used in concurrent separation logic.[1] However, in section 9.7 we show ARIES, an advanced algorithm that uses write-ahead logging. A different style of write-ahead logging is used by file systems called journaling [79], which we discuss in section 9.3.

[1] For an introduction to concurrent separation logic see [32].

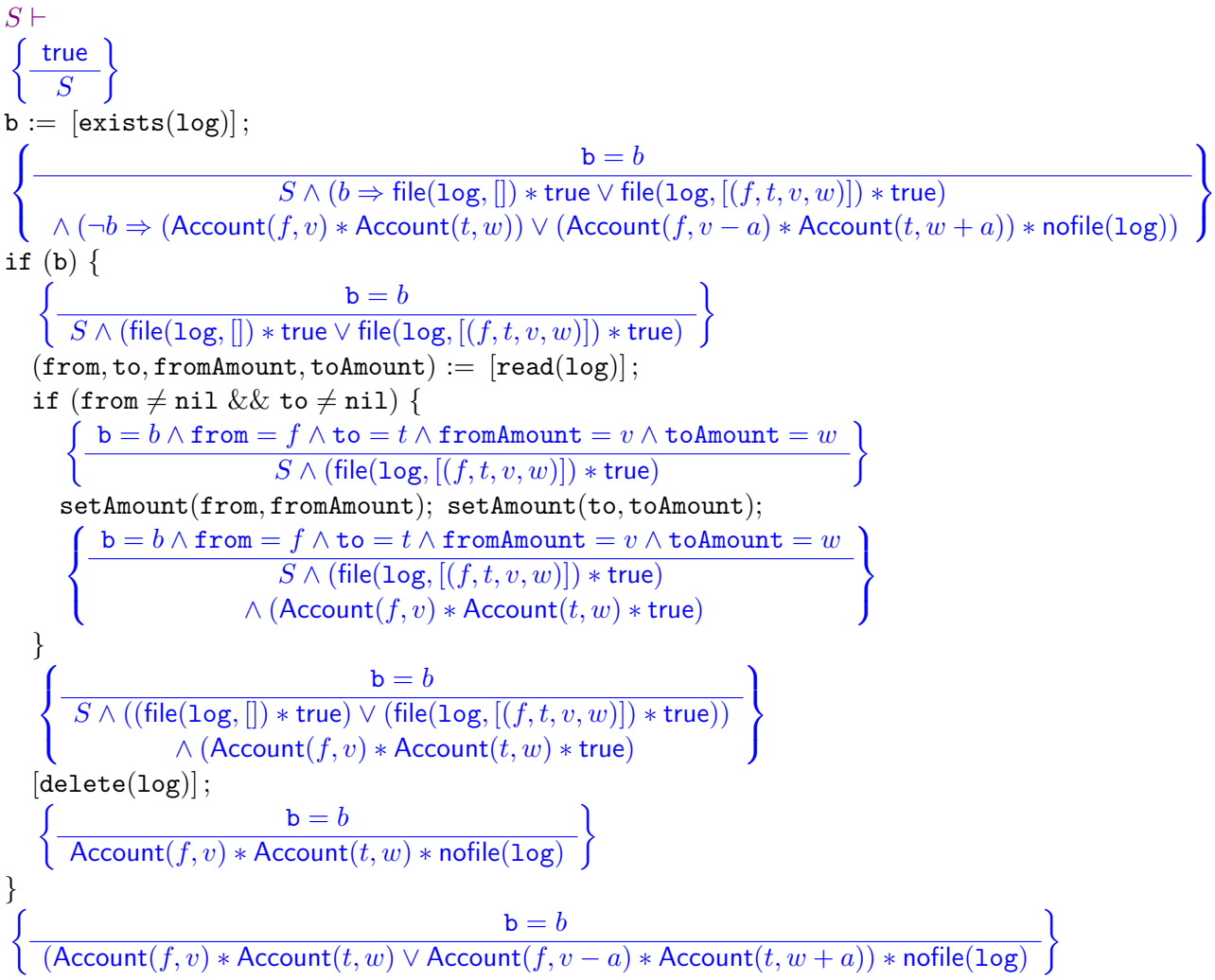

Figure 9.3.: Proof that the transfer recovery operation guarantees atomicity.

9.3. Journaling File Systems

Note that the recovery abstraction rule, introduced in section 9.2.3, does not constrain the fault condition established by the recovery. In the fault-tolerant bank transfer example, we have used this rule to show that `transferRecovery` guarantees atomicity. We could have limited recovery abstraction to just this atomic case. However, this is not the general case for fault tolerance guarantees.

Journaling file systems [79] are a prime example. Such file systems employ write ahead logging techniques so that file system consistency may be recovered if an operation is interrupted by a host failure. For example, consider the operation of appending data to a file, and its specification as given [72], where we simplify slightly by removing the size argument and assuming the write is appending to the file.

$$\{\mathsf{fd}(\mathtt{fd}, \iota) * \mathsf{buf}(\mathtt{buf}, y) \mid \mathsf{file}(\iota, b)\}\, \mathtt{write}(\mathtt{fd}, \mathtt{buf})\, \{\mathsf{fd}(\mathtt{fd}, \iota) * \mathsf{buf}(\mathtt{buf}, y) \mid \mathsf{file}(\iota, b \otimes y)\}$$

The specification says that we extend the file with inode ι, associated with the file descriptor `fd`, with the contents of the supplied memory buffer y. For the purposes of this discussion we distinguish between durable (the file contents) and volatile (file descriptor and memory buffer) resources.

The implementation of this operation typically involves: i) updating the inode's metadata with the new file size, ii) allocating new space, and iii) writing the appended contents. Journaling file systems make different choices regarding how much information to log and consequently the fault tolerance guarantees. One choice is to log all steps, referred to as physical journaling, in which case the operation is atomic with respect to host failures and we can extend its specification to the following:

$$\mathsf{file}(\iota, b) \vee \mathsf{file}(\iota, b \otimes y) \vdash \begin{array}{c} \left\{ \mathsf{fd}(\mathtt{fd}, \iota) * \mathsf{buf}(\mathtt{buf}, y) \mid \mathsf{file}(\iota, b) \right\} \\ [\mathtt{write}(\mathtt{fd}, \mathtt{buf})] \\ \left\{ \mathsf{fd}(\mathtt{fd}, \iota) * \mathsf{buf}(\mathtt{buf}, y) \mid \mathsf{file}(\iota, b \otimes y) \right\} \end{array}$$

However, physical journaling has a significant performance overhead since every update must be committed twice. Alternatively, file system perform logical journaling, where only metadata changes are logged. In our example, this means the third step is not logged. The specification in this case is the following:

$$\mathsf{file}(\iota, b) \vee \exists z.\, \mathsf{sizeof}(y) = \mathsf{sizeof}(z) \wedge \mathsf{file}(\iota, b \otimes z) \vdash \begin{array}{c} \left\{ \mathsf{fd}(\mathtt{fd}, \iota) * \mathsf{buf}(\mathtt{buf}, y) \mid \mathsf{file}(\iota, b) \right\} \\ [\mathtt{write}(\mathtt{fd}, \mathtt{buf})] \\ \left\{ \mathsf{fd}(\mathtt{fd}, \iota) * \mathsf{buf}(\mathtt{buf}, y) \mid \mathsf{file}(\iota, b \otimes y) \right\} \end{array}$$

As the first and second steps are logged, in the fault-condition we know that file may be extended to the correct size, but if it is, the new data is effectively garbage. If we restricted recovery abstraction to the atomic case, we would not be able to derive useful specifications for such strategies.

9.4. FTCSL Program Logic

Until now, we have only seen how to reason about sequential programs. For concurrent programs, we use resource invariants, in the style of concurrent separation logic [76], that are updated by primitive

atomic operations. Here primitive atomic is used to mean that the operation takes effect at a single, discrete instant in time, and that it is atomic with respect to host failures.

The general judgement that enables us to reason about host failing concurrent programs is:

$$\boxed{J_V \mid J_D}; S \vdash \{P_V \mid P_D\}\, \mathbb{C}\, \{Q_V \mid Q_D\}$$

Here, $P_V \mid P_D$ and $Q_V \mid Q_D$ are pre- and postconditions as usual and describe the volatile and durable resource. S is a durable assertion, which we refer to as the *fault-condition*, describing the durable resource of the program $\mathbb{C}$ after a host failure and possible recovery. The interpretation of these triples is partial *resource fault avoiding* and *host failing*. Starting from an initial state satisfying the precondition $P_V \mid P_D$, it is safe to execute $\mathbb{C}$ without causing a resource fault. If no host failure occurs and $\mathbb{C}$ terminates, the resulting state will satisfy the postcondition $Q_V \mid Q_D$. The shared resource invariant $J_V \mid J_D$ is maintained throughout the execution of $\mathbb{C}$. If a host failure occurs, all volatile resource is lost and the durable state will (after possible recoveries) satisfy $S * J_D$.

We give an overview of the key proof rules of Fault-tolerant Concurrent Separation Logic (FTCSL) in figure 9.4. Here we do not formally define the syntax of our assertions, although we describe the semantics in section 9.6. In general, volatile and durable assertions can be parameterised by any separation algebra.

The *sequence rule* allows us to combine two programs in sequence as long as they have the same fault-condition and resource invariant. Typically, when the fault-conditions differ, we can weaken them using the *consequence rule*, which adds fault-condition weakening to the standard consequence rule of Hoare logic. The *frame rule*, as in separation logic, allows us to extend the pre- and postconditions with the same unmodified resource $R_V * R_D$. However, here the durable part, R_D, is also added to the fault-condition.

The *atomic rule* allows us to use the resource invariant $J_V \mid J_D$ using a primitive atomic operation. Since the operation executes in a single, discrete, moment in time, we can think of the operation temporarily owning the resources $J_V \mid J_D$. However, they must be reestablished at the end. This guarantees that every primitive atomic operation maintains the resource invariant. Note that the rule enforces atomicity with respect to host failures. The *share rule* allows us to use local resources to extend the shared resource invariant.

The *parallel rule*, in terms of pre- and postconditions is as in concurrent separation logic. However, the fault-condition describes the possible durable resources that may result from a host failure while running $\mathbb{C}_1$ and $\mathbb{C}_2$ in parallel. In particular, a host-failure may occur while both $\mathbb{C}_1$ and $\mathbb{C}_2$ are running, in which case the fault-condition is $S_1 * S_2$, or when either one of $\mathbb{C}_1$, $\mathbb{C}_2$ has finished, in which case the fault-condition is $S_1 * Q_{D2}$ and $S_2 * Q_{D1}$ respectively.

Finally, the *recovery abstraction rule* allows us to prove that a recovery operation $\mathbb{C}_R$ establishes the fault-condition R we wish to expose to the client. The first premiss requires operation $\mathbb{C}_R$ to be the recovery of $\mathbb{C}$, i.e. it is executed on reboot after a host failure during execution of $\mathbb{C}$. The second premiss guarantees that in such case, the durable resources satisfy S and the shared resource invariant satisfies J_D, while the volatile state is lost after a host failure. The third premiss, takes the resource after the reboot and runs the recovery operation in order to establish R. Note that J_D is an invariant, as there can be potentially parallel recovery operations accessing it using primitive atomic operations. While the recovery operation $\mathbb{C}_R$ is running, there can be any number of host failures, which restart

FTCSLSeq

$$\frac{\boxed{J_V \mid J_D};S \vdash \{P_V \mid P_D\}\,\mathbb{C}_1\,\{R_V \mid R_D\} \qquad \boxed{J_V \mid J_D};S \vdash \{R_V \mid R_D\}\,\mathbb{C}_2\,\{Q_V \mid Q_D\}}{\boxed{J_V \mid J_D};S \vdash \{P_V \mid P_D\}\,\mathbb{C}_1;\mathbb{C}_2\,\{Q_V \mid Q_D\}}$$

FTCSLCons

$$\frac{P_V \mid P_D \Rightarrow P'_V \mid P'_D \qquad \boxed{J_V \mid J_D};S' \vdash \{P'_V \mid P'_D\}\,\mathbb{C}\,\{Q'_V \mid Q'_D\} \qquad Q'_V \mid Q'_D \Rightarrow Q_V \mid Q_D \qquad S' \Rightarrow S}{\boxed{J_V \mid J_D};S \vdash \{P_V \mid P_D\}\,\mathbb{C}\,\{Q_V \mid Q_D\}}$$

FTCSLFrame

$$\frac{\boxed{J_V \mid J_D};S \vdash \{P_V \mid P_D\}\,\mathbb{C}\,\{Q_V \mid Q_D\}}{\boxed{J_V \mid J_D};S * R_D \vdash \{P_V * R_V \mid P_D * R_D\}\,\mathbb{C}\,\{Q_V * R_V \mid Q_D * R_D\}}$$

FTCSLAtomic

$$\frac{\boxed{\mathsf{true} \mid \mathsf{true}};P_D * J_D \vee Q_D * J_D \vdash \{P_V * J_V \mid P_D * J_D\}\,\mathbb{C}\,\{Q_V * J_V \mid Q_D * J_D\}}{\boxed{J_V \mid J_D};P_D \vee Q_D \vdash \{P_V \mid P_D\}\,\langle\mathbb{C}\rangle\,\{Q_V \mid Q_D\}}$$

FTCSLShare

$$\frac{\boxed{J_V * R_V \mid J_D * R_D};S \vdash \{P_V \mid P_D\}\,\mathbb{C}\,\{Q_V \mid Q_D\}}{\boxed{J_V \mid J_D};S * R_D \vdash \{P_V * R_V \mid P_D * R_D\}\,\mathbb{C}\,\{Q_V * R_V \mid Q_D * R_D\}}$$

FTCSLParallel

$$\frac{\boxed{J_V \mid J_D};S_1 \vdash \{P_{V1} \mid P_{D1}\}\,\mathbb{C}_1\,\{Q_{V1} \mid Q_{D1}\} \qquad \boxed{J_V \mid J_D};S_2 \vdash \{P_{V2} \mid P_{D2}\}\,\mathbb{C}_2\,\{Q_{V2} \mid Q_{D2}\}}{\boxed{J_V \mid J_D};(S_1 * S_2) \vee (S_1 * Q_{D2}) \vee (Q_{D1} * S_2) \vdash \left\{\begin{array}{c} P_{V1} * P_{V2} \mid P_{D1} * P_{D2} \\ \mathbb{C}_1 \parallel \mathbb{C}_2 \\ Q_{V1} * Q_{V2} \mid Q_{D1} * Q_{D2} \end{array}\right\}}$$

FTCSLRecoveryAbstract

$$\frac{\mathbb{C}_R \textbf{ recovers } \mathbb{C} \qquad \boxed{J_V \mid J_D};S \vdash \{P_V \mid P_D\}\,\mathbb{C}\,\{Q_V \mid Q_D\} \qquad \boxed{\mathsf{true} \mid J_D};S \vdash \{\mathsf{true} \mid S\}\,\mathbb{C}_R\,\{\mathsf{true} \mid R\}}{\boxed{J_V \mid J_D};R \vdash \{P_V \mid P_D\}\,[\mathbb{C}]\,\{Q_V \mid Q_D\}}$$

Figure 9.4.: Selected proof rules of FTCSL.

the recovery. This means that the recovery operation must be able to recover from itself. We allow recovery abstraction to derive any fault-condition that is established by the recovery operation. If the fault-condition is a disjunction between the durable pre- and post-conditions, $P_V \vee Q_D$, then the operation $[\mathbb{C}]$ appears to be atomic with respect to host failures.

9.4.1. Unsound Rules

We have explored including a rule for deriving a FTCSL triple from a standard CSL triple. This gives rise to two questions: how do we distinguish volatile and durable resource, and what will the added fault-condition be? Consider the following candidate rule:

$$\frac{\{P_V * P_D\}\,\alpha\,\{Q_V * Q_D\}}{P_D \vee Q_D \vdash \{P_V \mid Q_D\}\,[\alpha]\,\{Q_V \mid Q_D\}}$$

where we assume that α is a primitive operation (the premiss is an axiom). Such a rule would be useful to easily import existing specifications, e.g. those given with separation logics for file systems [47, 72], into FTCSL.

There two issues. First, even though the fault-condition is sensible, it is overly restrictive for the same reasons discussed previously on file appends in journaling file systems. Second, the choice of what $*$-junct to assign to durable resource seems arbitrary. The solution is to require the CSL of the premiss to logically distinguish between volatile and durable resources, yet this introduces extra requirements typically not being met. Even though such a rule would be sound, we do not include it in FTCSL for the aforementioned reasons.

Assume a CSL which does distinguish between volatile and durable resource, and consider the following tempting, albeit unsound, rule:

$$\frac{\{P_V \mid \mathsf{true}\}\,\mathbb{C}\,\{Q_V \mid \mathsf{true}\}}{\mathsf{true} \vdash \{P_V \mid \mathsf{true}\}\,\mathbb{C}\,\{Q_V \mid \mathsf{true}\}}$$

The intuition of this rule is that if we know a program does not use durable resources, then we can infer its fault-condition to be empty. However, the fact that the pre- and post-condition durable resource is empty does not mean the program uses no durable resource at all. For example, $\mathbb{C}$ can allocate and subsequently deallocate something durable, in which case the fault-condition should reflect this. This is an instance of the ABA problem. The bottom line is that an empty fault-condition does not generally mean durable resource is not used.

9.5. Example: Concurrent Bank Transfer

Consider two threads that both perform a transfer operation from account f to account t as shown in section 9.2. The parallel rule requires that each operation acts on disjoint resources in the precondition. Since both threads update the same accounts, we synchronise their use with the atomic blocks denoted

by $\langle \_ \rangle$. A possible specification for the program is the following:

$$\boxed{\mathsf{true} \mid \mathsf{true}}; \exists v, w.\, \mathsf{Account}(f, v) * \mathsf{Account}(t, w) * \mathsf{nofile}(\mathtt{log}) \vdash$$
$$\left\{ \begin{array}{l} \mathtt{from} = f \wedge \mathtt{to} = t \wedge \mathtt{amount} = a \wedge \mathtt{amount2} = b \\ \hline \exists v, w.\, \mathsf{Account}(f, v) * \mathsf{Account}(t, w) * \mathsf{nofile}(\mathtt{log}) \end{array} \right\}$$
$$\langle [\mathtt{transfer}(\mathtt{from}, \mathtt{to}, \mathtt{amount})] \rangle;\ \|\ \langle [\mathtt{transfer}(\mathtt{from}, \mathtt{to}, \mathtt{amount2})] \rangle;$$
$$\left\{ \begin{array}{l} \mathtt{from} = f \wedge \mathtt{to} = t \wedge \mathtt{amount} = a \wedge \mathtt{amount2} = b \\ \hline \exists v, w.\, \mathsf{Account}(f, v) * \mathsf{Account}(t, w) * \mathsf{nofile}(\mathtt{log}) \end{array} \right\}$$

A sketch proof of this specification is given in figure 9.5. We first move the shared resources of the two `transfer` operations to the shared invariant (share rule). We then prove each thread independently by making use of the atomic rule to gain temporary access to the shared invariant within the atomic block, and reuse the specification given in section 9.2.3. It is possible to get stronger postconditions, that maintain exact information about the amounts of each bank account, using complementary approaches such as Owicki-Gries [75] or other forms of resource ownership [32]. The sequential examples in this dissertation can be adapted to concurrent applications using these techniques.

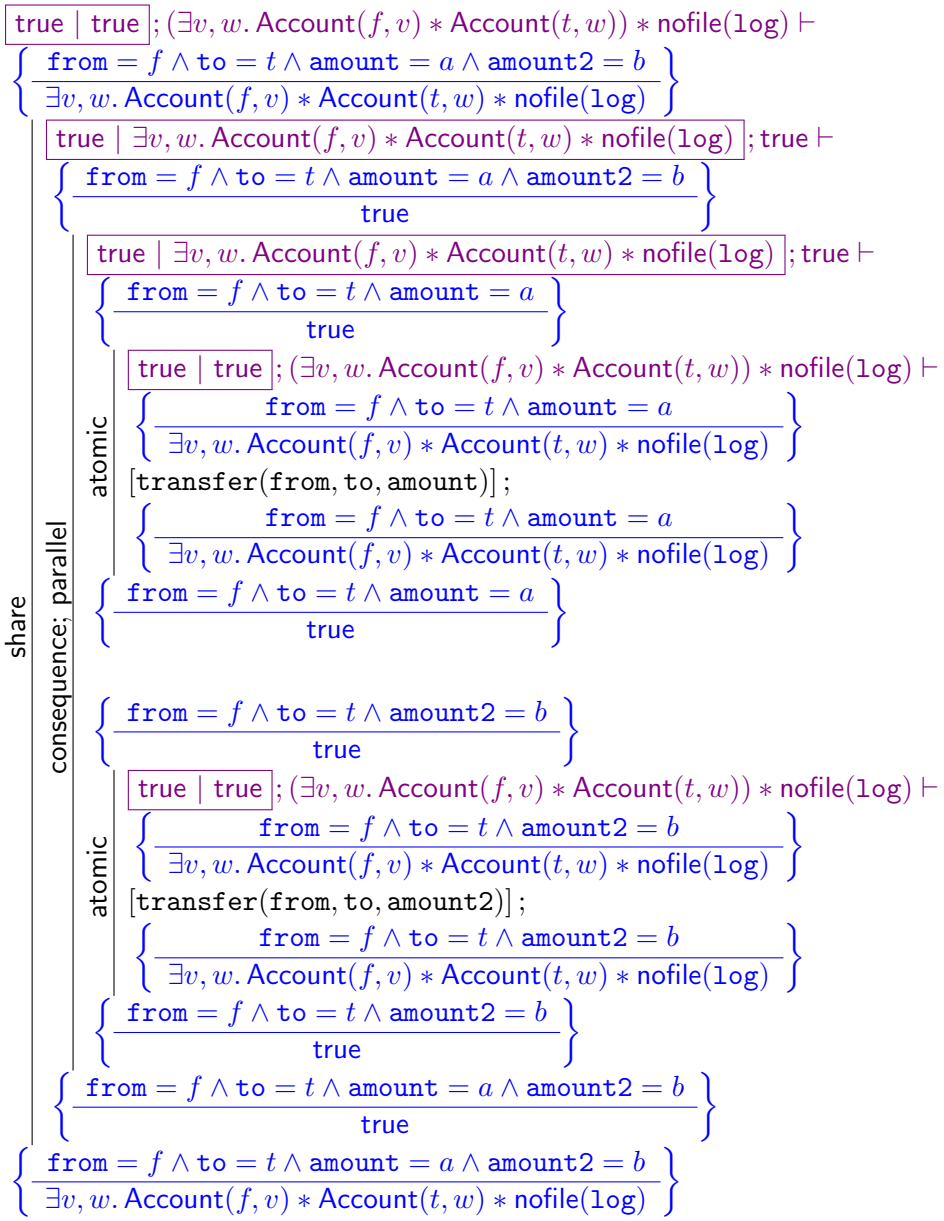

Figure 9.5.: Sketch proof of two concurrent transfers over the same accounts.

9.6. Semantics and Soundness

We give a brief overview of the semantics of our reasoning and the intuitions behind its soundness. A detailed account is given in appendix C.

9.6.1. Fault-tolerant Views

We define a general fault-tolerant reasoning framework using Hoare triples with fault-conditions in the style of the Views framework [35]. Pre- and postcondition assertions are modelled as pairs of *volatile* and *durable views* (commutative monoids). Fault-condition assertions are modelled as durable views [2]. Volatile and durable views provide partial knowledge reified to concrete volatile and durable program states respectively. Concrete volatile states include the distinguished host-failed state $\lightning$. The semantic interpretation of a primitive operation is given as a state transformer function from concrete states to sets of concrete states.

To prove soundness, we encode our Fault-tolerant Views (FTV) framework into Views [35]. A judgement[3] $s \vdash \{(p_v, p_d)\}\, \mathbb{C}\, \{(q_v, q_d)\}$, where s, p_d, q_d are durable views and p_v, q_v are volatile views is encoded as the Views judgement: $\{(p_v, q_d)\}\, \mathbb{C}\, \big\{(q_v, q_d) \vee (\lightning, s)\big\}$, where volatile views are extended to include $\lightning$ and $\vee$ is disjunction of views. For the general abstraction recovery rule we encode $[\mathbb{C}]$ as a program which can test for host failures, beginning with $\mathbb{C}$ and followed by as many iterations of the recovery $\mathbb{C}_R$ as required in case of a host failure.

We require the following properties for a sound instance of the framework:

Host failure: For each primitive operation, its interpretation function must transform non host-failed states to states including a host-failed state. This guarantees that each operation can be abruptly interrupted by a host failure.

Host failure propagation: For each primitive operation, its interpretation function must leave all host-failed states intact. That is, when the state says there is a host failure, it stays a host failure.

Axiom soundness: The axiom soundness property (property [G] of Views [35]).

The first two are required to justify the general FTV rules, while the final property establishes soundness of the Views encoding itself. When all the parameters are instantiated and the above properties established then the instantiation of the framework is sound.

9.6.2. Fault-tolerant Concurrent Separation Logic

We justify the soundness of FTCSL by an encoding into the Fault-tolerant Views framework discussed earlier. The encoding is similar to the concurrent separation logic encoding into Views. We instantiate volatile and durable views as pairs of local views and shared invariants.

The FTCSL judgement $\boxed{(j_v, j_d)}; s \vdash \{(p_v, p_d)\}\, \mathbb{C}\, \{(q_v, q_d)\}$ is encoded as:

$$s \vdash \{((p_v, j_v), (p_d, j_d))\}\, \mathbb{C}\, \{((q_v, j_v), (q_d, j_d))\}$$

The proof rules in figure 9.4 are justified by soundness of the encoding and simple application of FTV

[2] We use "Views" to refer to the Views framework of Dinsdale-Young et al. [35], and "views" to refer to the monoid structures used within it.

[3] Note that judgements, such as those in figure 9.4, using assertions (capital P, Q, S) are equivalent to judgements using views (models of assertions, little p, q, s).

proof rules. Soundness of the encoding is established by proving the properties stated in section 9.6.1.

Theorem 6 (FTCSL Soundness). *If the judgement* $\boxed{J_V \mid J_D}; S \vdash \{P_V \mid P_D\}\,\mathbb{C}\,\{Q_V \mid Q_D\}$ *is derivable in the program logic, then if we run the program* $\mathbb{C}$ *from state satisfying* $P_V * J_V \mid P_D * J_D$*, then* $\mathbb{C}$ *will either not terminate, or terminate in state satisfying* $Q_V * J_V \mid Q_D * J_D$*, or a host failure will occur destroying any volatile state and the remaining durable state (after potential recoveries) will satisfy* $S * J_D$*. The resource invariant* $J_V \mid J_D$ *holds throughout the execution of* $\mathbb{C}$*.*

9.7. Case Study: ARIES

In section 9.2 we saw an example of a very simple transaction and its associated recovery operation employing write-ahead logging. Relational databases support concurrent execution of complex transactions following the established ACID (Atomicity, Consistency, Isolation and Durability) set of properties. ARIES (Algorithms for Recovery and Isolation Exploiting Semantics) [69], is a collection of algorithms involving, concurrent execution, write-ahead-logging and failure recovery of transactions, that is widely-used to establish ACID properties.

It is beyond the scope of this dissertation to verify that the full set of ARIES algorithms guarantees ACID properties. Instead, we focus on a stylised version of the recovery algorithm of ARIES proving that: a) it is idempotent with respect to host failures, b) after recovery, all transactions recorded in the write-ahead log have either been completed, or were rolled-back.

Transactions update database records stored in durable memory, which for the purposes of this discussion we assume to be a single file in a file system. To increase performance, the database file is divided into fixed-size blocks, called pages, containing multiple records. Thus input/output to the database file, instead of records, is in terms of pages, which are also typically cached in volatile memory. A single transaction may update multiple pages. In the event of a host failure, there may be transactions that have not yet completed, or have completed but their updated pages have not yet been written back to the database file.

ARIES employs write-ahead logging for page updates performed by transactions. The log is stored on a durable fault-tolerant medium. The recovery uses the logged information in a sequence of three phases. First, the *analysis phase*, scans the log to determine the (volatile) state, of any active transactions (committed or not), at the point of host failure. Next, the *redo phase*, scans the log and redos each logged page update, unless the associated page in the database file is already updated. Finally, the *undo phase*, scans the log and undos each page update for each uncommitted transaction. To cope with a possible host failure during the ARIES recovery, each undo action is logged beforehand. Thus, in the event of a host failure the undo actions will be retried as part of the redo phase.

In figure 9.6, we define the log and database model and describe the predicates we use in our specifications and proofs. We model the database state, db, as a set of pages, where each page comprises the page identifier, the log sequence number (defined later) of the last update performed on the page, and the page data. The log, lg, is structured as a sequence of log records, ordered by a *log sequence number*, $lsn \in \mathbb{N}$, each of which records a particular action performed by a transaction. The ordering follows the order in which transaction actions are performed on the database. The logged action, $U[tid, pid, op]$, records that the transaction identifier tid, performs the update $op : \textsc{Data} \to \textsc{Data}$ on the page identified by pid. We use op^{-1} to denote the operation undoing the update op. $B[tid]$,

records the start of a new transaction with identifier *tid*, and $C[tid]$, records that the transaction with id *tid* is committed. The information from the above actions is used to construct two auxiliary structures used by the recovery to determine the state of transactions and pages at the point of a host failure. The *transaction table* (TT), records the status of all active transactions (e.g. updating, committed) and the latest log sequence number associated with the transaction. The *dirty page table* (DPT), records which pages are modified but yet unwritten to the database together with the first log sequence number of the action that caused the first modification to each page. To avoid the cost of scanning the entire log, implementations regularly log snapshots of the TT and DPT in checkpoints, $CHK[tt, dpt]$. For simplicity, we assume the log contains exactly one checkpoint.

Let $lsn, tid, pid \in \mathbb{N}$, where we use *lsn* for log sequence numbers, *tid* for transaction identifiers, *pid* for page identifiers, *d* for page data and *op* for page-update operations. Let $\varnothing$ be an empty list.

Model:

Database state	$db \subseteq \mathbb{N} \times \mathbb{N} \times \textsc{Data}$, triples of pid, lsn, d
Logged actions	$act ::= U[tid, pid, op] \mid B[tid] \mid C[tid] \mid CHK[tt, dpt]$
Log state	$lg ::= \varnothing \mid (lsn, act) \mid lg \otimes lg$
Transaction table	$tt \subseteq \mathbb{N} \times \mathbb{N} \times \{C, U\}$, triples of lsn, pid and transaction status
Dirty page table	$dpt \subseteq \mathbb{N} \times \mathbb{N}$, tuples of pid, lsn

Predicates:

$\mathsf{log}(lg)$	the state of the log is given by lg (abstract predicate)
$\mathsf{db\_state}(db)$	the state of the database is given by db (abstract predicate)
$\mathsf{set}(\mathtt{x}, s)$	the set s identified by program variable `x` (abstract predicate)
$\mathsf{log\_tt}(lg, tt)$	log lg produces the TT entries in tt
$\mathsf{log\_dpt}(lg, dpt)$	log lg produces the DPT entries in dpt
$\mathsf{log\_rl}(lg, dpt, ops)$	given log lg and DPT dpt the list of redo updates is ops
$\mathsf{ul\_undo}(lg, tt, ops)$	given log lg and TT tt the list of undo updates is ops
$\mathsf{log\_undos}(ops, lg)$	given list of undos ops the additional log records are lg
$\mathsf{db\_acts}(db, ops, db')$	given the list of updates ops, the database db is updated to db'
$\mathsf{recovery\_log}(lg, lg')$	given log lg log records added by recovery are lg'
$\mathsf{recovery\_db}(db, lg, db')$	given database db and log lg the recovered database state is db'

Axioms:

$$\mathsf{log}(lg \otimes lg') \iff \mathsf{log\_bseg}(lg) \otimes \mathsf{log\_fseg}(lg')$$

Figure 9.6.: Abstract model of the database and ARIES log, and predicates.

```
function aries_recovery() {
  //ANALYSIS PHASE: restore dirty page table, transaction table
  //and undo list at point of host failure.
  tt, dpt := aries_analyse();
  //REDO PHASE: repeat actions to restore database state at host failure.
  aries_redo(dpt);
  //UNDO PHASE: Undo actions of uncommitted transactions.
  aries_undo(tt);
}
```

Figure 9.7.: ARIES recovery: high level structure.

Predicate definitions

We give the following definitions for the concrete predicates of figure 9.6:

$$
\begin{array}{c}
\mathsf{log\_tt}(lg, tt) \triangleq \\
(lg = \varnothing \wedge tt = \emptyset) \\
\vee \left(\begin{array}{c} \exists lsn, act, a, tid, lg', tt'.\ (lg = (lsn, act) \otimes lg') \\ \wedge \left(\left(\begin{array}{c} ((act = U[tid, -] \wedge a = U) \vee (act = C[tid] \wedge a = C)) \\ \wedge\ tt = (tid, lsn, a) \oplus tt' \end{array} \right) \right) \\ \vee\ ((act = CHK[-] \vee act = B[-]) \wedge tt' = tt) \\ \wedge\ \mathsf{log\_tt}(lg', tt') \end{array} \right)
\end{array}
$$

$$
\begin{array}{c}
\mathsf{log\_dpt}(lg, dpt) \triangleq \\
(lg = \varnothing \wedge dpt = \emptyset) \\
\vee \left(\begin{array}{c} \exists lsn, pid, lg', dpt'.\ (lg = (lsn, U[-, pid]) \otimes lg') \\ \wedge \left(\begin{array}{c} \big(pid \notin dpt_{\downarrow 1} \wedge dpt = \{(pid, lsn)\} \cup dpt \wedge \mathsf{log\_dpt}(lg', dpt')\big) \\ \vee\ \big(pid \in dpt_{\downarrow 1} \wedge \mathsf{log\_dpt}(lg', dpt)\big) \end{array} \right) \end{array} \right) \\
\vee \left(\begin{array}{c} \exists act, lg'.\ (lg = (-, act) \otimes lg') \wedge \mathsf{log\_dpt}(lg', dpt) \\ \wedge\ (act = CHK[-] \vee act = C[-] \vee act = B[-]) \end{array} \right)
\end{array}
$$

$$
\begin{array}{c}
\mathsf{log\_rl}(lg, dpt, ops) \triangleq \\
(lg = \varnothing \wedge ops = \varnothing) \\
\vee \left(\begin{array}{c} \exists lsn, lsn', tid, pid, lg', op, ops'.\ (lg = (lsn, U[tid, pid], op) \otimes lg') \\ \wedge \left(\begin{array}{c} (pid, lsn') \in dpt \wedge lsn \geq lsn' \wedge (ops = (tid, pid, op) \otimes ops') \\ \wedge\ \mathsf{log\_rl}(lg', dpt, ops') \end{array} \right) \\ \vee \left(\begin{array}{c} (pid) \notin dpt_{\downarrow 1} \wedge \mathsf{log\_rl}(lg', dpt, ops) \end{array} \right) \end{array} \right) \\
\vee \left(\begin{array}{c} \exists act, lg'.\ (act = CHK[-] \vee act = C[-] \vee act = B[-]) \\ \wedge\ (lg = (-, act) \otimes lg') \wedge \mathsf{log\_rl}(lg', dpt, ops) \end{array} \right)
\end{array}
$$

$$
\begin{array}{c}
\mathsf{ul\_undo}(lg, tt, ops) \triangleq \\
(lg = \varnothing \wedge ul = \varnothing) \\
\vee \left(\begin{array}{c} \exists lg', lsn, tid, pid, op, ops'.\ (lg = lg' \otimes (lsn, U[tid, pid, op])) \\ \wedge \left(\begin{array}{c} \left((tid, -, U) \in tt \wedge \big(ops = (tid, pid, op^{-1}) \otimes ops'\big) \wedge \mathsf{ul\_undo}(lg', tt, ops') \right) \\ \vee\ ((tid, -, U) \notin tt \wedge \mathsf{ul\_undo}(lg', tt, ops)) \end{array} \right) \end{array} \right) \\
\vee \left(\begin{array}{c} \exists lg', act.\ (act = CHK[-] \vee act = C[-] \vee act = B[-]) \\ \wedge\ (lg = lg' \otimes (-, act)) \wedge \mathsf{ul\_undo}(lg', tt, ops) \end{array} \right)
\end{array}
$$

$$
\begin{array}{c}
\mathsf{log\_undos}(ops, lg) \triangleq \\
(ops = \varnothing \wedge rl = \varnothing) \\
\vee \left(\begin{array}{c} \exists lsn, tid, pid, ops', lg'.\ ops = (tid, pid, op) \otimes ops' \\ \wedge\ lg = (lsn, U[tid, pid, op]) \otimes lg' \wedge \mathsf{log\_undos}(ops', lg') \end{array} \right)
\end{array}
$$

$$
\begin{array}{c}
\mathsf{db\_acts}(db, ops, db') \triangleq \\
(ops = \varnothing \wedge db = db') \\
\vee \left(\begin{array}{c} \exists lsn, tid, pid, ops'.\ ops = (tid, pid, op) \otimes ops' \\ \wedge ((lsn, pid, d) \in db \wedge db' = db \setminus \{(lsn, pid, d)\} \cup \{lsn, pid, op(d)\}) \\ \vee ((lsn, pid, d) \notin db \wedge \mathsf{db\_acts}(db, ops', db')) \end{array} \right)
\end{array}
$$

$$
\begin{array}{c}
\mathsf{recovery\_log}(lg, lg') \triangleq \\
\exists ops, lg'', lsn, act, tt, tt', tt'', dpt, dpt', dpt''. \\
\mathsf{log\_undos}(ops, lg') \wedge \mathsf{ul\_undo}(lg'' \otimes (lsn, act), tt, ops) \\
\wedge\ (lg = lg'' \otimes (lsn, act) \otimes -) \wedge lsn = max(tt \downarrow_2) \\
\wedge\ (tt = tt' \oplus tt'') \wedge (dpt = dpt' \uplus dpt'') \\
\wedge\ (lg = - \otimes (-, CHK[tt', dpt']) \otimes lg_c) \\
\wedge\ \mathsf{log\_tt}(lg_c, tt'') \wedge \mathsf{log\_dpt}(lg_c, dpt'')
\end{array}
$$

$$
\begin{array}{c}
\mathsf{recovery\_db}(db, lg, db') \triangleq \\
\exists ops, ops', ops'', lsn_{\leq}, act_{\leq}, lsn_{\geq}, act_{\geq}, dpt', dpt'', tt', tt'', lg_c, lg_b, lg'. \\
ops = ops' \otimes ops'' \wedge \mathsf{log\_rl}((lsn_{\leq}, act_{\leq}) \otimes lg', dpt, ops') \\
\wedge\ lsn_{\leq} = min(dpt \downarrow_2) \wedge (lg = - \otimes (lsn_{\leq}, act) \otimes lg') \\
\wedge\ dpt = dpt' \uplus dpt'' \wedge tt = tt' \oplus tt'' \\
\wedge\ (lg = - \otimes (-, CHK[tt', dpt']) \otimes lg_c) \wedge \mathsf{log\_dpt}(lg_c, dpt'') \\
\wedge\ \mathsf{log\_tt}(lg_c, tt'') \wedge \mathsf{ul\_undo}(lg_b \otimes (lsn_{\geq}, act_{\geq}), tt, ops'') \\
\wedge\ (lg = lg_b \otimes (lsn_{\geq}, act_{\geq}) \otimes -) \wedge lsn_{\geq} = max(tt \downarrow_2) \\
\wedge \mathsf{db\_acts}(db, ops, db')
\end{array}
$$

Log and Database Specifications

We assume simple log and database file modules with the following specification.

$$
\begin{array}{c}
\mathsf{log}(lg_i \otimes (lsn, CHK[tt, dpt]) \otimes lg_c\ ) \vdash \\
\left\{\mathsf{true} \mid \mathsf{log}(lg_i \otimes (lsn, CHK[tt, dpt]) \otimes lg_c\ )\right\} \\
\texttt{init\_from\_log}() \\
\left\{\texttt{ret} = (lsn, \texttt{tt}, \texttt{dpt}) \wedge \mathsf{set}(\texttt{tt}, tt) * \mathsf{set}(\texttt{dpt}, dpt) \mid \mathsf{log}(lg_i \otimes (\texttt{chkLsn}, CHK[tt, dpt]) \otimes lg_c\ )\right\}
\end{array}
$$

$$
\begin{array}{c}
\mathsf{log}((lsn, act) \otimes lg) \vdash \\
\left\{\texttt{lsn} = lsn \wedge \mathsf{true} \mid \mathsf{log}((lsn, act) \otimes lg)\right\} \\
\texttt{log\_mk\_f\_iter}(\texttt{lsn}) \\
\left\{\mathsf{fiter}(\texttt{ret}, lg) \wedge \texttt{lsn} = lsn \mid \mathsf{log}((lsn, act) \otimes lg)\right\}
\end{array}
$$

$$
\begin{array}{c}
\mathsf{true} \vdash \\
\left\{\mathsf{fiter}(\texttt{i}, (lsn, act) \otimes lg) \mid \mathsf{true}\right\} \\
\texttt{log\_f\_next}(\texttt{i}) \\
\left\{\texttt{ret} = (lsn, act) \wedge \mathsf{fiter}(\texttt{i}, lg) \mid \mathsf{true}\right\}
\end{array}
$$

$$
\begin{array}{c}
\mathsf{true} \vdash \\
\left\{\mathsf{fiter}(\texttt{i}, \varnothing) \mid \mathsf{true}\right\} \\
\texttt{log\_f\_next}(\texttt{i}) \\
\left\{\texttt{ret} = (\texttt{nil}, \texttt{nil})\mathsf{fiter}(\texttt{i}, \varnothing) \mid \mathsf{true}\right\}
\end{array}
$$

$$\begin{array}{c}\mathsf{true} \vdash \\ \left\{\mathsf{fiter}(\mathtt{i}, lg) \mid \mathsf{true}\right\} \\ \mathtt{log\_close\_fiter}(\mathtt{i}) \\ \left\{\mathsf{true} \mid \mathsf{true}\right\}\end{array}$$

$$\begin{array}{c}\mathsf{true} \vdash \\ \left\{\mathtt{act} = act \mid \mathsf{true}\right\} \\ \mathtt{action\_get\_type}(\mathtt{act}) \\ \left\{\begin{array}{c}\mathtt{ret} = \mathtt{U} \Rightarrow act = U[-] \\ \wedge\, \mathtt{ret} = \mathtt{B} \Rightarrow act = B[-] \\ \wedge\, \mathtt{ret} = \mathtt{C} \Rightarrow act = C[-] \\ \wedge\, \mathtt{ret} = CHK \Rightarrow act = CHK[-] \\ \wedge\, \mathtt{act} = act \wedge \mathsf{true} \\ \hline \mathsf{true}\end{array}\right\}\end{array}$$

$$\begin{array}{c}\mathsf{true} \vdash \\ \left\{\mathtt{act} = act \wedge \mathsf{true} \mid \mathsf{true}\right\} \\ \mathtt{action\_get\_tid}(\mathtt{act}) \\ \left\{\mathsf{true} \wedge \mathtt{act} = act \wedge \mathtt{ret} = tid \wedge act = U[tid, -, -] \vee act = B[tid] \vee act = C[tid] \mid \mathsf{true}\right\}\end{array}$$

$$\begin{array}{c}\mathsf{true} \vdash \\ \left\{\mathtt{act} = U[tid, pid, op] \wedge \mathsf{true} \mid \mathsf{true}\right\} \\ \mathtt{action\_get\_pid}(\mathtt{act}) \\ \left\{\mathtt{act} = U[tid, pid, op] \wedge \mathtt{ret} = pid \wedge \mathsf{true} \mid \mathsf{true}\right\}\end{array}$$

$$\begin{array}{c}\mathsf{db\_state}(db) \vdash \\ \left\{\mathsf{true} \mid \mathsf{db\_state}(db) \wedge (\mathtt{pid}, lsn, -) \in db\right\} \\ \mathtt{db\_get\_page\_lsn}(\mathtt{pid}) \\ \left\{\mathtt{ret} = lsn \wedge \mathsf{true} \mid \mathsf{db\_state}(db) \wedge (\mathtt{pid}, lsn, -) \in db\right\}\end{array}$$

$$\begin{array}{c}\mathsf{db\_state}(db \cup \{(pid, lsn', d)\}) \vee \mathsf{db\_state}(db \cup \{(pid, lsn, op(d))\}) \vdash \\ \left\{\mathtt{pid} = pid \wedge \mathtt{lsn} = lsn \wedge \mathtt{op} = op \wedge \mathsf{true} \mid \mathsf{db\_state}(db \cup \{(pid, lsn', d)\})\right\} \\ [\mathtt{db\_update\_page}(\mathtt{pid}, \mathtt{lsn}, \mathtt{op})] \\ \left\{\mathtt{pid} = pid \wedge \mathtt{lsn} = lsn \wedge \mathtt{op} = op \wedge \mathsf{true} \mid \mathsf{db\_state}(db \cup \{(pid, lsn, op(d))\})\right\}\end{array}$$

$$\begin{array}{c}\mathsf{true} \vdash \\ \left\{\mathsf{set}(\mathtt{tt}, tt) \wedge (\mathtt{tid}, -) \notin tt \mid \mathsf{true}\right\} \\ \mathtt{tt\_insert}(\mathtt{tt}, \mathtt{tid}, (\mathtt{lsn}, \mathtt{a})) \\ \left\{\mathsf{set}(\mathtt{tt}, tt \cup \{(\mathtt{tid}, (\mathtt{lsn}, \mathtt{a}))\}) \mid \mathsf{true}\right\}\end{array}$$

$$\begin{array}{c}\mathsf{true} \vdash \\ \left\{\mathsf{set}(\mathtt{tt}, tt \cup \{(\mathtt{tid}, (lsn', a'))\}) \mid \mathsf{true}\right\} \\ \mathtt{tt\_insert}(\mathtt{tt}, \mathtt{tid}, (\mathtt{lsn}, \mathtt{a})) \\ \left\{\mathsf{set}(\mathtt{tt}, tt \cup \{(\mathtt{tid}, (\mathtt{lsn}, \mathtt{a}))\}) \mid \mathsf{true}\right\}\end{array}$$

$$\begin{array}{c}\mathsf{true} \vdash \\ \left\{\mathsf{set}(\mathtt{dpt}, dpt) \wedge (\mathtt{pid}, -) \notin dpt \mid \mathsf{true}\right\} \\ \mathtt{dpt\_insert}(\mathtt{dpt}, (\mathtt{pid}, \mathtt{lsn})) \\ \left\{\mathsf{set}(\mathtt{dpt}, dpt \cup \{(\mathtt{pid}, \mathtt{lsn})\}) \mid \mathsf{true}\right\}\end{array}$$

$$\begin{array}{c}\mathsf{true} \vdash \\ \left\{\mathsf{set}(\mathtt{dpt}, dpt \cup \{(\mathtt{pid}, lsn')\}) \mid \mathsf{true}\right\} \\ \mathtt{dpt\_insert}(\mathtt{dpt}, (\mathtt{pid}, \mathtt{lsn})) \\ \left\{\mathsf{set}(\mathtt{dpt}, dpt \cup \{(\mathtt{pid}, \mathtt{lsn})\}) \mid \mathsf{true}\right\}\end{array}$$

$$\begin{array}{c}\mathsf{true} \vdash \\ \left\{\mathsf{set}(\mathtt{dpt}, dpt \cup \{(\mathtt{pid}, lsn)\}) \mid \mathsf{true}\right\} \\ \mathtt{dpt\_search}(\mathtt{dpt}, \mathtt{pid}) \\ \left\{\mathtt{ret} = (\mathtt{pid}, lsn) \wedge \mathsf{set}(\mathtt{dpt}, dpt \cup \{(\mathtt{pid}, lsn)\}) \mid \mathsf{true}\right\}\end{array}$$

The high level overview of the recovery algorithm in terms of its analysis, redo and undo phases is given in figure 9.7. The analysis phase first finds the checkpoint and restores the TT and DPT. Then, it proceeds to scan the log forwards from the checkpoint, updating the TT and DPT. Any new transaction is added to the TT. For any commit log record we update the TT to record that the transaction is committed. For any update log record, we add an entry for the associated page to the DPT, also recording the log sequence number, unless an entry for the same page is already in it. We give the following specification for the analysis phase:

$$\begin{array}{c}\mathsf{log}(lg_i \otimes (lsn, CHK[tt, dpt]) \otimes lg_c\ ) \vdash \\ \left\{\dfrac{\mathsf{true}}{\mathsf{log}(lg_i \otimes (lsn, CHK[tt, dpt]) \otimes lg_c\ )}\right\} \\ \mathtt{tt}, \mathtt{dpt} := \mathtt{aries\_analyse}() \\ \left\{\dfrac{\exists tt', dpt'.\, \mathsf{log\_tt}(lg_c, tt') \wedge \mathsf{log\_dpt}(lg_c, dpt') \wedge \mathsf{set}(\mathtt{tt}, tt \oplus tt') * \mathsf{set}(\mathtt{dpt}, dpt \uplus dpt')}{\mathsf{log}(lg_i \otimes (-, CHK[tt, dpt]) \otimes lg_c\ )}\right\}\end{array}$$

The specification states that given the database log, the TT and DPT in the log's checkpoint are restored and updated according to the log records following the checkpoint. The analysis does not modify any durable state.

The redo phase, follows analysis and repeats the logged updates. Specifically, redo scans the log forward from the record with the lowest sequence number in the DPT. This is the very first update that is logged, but (potentially) not yet written to the database. The updates are redone unless the recorded page associated with that update is not present in the DPT, or a more recent update has

modified it. We give the following specification to redo:

$$\exists ops, ops', ops''.\, (ops = ops' \otimes ops'') \wedge \mathsf{db\_acts}(db, ops', db'')$$
$$\wedge\, \mathsf{log\_fseg}((lsn, act) \otimes lg) * \mathsf{db\_state}(db'') \vdash$$
$$\left\{ \frac{\mathsf{set}(\mathtt{dpt}, dpt) \wedge lsn = min(dpt_{\downarrow 2})}{\mathsf{log\_fseg}((lsn, act) \otimes lg) * \mathsf{db\_state}(db)} \right\}$$
$$\mathtt{aries\_redo}(\mathtt{dpt})$$
$$\left\{ \frac{\mathsf{set}(\mathtt{dpt}, dpt) \wedge lsn = min(dpt_{\downarrow 2})}{\begin{array}{c}\mathsf{log\_fseg}((lsn, act) \otimes lg) * \mathsf{db\_state}(db') \wedge \mathsf{db\_acts}(db, ops, db') \\ \wedge\, \mathsf{log\_rl}((lsn, act) \otimes lg, dpt, ops)\end{array}} \right\}$$

The specification states that the database is updated according to the logged update records following the smallest log sequence number in the DPT. The fault-condition specifies that after a host failure, all, some or none of the redos have happened. Since redo does not log anything, the log is not affected.

The last phase is undo, which reverts the updates of any transaction that is not committed. In particular, undo scans the log backwards from the log record with the largest log sequence number in the TT. This is the log sequence number of the very last update. For each update record scanned, if the transaction exists in the TT and is not marked as committed, the update is reversed. However, each reverting update is logged beforehand. This ensures, that undos will happen even in case of host failure, since they will be re-done in the redo phase of the subsequent recovery run. We give the following specification for the undo phase:

$$\exists lg', lg'', lg''', ops, ops', ops''.\, lg' = lg'' \otimes lg''' \wedge ops = ops' \otimes ops''$$
$$\wedge\, \mathsf{db\_acts}(db, ops', db'') \wedge \mathsf{log\_bseg}(lg \otimes (lsn, act) \otimes lg'') * \mathsf{db\_state}(db'') \vdash$$
$$\left\{ \frac{\mathsf{set}(\mathtt{tt}, tt) \wedge lsn = max(tt_{\downarrow 2})}{\mathsf{log\_bseg}(lg \otimes (lsn, act)) * \mathsf{db\_state}(db)} \right\}$$
$$\mathtt{aries\_undo}(\mathtt{tt})$$
$$\left\{ \frac{\mathsf{set}(\mathtt{tt}, tt) \wedge lsn = max(tt_{\downarrow 2}) \wedge \mathsf{ul\_undo}(tt, lg \otimes (lsn, act), ops)}{\begin{array}{c}\mathsf{log\_bseg}(lg \otimes (lsn, act) \otimes lg') \wedge \mathsf{log\_undos}(ops, lg') \\ * \mathsf{db\_state}(db') \wedge \mathsf{db\_acts}(db, ops, db')\end{array}} \right\}$$

The specification states that the database is updated with actions reverting previous updates as obtained from the log. These undo actions are themselves logged. In the event of a host failure the fault-condition specifies that all, some, or none of the operations are undone and logged.

Using the specification for each phase and using our logic we can derive the following specification for this ARIES recovery algorithm:

$$\exists lg', lg'', db'.\, \mathsf{log}(lg \otimes (lsn, CHK[tt, dpt]) \otimes lg') * \mathsf{db\_state}(db) \vdash$$
$$\left\{ \frac{\mathsf{true}}{\mathsf{log}(lg \otimes (lsn, CHK[tt, dpt]) \otimes lg') * \mathsf{db\_state}(db)} \right\}$$
$$\mathtt{aries\_recovery}()$$
$$\left\{ \frac{\mathsf{true}}{\begin{array}{c}\mathsf{log}(lg \otimes (lsn, CHK[tt, dpt]) \otimes lg' \otimes lg'') \\ \wedge\, \mathsf{recovery\_log}(lg \otimes (lsn, CHK[tt, dpt]) \otimes lg', lg'') \\ * \mathsf{db\_state}(db') \wedge \mathsf{recovery\_db}(db, lg \otimes (lsn, CHK[tt, dpt]) \otimes lg', db')\end{array}} \right\}$$

The proof that the high level structure of the ARIES algorithm satisfies this specification is given in figure 9.8. The key property of the ARIES recovery specification is that the durable precondition is the same as the fault-condition. This guarantees that the recovery is idempotent with respect to

host failures. This is crucial for any recovery operation, as witnessed in the recovery abstraction rule, guaranteeing that the recovery itself is robust against crashes. Furthermore, the specification states that any transaction logged as committed at the time of host failure, is committed after recovery. Otherwise transactions are rolled back.

Phase implementations and proofs

The implementation of the analysis phase together with the proof that it meets the specification given in §9.7 is given in figure 9.9. The implementation and proof of the redo phase is given in figure 9.10. In figure 9.11 we give the implementation of the undo phase together with the proof it meets the specification we have given in §9.7.

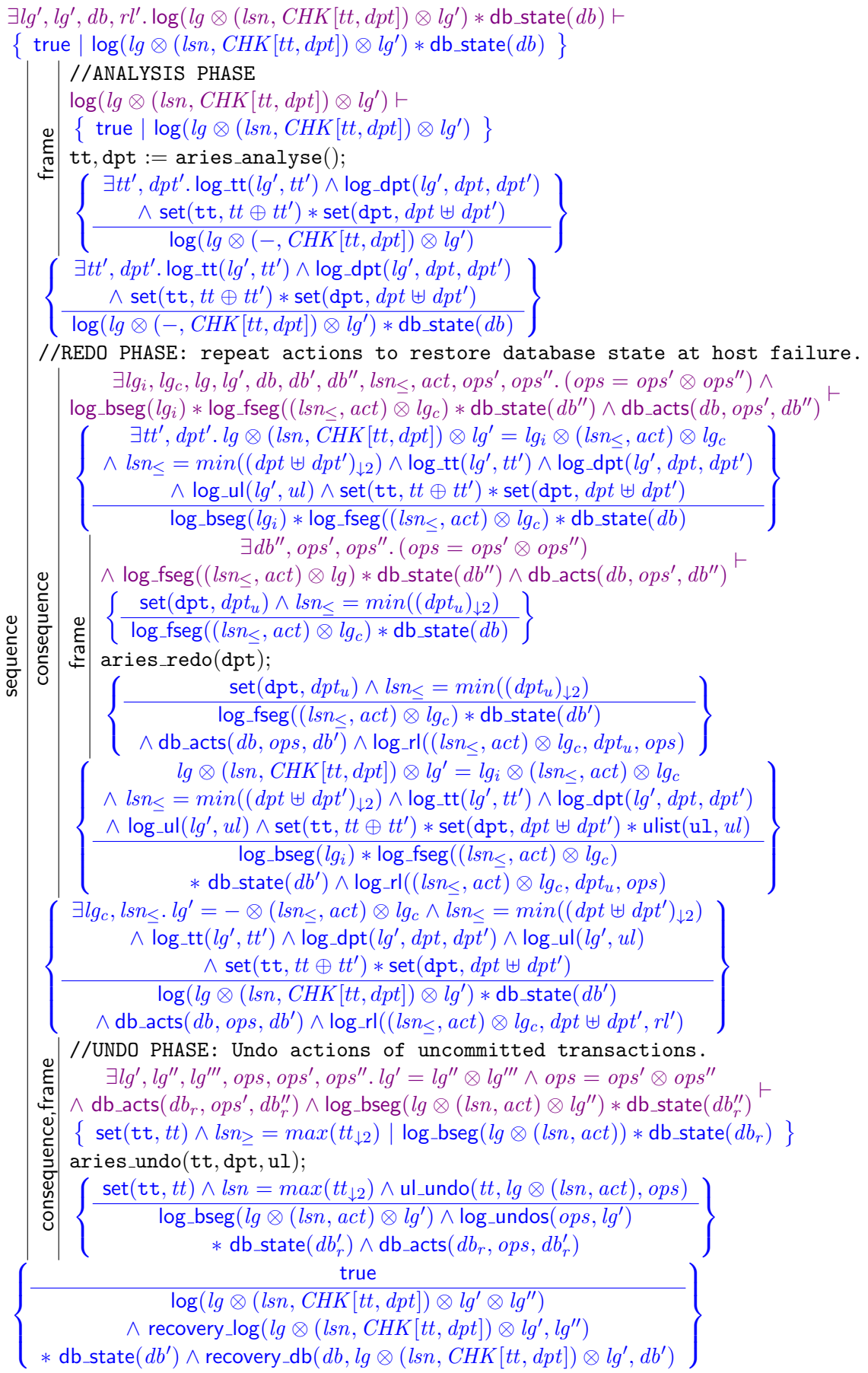

Figure 9.8.: Proof of the high level structure of ARIES recovery.

$\mathsf{log}(lg_i \otimes (lsn, CHK[tt, dpt]) \otimes lg_c\ ) \vdash$
$\{\ \mathsf{true} \mid \mathsf{log}(lg_i \otimes (lsn, CHK[tt, dpt]) \otimes lg_c\ )\ \}$
$\mathtt{chkLsn}, \mathtt{tt}, \mathtt{dpt} := \mathtt{init\_from\_log}();$
$\left\{ \dfrac{\mathsf{set}(\mathtt{tt}, tt) * \mathsf{set}(\mathtt{dpt}, dpt)}{\mathsf{log}(lg_i \otimes (\mathtt{chkLsn}, CHK[tt, dpt]) \otimes lg_c\ )} \right\}$
$\mathtt{i} := \mathtt{log\_mk\_f\_iter}(\mathtt{chkLsn});$
$\left\{ \dfrac{\mathsf{log\_out\_chk}(lg_i) \wedge \mathsf{set}(\mathtt{tt}, tt) * \mathsf{set}(\mathtt{dpt}, dpt) * \mathsf{fiter}(\mathtt{i}, lg_c)}{\mathsf{log}(lg_i \otimes (\mathtt{chkLsn}, CHK[tt, dpt]) \otimes lg_c\ )} \right\}$
$\mathtt{lsn}, \mathtt{act} := \mathtt{log\_f\_next}(\mathtt{i});$
$\left\{ \dfrac{\begin{array}{c}\exists lg_p, lg_u, lg_r, tt', dpt'.\ \mathsf{log\_out\_chk}(lg_i) \\ \wedge\ \left(\begin{array}{c}(\mathtt{lsn} \neq \mathtt{nil} \wedge (lg_c = lg_p \otimes (\mathtt{lsn}, \mathtt{act}) \otimes lg_r) \wedge lg_u = lg_p) \\ \vee\ (\mathtt{lsn} = \mathtt{nil} \wedge lg_r = \varnothing \wedge lg_u = lg_c)\end{array}\right) \\ \wedge\ \mathsf{log\_tt}(lg_u, tt') \wedge \mathsf{log\_dpt}(lg_u, dpt \uplus dpt') \wedge \mathsf{set}(\mathtt{tt}, tt \oplus tt') * \mathsf{set}(\mathtt{dpt}, dpt \uplus dpt') \\ * \mathsf{fiter}(\mathtt{i}, lg_r)\end{array}}{\mathsf{log}(lg_i \otimes (\mathtt{chkLsn}, CHK[tt, dpt]) \otimes lg_c\ )} \right\}$
$\mathtt{while}\ (\mathtt{lsn} \neq \mathtt{nil})\ \{$
$\quad \left\{ \dfrac{\exists lg_p, lg_r, tt', dpt'.\ \mathsf{log\_tt}(lg_p, tt') \wedge \mathsf{log\_dpt}(lg_p, dpt \uplus dpt') \wedge \mathsf{set}(\mathtt{tt}, tt \oplus tt') * \mathsf{set}(\mathtt{dpt}, dpt \uplus dpt') * \mathsf{fiter}(\mathtt{i}, lg_r)}{\mathsf{log}(lg_i \otimes (\mathtt{chkLsn}, CHK[tt, dpt]) \otimes lg_p \otimes (\mathtt{lsn}, \mathtt{act}) \otimes lg_r)} \right\}$
$\quad \mathtt{at} := \mathtt{action\_get\_type}(\mathtt{act});$
$\quad \mathtt{tid} := \mathtt{action\_get\_tid}(\mathtt{act});$
$\quad \mathtt{if}\ (\mathtt{at} = \mathtt{U})\ \{$
$\quad\quad \left\{ \dfrac{\exists lg_p, lg_r, tt', dpt', tid, pid, op.\ \mathtt{at} = U \wedge \mathsf{log\_tt}(lg_p, tt') \wedge \mathsf{log\_dpt}(lg_p, dpt \uplus dpt') \wedge \mathsf{set}(\mathtt{tt}, tt \oplus tt') * \mathsf{set}(\mathtt{dpt}, dpt \uplus dpt') * \mathsf{fiter}(\mathtt{i}, lg_r)}{\mathsf{log}(lg_i \otimes (\mathtt{chkLsn}, CHK[tt, dpt]) \otimes lg_p \otimes (\mathtt{lsn}, U[tid, pid, op]) \otimes lg_r)} \right\}$
$\quad\quad \mathtt{pid} := \mathtt{action\_get\_pid}(\mathtt{act});$
$\quad\quad \mathtt{tt\_insert}(\mathtt{tt}, \mathtt{tid}, (\mathtt{lsn}, \mathtt{U}));$
$\quad\quad \left\{ \dfrac{\exists lg_p, lg_r, tt', dpt', tid, pid, op.\ \mathtt{at} = U \wedge \mathsf{log\_tt}(lg_p, tt') \wedge \mathsf{log\_dpt}(lg_p, dpt \uplus dpt') \wedge \mathsf{set}(\mathtt{tt}, tt \oplus tt' \oplus \{(tid, lsn, U)\}) * \mathsf{set}(\mathtt{dpt}, dpt \uplus dpt') * \mathsf{fiter}(\mathtt{i}, lg_r)}{\mathsf{log}(lg_i \otimes (\mathtt{chkLsn}, CHK[tt, dpt]) \otimes lg_p \otimes (\mathtt{lsn}, U[tid, pid, op]) \otimes lg_r)} \right\}$
$\quad\quad \mathtt{pid}', \_ := \mathtt{dpt\_search}(\mathtt{dpt}, \mathtt{pid});$
$\quad\quad \mathtt{if}\ (\mathtt{pid}' \neq \mathtt{nil})\ \{$
$\quad\quad\quad \mathtt{dpt\_insert}(\mathtt{dpt}, \mathtt{pid}, \mathtt{lsn});$
$\quad\quad\quad \left\{ \dfrac{\exists lg_p, lg_r, tt', dpt', tid, pid, op.\ \mathtt{at} = U \wedge \mathsf{log\_tt}(lg_p, tt') \wedge \mathsf{log\_dpt}(lg_p, dpt \uplus dpt') \wedge \mathsf{set}(\mathtt{tt}, tt \oplus tt' \oplus \{(tid, lsn, U)\}) * \mathsf{set}(\mathtt{dpt}, dpt \uplus dpt' \uplus \{(pid, lsn)\}) * \mathsf{fiter}(\mathtt{i}, lg_r)}{\mathsf{log}(lg_i \otimes (\mathtt{chkLsn}, CHK[tt, dpt]) \otimes lg_p \otimes (\mathtt{lsn}, U[tid, pid, op]) \otimes lg_r)} \right\}$
$\quad\quad \}$
$\quad \}\ \mathtt{else\ if}\ (\mathtt{at} = \mathtt{C})\ \{$
$\quad\quad \mathtt{tt\_insert}(\mathtt{tt}, \mathtt{tid}, (\mathtt{lsn}, \mathtt{C}));$
$\quad\quad \left\{ \dfrac{\exists lg_p, lg_r, tt', dpt', tid, pid, op.\ \mathtt{at} = C \wedge \mathsf{log\_tt}(lg_p, tt') \wedge \mathsf{log\_dpt}(lg_p, dpt \uplus dpt') \wedge \mathsf{set}(\mathtt{tt}, tt \oplus tt' \oplus \{(tid, lsn, C)\}) * \mathsf{set}(\mathtt{dpt}, dpt \uplus dpt') * \mathsf{fiter}(\mathtt{i}, lg_r)}{\mathsf{log}(lg_i \otimes (\mathtt{chkLsn}, CHK[tt, dpt]) \otimes lg_p \otimes (\mathtt{lsn}, U[tid, pid, op]) \otimes lg_r)} \right\}$
$\quad \}$
$\quad \mathtt{lsn}, \mathtt{act} := \mathtt{log\_f\_next}(\mathtt{i});$
$\}$
$\left\{ \dfrac{\exists tt', dpt'.\ \mathsf{log\_tt}(lg_c, tt') \wedge \mathsf{log\_dpt}(lg_c, dpt \uplus dpt') \wedge \mathsf{set}(\mathtt{tt}, tt \oplus tt') * \mathsf{set}(\mathtt{dpt}, dpt \uplus dpt') * \mathsf{fiter}(\mathtt{i}, \varnothing)}{\mathsf{log}(lg_i \otimes (\mathtt{chkLsn}, CHK[tt, dpt]) \otimes lg_c\ )} \right\}$
$\mathtt{log\_close\_f\_iter}(\mathtt{i});$
$\mathtt{return}\ \mathtt{tt}, \mathtt{dtp}, \mathtt{ut};$
$\left\{ \dfrac{\exists tt', dpt'.\ \mathsf{log\_tt}(lg_c, tt') \wedge \mathsf{log\_dpt}(lg_c, dpt') \wedge \mathsf{set}(\mathtt{tt}, tt \oplus tt') * \mathsf{set}(\mathtt{dpt}, dpt \uplus dpt')}{\mathsf{log}(lg_i \otimes (-, CHK[tt, dpt]) \otimes lg_c\ )} \right\}$

Figure 9.9.: Implementation and proof of the ARIES analysis phase.

$\exists rl', rl'', rl'''.\, (rl' = rl'' \otimes rl''') \wedge \mathsf{log\_fseg}((lsn, act) \otimes lg) * \mathsf{db\_state}(rl \otimes rl'') \vdash$

$\left\{ \dfrac{\mathsf{set}(\mathtt{dpt}, dpt) \wedge lsn = min(dpt_{\downarrow 2})}{\mathsf{log\_fseg}((lsn, act) \otimes lg) * \mathsf{db\_state}(rl)} \right\}$

$\mathtt{lsn} := \mathtt{dpt\_get\_least\_lsn}(\mathtt{dpt});$

$\left\{ \dfrac{\exists dpt'.\, \mathsf{log\_dpt}(lg_c, dpt, dpt') \wedge \mathtt{lsn} = min((dpt \uplus dpt')_{\downarrow 2}) \wedge \mathsf{set}(\mathtt{dpt}, dpt \uplus dpt')}{\mathsf{true}} \right\}$

$\left\{ \dfrac{\exists lg_m.\, \mathsf{log\_dpt}(lg_c, dpt, dpt') \wedge \mathtt{lsn} = min((dpt \uplus dpt')_{\downarrow 2}) \wedge \mathsf{set}(\mathtt{dpt}, dpt \uplus dpt')}{\mathsf{log}(lg_i \otimes (-, CHK[tt, dpt]) \otimes - \otimes\ (\mathtt{lsn}, -) \otimes lg_m)} \right\}$

$\mathtt{i} := \mathtt{log\_mk\_f\_iter}(\mathtt{lsn});$

$\mathtt{lsn}, \mathtt{act} := \mathtt{log\_f\_next}(\mathtt{i});$

$\left\{ \dfrac{\exists tt', dpt', lsn_{\leq}, lg_m, lg_p, lg_u.\, \mathsf{log\_dpt}(lg_c, dpt, dpt') \wedge \left(\begin{array}{c} (\mathtt{lsn} \neq \mathtt{nil} \wedge (lg_m = lg_p \otimes (\mathtt{lsn}, \mathtt{act}) \otimes lg_r) \wedge lg_u = lg_p) \\ \vee\ (\mathtt{lsn} = \mathtt{nil} \wedge lg_r = \varnothing \wedge lg_u = lg_m) \end{array} \right) \wedge \mathsf{log\_rl}(lg_u, dpt \uplus dpt', rl') \wedge \mathsf{set}(\mathtt{dpt}, dpt \uplus dpt') * \mathsf{fiter}(\mathtt{i}, lg_r)}{\mathsf{log}(lg_i \otimes (-, CHK[tt, dpt]) \otimes - \otimes (lsn_{\leq}, -) \otimes lg_m) * \mathsf{db\_state}(rl \otimes rl')} \right\}$

```
while (lsn ≠ nil) {
  at := action_get_type(act);
  if (at = U) {
    tid := action_get_tid(act);
    pid := action_get_pid(act);
    pid', lsn' := dpt_search(dpt, pid);
    if (pid' ≠ nil ∧ lsn ≥ lsn') {
      db_update(U[tid, pid]);
    }
  }
  lsn, act := log_f_next(i);
}
```

$\left\{ \dfrac{\exists tt', dpt', lsn_{\leq}, lg_m.\, \mathsf{log\_dpt}(lg_c, dpt, dpt') \wedge \mathsf{log\_rl}(lg_m, dpt \uplus dpt', rl) \wedge \mathsf{set}(\mathtt{dpt}, dpt \uplus dpt') * \mathsf{fiter}(\mathtt{i}, \varnothing)}{\mathsf{log}(lg_i \otimes (-, CHK[tt, dpt]) \otimes - \otimes (lsn_{\leq}, -) \otimes lg_m) * \mathsf{db\_state}(rl \otimes rl')} \right\}$

$\mathtt{log\_close\_f\_iter}(\mathtt{i});$

$\left\{ \dfrac{\mathsf{set}(\mathtt{dpt}, dpt) \wedge lsn = min(dpt_{\downarrow 2})}{\mathsf{log\_fseg}((lsn, act) \otimes lg) * \mathsf{db\_state}(rl \otimes rl') \wedge \mathsf{log\_rl}((lsn, act) \otimes lg, dpt, rl')} \right\}$

Figure 9.10.: Implementation and proof of the ARIES redo phase.

$\exists lg', lg'', lg''', rl', ul', ul''.\ lg' = lg'' \otimes lg''' \wedge ul = ul' \otimes ul''$ $\vdash$
$\quad \wedge\ \mathsf{log\_bseg}(lg \otimes (lsn, act) \otimes lg'') * \mathsf{db\_state}(rl \otimes ul')$

$$\left\{ \frac{\mathsf{set}(\mathtt{tt}, tt) \wedge lsn = max(tt_{\downarrow 2})}{\mathsf{log\_bseg}(lg \otimes (lsn, act)) * \mathsf{db\_state}(rl)} \right\}$$

```
ul := ul_new();
lsn := tt_get_max_lsn(tt);
i := log_mk_b_iter(lsn);
lsn, act := log_b_next(i);
tid := action_get_tid(act);
tid, _, st := tt_search(tt, tid);
while (lsn ≠ nil ∧ tid ≠ nil) {
  at := action_get_type(act);
  if (at = U ∧ st = U) {
    pid := action_get_pid(act);
    ul_add(ul, (tid, pid));
  }
  lsn, act := log_b_next(i);
  tid := action_get_tid(act);
  tid, _, st := tt_search(tt, tid);
}
log_close_b_iter(i);
```

$$\left\{ \frac{\begin{array}{c} x = (lsn_{\leq}, -) \wedge lsn_{\leq} = max(tt_{\downarrow 2}) \wedge \mathsf{log\_tids}(lg_c \otimes x) \subseteq tt_{\downarrow 1} \\ \wedge\ \mathsf{ul\_undo}(tt, lg_c \otimes x, ul') \wedge \mathsf{set}(\mathtt{tt}, tt) * \mathsf{ulist}(\mathtt{ul}, ul \otimes ul') \end{array}}{\mathsf{log}(- \otimes (-, CHK[-]) \otimes lg_c \otimes x \otimes -) * \mathsf{db\_state}(rl)} \right\}$$

```
i := ul_mk_iter(ul);
```

$$\left\{ \frac{\begin{array}{c} x = (lsn_{\leq}, -) \wedge lsn_{\leq} = max(tt_{\downarrow 2}) \wedge \mathsf{log\_tids}(lg_c \otimes x) \subseteq tt_{\downarrow 1} \\ \wedge\ \mathsf{ul\_undo}(tt, lg_c \otimes x, ul') \wedge \mathsf{set}(\mathtt{tt}, tt) * \mathsf{ulist}(\mathtt{ul}, ul \otimes ul') * \mathsf{uliter}(\mathtt{i}, ul \otimes ul') \end{array}}{\mathsf{log}(- \otimes (-, CHK[-]) \otimes lg_c \otimes x \otimes -) * \mathsf{db\_state}(rl)} \right\}$$

```
lsn, tid, pid := ul_next(i);
```

$$\left\{ \frac{\begin{array}{c} \exists ul_r.\ lg = - \otimes (-, CHK[-]) \otimes lg_c \otimes x \otimes - \\ \wedge\ ul \otimes ul' = ul_p \otimes ul_r \wedge ((\mathtt{lsn} \neq \mathtt{nil}) \vee (\mathtt{lsn} = \mathtt{nil} \wedge ul_r = \varnothing)) \\ \wedge\ \mathsf{log\_undos}(ul_p, lg') \wedge \mathsf{db\_undos}(ul_p, rl') \wedge x = (lsn_{\leq}, -) \\ \wedge\ lsn_{\leq} = max(tt_{\downarrow 2}) \wedge \mathsf{log\_tids}(lg_c \otimes x) \subseteq tt_{\downarrow 1} \wedge \mathsf{ul\_undo}(tt, lg_c \otimes x, ul') \\ \wedge\ \mathsf{set}(\mathtt{tt}, tt) * \mathsf{ulist}(\mathtt{ul}, ul \otimes ul') * \mathsf{uliter}(\mathtt{i}, ul_r) \end{array}}{\mathsf{log}(lg \otimes lg') * \mathsf{db\_state}(rl \otimes rl')} \right\}$$

```
while (lsn ≠ nil) {
  lsn' := log_mk_lsn();
  log_append(lsn', R[tid, pid]);
  db_update(R[tid, pid]);
  lsn, tid, pid := ul_next(i);
}
```

$$\left\{ \frac{\begin{array}{c} \mathtt{lsn} = \mathtt{nil} \wedge lg = - \otimes (-, CHK[-]) \otimes lg_c \otimes x \otimes - \\ \wedge\ \mathsf{log\_undos}(ul \otimes ul', lg') \wedge \mathsf{db\_undos}(ul \otimes ul', rl') \wedge x = (lsn_{\leq}, -) \\ \wedge\ lsn_{\leq} = max(tt_{\downarrow 2}) \wedge \mathsf{log\_tids}(lg_c \otimes x) \subseteq tt_{\downarrow 1} \\ \wedge\ \mathsf{ul\_undo}(tt, lg_c \otimes x, ul') \wedge \mathsf{set}(\mathtt{tt}, tt) * \mathsf{ulist}(\mathtt{ul}, ul \otimes ul') * \mathsf{uliter}(\mathtt{i}, \varnothing) \end{array}}{\mathsf{log}(lg \otimes lg') * \mathsf{db\_state}(rl \otimes rl')} \right\}$$

```
ul_close_iter(i);
ul_free(ul);
```

$$\left\{ \frac{\mathsf{set}(\mathtt{tt}, tt) \wedge lsn = max(tt_{\downarrow 2}) \wedge \mathsf{ul\_undo}(tt, lg \otimes (lsn, act), ul)}{\mathsf{log\_bseg}(lg \otimes (lsn, act) \otimes lg') \wedge \mathsf{log\_undos}(ul, lg') * \mathsf{db\_state}(rl \otimes ul)} \right\}$$

Figure 9.11.: Implementation and proof of the ARIES undo phase.

9.8. Related Work

There has been a significant amount of work in critical systems, such as file systems and databases, to develop defensive methods against the types of failures covered in this chapter [79, 84, 15, 69]. The verification of these techniques has mainly been through testing [78, 64] and model checking [100]. However, these techniques have been based on building models that are specific to the particular application and recovery strategy, and are difficult to reuse.

Program logics based on separation logic have been successful in reasoning about file systems [47, 72] and concurrent indexes [29] on which database and file systems depend. However, as is typical with Hoare logics, their specifications avoid host failures, assuming that if a precondition holds then associated operations will not fail. An exception to this is early work by Schlichting and Schneider [85], where they propose a methodology based on Hoare logic proof outlines to reason about fault-tolerance properties of programs. However, Hoare logic itself is not fundamentally modified. In contrast, our reasoning is based on a fundamental re-interpretation of Hoare triples and dedicated inference rules. Faulty Logic [68] by Meola and Walker is another exception to the standard fault-avoiding program logics. Faulty logic is designed to reason about transient faults, such as random bit flips due to background radiation, which are different in nature from host failure.

Zengin and Vafeiadis propose a purely functional programming language with an operational semantics providing tolerance against processor failures in parallel programs [102]. Computations are check-pointed to durable storage before execution and, upon detection of a failure, the failed computations are restarted. In general, this approach does not work for concurrent imperative programs which mutate the durable store.

In independent work, Chen *et al.* introduced Crash Hoare Logic (CHL) to reason about host failures and applied it to a substantial sequential journaling file system (FSCQ) written in Coq [27, 26]. CHL extends Hoare triples with fault-conditions and provides highly automated reasoning about host failures. FSCQ performs physical journaling, meaning it uses a write-ahead log for both data and metadata, so that the recovery can guarantee atomicity with respect to host failures. The authors use CHL to prove that this property is indeed true. The resource stored in the disk is treated as durable. Since FSCQ is implemented in the functional language of Coq, which lacks the traditional process heap, the volatile state is stored in immutable variables.

The aim of FSCQ and CHL is to provide a verified implementation of a sequential file system which tolerates host failures. In contrast, our aim is to provide a general methodology for fault-tolerant resource reasoning about concurrent programs. We extend the Views framework [35] to provide a general concurrent framework for reasoning about host failure and recovery. Like CHL, we extend Hoare triples with fault-conditions. We instantiate our framework to concurrent separation logic, and demonstrate that an ARIES recovery algorithm uses the write-ahead log correctly to guarantee the atomicity of transactions.

As we are defining a framework, our reasoning of the durable and volatile state (given by arbitrary view monoids) is general. In contrast, CHL reasoning is specific to the durable state on the disk and the volatile state in the immutable variable store. CHL is able to reason modularly about different layers of abstraction of a file-system implementation, using *logical address spaces* which give a systematic pattern of use for standard predicates. We do not explore modular reasoning about layers of abstractions in this chapter, since it is orthogonal to reasoning about host failures, and

examples have already been studied in instances of the Views framework and other separation logic literature [77, 36, 30, 88, 32].

We can certainly benefit from the practical CHL approach to mechanisation and proof automation. We also believe that future work on CHL, especially on extending the reasoning to heap-manipulating concurrent programs, can benefit from our general approach.

9.9. Conclusions

We have developed an extension of resource reasoning for fault tolerance by extending the Views framework [35] to reason about programs that experience host failures. The fault-tolerant Views framework is a generic framework for constructing sound fault-tolerant concurrent program logics. The details of the framework and its soundness result are given in appendix C. We have demonstrated our approach with a particular instantiation of this framework, FTCSL; a fault-tolerant extension of concurrent separation logic. Many other program logics that have been shown to be instances of the Views framework can be similarly extended, such as CAP [36] and Rely-Guarantee [60].

We have discussed our approach can be used to specify journaling file systems. Our framework is directly applicable to program logics used for sequential fragments of POSIX file systems, such as structural separation logic [47] and fusion logic [72]. It is not directly applicable to the concurrent specification developed in this dissertation, since the Views framework does not handle atomicity and contextual refinement. Extending reasoning about fault tolerance to the specification language and refinement calculus developed in chapter 7 is a matter of future work. The work presented in this chapter is the first step towards that direction.

10. Conclusions

In this dissertation we have tackled the subject of reasoning about POSIX file systems with a primary focus on their concurrent behaviour. We have identified two main challenges in the style of concurrency exhibited by POSIX file systems: file-system operations perform complex sequences of atomic steps, and POSIX file systems are a public namespace.

We addressed the first challenge by developing a specification language capable of specifying operations in terms of multiple atomic steps. Our specification language is the combination of two concepts from prior work on reasoning about fine-grained concurrency: specifications of abstract atomicity from the program logic TaDA [30], and the specification language and contextual refinement reasoning by Turon and Wand [94]. We have given a formal specification of a core fragment of the POSIX file-system interface, capturing the complex concurrent behaviour specified informally in the POSIX standard. To the best of our knowledge, this is the first formal specification of file-system concurrency to achieve this. However, we do not claim that our formalisation is complete or even definitive. By focusing on concurrency we have only covered a fragment of the POSIX file-system interface, and ambiguities as well as possible errors in the POSIX standard prevents us from making such a claim. However, we have demonstrated that our specification approach is flexible and amenable to extensions and revisions in line with future revisions of the standard.

We developed a refinement calculus for reasoning about client applications using the file system. We applied our method to the examples of lock files and an implementation of named pipes to demonstrate the scalability of our reasoning. To address the second challenge, the fact that file systems are a public namespace, we introduced client specifications conditional on context invariants to restrict the interference on file-system operations.

Additionally, we have developed a general framework, through an extension of the Views framework [35], for reasoning about concurrent programs in the presence of host failures. To account for host failures, we distinguish between volatile resource, that is lost in the event of a host failure, and durable resource, that persists between host failures. To reason about fault-tolerance properties, we introduce a rule for abstracting the behaviour of recovery procedures. With an eye towards file systems, we do not limit the verification of fault tolerance to just atomicity with respect to host failures. We have presented a particular instance of this framework, a fault-tolerant concurrent separation logic. We have demonstrated our reasoning by studying an ARIES recovery algorithm, showing that it is idempotent and that it guarantees atomicity of database transactions in the event of a host failure.

10.1. Future Work

Our research on the specification of POSIX file systems and reasoning about fault tolerance is far from over. Both strands of work present several avenues for future work. The overarching goal is for both strands of work to eventually converge into a unified reasoning theory. Our vision is that the work

developed in this dissertation will form the basis of theories and tools that will assist the development and improvement of future revisions of the POSIX file-system specification as well as assist developers in implementation conformance.

10.1.1. Mechanisation and Automation

Our file-system specifications, refinement calculus for atomicity, reasoning framework for fault tolerance, as well as their soundness proofs in this dissertation have all been done by hand. In future we plan to mechanise our reasoning systems and specifications in an interactive theorem prover such as Coq or HOL. Ideally, we would like to extract an executable version of our specification from this mechanisation. All other future directions discussed subsequently will stand to benefit from a mechanisation of the content presented in this dissertation. Additionally, we plan to develop automated verification tools. For automating the verification of fault-tolerance properties, one plausible direction is to extend existing separation-logic based automated tools such as Infer [23]. However, automating the verification of file-system clients will require the development of new tooling, as automated tools for verifying abstract atomicity have yet to be developed.

10.1.2. Total Correctness

Both our file-system reasoning and fault-tolerance reasoning is done from the point of partial correctness. Our specifications do not require implementations to terminate nor can we prove termination. This is in line with POSIX which purposefully does not mandate termination properties. However, total correctness is desirable for reasoning about particular file-system implementations. Extending our specification to a total correctness interpretation will require modifications to our semantics of contextual refinement and atomicity. We plan to use the recent of extension of TaDA to total correctness by da Rocha Pinto *et al.* [32] as a starting point.

Extending our reasoning for fault tolerance to total correctness is more challenging. If we assume there are no host failures, total correctness should work as in other concurrent separation logics. However, in the presence of host failures, the termination or non-termination of a program becomes much more subtle, and actually depends whether the program is associated with a recovery. Assume a program that is not associated with a recovery. Then, even if the program does not terminate on its own accord, it is guaranteed to be eventually terminated by a host failure. On the other hand, if the program is associated with a recovery, the recovery will execute after a host failure. However, in the event of a host failure, there is no way to guarantee that the recovery operation will terminate. The recovery operation itself may be terminated by a host failure, at which point it will restart and this process may continue indefinitely.

10.1.3. Helping and Speculation

Our refinement calculus for atomicity limits the verification of atomicity to operations that do not employ *helping* or *speculation* [10] in their implementation. Helping occurs when the linearisation point for the current thread is performed by different thread. Speculation occurs when the linearisation point for the current thread can only be determined after the operation returns and the environment subsequently performs some other action. We inherit these limitations from the semantics and proof

system of TaDA. In future, we plan to extend our semantics and refinement calculus to support atomicity verification of operations employing helping and speculation. Future extensions of TaDA will also benefit from this work.

10.1.4. File-System Implementation and Testing

In this dissertation we justify our POSIX file-system specification by appealing to the standard as well as the Austin Group mailing list. In future, we plan to justify our specification with respect to implementations. One approach is to justify the specification against real-world implementations by generating tests and using the specification as a test oracle, similarly to the approach of Ridge *et al* [83]. Another approach, following the laws of our refinement calculus, is to refine the specification to a fine-grained concurrent reference implementation. Both approaches will require a mechanised version of our POSIX specification.

10.1.5. Fault-tolerance of File Systems

Our POSIX file-system specification does not specify any fault-tolerance properties of the file-system operations. The POSIX standard does not specify what implementations should do in this case. However, most popular file-system implementations strive to provide some guarantees in the case of host failures. At the very least, all major file-system implementations preserve the integrity of the overall file-system structure, *i.e.* the file system remains a file system even if the implementation allows for some data loss. We plan to extend our file-system specifications with the fault-tolerance guarantees provided by a range of file-system implementations. However, our reasoning framework for fault tolerance does not support reasoning about operations that are sequences of atomic steps, as are file-system operations in POSIX. Therefore, we will have to extend our specification language and refinement calculus for atomicity to account for host failure.

10.1.6. Atomicity? Which Atomicity?

In this dissertation we have used the term "atomicity" in two different contexts: concurrency and host failures. The meaning of the term is subtly different but also related between the two contexts in which it is used. In the context of concurrency, an operation is atomic if it appears to take effect at a single, discrete point in time. In the context of host failures, an operation is atomic if we always observe it happening completely or not at all, even if it is interrupted by a host failure. The rules of fault-tolerant concurrent separation logic we have developed in chapter 9, require an operation that is atomic in the context of concurrency to also be atomic with respect to host failures. On the one hand this is sensible: if the operation is not atomic with respect to host failures, after recovery we may observe an intermediate state, which would violate it being atomic in the sense of concurrency. On the other hand, in several file-system implementations the `write` operation is atomic in the concurrency sense, but it is not guaranteed to be atomic with respect to host failures. We can also ask the question: if an operation is atomic with respect to host-failures, does it have to be atomic in the sense of concurrency? In transactions systems that adhere to ACID properties this certainly is the case. The exact relationship between these two notions of atomicity remains an open question.

Bibliography

[1] Hfs plus volume format. http://dubeiko.com/development/FileSystems/HFSPLUS/tn1150.html. Accessed: September 30, 2016. 27

[2] How Time Machine Works its Magic. http://pondini.org/TM/Works.html. Accessed: September 30, 2016. 26

[3] Introduction to Microsoft Windows Services for UNIX 3.5. https://technet.microsoft.com/en-gb/library/bb463212.aspx. Accessed: September 30, 2016. 27

[4] Linux manpages, link(2). http://man7.org/linux/man-pages/man2/link.2.html. Accessed: September 30, 2016. 26

[5] Microsoft extensible firmware initiative fat32 file system specification. http://www.microsoft.com/whdc/system/platform/firmware/fatgen.mspx. Accessed: September 30, 2016. 27

[6] Ntfs technical reference. http://technet.microsoft.com/en-us/library/cc758691(WS.10).aspx. Accessed: September 30, 2016. 27

[7] POSIX.1-2008, IEEE 1003.1-2008, The Open Group Base Specifications Issue 7. 19, 24, 27, 28, 29, 30, 31, 32, 33, 34, 39, 55

[8] The Austin Group. http://www.opengroup.org/austin/. Accessed: September 30, 2016. 27

[9] The Austin Group Mailing List. https://www.opengroup.org/austin/lists.html. Accessed: September 30, 2016. 34, 39

[10] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253 – 284, 1991. 218

[11] K. Arkoudas, K. Zee, V. Kuncak, and M. Rinard. Verifying a file system implementation. In J. Davies, W. Schulte, and M. Barnett, editors, *Formal Methods and Software Engineering*, volume 3308 of *Lecture Notes in Computer Science*, pages 373–390. Springer Berlin Heidelberg, 2004. 19, 53

[12] E. C. M. Association et al. *Volume and File Structure of CDROM for Information Interchange*. ECMA, 1987. 24

[13] R.-J. Back and J. Wright. *Refinement calculus: a systematic introduction*. Springer Science & Business Media, 2012. 19, 48, 130

[14] J. Berdine, C. Calcagno, and P. W. O'Hearn. *Smallfoot: Modular Automatic Assertion Checking with Separation Logic*, pages 115–137. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. 44

[15] J. Bonwick, M. Ahrens, V. Henson, M. Maybee, and M. Shellenbaum. The zettabyte file system. In *Proc. of the 2nd Usenix Conference on File and Storage Technologies*, 2003. 215

[16] R. Bornat. *Proving Pointer Programs in Hoare Logic*, pages 102–126. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000. 42

[17] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 259–270, New York, NY, USA, 2005. ACM. 44, 46

[18] R. Bornat, C. Calcagno, and H. Yang. Variables as resource in separation logic. *Electronic Notes in Theoretical Computer Science*, 155:247 – 276, 2006. 43

[19] M. Botinčan, D. Distefano, M. Dodds, R. Grigore, and M. J. Parkinson. corestar: The core of jstar. In *In Boogie*, pages 65–77, 2011. 44

[20] J. Boyland. *Checking Interference with Fractional Permissions*, pages 55–72. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003. 44, 54

[21] S. Brookes. Full abstraction for a shared-variable parallel language. *Information and Computation*, 127(2):145 – 163, 1996. 69, 108, 109, 116

[22] C. Calcagno and D. Distefano. *Infer: An Automatic Program Verifier for Memory Safety of C Programs*, pages 459–465. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. 44

[23] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez. *Moving Fast with Software Verification*, pages 3–11. Springer International Publishing, Cham, 2015. 44, 218

[24] C. Calcagno, P. Gardner, and U. Zarfaty. Context logic and tree update. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 271–282, New York, NY, USA, 2005. ACM. 54

[25] C. Calcagno, P. W. O'Hearn, and H. Yang. Local action and abstract separation logic. In *22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007)*, pages 366–378, July 2007. 45, 77, 97

[26] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich. Using crash hoare logic for certifying the fscq file system. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 18–37, New York, NY, USA, 2015. ACM. 19, 53, 215

[27] H. Chen, D. Ziegler, A. Chlipala, M. F. Kaashoek, E. Kohler, and N. Zeldovich. Specifying crash safety for storage systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, 2015. USENIX Association. 53, 215

[28] P. da Rocha Pinto. *Reasoning with Time and Data Abstractions*. PhD thesis, Imperial College London, 2016. 20, 22, 49, 51, 94, 97

[29] P. da Rocha Pinto, T. Dinsdale-Young, M. Dodds, P. Gardner, and M. Wheelhouse. A simple abstraction for complex concurrent indexes. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 845–864, New York, NY, USA, 2011. ACM. 20, 47, 215

[30] P. da Rocha Pinto, T. Dinsdale-Young, and P. Gardner. Tada: A logic for time and data abstraction. In R. Jones, editor, *ECOOP 2014 – Object-Oriented Programming*, volume 8586 of *Lecture Notes in Computer Science*, pages 207–231. Springer Berlin Heidelberg, 2014. 20, 22, 49, 51, 59, 61, 73, 76, 91, 94, 117, 124, 130, 134, 216, 217

[31] P. da Rocha Pinto, T. Dinsdale-Young, and P. Gardner. TaDA: A Logic for Time and Data Abstraction. Technical report, Imperial College London, 2014. 97, 106

[32] P. da Rocha Pinto, T. Dinsdale-Young, and P. Gardner. Steps in modular specifications for concurrent modules (invited tutorial paper). *Electronic Notes in Theoretical Computer Science*, 319:3 – 18, 2015. 20, 46, 47, 194, 200, 216, 218

[33] K. Damchoom and M. Butler. *Applying Event and Machine Decomposition to a Flash-Based Filestore in Event-B*, pages 134–152. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. 53

[34] K. Damchoom, M. Butler, and J.-R. Abrial. *Modelling and Proof of a Tree-Structured File System in Event-B and Rodin*, pages 25–44. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. 53

[35] T. Dinsdale-Young, L. Birkedal, P. Gardner, M. Parkinson, and H. Yang. Views: Compositional reasoning for concurrent programs. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 287–300, New York, NY, USA, 2013. ACM. 21, 22, 45, 54, 97, 106, 116, 188, 189, 202, 215, 216, 217, 291

[36] T. Dinsdale-Young, M. Dodds, P. Gardner, M. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In T. D'Hondt, editor, *ECOOP 2010 – Object-Oriented Programming*, volume 6183 of *Lecture Notes in Computer Science*, pages 504–528. Springer Berlin Heidelberg, 2010. 46, 47, 98, 132, 134, 216

[37] D. Distefano. *Attacking Large Industrial Code with Bi-abductive Inference*, pages 1–8. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. 44

[38] D. Distefano and M. J. Parkinson J. jstar: Towards practical verification for java. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA '08, pages 213–226, New York, NY, USA, 2008. ACM. 44

[39] M. Dodds, X. Feng, M. Parkinson, and V. Vafeiadis. *Deny-Guarantee Reasoning*, pages 363–377. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. 46

[40] G. Ernst, G. Schellhorn, D. Haneberg, J. Pfähler, and W. Reif. Verification of a virtual filesystem switch. In E. Cohen and A. Rybalchenko, editors, *Verified Software: Theories, Tools, Experiments*, volume 8164 of *Lecture Notes in Computer Science*, pages 242–261. Springer Berlin Heidelberg, 2014. 53

[41] X. Feng. Local rely-guarantee reasoning. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 315–327, New York, NY, USA, 2009. ACM. 46

[42] M. A. Ferreira and J. N. Oliveira. *An Integrated Formal Methods Tool-Chain and Its Application to Verifying a File System Model*, pages 153–169. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. 53

[43] I. Filipović, P. O'Hearn, N. Rinetzky, and H. Yang. Abstraction for concurrent objects. *Theoretical Computer Science*, 411(51):4379 – 4398, 2010. 48

[44] L. Freitas, Z. Fu, and J. Woocock. Posix file store in z/eves: an experiment in the verified software repository. In *Engineering Complex Computer Systems, 2007. 12th IEEE International Conference on*, pages 3–14, July 2007. 19, 53

[45] L. Freitas, J. Woodcock, and A. Butterfield. Posix and the verification grand challenge: A roadmap. *2014 19th International Conference on Engineering of Complex Computer Systems*, 0:153–162, 2008. 19, 53

[46] A. Galloway, G. Lüttgen, J. Mühlberg, and R. Siminiceanu. Model-checking the linux virtual file system. In N. Jones and M. Müller-Olm, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 5403 of *Lecture Notes in Computer Science*, pages 74–88. Springer Berlin Heidelberg, 2009. 52

[47] P. Gardner, G. Ntzik, and A. Wright. Local reasoning for the posix file system. In Z. Shao, editor, *Programming Languages and Systems*, volume 8410 of *Lecture Notes in Computer Science*, pages 169–188. Springer Berlin Heidelberg, 2014. 19, 20, 54, 199, 215, 216

[48] P. Gardner and M. Wheelhouse. *Small Specifications for Tree Update*, pages 178–195. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. 54

[49] P. A. Gardner, G. D. Smith, M. J. Wheelhouse, and U. D. Zarfaty. Local hoare reasoning about dom. In *Proceedings of the Twenty-seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '08, pages 261–270, New York, NY, USA, 2008. ACM. 54

[50] A. Groce, G. Holzmann, and R. Joshi. Randomized differential testing as a prelude to formal verification. In *29th International Conference on Software Engineering (ICSE'07)*, pages 621–631, May 2007. 52

[51] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990. 19, 47

[52] W. Hesselink and M. Lali. Formalizing a hierarchical file system. *Formal Aspects of Computing*, 24(1):27–44, 2012. 19, 53

[53] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, Oct. 1969. 42

[54] C. A. R. Hoare. Proof of a program: Find. *Commun. ACM*, 14(1):39–45, Jan. 1971. 42

[55] T. Hoare. *The Verifying Compiler: A Grand Challenge for Computing Research*, pages 25–35. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003. 52

[56] T. Hoare. Generic models of the laws of programming. In Z. Liu, J. Woodcock, and H. Zhu, editors, *Theories of Programming and Formal Methods - Essays Dedicated to Jifeng He on the Occasion of His 70th Birthday*, volume 8051 of *Lecture Notes in Computer Science*, pages 213–226. Springer, 2013. 65, 114

[57] T. Hoare and S. van Staden. In praise of algebra. *Formal Aspects of Computing*, 24(4-6):423–431, 2012. 114

[58] B. Jacobs and F. Piessens. Expressive modular fine-grained concurrency specification. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 271–282, New York, NY, USA, 2011. ACM. 51

[59] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. *VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java*, pages 41–55. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. 44

[60] C. B. Jones. Specification and design of (parallel) programs. In *IFIP congress*, volume 83, pages 321–332, 1983. 46, 216

[61] R. Joshi and G. Holzmann. A mini challenge: build a verifiable filesystem. *Formal Aspects of Computing*, 19(2):269–272, 2007. 52

[62] R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 637–650, New York, NY, USA, 2015. ACM. 51

[63] E. Kang and D. Jackson. *Formal Modeling and Analysis of a Flash Filesystem in Alloy*, pages 294–308. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. 53

[64] N. Kropp, P. Koopman, and D. Siewiorek. Automated robustness testing of off-the-shelf software components. In *Fault-Tolerant Computing, 1998. Digest of Papers. Twenty-Eighth Annual International Symposium on*, pages 230–239, June 1998. 215

[65] H. Liang, X. Feng, and Z. Shao. Compositional verification of termination-preserving refinement of concurrent programs. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, CSL-LICS '14, pages 65:1–65:10, New York, NY, USA, 2014. ACM. 49

[66] R. J. Lipton. Reduction: A method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, Dec. 1975. 131

[67] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux symposium*, volume 2, pages 21–33, 2007. 27

[68] M. Meola and D. Walker. Faulty logic: Reasoning about fault tolerant programs. In A. Gordon, editor, *Programming Languages and Systems*, volume 6012 of *Lecture Notes in Computer Science*, pages 468–487. Springer Berlin Heidelberg, 2010. 215

[69] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, Mar. 1992. 21, 189, 203, 215

[70] C. Morgan. The refinement calculus. In M. Broy, editor, *Program Design Calculi*, volume 118 of *NATO ASI Series*, pages 3–52. Springer Berlin Heidelberg, 1993. 48

[71] C. Morgan and B. Sufrin. Specification of the unix filing system. *Software Engineering, IEEE Transactions on*, SE-10(2):128–142, March 1984. 19, 53

[72] G. Ntzik and P. Gardner. Reasoning about the posix file system: Local update and global pathnames. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 201–220, New York, NY, USA, 2015. ACM. 19, 20, 54, 196, 199, 215, 216

[73] G. Ntzik, P. Rocha Pinto, and P. Gardner. *Programming Languages and Systems: 13th Asian Symposium, APLAS 2015, Pohang, South Korea, November 30 - December 2, 2015, Proceedings*, chapter Fault-Tolerant Resource Reasoning, pages 169–188. Springer International Publishing, Cham, 2015. 22

[74] P. O'Hearn, J. Reynolds, and H. Yang. *Local Reasoning about Programs that Alter Data Structures*, pages 1–19. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001. 42

[75] S. Owicki and D. Gries. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM*, 19(5):279–285, May 1976. 200

[76] P. W. O'Hearn. Resources, concurrency, and local reasoning. *Theoretical Computer Science*, 375(1–3):271 – 307, 2007. Festschrift for John C. Reynolds's 70th birthday. 21, 44, 45, 46, 116, 128, 132, 188, 189, 196

[77] M. Parkinson and G. Bierman. Separation logic and abstraction. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 247–258, New York, NY, USA, 2005. ACM. 46, 190, 216

[78] V. Prabhakaran, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Model-based failure analysis of journaling file systems. In *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*, pages 802–811, June 2005. 215

[79] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Analysis and evolution of journaling file systems. In *USENIX Annual Technical Conference, General Track*, 2005. 194, 196, 215

[80] H. Reiser. Reiserfs. http://cs.uns.edu.ar/~jechaiz/sosd/clases/slides/07-FileSystemsExtra4_Reiser.pdf, 2004. 27

[81] J. Reynolds. Separation logic: a logic for shared mutable data structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 55–74, 2002. 19, 20, 42, 94, 116, 128, 188, 189, 190

[82] J. C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In *Millennial Perspectives in Computer Science*, pages 303–321. Palgrave, 2000. 42

[83] T. Ridge, D. Sheets, T. Tuerk, A. Giugliano, A. Madhavapeddy, and P. Sewell. Sibylfs: Formal specification and oracle-based testing for posix and real-world file systems. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 38–53, New York, NY, USA, 2015. ACM. 19, 36, 53, 70, 219

[84] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, Feb. 1992. 215

[85] R. D. Schlichting and F. B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Trans. Comput. Syst.*, 1(3):222–238, Aug. 1983. 215

[86] P. Snyder. tmpfs: A virtual memory file system. In *Proceedings of the Autumn 1990 EUUG Conference*, pages 241–248, 1990. 27

[87] W. R. Stevens and S. A. Rago. *Advanced Programming in the UNIX Environment*. Addison-Wesley Professional, 3rd edition, 2013. 24, 26, 27, 31

[88] K. Svendsen and L. Birkedal. Impredicative concurrent abstract predicates. In Z. Shao, editor, *Programming Languages and Systems*, volume 8410 of *Lecture Notes in Computer Science*, pages 149–168. Springer Berlin Heidelberg, 2014. 51, 98, 216

[89] K. Svendsen, L. Birkedal, and M. Parkinson. Modular reasoning about separation of concurrent data structures. In M. Felleisen and P. Gardner, editors, *Programming Languages and Systems*, volume 7792 of *Lecture Notes in Computer Science*, pages 169–188. Springer Berlin Heidelberg, 2013. 51

[90] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007. 23, 24

[91] R. K. Treiber. Systems programming: Coping with parallelism. Technical report, IBM Almaden Research Center, 1986. 48

[92] H. Trickey. Ape - the ansi/posix environment. *World-Wide Web document, Computing Sciences Research Center, Bell Laboratories, Murray Hill, NJ, USA*, 2000. 27

[93] A. Turon, D. Dreyer, and L. Birkedal. Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 377–390, New York, NY, USA, 2013. ACM. 49, 51

[94] A. J. Turon and M. Wand. A separation logic for refining concurrent objects. *ACM SIGPLAN Notices*, 46(1):247–258, 2011. 19, 48, 51, 59, 108, 109, 116, 130, 217

[95] V. Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, 2008. 46

[96] V. Vafeiadis and M. Parkinson. *A Marriage of Rely/Guarantee and Separation Logic*, pages 256–271. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. 46

[97] A. D. Wright. *Structural Separation Logic*. PhD thesis, Imperial College London, 2013. 46, 54

[98] S. Xiong, P. da Rocha Pinto, G. Ntzik, and P. Gardner. Abstract Specifications for Concurrent Maps. In *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017*, Lecture Notes in Computer Science. Springer, 2017. 91

[99] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. O'Hearn. *Scalable Shape Analysis for Systems Code*, pages 385–398. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. 44

[100] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. *ACM Trans. Comput. Syst.*, 24(4):393–423, Nov. 2006. 52, 215

[101] U. Zarfaty and P. Gardner. Local reasoning about tree update. *Electronic Notes in Theoretical Computer Science*, 158(0):399 – 424, 2006. Proceedings of the 22nd Annual Conference on Mathematical Foundations of Programming Semantics (MFPS XXII) Mathematical Foundations of Programming Semantics {XXII}. 54

[102] M. Zengin and V. Vafeiadis. A programming language approach to fault tolerance for fork-join parallelism. In *Theoretical Aspects of Software Engineering (TASE), 2013 International Symposium on*, pages 105–112, July 2013. 215

A. POSIX Fragment Formalisation

For the POSIX specifications in this dissertation, we extend the set of values to include paths and error codes, file-system graphs, inodes, filenames, bytes, file data, file descriptor flags, bounded integers, the special values `int` denoting the use of bounded integers in heap memory operations, and the special value $\texttt{STR}(n)$ denoting the use of string of size n in heap memory operations.

$$\textsc{FileFlags} \triangleq \{\texttt{O\_CREAT}, \texttt{O\_EXCL}, \texttt{O\_RDONLY}, \texttt{O\_WRONLY}, \texttt{O\_RDWR}\}$$

$$\begin{aligned}\textsc{Val} \triangleq \ldots \cup \textsc{Paths} \cup \textsc{Errs} \cup \mathcal{FS} \cup \textsc{Inodes} \cup \textsc{FNames} \cup \{\text{“.”}, \text{“..”}\} \cup \textsc{Bytes} \cup \textsc{FileData} \\ \cup \textsc{FileFlags} \cup \textsc{Int} \cup \{\texttt{int}\} \cup \{\texttt{STR}(n) \mid n \in \mathbb{N}^+\}\end{aligned}$$

We extend expressions to include pathname concatenation, head, tail, basename and directory path expressions.

$$e, e' ::= \ldots \mid e/e' \mid \texttt{head}(e) \mid \texttt{tail}(e) \mid \texttt{basename}(e) \mid \texttt{dirname}(e)$$

$$\begin{aligned}
[\![e/e']\!]^\rho &\triangleq [\![e]\!]^\rho \,/\, [\![e']\!]^\rho && \text{if } [\![e]\!]^\rho \in \textsc{Paths} \text{ and } [\![e']\!]^\rho \in \textsc{Paths} \text{ and } [\![e]\!]^\rho \,/\, [\![e']\!]^\rho \in \textsc{Paths} \\
[\![\texttt{head}(e)]\!]^\rho &\triangleq \texttt{null} && \text{if } [\![e]\!]^\rho \in \{\varnothing_p, \varnothing_p/\varnothing_p\} \\
[\![\texttt{head}(e)]\!]^\rho &\triangleq a && \text{if } [\![e]\!]^\rho \in \{a, a/p, \varnothing_p/a, \varnothing_p/a/p\} \\
[\![\texttt{tail}(e)]\!]^\rho &\triangleq \texttt{null} && \text{if } [\![e]\!]^\rho \in \{\varnothing_p, \varnothing_p/\varnothing_p, a, a/\varnothing_p, \varnothing_p/a, \varnothing_p/a/\varnothing_p\} \\
[\![\texttt{tail}(e)]\!]^\rho &\triangleq p && \text{if } [\![e]\!]^\rho \in \{a/p, \varnothing_p/a/p\} \text{ and } p \neq \varnothing_p \\
[\![\texttt{basename}(e)]\!]^\rho &\triangleq \texttt{null} && \text{if } [\![e]\!]^\rho \in \{\varnothing_p, \varnothing_p/\varnothing_p\} \\
[\![\texttt{basename}(e)]\!]^\rho &\triangleq a && \text{if } [\![e]\!]^\rho \in \{a, a/\varnothing_p, p/a, p/a/\varnothing_p\} \\
[\![\texttt{dirname}(e)]\!]^\rho &\triangleq \texttt{null} && \text{if } [\![e]\!]^\rho \in \{\varnothing_p, \varnothing_p/\varnothing_p, \varnothing_p/a, \varnothing_p/a/\varnothing_p, a, a/\varnothing_p\} \\
[\![\texttt{dirname}(e)]\!]^\rho &\triangleq p && \text{if } [\![e]\!]^\rho \in \{p/a, p/vara/\varnothing_p\} \text{ and } p \neq \varnothing_p
\end{aligned}$$

Next, we extend expressions to include file-descriptor flag expressions.

$$e ::= \ldots \mid \mathsf{fdflags}(e)$$

$$[\![\mathsf{fdflags}(e)]\!]^\rho \triangleq [\![e]\!]^\rho \setminus \{\texttt{O\_CREAT}, \texttt{O\_EXCL}\} \qquad \text{if } [\![e]\!]^\rho \in \textsc{FileFlags}$$

Furthermore, we extend expression to include the following expressions on byte sequences (in the

sense of FileData).

$$
\begin{array}{lll}
e, e', e'' ::= & \ldots & \\
& \mid \mathsf{len}(e) & \text{length of byte sequence } e \\
& \mid \mathsf{seqn}(e, e') & \text{byte sequence of } e\text{s of length } e' \\
& \mid \mathsf{skipseq}(e, e') & \text{byte sequence which skips } e' \text{ number of bytes from byte sequence } e \\
& \mid \mathsf{seqtake}(e, e') & \text{byte sequence which keeps } e' \text{ number of bytes from byte sequence } e \\
& \mid \mathsf{subseq}(e, e', e'') & \text{sub-sequence of } e, \text{ starting from } e', \text{ of length } e'' \\
& \mid \mathsf{zeroise}(e) & \text{byte sequence in which every } \emptyset \text{ element is replaced with } 0 \\
& \mid e \upharpoonleft e' & \text{overwriting of sequence } e \text{ with } e' \\
& \mid e[e' \leftarrow e''] & \text{update of file data } e, \text{ at offset } e' \text{ with byte sequence } e'' \\
& \mid e[e', e''] & \text{read file data } e, \text{ from offset } e' \text{ of at most length } e''
\end{array}
$$

The denotations of the byte-sequence expressions are defined as follows:

$$
\begin{aligned}
[\![\mathsf{len}(\epsilon)]\!]^\rho &= 0 \\
[\![\mathsf{len}(y : e)]\!]^\rho &= 1 + [\![\mathsf{len}(e)]\!]^\rho && \text{if } [\![e]\!]^\rho \in \textsc{FileData} \\
[\![\mathsf{seqn}(y, e)]\!]^\rho &= \epsilon && \text{if } y \in \textsc{Bytes} \wedge [\![e]\!]^\rho = 0 \\
[\![\mathsf{seqn}(y, e)]\!]^\rho &= y : [\![\mathsf{seqn}(y, e-1)]\!]^\rho && \text{if } y \in \textsc{Bytes} \wedge [\![e]\!]^\rho > 0 \\
[\![\mathsf{skipseq}(e, e')]\!]^\rho &= [\![e]\!]^\rho && \text{if } [\![e]\!]^\rho \in \textsc{FileData} \wedge [\![e']\!]^\rho = 0 \\
[\![\mathsf{skipseq}(y : e, e')]\!]^\rho &= [\![\mathsf{skipseq}(e), e' - 1]\!]^\rho && \text{if } [\![e]\!]^\rho \in \textsc{FileData} \wedge [\![e']\!]^\rho > 0 \\
[\![\mathsf{seqtake}(e, e')]\!]^\rho &= \epsilon && \text{if } [\![e]\!]^\rho \in \textsc{FileData} \wedge [\![e']\!]^\rho = 0 \\
[\![\mathsf{seqtake}(y : e, e')]\!]^\rho &= y : [\![\mathsf{seqtake}(e), e' - 1]\!]^\rho && \text{if } [\![e]\!]^\rho \in \textsc{FileData} \wedge [\![e']\!]^\rho > 0 \\
[\![\mathsf{subseq}(e, e', e'')]\!]^\rho &= [\![\mathsf{seqtake}(\mathsf{skipseq}(e, e'), e'')]\!]^\rho \\
[\![\mathsf{zeroise}(\epsilon)]\!]^\rho &= \epsilon \\
[\![\mathsf{zeroise}(y : e)]\!]^\rho &= 0 : [\![\mathsf{zeroise}(e)]\!]^\rho && \text{if } y = \emptyset \wedge [\![e]\!]^\rho \in \textsc{FileData} \\
[\![\mathsf{zeroise}(y : e)]\!]^\rho &= y : [\![\mathsf{zeroise}(e)]\!]^\rho && \text{if } y \neq \emptyset \wedge [\![e]\!]^\rho \in \textsc{FileData} \\
[\![e \upharpoonleft e']\!]^\rho &= [\![e']\!]^\rho :: [\![\mathsf{subseq}(e, \mathsf{len}(e'), \mathsf{len}(e) - \mathsf{len}(e'))]\!]^\rho && \text{if } [\![e]\!]^\rho, [\![e']\!]^\rho \in \textsc{FileData} \\
[\![e[e' \leftarrow e'']]\!]^\rho &= [\![\mathsf{subseq}(e, 0, e')]\!]^\rho :: [\![e'']\!]^\rho && \text{if } [\![e]\!]^\rho, [\![e'']\!]^\rho \in \textsc{FileData} \wedge [\![e']\!]^\rho \in \mathbb{N} \\
& && \quad \wedge [\![e' \leq \mathsf{len}(e)]\!]^\rho \\
[\![e[e' \leftarrow e'']]\!]^\rho &= [\![e]\!]^\rho :: [\![\mathsf{seqn}(\emptyset, e' - \mathsf{len}(e))]\!]^\rho :: [\![e'']\!]^\rho && \text{if } [\![e]\!]^\rho, [\![e']\!]^\rho \in \textsc{FileData} \wedge [\![e']\!]^\rho \in \mathbb{N} \\
& && \quad \wedge [\![e' > \mathsf{len}(e)]\!]^\rho \\
[\![e[e', e'']]\!]^\rho &= [\![\mathsf{zeroise}(\mathsf{subseq}(e, e', e''))]\!]^\rho && \text{if } [\![e]\!]^\rho \in \textsc{FileData} \wedge [\![e']\!]^\rho, [\![e'']\!]^\rho \in \mathbb{N} \\
& && \quad \wedge [\![e' + e'' \leq \mathsf{len}(e)]\!]^\rho \\
[\![e[e', e'']]\!]^\rho &= [\![\mathsf{zeroise}(\mathsf{subseq}(e, e', \mathsf{len}(e) - e'))]\!]^\rho && \text{if } [\![e]\!]^\rho \in \textsc{FileData} \wedge [\![e']\!]^\rho, [\![e'']\!]^\rho \in \mathbb{N} \\
& && \quad \wedge [\![e' + e'' > \mathsf{len}(e)]\!]^\rho
\end{aligned}
$$

Additionally, we extend Val with sets of values. We extend expressions and their evaluation with set operations in the standard manner.

A.1. Path Resolution

```
letrec resolve(path, ι) ≜
  if path = null then return ι else
    let a = head(path);
    let p = tail(path);
    let r = link_lookup(ι, a);
    if iserr(r) then return r
    else return resolve(p, r) fi
  fi
```

The `link_lookup` operations is defined in section A.2.

A.2. Operations on Links

```
stat(path)
  ⊑ let p = dirname(path);
    let a = basename(path);
    let r = resolve(p, ι0);
    if ¬iserr(r) then
      return link_stat(r, a)
    else return r fi
```

```
link(source, target)
  ⊑ let p_s = dirname(source);
    let a = basename(source);
    let p_t = dirname(target);
    let b = basename(target);
    let r_s, r_t = resolve(p_s, ι0) ‖ resolve(p_t, ι0);
    if ¬iserr(r_s) ∧ ¬iserr(r_t) then
      return link_insert(r_s, a, r_t, b)
             ⊔ link_insert_notdir(r_s, a)
    else if iserr(r_s) ∧ ¬iserr(r_t) then return r_s
    else if ¬iserr(r_s) ∧ iserr(r_t) then return r_t
    else if iserr(r_s) ∧ iserr(r_t) then return r_s ⊔ return r_t fi
```

```
unlink(path)
    let p = dirname(path);
    let a = basename(path);
    let r = resolve(p, ι0);
  ⊑ if ¬iserr(r) then
      return link_delete(r, a)
             ⊔ link_delete_notdir(r, a)
    else return r fi
```

$$
\begin{array}{l}
\texttt{rename}(source, target) \\
\quad \sqsubseteq \textbf{let } p_s = \texttt{dirname}(source); \\
\qquad \textbf{let } a = \texttt{basename}(source); \\
\qquad \textbf{let } p_t = \texttt{dirname}(target); \\
\qquad \textbf{let } b = \texttt{basename}(target); \\
\qquad \textbf{let } r_s, r_t = \texttt{resolve}(p_s, \iota_0) \parallel \texttt{resolve}(p_t, \iota_0); \\
\qquad \textbf{if } \neg\mathsf{iserr}(r_s) \wedge \neg\mathsf{iserr}(r_t) \textbf{ then} \\
\qquad\quad \texttt{return //Success cases} \\
\qquad\qquad\quad \texttt{link\_move\_noop}(r_s, a, r_t, b) \\
\qquad\qquad \sqcap \texttt{link\_move\_file\_target\_not\_exists}(r_s, a, r_t, b) \\
\qquad\qquad \sqcap \texttt{link\_move\_file\_target\_exists}(r_s, a, r_t, b) \\
\qquad\qquad \sqcap \texttt{link\_move\_dir\_target\_not\_exists}(r_s, a, r_t, b) \\
\qquad\qquad \sqcap \texttt{link\_move\_dir\_target\_exists}(r_s, a, r_t, b) \\
\qquad\qquad \texttt{//Error cases} \\
\qquad\qquad \sqcap \texttt{enoent}(r_s, a) \\
\qquad\qquad \sqcap \texttt{enotdir}(r_s) \\
\qquad\qquad \sqcap \texttt{enotdir}(r_t) \\
\qquad\qquad \sqcap \texttt{err\_source\_isfile\_target\_isdir}(r_s, a, r_t, b) \\
\qquad\qquad \sqcap \texttt{err\_source\_isdir\_target\_isfile}(r_s, a, r_t, b) \\
\qquad\qquad \sqcap \texttt{err\_target\_notempty}(r_t, b) \\
\qquad\qquad \sqcap \texttt{err\_target\_isdescendant}(r_s, a, r_t, b) \\
\qquad \textbf{else if } \mathsf{iserr}(r_s) \wedge \neg\mathsf{iserr}(r_t) \textbf{ then } \texttt{return } r_s \\
\qquad \textbf{else if } \neg\mathsf{iserr}(r_s) \wedge \mathsf{iserr}(r_t) \textbf{ then } \texttt{return } r_t \\
\qquad \textbf{else if } \mathsf{iserr}(r_s) \wedge \mathsf{iserr}(r_t) \textbf{ then } \texttt{return } r_s \sqcup \texttt{return } r_t \textbf{ fi}
\end{array}
$$

$$
\begin{array}{l}
\textbf{let } \texttt{link\_lookup}(\iota, a) \triangleq \\
\quad \forall FS.\, \langle \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota))\,,\ a \in FS(\iota) \Rightarrow \mathsf{fs}(FS) * \texttt{ret} = FS(\iota)(a) \rangle \\
\quad \sqcap \texttt{return enoent}(\iota, a) \\
\quad \sqcap \texttt{return enotdir}(\iota)
\end{array}
$$

$$
\begin{array}{l}
\texttt{link\_stat}(\iota, a) \triangleq \\
\quad \forall FS.\, \langle \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota))\,,\ a \in FS(\iota) \Rightarrow \mathsf{fs}(FS) * \texttt{ret} = \mathsf{ftype}(FS(FS(\iota)(a))) \rangle \\
\quad \sqcap \texttt{return enotdir}(\iota) \sqcap \texttt{return enoent}(\iota, a)
\end{array}
$$

$$
\begin{array}{l}
\textbf{let } \texttt{link\_delete}(\iota, a) \triangleq \\
\quad \forall FS.\, \langle \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota))\,,\ a \in FS(\iota) \Rightarrow \mathsf{fs}(FS[\iota \mapsto FS(\iota) \setminus \{a\}]) * \texttt{ret} = 0 \rangle \\
\quad \sqcap \texttt{return enoent}(\iota, a) \\
\quad \sqcap \texttt{return enotdir}(\iota)
\end{array}
$$

$$\textbf{let}\ \texttt{link\_delete\_notdir}(\iota, a) \triangleq$$
$$\forall FS.\ \langle \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota))\,,\ \mathsf{isfile}(FS(FS(\iota)(a))) \Rightarrow \mathsf{fs}(FS[\iota \mapsto FS(\iota) \setminus \{a\}]) * \texttt{ret} = 0 \rangle$$
$$\sqcap\ \texttt{return enoent}(\iota, a)$$
$$\sqcap\ \texttt{return enotdir}(\iota)$$
$$\sqcap\ \texttt{return err\_nodir\_hlinks}(\iota, a)$$

$$\textbf{let}\ \texttt{link\_insert}(\iota, a, j, b) \triangleq$$
$$\forall FS.\ \left\langle \begin{array}{l} \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota)) \wedge \mathsf{isdir}(FS(j))\,, \\ a \in FS(\iota) \wedge b \notin FS(j) \Rightarrow \mathsf{fs}(FS[j \mapsto FS(j)[b \mapsto FS(\iota)(a)]]) * \texttt{ret} = 0 \end{array} \right\rangle$$
$$\sqcap\ \texttt{return enoent}(\iota, a)$$
$$\sqcap\ \texttt{return eexist}(j, b)$$
$$\sqcap\ \texttt{return enotdir}(\iota)$$
$$\sqcap\ \texttt{return enotdir}(j)$$

$$\textbf{let}\ \texttt{link\_insert\_notdir}(\iota, a, j, b) \triangleq$$
$$\forall FS.\ \left\langle \begin{array}{l} \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota)) \wedge \mathsf{isdir}(FS(j))\,, \\ \mathsf{isfile}(FS(FS(\iota)(a))) \wedge b \notin FS(j) \Rightarrow \mathsf{fs}(FS[j \mapsto FS(j)[b \mapsto FS(\iota)(a)]]) * \texttt{ret} = 0 \end{array} \right\rangle$$
$$\sqcap\ \texttt{return enoent}(\iota, a)$$
$$\sqcap\ \texttt{return eexist}(j, b)$$
$$\sqcap\ \texttt{return enotdir}(\iota)$$
$$\sqcap\ \texttt{return enotdir}(j)$$
$$\sqcap\ \texttt{return err\_nodir\_hlinks}(\iota, a)$$

$$\textbf{let}\ \texttt{link\_move\_noop}(\iota_s, a, \iota_t, b) \triangleq$$
$$\forall FS.\ \langle \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota_s)) \wedge \mathsf{isdir}(FS(\iota_t))\,,\ FS(\iota_s)(a) = FS(\iota_t)(b) \Rightarrow \mathsf{fs}(FS) * \texttt{ret} = 0 \rangle$$

$$\textbf{let}\ \texttt{link\_move\_file\_target\_not\_exists}(\iota_s, a, \iota_t, b) \triangleq$$
$$\forall FS.\ \left\langle \begin{array}{l} \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota_s)) \wedge \mathsf{isdir}(FS(\iota_t))\,, \\ \mathsf{isfile}(FS(FS(\iota_s)(a))) \wedge b \notin FS(\iota_t) \\ \qquad \Rightarrow \mathsf{fs}(FS[\iota_s \mapsto FS(\iota_s) \setminus \{a\}][\iota_t \mapsto FS(\iota_t)[b \mapsto FS(\iota_s)(a)]]) * \texttt{ret} = 0 \end{array} \right\rangle$$

$$\textbf{let}\ \texttt{link\_move\_file\_target\_exists}(\iota_s, a, \iota_t, b) \triangleq$$
$$\forall FS.\ \left\langle \begin{array}{l} \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota_s)) \wedge \mathsf{isdir}(FS(\iota_t))\,, \\ \mathsf{isfile}(FS(FS(\iota_s)(a))) \wedge \mathsf{isfile}(FS(FS(\iota_t)(b))) \\ \qquad \Rightarrow \mathsf{fs}(FS[\iota_s \mapsto FS(\iota_s) \setminus \{a\}][\iota_t \mapsto FS(\iota_t)[b \mapsto FS(\iota_s)(a)]]) * \texttt{ret} = 0 \end{array} \right\rangle$$

$$\textbf{let } \texttt{link\_move\_dir\_target\_not\_exists}(\iota_s, a, \iota_t, b) \triangleq$$

$$\mathbb{V}FS. \left\langle \begin{array}{l} \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota_s)) \wedge \mathsf{isdir}(FS(\iota_t)), \\ \mathsf{isdir}(FS(FS(\iota_s)(a))) \wedge \iota_t \notin \mathsf{descendants}(FS, \iota_s) \wedge b \notin FS(\iota_t) \\ \quad \Rightarrow \exists FS'.\, FS' = FS[\iota_s \mapsto FS(\iota_s) \setminus \{a\}][\iota_t \mapsto FS(\iota_t)[b \mapsto FS(\iota_s)(a)]] \\ \qquad \wedge (FS'(\iota_t)(b)(\text{“..”}) \neq \iota_s \Rightarrow \mathsf{fs}(FS')) \\ \qquad \wedge (FS'(\iota_t)(b)(\text{“..”}) = \iota_s \Rightarrow \mathsf{fs}(FS'[FS'(\iota_t)(b) \mapsto FS'(\iota_t)(b)[\text{“..”} \mapsto \iota_t]])) \\ \qquad * \texttt{ret} = 0 \end{array} \right\rangle$$

$$\textbf{let } \texttt{link\_move\_dir\_target\_exists}(\iota_s, a, \iota_t, b) \triangleq$$

$$\mathbb{V}FS. \left\langle \begin{array}{l} \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota_s)) \wedge \mathsf{isdir}(FS(\iota_t)), \\ \mathsf{isdir}(FS(FS(\iota_s)(a))) \wedge \iota_t \notin \mathsf{descendants}(FS, \iota_s) \wedge \mathsf{isempdir}(FS(FS(\iota_t)(b))) \\ \quad \Rightarrow \exists FS'.\, FS' = FS[\iota_s \mapsto FS(\iota_s) \setminus \{a\}][\iota_t \mapsto FS(\iota_t)[b \mapsto FS(\iota_s)(a)]] \\ \qquad \wedge (FS'(\iota_t)(b)(\text{“..”}) \neq \iota_s \Rightarrow \mathsf{fs}(FS')) \\ \qquad \wedge (FS'(\iota_t)(b)(\text{“..”}) = \iota_s \Rightarrow \mathsf{fs}(FS'[FS'(\iota_t)(b) \mapsto FS'(\iota_t)(b)[\text{“..”} \mapsto \iota_t]])) \\ \qquad * \texttt{ret} = 0 \end{array} \right\rangle$$

$$\textbf{let } \texttt{enoent}(\iota, a) \triangleq \mathbb{V}FS. \langle \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota)),\ a \notin FS(\iota) \Rightarrow \mathsf{fs}(FS) * \texttt{ret} = \texttt{ENOENT} \rangle$$

$$\textbf{let } \texttt{enotdir}(\iota) \triangleq \mathbb{V}FS. \langle \mathsf{fs}(FS) \wedge \neg\mathsf{isdir}(FS(\iota)),\ \mathsf{fs}(FS) * \texttt{ret} = \texttt{ENOTDIR} \rangle$$

$$\textbf{let } \texttt{eexist}(\iota, a) \triangleq \mathbb{V}FS. \langle \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota)),\ a \in FS(\iota) \Rightarrow \mathsf{fs}(FS) * \texttt{ret} = \texttt{EEXIST} \rangle$$

$$\textbf{let } \texttt{err\_nodir\_hlinks}(\iota, a) \triangleq$$
$$\mathbb{V}FS. \langle \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota)),\ \mathsf{isdir}(FS(FS(\iota)(a))) \Rightarrow \mathsf{fs}(FS) * \texttt{ret} = \texttt{EPERM} \rangle$$

$$\textbf{let } \texttt{err\_source\_isfile\_target\_isdir}(\iota_s, a, \iota_t, b) \triangleq$$
$$\mathbb{V}FS. \left\langle \begin{array}{l} \mathsf{fs}(FS) \wedge \iota_s \in FS \wedge \iota_t \in FS, \\ \mathsf{isfile}(FS(FS(\iota_s)(a))) \wedge \mathsf{isdir}(FS(FS(\iota_t)(b))) \Rightarrow \mathsf{fs}(FS) * \texttt{ret} = \texttt{EISDIR} \end{array} \right\rangle$$

$$\textbf{let } \texttt{err\_source\_isdir\_target\_isfile}(\iota_s, a, \iota_t, b) \triangleq$$
$$\mathbb{V}FS. \left\langle \begin{array}{l} \mathsf{fs}(FS) \wedge \iota_s \in FS \wedge \iota_t \in FS, \\ \mathsf{isdir}(FS(FS(\iota_s)(a))) \wedge \mathsf{isfile}(FS(FS(\iota_t)(b))) \Rightarrow \mathsf{fs}(FS) * \texttt{ret} = \texttt{ENOTDIR} \end{array} \right\rangle$$

$$\textbf{let } \texttt{err\_target\_notempty}(\iota_t, b) \triangleq$$
$$\mathbb{V}FS. \langle \mathsf{fs}(FS) \wedge \iota_t \in FS,\ \mathsf{isempdir}(FS(FS(\iota_t)(b))) \Rightarrow \mathsf{fs}(FS) * \texttt{ret} \in \{\texttt{EEXIST}, \texttt{ENOTEMPTY}\} \rangle$$

$$\textbf{let } \texttt{err\_target\_isdescendant}(\iota_s, a, \iota_t, b) \triangleq$$
$$\mathbb{V}FS. \langle \mathsf{fs}(FS) \wedge \iota_s \in FS \wedge \iota_t \in FS,\ \iota_t \in \mathsf{descendants}(FS, FS(\iota_s)(a)) \Rightarrow \mathsf{fs}(FS) * \texttt{ret} = \texttt{EINVAL} \rangle$$

A.3. Operations on Directories

$\texttt{mkdir}(path)$
$\quad \sqsubseteq$ **let** $p = \texttt{dirname}(path)$;
$\quad\quad$ **let** $a = \texttt{basename}(path)$;
$\quad\quad$ **let** $r = \texttt{resolve}(p, \iota_0)$;
$\quad\quad$ **if** $\neg\mathsf{iserr}(r)$ **then**
$\quad\quad\quad$ `return` $\texttt{link\_new\_dir}(r, a)$
$\quad\quad\quad\quad\quad \sqcap\, \texttt{eexist}(\iota, a)$
$\quad\quad\quad\quad\quad \sqcap\, \texttt{enotdir}(\iota)$
$\quad\quad$ **else** `return` r **fi**

$\texttt{rmdir}(path)$
$\quad \sqsubseteq$ **let** $p = \texttt{dirname}(path)$;
$\quad\quad$ **let** $a = \texttt{basename}(path)$;
$\quad\quad$ **let** $r = \texttt{resolve}(p, \iota_0)$;
$\quad\quad$ **if** $\neg\mathsf{iserr}(r)$ **then**
$\quad\quad\quad$ `return` $\texttt{link\_del\_dir}(r, a)$
$\quad\quad\quad\quad\quad \sqcap\, \texttt{enoent}(\iota, a)$
$\quad\quad\quad\quad\quad \sqcap\, \texttt{enotdir}(\iota)$
$\quad\quad$ **else** `return` r **fi**

$$\textbf{let}\ \texttt{link\_new\_dir}(\iota, a) \triangleq$$
$$\forall FS.\left\langle \begin{array}{l} \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota))\,, \\ a \notin FS(\iota) \Rightarrow \exists \iota'.\, \mathsf{fs}(FS[\iota \mapsto FS(\iota)[a \mapsto \iota']] \uplus \iota' \mapsto \emptyset[\text{“.”} \mapsto \iota'][\text{“..”} \mapsto \iota]) * \texttt{ret} = 0 \end{array} \right\rangle$$

$$\textbf{let}\ \texttt{link\_del\_dir}(\iota, a) \triangleq$$
$$\forall FS.\, \langle \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota))\,,\ \mathsf{isdir}(FS(FS(\iota)(a))) \Rightarrow \mathsf{fs}(FS[\iota \mapsto FS(\iota) \setminus \{a\}]) * \texttt{ret} = 0 \rangle$$

A.4. I/O Operations on Regular Files

$$\texttt{open}(\mathit{path}, \mathit{flags})$$
$$\sqsubseteq \textbf{let } p = \texttt{dirname}(\mathit{path});$$
$$\textbf{let } a = \texttt{basename}(\mathit{path});$$
$$\textbf{let } r = \texttt{resolve}(p, \iota_0);$$
$$\textbf{if } \neg\mathsf{iserr}(r) \textbf{ then}$$
$$\textbf{if } \texttt{O\_CREAT} \in \mathit{flags} \wedge \texttt{O\_EXCL} \in \mathit{flags} \textbf{ then}$$
$$\texttt{return link\_new\_file}(r, a, \mathit{flags})$$
$$\sqcap\ \texttt{eexist}(r, a)$$
$$\textbf{else if } \texttt{O\_CREAT} \in \mathit{flags} \wedge \texttt{O\_EXCL} \notin \mathit{flags} \textbf{ then}$$
$$\texttt{return link\_new\_file}(r, a, \mathit{flags})$$
$$\sqcap\ \texttt{open\_file}(r, a, \mathit{flags})$$
$$\textbf{else}$$
$$\texttt{return open\_file}(r, a, \mathit{flags})$$
$$\sqcap\ \texttt{enoent}(r, a)$$
$$\textbf{fi}$$
$$\sqcap\ \texttt{return enotdir}(r)$$
$$\textbf{else return } r \textbf{ fi}$$

$$\texttt{close}(\mathit{fd}) \sqsubseteq \langle \mathsf{fd}(\mathit{fd}, -, -, -)\ ,\ \mathsf{true} \rangle$$

$$\texttt{write}(\mathit{fd}, \mathit{ptr}, \mathit{sz}) \sqsubseteq \texttt{return write\_off}(\mathit{fd}, \mathit{ptr}, \mathit{sz}) \sqcap \texttt{write\_badf}(\mathit{fd})$$

$$\texttt{read}(\mathit{fd}, \mathit{ptr}, \mathit{sz}) \sqsubseteq \texttt{return read\_norm}(\mathit{fd}, \mathit{ptr}, \mathit{sz}) \sqcap \texttt{read\_badf}(\mathit{fd})$$

$$\texttt{lseek}(\mathit{fd}, \mathit{off}, \mathit{whence})$$
$$\sqsubseteq \textbf{if } \mathit{whence} = \texttt{SEEK\_SET} \textbf{ then}$$
$$\texttt{return lseek\_set}(\mathit{fd}, \mathit{off})$$
$$\textbf{else if } \mathit{whence} = \texttt{SEEK\_CUR} \textbf{ then}$$
$$\texttt{return lseek\_cur}(\mathit{fd}, \mathit{off})$$
$$\textbf{else if } \mathit{whence} = \texttt{SEEK\_END} \textbf{ then}$$
$$\texttt{return lseek\_end}(\mathit{fd}, \mathit{off})$$
$$\textbf{fi}$$

$$\texttt{pwrite}(\mathit{fd}, \mathit{ptr}, \mathit{sz}, \mathit{off}) \sqsubseteq \texttt{pwrite\_off}(\mathit{fd}, \mathit{ptr}, \mathit{sz}, \mathit{off}) \sqcap \texttt{write\_badf}(\mathit{fd})$$

$$\texttt{pread}(\mathit{fd}, \mathit{ptr}, \mathit{sz}, \mathit{off}) \sqsubseteq \texttt{pread\_norm}(\mathit{fd}, \mathit{ptr}, \mathit{sz}, \mathit{off}) \sqcap \texttt{read\_badf}(\mathit{fd})$$

$$\textbf{let } \texttt{link\_new\_file}(\iota, a, \mathit{flags}) \triangleq$$
$$\forall FS. \left\langle \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota)),\ a \notin FS(\iota) \Rightarrow \begin{array}{c} \exists \iota'.\ \mathsf{fs}(FS[\iota \mapsto FS(\iota)[a \mapsto \iota']] \uplus \iota' \mapsto \epsilon) \\ * \mathsf{fd}(\texttt{ret}, \iota', 0, \mathsf{fdflags}(\mathit{flags})) \end{array} \right\rangle$$

$$\textbf{let } \texttt{open\_file}(\iota, a, \mathit{flags}) \triangleq$$
$$\forall FS.\, \langle \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota))\,,\ \mathsf{isfile}(FS(FS(\iota)(a))) \Rightarrow \mathsf{fs}(FS) * \mathsf{fd}(\texttt{ret}, FS(\iota)(a), 0, \mathsf{fdflags}(\mathit{flags}))\rangle$$

$$\textbf{let } \texttt{write\_off}(\mathit{fd}, \mathit{ptr}, \mathit{sz}) \triangleq$$
$$\forall FS, o \in \mathbb{N}.\, \left\langle \begin{array}{l} \mathsf{fs}(FS) \wedge \mathsf{isfile}(FS(\iota)) * \mathsf{fd}(\mathit{fd}, \iota, o, \mathit{fl}) \wedge \mathsf{iswrfd}(\mathit{fl}) * \mathsf{buf}\big(\mathit{ptr}, \overline{b}\big) \wedge \mathsf{len}\big(\overline{b}\big) = \mathit{sz}, \\ \mathsf{fs}\big(FS[\iota \mapsto FS(\iota)[o \leftarrow \overline{b}]]\big) * \mathsf{fd}(\mathit{fd}, \iota, o + \mathit{sz}, \mathit{fl}) * \mathsf{buf}\big(\mathit{ptr}, \overline{b}\big) * \texttt{ret} = \mathit{sz} \end{array} \right\rangle$$

$$\textbf{let } \texttt{write\_badf}(\mathit{fd}) \triangleq \forall o \in \mathbb{N}.\, \langle \mathsf{fd}(\mathit{fd}, \iota, o, \mathit{fl}) \wedge \texttt{O\_RDONLY} \in \mathit{fl},\ \mathsf{fd}(\mathit{fd}, \iota, o, \mathit{fl}) * \texttt{ret} = \texttt{EBADF}\rangle$$

$$\textbf{let } \texttt{read\_norm}(\mathit{fd}, \mathit{ptr}, \mathit{sz}) \triangleq$$
$$\forall FS, o \in \mathbb{N}.\, \left\langle \begin{array}{l} \mathsf{fs}(FS) \wedge \mathsf{isfile}(FS(\iota)) * \mathsf{fd}(\mathit{fd}, \iota, o, \mathit{fl}) * \mathsf{buf}\big(\mathit{ptr}, \overline{b_s}\big) \wedge \mathsf{len}\big(\overline{b_s}\big) = \mathit{sz}, \\ \exists \overline{b_t}.\, \mathsf{fs}(FS) * \mathsf{fd}(\mathit{fd}, \iota, o + \texttt{ret}, \mathit{fl}) * \mathsf{buf}\big(\mathit{ptr}, \overline{b_s} \upharpoonright \overline{b_t}\big) \wedge \overline{b_t} = FS(\iota)[o, \mathit{sz}] * \texttt{ret} = \mathsf{len}\big(\overline{b_t}\big) \end{array} \right\rangle$$

$$\textbf{let } \texttt{read\_badf}(\mathit{fd}) \triangleq$$
$$\forall o \in \mathbb{N}.\, \langle \mathsf{fd}(\mathit{fd}, \iota, o, \mathit{fl}) \wedge \mathsf{testflag}(\mathit{fl}, \texttt{O\_WRONLY})\,,\ \mathsf{fd}(\mathit{fd}, \iota, o, \mathit{fl}) * \texttt{ret} = \texttt{EBADF}\rangle$$

$$\textbf{let } \texttt{lseek\_set}(\mathit{fd}, \mathit{off}) \triangleq$$
$$\langle \mathsf{fd}(\mathit{fd}, \iota, -, \mathit{fl})\,,\ \mathsf{fd}(\mathit{fd}, \iota, \mathit{off}, \mathit{fl}) * \texttt{ret} = \mathit{off}\rangle$$

$$\textbf{let } \texttt{lseek\_cur}(\mathit{fd}, \mathit{off}) \triangleq$$
$$\forall o \in \mathbb{N}.\, \langle \mathsf{fd}(\mathit{fd}, \iota, o, \mathit{fl})\,,\ \mathsf{fd}(\mathit{fd}, \iota, \texttt{ret}, \mathit{fl}) * \texttt{ret} = o + \mathit{off}\rangle$$

$$\textbf{let } \texttt{lseek\_end}(\mathit{fd}, \mathit{off}) \triangleq$$
$$\forall FS.\, \langle \mathsf{fs}(FS) \wedge \mathsf{isfile}(FS(\iota)) * \mathsf{fd}(\mathit{fd}, \iota, -, \mathit{fl})\,,\ \mathsf{fs}(FS) * \mathsf{fd}(\mathit{fd}, \iota, \texttt{ret}, \mathit{fl}) * \texttt{ret} = \mathsf{len}(FS(\iota)) + \mathit{off}\rangle$$

$$\textbf{let } \texttt{pwrite\_off}(\mathit{fd}, \mathit{ptr}, \mathit{sz}, \mathit{off}) \triangleq$$
$$\forall FS.\, \left\langle \begin{array}{l} \mathsf{fs}(FS) \wedge \mathsf{isfile}(FS(\iota)) * \mathsf{fd}(\mathit{fd}, \iota, -, \mathit{fl}) \wedge \mathsf{iswrfd}(\mathit{fl}) * \mathsf{buf}(\mathit{ptr}, \overline{y}) \wedge \mathsf{len}(\overline{y}) = \mathit{sz}, \\ \mathsf{fs}(FS[\iota \mapsto FS(\iota)[\mathit{off} \leftarrow \overline{y}]]) * \mathsf{fd}(\mathit{fd}, \iota, -, \mathit{fl}) \wedge \mathsf{iswrfd}(\mathit{fl}) * \mathsf{buf}(\mathit{ptr}, \overline{y}) \wedge \mathsf{len}(\overline{y}) = \mathit{sz} * \texttt{ret} = \mathit{sz} \end{array} \right\rangle$$

$$\textbf{let } \texttt{pread\_norm}(\mathit{fd}, \mathit{ptr}, \mathit{sz}, \mathit{off}) \triangleq$$
$$\forall FS.\, \left\langle \begin{array}{l} \mathsf{fs}(FS) \wedge \mathsf{isfile}(FS(\iota)) * \mathsf{fd}(\mathit{fd}, \iota, -, \mathit{fl}) \wedge \mathsf{isrdfd}(\mathit{fl}) * \mathsf{buf}(\mathit{ptr}, \overline{y}) \wedge \mathsf{len}(\overline{y}) = \mathit{sz}, \\ \exists \overline{y_t}.\, \mathsf{fs}(FS) * \mathsf{fd}(\mathit{fd}, \iota, -, \mathit{fl}) * \mathsf{buf}(\mathit{ptr}, \overline{y} \upharpoonright \overline{y_t}) \wedge \overline{y_t} = FS(\iota)[\mathit{off}, \mathit{sz}] * \texttt{ret} = \mathsf{len}(\overline{y_t}) \end{array} \right\rangle$$

A.5. I/O Operations on Directories

We specify `readdir` capturing the high non-determinism allowed by the POSIX standard. If the directory contents are being changed while the directory is being opened with `opendir`, `readdir` may or may not see the changes, with the proviso that a filename is not returned more than once. To specify this, we define the directory stream to hold a look-ahead set of the filenames in the directory, of non-deterministic size. The specification of `readdir` non-deterministically chooses to either fill the look-ahead set and then return some filename from that set, or simply return a filename straight from that set. We use the abstract predicate $\mathsf{dirstr}(\mathit{ds}, \iota, V, L, \mathit{off})$ to denote a directory stream ds

opened for the directory with inode ι, with the set V denoting the filenames read so far, the set L denoting the look-ahead set, and *off* storing the number of filenames read so far. The abstract predicate $\mathsf{imp\_ord\_ents}(dir)$ returns a sequence of the names of the links in the directory dir in an implementation-defined order.

$$\begin{array}{l}\texttt{opendir}(path) \\ \quad \sqsubseteq \textbf{let } r = \texttt{resolve}(path, \iota_0); \\ \qquad \textbf{if } \neg\mathsf{iserr}(r) \textbf{ then} \\ \qquad\quad \texttt{return open\_dirstr}(r) \\ \qquad \textbf{else return } r \textbf{ fi}\end{array} \qquad \texttt{closedir}(ds) \sqsubseteq \langle \mathsf{dirstr}(ds, -, -, -, -)\,,\ \mathsf{true}\rangle$$

$$\texttt{readdir}(ds) \sqsubseteq \texttt{return readdir\_buff\_fill}(ds) \sqcup \texttt{return readdir\_from\_buff}(ds)$$

$$\begin{array}{l}\textbf{let } \texttt{open\_dirstr}(\iota) \triangleq \\ \quad \forall FS.\, \langle \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota))\,,\ \mathsf{fs}(FS) * \mathsf{dirstr}(\texttt{ret}, \iota, \emptyset, \emptyset)\rangle \\ \quad \sqcap \texttt{return enotdir}(\iota)\end{array}$$

$$\begin{array}{l}\textbf{let } \texttt{readdir\_buff\_fill}(ds) \triangleq \\ \quad \forall FS,\, V \in \mathcal{P}(\mathrm{FNames}), L \in \mathcal{P}(\mathrm{FNames}), \mathit{off} \in \mathbb{N}. \\ \quad \left\langle \begin{array}{l} \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota)) * \mathsf{dirstr}(ds, \iota,\, V, L, \mathit{off})\,, \\ \exists V', L', L'', n > 0.\, \mathsf{fs}(FS) * \mathsf{dirstr}(ds, \iota,\, V', L'' \setminus \{\texttt{ret}\}\,, \mathit{off}) \wedge L = \emptyset \\ \wedge\ L' = \mathsf{setofseq}(\mathsf{subseq}(\mathit{off}, n, \mathsf{imp\_ord\_ents}(FS(\iota)))) \\ \wedge\ (L' \neq \emptyset \Rightarrow V' = V \uplus \{\texttt{ret}\} \wedge L'' = L' \setminus \{\texttt{ret}\}) \\ \wedge\ (L' = \emptyset \Rightarrow V' = V \wedge L'' = L' \wedge \texttt{ret} = \texttt{null}) \end{array} \right\rangle \end{array}$$

$$\begin{array}{l}\textbf{let } \texttt{readdir\_from\_buff}(ds) \triangleq \\ \quad \forall V \in \mathcal{P}(\mathrm{FNames}), L \in \mathcal{P}(\mathrm{FNames}), \mathit{off} \in \mathbb{N}. \\ \quad \langle \mathsf{dirstr}(ds, \iota,\, V, L, \mathit{off})\,,\ \mathsf{dirstr}(ds, \iota,\, V \uplus \{\texttt{ret}\}\,, L \setminus \{\texttt{ret}\}\,, \mathit{off}) \wedge L \neq \emptyset\rangle\end{array}$$

B. Atomicity and Refinement Technical Appendix

B.1. Adequacy Addendum

In chapter 7, we have defined the semantics of our specification language and refinement both in terms of operational and denotational semantics. With theorem 1, we have established the soundness of denotational refinement with respect to contextual refinement based on the operational semantics. The proof of this theorem relies on lemma 5, which equates the traces obtained by the denotational semantics to the traces obtained by the operational semantics under the stuttering, mumbling and faulting closure.

To prove lemma 5, we establish inequality between operational and denotational traces in both directions. In appendix B.1.1, we prove that operational traces are contained within denotational traces and in appendix B.1.2 we prove the reverse. As a stepping stone, in both directions, we will work with *raw* traces, that are not closed by the stuttering, mumbling and faulting closure. This simplifies the proof process by avoiding the need for mumbling and stuttering, not only for lemma 5, but also for the refinement laws.

Before we proceed with the proof, we define the raw denotational semantics, and establish several crucial lemmas.

Definition 61 (Raw Denotational Semantics)**.** *The raw denotational semantics,* $\mathcal{R}[\![-]\!]^- : \textsc{VarStore}_\mu \to \mathcal{L} \to \mathcal{P}(\textsc{Trace})$*, map specification programs to sets of traces, within a variable environment.*

$$
\begin{aligned}
\mathcal{R}[\![\phi; \psi]\!]^\rho &\triangleq \mathcal{R}[\![\phi]\!]^\rho \,;\mathcal{R}[\![\psi]\!]^\rho \\
\mathcal{R}[\![\phi \parallel \psi]\!]^\rho &\triangleq \mathcal{R}[\![\phi]\!]^\rho \parallel \mathcal{R}[\![\psi]\!]^\rho \\
\mathcal{R}[\![\phi \sqcup \psi]\!]^\rho &\triangleq \mathcal{R}[\![\phi]\!]^\rho \cup \mathcal{R}[\![\psi]\!]^\rho \\
\mathcal{R}[\![\phi \sqcap \psi]\!]^\rho &\triangleq \mathcal{R}[\![\phi]\!]^\rho \cap \mathcal{R}[\![\psi]\!]^\rho \\
\mathcal{R}[\![\exists x.\, \phi]\!]^\rho &\triangleq \bigcup_{v \in \textsc{Val}} \mathcal{R}[\![\phi]\!]^{\rho[x \mapsto v]} \\
\mathcal{R}[\![\textbf{let } f = F \textbf{ in } \phi]\!]^\rho &\triangleq \mathcal{R}[\![\phi]\!]^{\rho[f \mapsto \mathcal{R}[\![F]\!]^\rho]} \\
\mathcal{R}[\![F e]\!]^\rho &\triangleq \mathcal{R}[\![F]\!]^\rho \, [\![e]\!]^\rho \\
\mathcal{R}[\![f]\!]^\rho &\triangleq \rho(f) \\
\mathcal{R}[\![A]\!]^\rho &\triangleq \rho(A) \\
\mathcal{R}[\![\mu A.\, \lambda x.\, \phi]\!]^\rho &\triangleq \bigcap \left\{ T_f \in \textsc{Val} \to \mathcal{P}(\textsc{Trace}) \;\middle|\; \mathcal{R}[\![\lambda x.\, \phi]\!]^{\rho[A \mapsto T_f]} \subseteq T_f \right\}
\end{aligned}
$$

$$\mathcal{R}[\![\lambda x.\, \phi]\!]^\rho \triangleq \lambda v.\, \mathcal{R}[\![\phi]\!]^{\rho[x \mapsto v]}$$

$$\mathcal{R}\left[\!\!\left[a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_k\right]\!\!\right]^\rho \triangleq \left\{(h,h') \in \textsc{Move} \;\middle|\; \vec{v} \in \overrightarrow{\textsc{Val}} \wedge h' \in \mathsf{a}\left((\!|P|\!)^{\rho[\vec{x}\mapsto\vec{v}]}_{\mathcal{A}}, (\!|Q|\!)^{\rho[\vec{x}\mapsto\vec{v}]}_{\mathcal{A}}\right)^{\mathcal{A}}_k (h)\right\}$$
$$\cup \left\{(h,\lightning) \in \textsc{Heap}^{\lightning} \;\middle|\; \begin{array}{l}\vec{v} \in \overrightarrow{\textsc{Val}} \wedge \mathsf{a}\left((\!|P|\!)^{\rho[\vec{x}\mapsto\vec{v}]}_{\mathcal{A}}, (\!|Q|\!)^{\rho[\vec{x}\mapsto\vec{v}]}_{\mathcal{A}}\right)^{\mathcal{A}}_k (h) = \emptyset \\ \wedge\ (\!|Q|\!)^{\rho[\vec{x}\mapsto\vec{v}]}_{\neq} \emptyset\end{array}\right\}$$
$$\cup \left\{(\lightning,\lightning)\right\}$$

The argument for the existence of the least fixpoint is the same as for the denotation semantics of definition 51.

Lemma 8. $(\lightning,\lightning) \in \mathcal{R}[\![\phi]\!]^\rho$

Proof. Straightforward induction on ϕ. $(\lightning,\lightning) \in \mathcal{R}\left[\!\!\left[a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_k\right]\!\!\right]^\rho$ by definition 61. All inductive cases follow immediately from the inductive hypothesis. □

Lemma 9 (Function and Recursion Substitution)**.** *If ψ is closed, then $\mathcal{R}[\![\phi]\!]^{\rho[y \mapsto \mathcal{R}[\![\psi]\!]^\rho]} = \mathcal{R}[\![\phi\,[\psi/y]]\!]^\rho$, where y is a recursion variable A, or a function variable f.*

Proof. Straightforward induction on ϕ. Base case $\mathcal{R}[\![A]\!]$ trivial. Base case $\mathcal{R}[\![f]\!]$ trivial. Base Case $\mathcal{R}\left[\!\!\left[a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_k\right]\!\!\right]$ trivial, as recursion and function variables or not free in P or Q. Inductive cases follow immediately from the induction hypothesis. □

Lemma 10 (Variable Substitution)**.** *If e is an expression, where x is not free, then $\mathcal{R}[\![\phi]\!]^{\rho[x \mapsto [\![e]\!]^\rho]} = \mathcal{R}[\![\phi\,[[\![e]\!]^\rho/x]]\!]^\rho$ and $[\![\phi]\!]^{\rho[x \mapsto [\![e]\!]^\rho]} = [\![\phi\,[[\![e]\!]^\rho/x]]\!]^\rho$*

Proof. Straightforward induction on ϕ. □

The semantics of recursion are given as the Tarskian least fixpoint. However, in some proof steps, the Kleenian least fixpoint is more useful. In order to switch to the Kleenian fixpoint we require continuity.

Lemma 11 (Raw Denotation Continuity)**.** $\mathcal{R}[\![\phi]\!]^{\rho[A \mapsto -]}$ *is Scott-continuous.*

Proof. $\mathcal{R}[\![\phi]\!]^{\rho[A \mapsto -]} : (\textsc{Val} \to \mathcal{P}(\textsc{Trace})) \to \mathcal{P}(\textsc{Trace})$.
Let $D \subseteq \textsc{Val} \to \mathcal{P}(\textsc{Trace})$. D is a directed subset of $\textsc{Val} \to \mathcal{P}(\textsc{Trace})$, due to the fact that $\textsc{Val} \to \mathcal{P}(\textsc{Trace})$ is a lattice by pointwise extension of the powerset lattice.
For Scott-continuity we show that: $\sqcup(\mathcal{R}[\![\phi]\!]^{\rho[A \mapsto -]})[D] = \mathcal{R}[\![\phi]\!]^{\rho[A \mapsto \sqcup D]}$ by induction on ϕ.
Base case: Ae.

$$
\begin{aligned}
\sqcup(\mathcal{R}[\![Ae]\!]^{\rho[A\mapsto -]})[D] &= \bigcup_{T_f \in D} \mathcal{R}[\![Ae]\!]^{\rho[A\mapsto T_f]} \\
&= \text{by definition 61} \\
&\quad \bigcup_{T_f \in D} T_f e \\
&= \text{by pointwise extension of the powerset lattice} \\
&\quad \left(\bigsqcup_{T_f \in D} T_f \right) e \\
&= (\sqcup D)\, e \\
&= \text{by definition 61} \\
&\quad \mathcal{R}[\![Ae]\!]^{\rho[A\mapsto \sqcup D]}
\end{aligned}
$$

Base cases $fe, a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_k$ not applicable as the recursion variable A does not appear in these cases.
Cases $\phi;\psi$, $\phi \parallel \psi$, $\phi \sqcup \psi$ and $\phi \sqcap \psi$ from induction hypothesis and by the fact that $;$, $\parallel$, $\cup$ and $\cap$ preserve continuity respectively.
All other cases follow straightforwardly from the inductive hypothesis.

□

The next three lemmas establish properties of the stuttering, mumbling and faulting closure that we rely on in several proof steps.

Lemma 12 (Closure operator)**.** $-^\dagger$ *is a closure operator:*

$$
\begin{aligned}
T \subseteq T^\dagger &\quad (-^\dagger \textit{ is extensive}) \\
T \subseteq U \Rightarrow T^\dagger \subseteq U^\dagger &\quad (-^\dagger \textit{ is increasing}) \\
(T^\dagger)^\dagger = T^\dagger &\quad (-^\dagger \textit{ is idempotent})
\end{aligned}
$$

Proof. **Idempotent:** $T^\dagger \subseteq (T^\dagger)^\dagger$ follows directly from rule (7.23).
We show that $(T^\dagger)^\dagger \subseteq T^\dagger$ by induction on the derivation of $t \in T^\dagger$.
Base cases:
Rule (7.24). $(\lightning, \lightning) \in T^{\dagger\dagger}$ and $(\lightning, \lightning) \in T^\dagger$.
Rule (7.23). Let $t \in T^{\dagger\dagger}$. By premiss, $t \in T^\dagger$.
Inductive cases:
Rule CLStutter. Let $s(h,h)t \in T^{\dagger\dagger}$. By premiss, $st \in T^{\dagger\dagger}$. By the inductive hypothesis, $st \in T^\dagger$, thus $s(h,h)t \in T^\dagger$.
Rule CLMumble. Let $s(h,o)t \in T^{\dagger\dagger}$. By premiss, $s(h,h')(h',o)t \in T^{\dagger\dagger}$. By the inductive hypothesis, $s(h,h')(h',o)t \in T^\dagger$, thus also $s(h,o)t \in T^\dagger$.
Rule (7.25). Let $t(h,h')u \in T^{\dagger\dagger}$. By premiss, $t(h,\lightning) \in T^{\dagger\dagger}$. By the inductive hypothesis, $t(h,\lightning) \in T^\dagger$, thus also $t(h,h')u \in T^\dagger$.
Increasing: By induction on the derivation of $t \in T^\dagger$.

Base case:
Rule (7.24). $(\{\},\{\}) \in T^\dagger \Rightarrow (\{\},\{\}) \in U^\dagger$ holds trivially.
Rule (7.23). Let $t \in T^\dagger$. By premiss, $t \in T$. By assumption, $t \in U$. Then, by rule (7.23), $t \in U^\dagger$.
Inductive cases:
Rule CLStutter. Let $s(h,h)t \in T^\dagger$. By premiss, $st \in T^\dagger$. By the induction hypothesis, $st \in U^\dagger$, from which it follows that $s(h,h)t \in U^\dagger$.
Rule CLMumble. Let $s(h,o)t \in T^\dagger$. By premiss, $s(h,h')(h',o)t \in T^\dagger$. By the induction hypothesis, $s(h,h')(h',o)t \in U^\dagger$, from which it follows that $s(h,o)t \in U^\dagger$.
Rule (7.25). Let $t(h,h')u \in T^\dagger$. By premiss, $t(h,\{\}) \in T^\dagger$. By the induction hypothesis, $t(h,\{\}) \in U^\dagger$, from which it follows that $t(h,h')u \in U^\dagger$.
Extensive: $\forall t.\, t \in T$, by rule (7.23), $t \in T^\dagger$. □

Lemma 13 (Trace Closure Distributivity)**.**

1. $T^\dagger; U^\dagger \subseteq (T;U)^\dagger$
2. $T^\dagger \parallel U^\dagger \subseteq (T \parallel U)^\dagger$
3. $\bigcup(T_i{}^\dagger) \subseteq (\bigcup T_i)^\dagger$
4. $\bigcap(T_i{}^\dagger) \supseteq (\bigcap T_i)^\dagger$

Proof. (1): First we show that $T^\dagger; U \subseteq (T;U)^\dagger$ by induction on the derivation of $t \in T^\dagger$. Fix $u \in U$.
Base cases:
Rule (7.24). $(\{\},\{\}) \in T^\dagger; U^\dagger \Rightarrow (\{\},\{\}) \in (T;U)^\dagger$ holds trivially.
Rule 7.23. Let $t \in T^\dagger$. By premiss, $t \in T$. By trace concatenation, $tu \in T;U$. Then, by rule 7.23, $tu \in (T;U)^\dagger$.
Inductive cases:
Rule CLStutter. Let $s(h,h)t \in T^\dagger$. By premiss, $st \in T^\dagger$. By trace concatenation, $stu \in T^\dagger; U$. Then, by the induction hypothesis, $stu \in (T;U)^\dagger$, from which it follows that $s(h,h)tu \in (T;U)^\dagger$.
Rule CLMumble. Let $s(h,o)t \in T^\dagger$. By premiss, $s(h,h')(h',o)t \in T^\dagger$. By trace concatenation, $s(h,h')(h',o)tu \in T^\dagger; U$. Then, by the induction hypothesis, $s(h,h')(h',o)tu \in (T;U)^\dagger$, from which it follows that $s(h,o)t \in (T;U)^\dagger$.
Rule (7.25). Let $t(h,h')u \in T^\dagger$. By premiss, $t(h,\{\}) \in T^\dagger$. By trace concatenation, $t(h,\{\})u = t(h,\{\}) \in T^\dagger; U$. Then, by the induction hypothesis, $t(h,\{\})u \in (T;U)^\dagger$, from which it follows that $t(h,h')u \in (T;U)^\dagger$.
Furthermore, $T; U^\dagger \subseteq (T;U)^\dagger$ by induction on the derivation of $u \in U^\dagger$ similarly.
Then, it follows that: $T^\dagger; U^\dagger \subseteq (T; U^\dagger)^\dagger \subseteq ((T;U)^\dagger)^\dagger$. Then, by idempotence $T^\dagger; U^\dagger \subseteq (T;U)^\dagger$.
(2): Similarly to (1).
(3): Fix index I, $n \in I$. By induction on the derivation of $t \in {T_n}^\dagger$.
Base case:
Rule (7.24). $(\{\},\{\}) \in \bigcup(T_i{}^\dagger) \iff (\{\},\{\}) \in (\bigcup T_i)^\dagger$ holds trivially.
Inductive case:
Rule (7.23). Let $t \in {T_n}^\dagger$, then $t \in \bigcup(T_i{}^\dagger)$. Then, by the induction hypothesis, $t \in (\bigcup T_i)^\dagger$.

Rule CLStutter. Let $s(h,h)t \in \bigcup(T_i{}^\dagger)$. By premiss, $st \in \bigcup(T_i{}^\dagger)$. Then, by the induction hypothesis, $st \in (\bigcup T_i)^\dagger$, from which it follows that $s(h,h)t \in (\bigcup T_i)^\dagger$.
Rule CLMumble. Let $s(h,o)t \in \bigcup(T_i{}^\dagger)$. By premiss, $s(h,h')(h',o)t \in \bigcup(T_i{}^\dagger)$. Then, by the induction hypothesis, $s(h,h')(h',o)t \in \bigcup(T_i)^\dagger$, from which it follows that $s(h,o)t \in \bigcup(T_i)^\dagger$.
Rule (7.25). Let $t(h,h')u \in \bigcup(T_i{}^\dagger)$. By premiss, $t(h,\lightning) \in \bigcup(T_i{}^\dagger)$. Then, by the induction hypothesis, $t(h,\lightning) \in \bigcup(T_i)^\dagger$, from which it follows $t(h,h')u \in \bigcup(T_i)^\dagger$.
(4): Similarly to (3). □

Lemma 14. $\left(\bigcap(T_i{}^\dagger)\right)^\dagger = \bigcap(T_i{}^\dagger)$

Proof. By lemma 12 (extensive): $\bigcap(T_i{}^\dagger) \subseteq \left(\bigcap(T_i{}^\dagger)\right)^\dagger$.
By lemma 13: $\bigcap\left(T_i{}^{\dagger\dagger}\right) \supseteq \left(\bigcap(T_i{}^\dagger)\right)^\dagger$.
By lemma 12 (idempotence): $\bigcap(T_i{}^\dagger) \supseteq \left(\bigcap(T_i{}^\dagger)\right)^\dagger$. □

The following lemma reflects the fact that the denotational semantics are idempotent with respect to trace closure.

Lemma 15. $([\![\phi]\!]^\rho)^\dagger = [\![\phi]\!]^\rho$

Proof. Straightforward induction on ϕ using lemma 12.
Base case: $a(\forall\vec{x}.\, P,\, Q)^{\mathcal{A}}_k$

$$\left(\left[\!\left[a(\forall\vec{x}.\, P,\, Q)^{\mathcal{A}}_k\right]\!\right]^\rho\right)^\dagger = \left(\left(\left\{(h,h') \in \textsc{Move} \;\middle|\; \vec{v} \in \overrightarrow{\textsc{Val}} \wedge h' \in \mathsf{a}\left((\!|P|\!)^{\rho[\vec{x}\mapsto\vec{v}]}_{\mathcal{A}}, (\!|Q|\!)^{\rho[\vec{x}\mapsto\vec{v}]}_{\mathcal{A}}\right)^{\mathcal{A}}_k(h)\right\} \cup \left\{(h,\lightning) \in \textsc{Heap}^{\lightning} \;\middle|\; \vec{v} \in \overrightarrow{\textsc{Val}} \wedge \mathsf{a}\left((\!|P|\!)^{\rho[\vec{x}\mapsto\vec{v}]}_{\mathcal{A}}, (\!|Q|\!)^{\rho[\vec{x}\mapsto\vec{v}]}_{\mathcal{A}}\right)^{\mathcal{A}}_k(h) = \emptyset \wedge (\!|Q|\!)^{\rho[\vec{x}\mapsto\vec{v}]}_{\mathcal{A}} \neq \emptyset\right\}\right)^\dagger\right)^\dagger$$

$$= \text{by lemma 12 (idempotence)}$$

$$\left(\left\{(h,h') \in \textsc{Move} \;\middle|\; \vec{v} \in \overrightarrow{\textsc{Val}} \wedge h' \in \mathsf{a}\left((\!|P|\!)^{\rho[\vec{x}\mapsto\vec{v}]}_{\mathcal{A}}, (\!|Q|\!)^{\rho[\vec{x}\mapsto\vec{v}]}_{\mathcal{A}}\right)^{\mathcal{A}}_k(h)\right\} \cup \left\{(h,\lightning) \in \textsc{Heap}^{\lightning} \;\middle|\; \vec{v} \in \overrightarrow{\textsc{Val}} \wedge \mathsf{a}\left((\!|P|\!)^{\rho[\vec{x}\mapsto\vec{v}]}_{\mathcal{A}}, (\!|Q|\!)^{\rho[\vec{x}\mapsto\vec{v}]}_{\mathcal{A}}\right)^{\mathcal{A}}_k(h) = \emptyset \wedge (\!|Q|\!)^{\rho[\vec{x}\mapsto\vec{v}]}_{\mathcal{A}} \neq \emptyset\right\}\right)^\dagger$$

$$= \left[\!\left[a(\forall\vec{x}.\, P,\, Q)^{\mathcal{A}}_k\right]\!\right]^\rho$$

Base case: A

$$\begin{aligned}(\![A]\!]^\rho)^\dagger &= \left(\rho(A)^\dagger\right)^\dagger \\ &= \text{by lemma 12 (idempotence)} \\ &\quad \rho(A)^\dagger = [\![A]\!]^\rho\end{aligned}$$

Base case: f, same as A
Case: $\phi;\psi$

$$
\begin{aligned}
(\llbracket \phi ; \psi \rrbracket^\rho)^\dagger &= \left((\llbracket \phi \rrbracket^\rho \,;\, \llbracket \psi \rrbracket^\rho)^\dagger \right)^\dagger \\
&= \text{by lemma 12} \\
&\quad \llbracket \phi ; \psi \rrbracket^\rho
\end{aligned}
$$

Case: $\phi \parallel \psi$ as previous.

Case: $\phi \sqcup \psi$

$$
\begin{aligned}
(\llbracket \phi \sqcup \psi \rrbracket^\rho)^\dagger &= \left((\llbracket \phi \rrbracket^\rho \cup \llbracket \psi \rrbracket^\rho)^\dagger \right)^\dagger \\
&= \text{by lemma 12} \\
&\quad \llbracket \phi \sqcup \psi \rrbracket^\rho
\end{aligned}
$$

Case: $\phi \sqcap \psi$

$$
\begin{aligned}
(\llbracket \phi \sqcap \psi \rrbracket^\rho)^\dagger &= \left((\llbracket \phi \rrbracket^\rho \cap \llbracket \psi \rrbracket^\rho)^\dagger \right)^\dagger \\
&= \text{by lemma 12} \\
&\quad \llbracket \phi \sqcap \psi \rrbracket^\rho
\end{aligned}
$$

Case: $\exists x.\, \phi$

$$
\begin{aligned}
(\llbracket \phi \rrbracket^\rho)^\dagger &= \left(\left(\bigcup_v \llbracket \phi \rrbracket^{\rho[x \mapsto v]} \right)^\dagger \right)^\dagger \\
&= \text{by lemma 12} \\
&\quad \llbracket \exists x.\, \phi \rrbracket^\rho
\end{aligned}
$$

Case: $\mu A.\, \lambda x.\, \phi$

$$
\begin{aligned}
(\llbracket \mu A.\, \lambda x.\, \phi \rrbracket^\rho)^\dagger &= \left(\bigcap \left\{ T_f \in \textsc{Val} \to \mathcal{P}(\textsc{Trace}) \;\middle|\; \left(\llbracket \lambda x.\, \phi \rrbracket^{\rho[A \mapsto T_f]} \right)^\dagger \subseteq {T_f}^\dagger \right\} \right)^\dagger \\
&= \text{by lemma 14} \\
&\quad \bigcap \left\{ T_f \in \textsc{Val} \to \mathcal{P}(\textsc{Trace}) \;\middle|\; \left(\llbracket \lambda x.\, \phi \rrbracket^{\rho[A \mapsto T_f]} \right)^\dagger \subseteq {T_f}^\dagger \right\} \\
&= \llbracket \mu A.\, \lambda x.\, \phi \rrbracket^\rho
\end{aligned}
$$

Case: **let** $f = F$ **in** ϕ

$$
\begin{aligned}
(\llbracket \textbf{let } f = F \textbf{ in } \phi \rrbracket^\rho)^\dagger &= \left(\llbracket \phi \rrbracket^{\rho[f \mapsto \llbracket F \rrbracket^\rho]} \right)^\dagger \\
&= \text{by induction hypothesis} \\
&\quad \llbracket \phi \rrbracket^{\rho[f \mapsto \llbracket F \rrbracket^\rho]} \\
&= \llbracket \textbf{let } f = F \textbf{ in } \phi \rrbracket^\rho
\end{aligned}
$$

Case: $\lambda x.\, \phi$

$$
\begin{aligned}
(\llbracket \lambda x.\, \phi \rrbracket^\rho)^\dagger &= \left(\lambda v.\, \llbracket \phi \rrbracket^{\rho[x \mapsto v]} \right)^\dagger \\
&= \text{by induction hypothesis} \\
&\quad \lambda v.\, \llbracket \phi \rrbracket^{\rho[x \mapsto v]} \\
&= \llbracket \lambda x.\, \phi \rrbracket^\rho
\end{aligned}
$$

Case: Fe

$$
\begin{aligned}
(\llbracket Fe \rrbracket^\rho)^\dagger &= (\llbracket F \rrbracket^\rho \, \llbracket e \rrbracket^\rho)^\dagger \\
&= \text{by induction hypothesis} \\
&\quad \llbracket F \rrbracket^\rho \, \llbracket e \rrbracket^\rho \\
&= \llbracket Fe \rrbracket^\rho
\end{aligned}
$$

□

The raw denotational semantics produce the denotational semantics by adding the trace closure, as indicated by the following lemma.

Lemma 16. $\llbracket \phi \rrbracket^\rho = (\mathcal{R} \llbracket \phi \rrbracket^\rho)^\dagger$.

Proof. First, we establish $(\mathcal{R} \llbracket \phi \rrbracket^\rho)^\dagger \subseteq \llbracket \phi \rrbracket^\rho$.

Trivially, $\mathcal{R} \llbracket \phi \rrbracket^\rho \subseteq \llbracket \phi \rrbracket^\rho$.

By lemma 12 (increasing), $(\mathcal{R} \llbracket \phi \rrbracket^\rho)^\dagger \subseteq (\llbracket \phi \rrbracket^\rho)^\dagger$.

By lemma 15, $(\mathcal{R} \llbracket \phi \rrbracket^\rho)^\dagger \subseteq \llbracket \phi \rrbracket^\rho$.

Second, we establish $(\mathcal{R} \llbracket \phi \rrbracket^\rho)^\dagger \supseteq \llbracket \phi \rrbracket^\rho$, by induction on ϕ.

Base case: $\left(\mathcal{R} \left\llbracket a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_k \right\rrbracket^\rho \right)^\dagger \supseteq \left\llbracket a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_k \right\rrbracket^\rho$ by the definitions.

Base case: A. $\mathcal{R} \llbracket A \rrbracket^{\rho\dagger} \supseteq \rho(A)^\dagger \supseteq \llbracket A \rrbracket^\rho$.

Base case: f, as previous.

Case: $\phi; \psi$.

$$
\begin{aligned}
(\mathcal{R}[\![\phi ; \psi]\!]^\rho)^\dagger &= (\mathcal{R}[\![\phi]\!]^\rho \,;\mathcal{R}[\![\psi]\!]^\rho)^\dagger \\
&\supseteq \text{by lemma 13} \\
&\quad (\mathcal{R}[\![\phi]\!]^\rho)^\dagger ; (\mathcal{R}[\![\psi]\!]^\rho)^\dagger \\
&\supseteq \text{by inductive hypothesis} \\
&\quad [\![\phi]\!]^\rho \,; [\![\psi]\!]^\rho \\
&\supseteq \text{by lemma 12 (increasing on both sides, idempotence on left of } \supseteq) \\
&\quad ([\![\phi]\!]^\rho \,; [\![\psi]\!]^\rho)^\dagger \\
&= \text{by definition 51} \\
&\quad [\![\phi ; \psi]\!]^\rho
\end{aligned}
$$

Case: $\phi \parallel \phi$, similar to previous.

Case: $\phi \sqcup \psi$

$$
\begin{aligned}
(\mathcal{R}[\![\phi \sqcup \psi]\!]^\rho)^\dagger &= (\mathcal{R}[\![\phi]\!]^\rho \cup \mathcal{R}[\![\psi]\!]^\rho)^\dagger \\
&= \text{by lemma 12} \\
&\quad \left((\mathcal{R}[\![\phi]\!]^\rho \cup \mathcal{R}[\![\psi]\!]^\rho)^\dagger\right)^\dagger \\
&\supseteq \text{by lemma 13} \\
&\quad \left((\mathcal{R}[\![\phi]\!]^\rho)^\dagger \cup (\mathcal{R}[\![\psi]\!]^\rho)^\dagger\right)^\dagger \\
&\supseteq \text{by inductive hypothesis} \\
&\quad ([\![\phi]\!]^\rho \cup [\![\psi]\!]^\rho)^\dagger \\
&= \text{by definition 51} \\
&\quad [\![\phi \sqcup \psi]\!]^\rho
\end{aligned}
$$

Case: $\phi \sqcap \psi$

$$
\begin{aligned}
[\![\phi \sqcap \psi]\!]^\rho &= ([\![\phi]\!]^\rho \cap [\![\psi]\!]^\rho)^\dagger \\
&\subseteq \text{by lemma 13} \\
&\quad ([\![\phi]\!]^\rho)^\dagger \cap ([\![\psi]\!]^\rho)^\dagger \\
&= \text{by lemma 12} \\
&\quad [\![\phi]\!]^\rho \cap [\![\psi]\!]^\rho \\
&\subseteq \text{by inductive hypothesis} \\
&\quad (\mathcal{R}[\![\phi]\!]^\rho)^\dagger \cap (\mathcal{R}[\![\psi]\!]^\rho)^\dagger \\
&\subseteq \text{by lemma 13}
\end{aligned}
$$

$$
\begin{aligned}
&(\mathcal{R}[\![\phi]\!]^\rho \cap \mathcal{R}[\![\psi]\!]^\rho)^\dagger \\
&= \text{by definition 61} \\
&\mathcal{R}[\![\phi \sqcap \psi]\!]^\rho
\end{aligned}
$$

Case: $\exists x.\, \phi$, similar to $\phi \sqcup \psi$.
Case: $\mu A.\, \lambda x.\, \phi$

$$
\begin{aligned}
(\mathcal{R}[\![\mu A.\, \lambda x.\, \phi]\!]^\rho)^\dagger &= \left(\bigcap \left\{ T_f \in \textsc{Val} \to \mathcal{P}(\textsc{Trace}) \;\middle|\; \mathcal{R}[\![\lambda x.\, \phi]\!]^{\rho[A \mapsto T_f]} \subseteq T_f \right\}\right)^\dagger \\
&= \text{by lemma 11 and Kleene's fixpoint theorem} \\
&\quad \left(\bigcup \left\{ T_f \in \textsc{Val} \to \mathcal{P}(\textsc{Trace}) \;\middle|\; n \in \mathbb{N} \wedge T_f = (\mathcal{R}[\![\lambda x.\, \phi]\!]^{\rho[A \mapsto \emptyset]})^n \right\}\right)^\dagger \\
&\supseteq \text{by lemma 13} \\
&\quad \bigcup \left\{ T_f \in \textsc{Val} \to \mathcal{P}(\textsc{Trace}) \;\middle|\; n \in \mathbb{N} \wedge T_f = (\mathcal{R}[\![\lambda x.\, \phi]\!]^{\rho[A \mapsto \emptyset]})^n \right\}^\dagger \\
&= \bigcup \left\{ T_f \in \textsc{Val} \to \mathcal{P}(\textsc{Trace}) \;\middle|\; n \in \mathbb{N} \wedge T_f = \left(\left(\mathcal{R}[\![\lambda x.\, \phi]\!]^{\rho[A \mapsto \emptyset]}\right)^\dagger\right)^n \right\} \\
&= \text{by lemma 11 and Kleene's fixpoint theorem} \\
&\quad \bigcap \left\{ T_f \in \textsc{Val} \to \mathcal{P}(\textsc{Trace}) \;\middle|\; \left([\![\lambda x.\, \phi]\!]^{\rho[A \mapsto T_f]}\right)^\dagger \subseteq {T_f}^\dagger \right\} \\
&= \text{by definition 51} \\
&\quad [\![\mu A.\, \lambda x.\, \phi]\!]^\rho
\end{aligned}
$$

All other cases follow directly from the inductive hypothesis. □

Lemma 17 (Trace-Set Interleaving is Associative and Commutative)**.**

$$T \parallel (S \parallel U) = (T \parallel S) \parallel U \qquad\qquad T \parallel U = U \parallel T$$

Proof. Immediate by definition 49. □

B.1.1. Operational traces are denotational traces

In definition 48 we defined the observed traces on the reflexive, transitive closure of $\rightsquigarrow$. As a stepping stone, we also define single step traces which we relate to the raw denotational semantics.

Definition 62 (Single-Step Observed Traces)**.** *The single-step observed traces relation,* $\mathcal{O}_1[\![-]\!] \subseteq \mathcal{L} \times \mathcal{P}(\textsc{Trace})$*, is the smallest relation that satisfies the following rules:*

$$
\begin{array}{lll}
\text{(B.1)} & \text{(B.2)} & \text{(B.3)} \\
\dfrac{}{(\lightning, \lightning) \in \mathcal{O}_1[\![\phi]\!]} & \dfrac{\phi, h \rightsquigarrow o}{(h, o) \in \mathcal{O}_1[\![\phi]\!]} & \dfrac{\phi, h \rightsquigarrow \psi, h' \qquad t \in \mathcal{O}_1[\![\psi]\!]}{(h, h')t \in \mathcal{O}_1[\![\phi]\!]}
\end{array}
$$

Traces in $\mathcal{O}_1[\![\phi]\!]$ are not closed by the stuttering, mumbling and faulting closure.

We now relate the operational semantics to the raw denotational semantics: every single step in the operational semantics must be present as a move in the raw denotational semantics. Consequently, the traces observed by $\mathcal{O}_1[\![\phi]\!]$ are contained within $\mathcal{R}[\![\phi]\!]^\emptyset$, when ϕ is closed.

Lemma 18.

- *If* $\phi, h \rightsquigarrow \psi, h'$ *and* $t \in \mathcal{R}[\![\psi]\!]^\emptyset$ *then* $(h, h')t \in \mathcal{R}[\![\phi]\!]^\emptyset$
- *If* $\phi, h \rightsquigarrow o$ *then* $(h, o) \in \mathcal{R}[\![\phi]\!]^\emptyset$

Proof. By induction on the derivation of $\phi, h \rightsquigarrow \kappa$
Base case: rule (7.14).
Let $a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_k, h \rightsquigarrow h'$. By the premiss, $(h, h') \in \left\{ \mathsf{a}(P([\vec{x} \mapsto \vec{v}]), Q([\vec{x} \mapsto \vec{v}]))^{\mathcal{A}}_k \;\middle|\; \vec{v} \in \overrightarrow{\text{VAL}} \right\}$. By definition 61, $(h, h') \in \mathcal{R}\left[\!\left[a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_k \right]\!\right]^\emptyset$.
Base case: rule (7.15).
Let $a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_k, h \rightsquigarrow \natural$. By the premiss,
For all $\vec{v} \in \overrightarrow{\text{VAL}}$, $\mathsf{a}(P([\vec{x} \mapsto \vec{v}]), Q([\vec{x} \mapsto \vec{v}]))^{\mathcal{A}}_k(h) = \emptyset$. By definition 61, $(h, \natural) \in \mathcal{R}\left[\!\left[a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_k \right]\!\right]^\emptyset$.
Case: rule (7.1).
Let $s = t; u$, such that, $s \in \mathcal{R}[\![\phi'; \psi]\!]^\emptyset$, $t \in \mathcal{R}[\![\phi']\!]^\emptyset$ and $u \in \mathcal{R}[\![\psi]\!]^\emptyset$. By the premiss and the inductive hypothesis, $(h, h')t \in \mathcal{R}[\![\phi]\!]^\emptyset$. Then, by definition 61, $(h, h')s = (h, h')tu \in \mathcal{R}[\![\phi; \psi]\!]^\emptyset$.
Case: rule (7.2).
Let $s \in \mathcal{R}[\![\psi]\!]^\emptyset$. By the premiss and the inductive hypothesis, $(h, h') \in \mathcal{R}[\![\phi]\!]^\emptyset$. Then, by definition 61, $(h, h')s \in \mathcal{R}[\![\phi; \psi]\!]^\emptyset$.
Case: rule (7.3).
By the premiss and the inductive hypothesis, $(h, \natural) \in \mathcal{R}[\![\phi]\!]^\emptyset$. By lemma 8, $(\natural, \natural) \in \mathcal{R}[\![\psi]\!]^\emptyset$. By trace concatenation $(h, \natural) = (h, \natural); (\natural, \natural)$. Then, by definition 61, $(h, \natural) \in \mathcal{R}[\![\phi; \psi]\!]^\emptyset$.
Case: rule (7.4).
By case analysis on κ. First, let $\kappa = \phi', h'$. Let $t \in \mathcal{R}[\![\phi']\!]^\emptyset$. By the premiss and the inductive hypothesis, $(h, h')t \in \mathcal{R}[\![\phi \parallel \psi]\!]^\emptyset$. Then, by definition 61 and lemma 17, $(h, h')t \in \mathcal{R}[\![\psi \parallel \phi]\!]^\emptyset$. Second, let $\kappa = o$. By the premiss and the inductive hypothesis, $(h, o) \in \mathcal{R}[\![\phi \parallel \psi]\!]^\emptyset$. Then, by definition 61 and lemma 17, $(h, o) \in \mathcal{R}[\![\psi \parallel \phi]\!]^\emptyset$.
Case: rule (7.5).
Let $s \in t \parallel u$, such that $s \in \mathcal{R}[\![\phi' \parallel \psi]\!]^\emptyset$, $t \in \mathcal{R}[\![\phi']\!]^\emptyset$ and $u \in \mathcal{R}[\![\psi]\!]^\emptyset$. By the premiss and the inductive hypothesis, $(h, h')t \in \mathcal{R}[\![\phi]\!]^\emptyset$. Then, by definition 49, $(h, h')s \in (h, h')t \parallel u$. By definitions 61 and 49, $(h, h')t \parallel u \subseteq \mathcal{R}[\![\phi \parallel \psi]\!]^\emptyset$, from which it follows that $(h, h')s \in \mathcal{R}[\![\phi \parallel \psi]\!]^\emptyset$.
Case: rule (7.6).
Let $s \in \mathcal{R}[\![\psi]\!]^\emptyset$. By the premiss and the inductive hypothesis, $(h, h') \in \mathcal{R}[\![\phi]\!]^\emptyset$. Then, by definition 49, $(h, h')s \in \mathcal{R}[\![\phi \parallel \psi]\!]^\emptyset$.
Case: rule (7.7).
By the premiss and the inductive hypothesis, $(h, \natural) \in \mathcal{R}[\![\phi]\!]^\emptyset$. By lemma 8, $(\natural, \natural)\mathcal{R}[\![\psi]\!]^\emptyset$. By definition 49, $(h, \natural) \in (h, \natural) \parallel (\natural, \natural)$. By definition 61 and definition 49, $(h, \natural) \parallel (\natural, \natural) \subseteq \mathcal{R}[\![\phi \parallel \psi]\!]^\emptyset$, from which it follows $(h, \natural) \in \mathcal{R}[\![\phi \parallel \psi]\!]^\emptyset$.
Case: rule (7.8).

Case analysis on κ. First, let $\kappa = \phi', h'$. Let $t \in \mathcal{R}[\![\phi']\!]^{\emptyset}$. By the premiss, by the inductive hypothesis, for some $i \in \{0,1\}$, $(h,h')t \in \mathcal{R}[\![\phi_i]\!]^{\emptyset}$. Thus, $(h,h')t \in \mathcal{R}[\![\phi_i]\!]^{\emptyset} \cup \mathcal{R}\left[\!\left[\phi_{(i+1) \bmod 2}\right]\!\right]^{\emptyset}$. Then, by definition 61, $(h,h')t \in \mathcal{R}[\![\phi_0 \sqcup \phi_1]\!]^{\emptyset}$. Second, let $\kappa = o$. By the premiss, by the inductive hypothesis, for some $i \in \{0,1\}$, $(h,o) \in \mathcal{R}[\![\phi_i]\!]^{\emptyset}$. Thus, $(h,o) \in \mathcal{R}[\![\phi_i]\!]^{\emptyset} \cup \mathcal{R}\left[\!\left[\phi_{(i+1) \bmod 2}\right]\!\right]^{\emptyset}$. Then, by definition 61, $(h,o) \in \mathcal{R}[\![\phi_0 \sqcup \phi_1]\!]^{\emptyset}$.

Case: rule (7.9).

Case analysis on κ. First, let $\kappa = \phi', h'$. Let $t \in \mathcal{R}[\![\phi']\!]^{\emptyset}$. By the premiss, by the inductive hypothesis, for all $i \in \{0,1\}$, $(h,h')t \in \mathcal{R}[\![\phi_i]\!]^{\emptyset}$. Thus, $(h,h')t \in \mathcal{R}[\![\phi_i]\!]^{\emptyset} \cap \mathcal{R}\left[\!\left[\phi_{(i+1) \bmod 2}\right]\!\right]^{\emptyset}$. Then, by definition 61, $(h,h')t \in \mathcal{R}[\![\phi_0 \sqcap \phi_1]\!]^{\emptyset}$. Second, let $\kappa = o$. By the premiss, by the inductive hypothesis, for all $i \in \{0,1\}$, $(h,o) \in \mathcal{R}[\![\phi_i]\!]^{\emptyset}$. Thus, $(h,o) \in \mathcal{R}[\![\phi_i]\!]^{\emptyset} \cap \mathcal{R}\left[\!\left[\phi_{(i+1) \bmod 2}\right]\!\right]^{\emptyset}$. Then, by definition 61, $(h,o) \in \mathcal{R}[\![\phi_0 \sqcap \phi_1]\!]^{\emptyset}$.

Case: rule (7.10).

Case analysis on κ. First, let $\kappa = \phi', h'$. Let $t \in \mathcal{R}[\![\phi']\!]^{\emptyset}$. Fix v. By the premiss and the inductive hypothesis, $(h,h')t \in \mathcal{R}[\![\phi\,[v/x]]\!]^{\emptyset}$. Then, by definition 61, $(h,h')t \in \mathcal{R}[\![\exists x.\,\phi]\!]^{\emptyset}$. Second, let $\kappa = o$. By the premiss and the inductive hypothesis, $(h,o) \in \mathcal{R}[\![\phi\,[v/x]]\!]^{\emptyset}$. Then, by definition 61, $(h,o) \in \mathcal{R}[\![\exists x.\,\phi]\!]^{\emptyset}$.

Case: rule (7.11).

Case analysis on κ. First, let $\kappa = \phi', h'$. Let $t \in \mathcal{R}[\![\phi']\!]^{\emptyset}$. By the premiss and the inductive hypothesis, $(h,h')t \in \mathcal{R}[\![\phi\,[F/f]]\!]^{\emptyset}$. Then, by lemma 9, $(h,h')t \in \mathcal{R}[\![\phi]\!]^{\emptyset[f \mapsto \mathcal{R}[\![F]\!]^{\emptyset}]}$. Then, by definition 61, $(h,h')t \in \mathcal{R}[\![\textbf{let } f = F \textbf{ in } \phi]\!]^{\emptyset}$. Second, let $\kappa = o$. By the premiss and the inductive hypothesis, $(h,o) \in \mathcal{R}[\![\phi\,[F/f]]\!]^{\emptyset}$. Then, by lemma 9, $(h,o) \in \mathcal{R}[\![\phi]\!]^{\emptyset[f \mapsto \mathcal{R}[\![F]\!]^{\emptyset}]}$. Then, by definition 61, $(h,o) \in \mathcal{R}[\![\textbf{let } f = F \textbf{ in } \phi]\!]^{\emptyset}$.

Case: rule (7.12).

Case analysis on κ. First, let $\kappa = \phi', h'$. Let $t \in \mathcal{R}[\![\phi']\!]^{\emptyset}$. By the premiss and the inductive hypothesis, $(h,h')t \in \mathcal{R}[\![(\lambda x.\,\phi\,[\mu A.\,\lambda x.\,\phi/A])\,e]\!]^{\emptyset}$. Then, from the fact that μ is denotationally the least fixpoint, $(h,h')t \in \mathcal{R}[\![(\mu A.\,\lambda x.\,\phi)\,e]\!]^{\emptyset}$. Second, let $\kappa = o$. By the premiss and the inductive hypothesis, $(h,o) \in \mathcal{R}[\![(\lambda x.\,\phi\,[\mu A.\,\lambda x.\,\phi/A])\,e]\!]^{\emptyset}$. Then, from the fact that μ is denotationally the least fixpoint, $(h,o) \in \mathcal{R}[\![(\mu A.\,\lambda x.\,\phi)\,e]\!]^{\emptyset}$.

Case: rule (7.13).

Case analysis on κ. First, let $\kappa = \phi', h'$. Let $t \in \mathcal{R}[\![\phi']\!]^{\emptyset}$. By the premiss and the inductive hypothesis, $(h,h')t \in \mathcal{R}\left[\!\left[\phi\left[[\![e]\!]^{\emptyset}/x\right]\right]\!\right]^{\emptyset}$. Then, by lemma 10, $(h,h')t \in \mathcal{R}[\![\phi]\!]^{\emptyset[x \mapsto [\![e]\!]]}$. Then, by definition 61, $(h,h')t \in \mathcal{R}[\![\lambda x.\,\phi]\!]^{\emptyset}$. Second, let $\kappa = o$. By the premiss and the inductive hypothesis, $(h,o) \in \mathcal{R}\left[\!\left[\phi\left[[\![e]\!]^{\emptyset}/x\right]\right]\!\right]^{\emptyset}$. Then, by lemma 10, $(h,o) \in \mathcal{R}[\![\phi]\!]^{\emptyset[x \mapsto [\![e]\!]]}$. Then, by definition 61, $(h,o) \in \mathcal{R}[\![\lambda x.\,\phi]\!]^{\emptyset}$. □

Corollary 1. *If* $t \in \mathcal{O}_1[\![\phi]\!]$ *then* $t \in \mathcal{R}[\![\phi]\!]^{\emptyset}$.

Proof. Straightforward induction on the derivation of $t \in \mathcal{O}_1[\![\phi]\!]$, using lemma 18.

Base case: rule (B.1).

$(\wr, \wr) \in \mathcal{O}_1[\![\phi]\!]$ and by lemma 8, $(\wr, \wr) \in \mathcal{R}[\![\phi]\!]^{\emptyset}$.

Base case: rule (B.2).

By the premiss and lemma 18, $(h,\kappa) \in \mathcal{R}[\![\phi]\!]^{\emptyset}$.

Case: rule (B.3).

By the premisses and lemma 18, $(h, h')t \in \mathcal{R}[\![\phi]\!]^{\emptyset}$ □

Traces in $\mathcal{O}_1[\![\phi]\!]$ are obtained by observing every single transition in the operational semantics. They relate to traces in $\mathcal{O}[\![\phi]\!]$, which are obtained by observing transitions in the transitive and reflexive closure of the operational semantics, by the stuttering, mumbling and faulting closure.

Lemma 19. *If $t \in \mathcal{O}[\![\phi]\!]$ then $t \in (\mathcal{O}_1[\![\phi]\!])^{\dagger}$.*

Proof. Induction on the derivation of $t \in \mathcal{O}[\![\phi]\!]$, with nested induction on steps $\kappa \rightsquigarrow^* \kappa'$.
Base case: rule (7.16).
$(\lightning, \lightning) \in \mathcal{O}[\![\phi]\!]$ and by definition 50, $(\lightning, \lightning) \in (\mathcal{O}_1[\![\phi]\!])^{\dagger}$.
Base case: rule (7.17).
By induction on the derivation of the premiss.
Nested base case: $\phi, h \rightsquigarrow o$
By definition 62, $(h, o) \in \mathcal{O}_1[\![\phi]\!]$. Then by definition 50, $(h, o) \in (\mathcal{O}_1[\![\phi]\!])^{\dagger}$.
Nested case: $\phi, h, \rightsquigarrow \psi, h'$ and $\psi, h' \rightsquigarrow^* o$
By the inductive hypothesis, $(h', o) \in (\mathcal{O}_1[\![\psi]\!])^{\dagger}$. Then, by definition 50, there exist h'', h''', t such that if $t = \epsilon$, then $h''' = h''$, and $(h', h'')t(h''', o) \in \mathcal{O}_1[\![\psi]\!]$. Then, by definition 62, $(h, h')(h', h'')t(h''', o) \in \mathcal{O}_1[\![\phi]\!]$. Then, by definition 50, $(h, o) \in (\mathcal{O}[\![\phi]\!])^{\dagger}$.
Case: rule (7.18).
Let $t \in \mathcal{O}[\![\psi]\!]$. By induction on the derivation of the premiss.
Nested base case: $\phi, h \rightsquigarrow \psi, h'$ By the inductive hypothesis $t \in (\mathcal{O}_1[\![\psi]\!])^{\dagger}$. By definition 50, there exists $u \in \mathcal{O}_1[\![\psi]\!]$ such that $t \in (\mathcal{O}_1[\![\psi]\!])^{\dagger}$. Then, by definition 62, $(h, h')u \in \mathcal{O}_1[\![\phi]\!]$. Then, by definition 50, $(h, h')t \in (\mathcal{O}_1[\![\phi]\!])^{\dagger}$.
Nested case: $\phi, h \rightsquigarrow \phi', h''$ and $\phi', h'' \rightsquigarrow^* \psi, h'$. By the inductive hypothesis, $(h'', h')t \in (\mathcal{O}_1[\![\phi']\!])^{\dagger}$. By definition 50, there exists u, v, h''', h'''', such that $(h'', h''')u(h'''', h')v \in \mathcal{O}_1[\![\phi']\!]$. Then, by definition 62, $(h, h'')(h'', h''')u(h'''', h')v \in \mathcal{O}_1[\![\phi]\!]$. Then, by definition 50, $(h, h')t \in (\mathcal{O}_1[\![\phi]\!])^{\dagger}$. □

Corollary 2. *If $t \in \mathcal{O}[\![\phi]\!]$ then $t \in [\![\phi]\!]^{\emptyset}$.*

Proof.

$$
\begin{aligned}
\mathcal{O}[\![\phi]\!] &\subseteq \text{by lemma 19} \\
&\quad (\mathcal{O}_1[\![\phi]\!])^{\dagger} \\
&\subseteq \text{by corollary 1 and lemma 12} \\
&\quad \left(\mathcal{R}[\![\phi]\!]^{\emptyset}\right)^{\dagger} \\
&= \text{by lemma 16} \\
&\quad [\![\phi]\!]^{\emptyset}
\end{aligned}
$$

□

B.1.2. Denotational traces are operational traces

The following two intermediate lemmas, establish compositional properties for $\mathcal{O}_1[\![\phi]\!]$.

Lemma 20. $\mathcal{O}_1[\![\phi]\!]; \mathcal{O}_1[\![\psi]\!] \subseteq \mathcal{O}_1[\![\phi; \psi]\!]$

Proof. We prove that if $t \in \mathcal{O}_1[\![\phi]\!]$ and $u \in \mathcal{O}_1[\![\psi]\!]$, then $tu \in \mathcal{O}_1[\![\phi; \psi]\!]$, by induction on the derivation of $t \in \mathcal{O}_1[\![\phi]\!]$.
Base case: rule (B.1).
$(\lightning, \lightning) \in \mathcal{O}_1[\![\phi]\!]$, $(\lightning, \lightning) \in \mathcal{O}_1[\![\psi]\!]$ and $(\lightning, \lightning) \in \mathcal{O}_1[\![\phi; \psi]\!]$.
Base case: rule (B.2).
Let $u \in \mathcal{O}_1[\![\psi]\!]$. Let $(h, o) \in \mathcal{O}_1[\![\phi]\!]$. Case analysis on o. First, let $o = \lightning$. Then, by the premiss and definition 44, $\phi; \psi, h \rightsquigarrow \lightning$, by which $(h, \lightning) \in \mathcal{O}_1[\![\phi; \psi]\!]$. By trace concatenation, $(h, \lightning) = (h, \lightning)u$, by which $(h, \lightning)u \in \mathcal{O}_1[\![\phi; \psi]\!]$. Second, let $o = h'$. By definition 44, $\phi; \psi, h \rightsquigarrow \psi, h'$. Then, by rule (B.3), $(h, h')u \in \mathcal{O}_1[\![\phi; \psi]\!]$.
Case: rule (B.3).
Let $u \in \mathcal{O}_1[\![\psi]\!]$. Let $(h, h')t \in \mathcal{O}_1[\![\phi]\!]$. Then, by the premiss, there exists ϕ', such that $\phi, h \rightsquigarrow \phi', h'$ and $t \in \mathcal{O}_1[\![\phi']\!]$. By the induction hypothesis, $tu \in \mathcal{O}_1[\![\phi'; \psi]\!]$. From the premiss and definition 44, $\phi; \psi, h \rightsquigarrow \phi'; \psi, h'$. Then, $(h, h')tu \in \mathcal{O}_1[\![\phi; \psi]\!]$. □

Lemma 21. $\mathcal{O}_1[\![\phi]\!] \parallel \mathcal{O}_1[\![\psi]\!] \subseteq \mathcal{O}_1[\![\phi \parallel \psi]\!]$.

Proof. We prove that if $t \in \mathcal{O}_1[\![\phi]\!]$, $u \in \mathcal{O}_1[\![\psi]\!]$, and $s \in t \parallel u$, then $s \in \mathcal{O}_1[\![\phi \parallel \psi]\!]$, by induction on the derivation of $t \in \mathcal{O}_1[\![\phi]\!]$.
Base case: rule (B.1). $(\lightning, \lightning) \in \mathcal{O}_1[\![\phi]\!]$, $(\lightning, \lightning) \in \mathcal{O}_1[\![\psi]\!]$ and $(\lightning, \lightning) \in \mathcal{O}_1[\![\phi \parallel \psi]\!]$.
Base case: rule (B.2).
Let $u \in \mathcal{O}_1[\![\psi]\!]$. Let $(h, o) \in \mathcal{O}_1[\![\phi]\!]$. Case analysis on o. First, let $o = \lightning$. By definition 49, $(h, \lightning) \in (h, \lightning) \parallel u$. Then, by premiss and by definition 44, $\phi \parallel \psi, h \rightsquigarrow \lightning$, by which $(h, \lightning) \in \mathcal{O}_1[\![\phi \parallel \psi]\!]$. Second, let $o = h'$. By definition 49, $(h, h')u \in (h, h') \parallel u$. Then, by premiss and by definition 44, $\phi \parallel \psi, h \rightsquigarrow \psi, h'$. Then, by rule (B.3), $(h, h')u \in \mathcal{O}_1[\![\phi \parallel \psi]\!]$.
Case: rule (B.3).
Let $u \in \mathcal{O}_1[\![\psi]\!]$. Let $(h, h')s \in \mathcal{O}_1[\![\phi]\!]$. Then, by the premiss, there exists ϕ', such that $\phi, h \rightsquigarrow \phi', h'$ and $s \in \mathcal{O}_1[\![\phi']\!]$. Let $w \in s \parallel u$. By definition 49, $(h, h')w \in (h, h')s \parallel u$. By the inductive hypothesis, $w \in \mathcal{O}_1[\![\phi' \parallel \psi]\!]$. From the premiss and definition 44, $\phi \parallel \psi, h \rightsquigarrow \phi' \parallel \psi, h'$. Then, $(h, h')w \in \mathcal{O}_1[\![\phi \parallel \psi]\!]$. □

Lemma 22. $\mathcal{O}_1[\![\phi]\!] \cup \mathcal{O}_1[\![\psi]\!] \subseteq \mathcal{O}_1[\![\phi \sqcup \psi]\!]$.

Proof. We prove that if $t \in \mathcal{O}_1[\![\phi]\!]$ or $t \in \mathcal{O}_1[\![\phi]\!]$, then $t \in \mathcal{O}_1[\![\phi \sqcup \psi]\!]$, by proving: a). if $t \in \mathcal{O}_1[\![\phi]\!]$, then $t \in \mathcal{O}_1[\![\phi \sqcup \psi]\!]$, and b). if $t \in \mathcal{O}_1[\![\psi]\!]$, then $t \in \mathcal{O}_1[\![\phi \sqcup \psi]\!]$, each by induction on the derivation of $t \in \mathcal{O}_1[\![\phi]\!]$.
Proof of a).
Base case: rule (B.1). $(\lightning, \lightning) \in \mathcal{O}_1[\![\phi]\!]$, $(\lightning, \lightning) \in \mathcal{O}_1[\![\psi]\!]$ and $(\lightning, \lightning) \in \mathcal{O}_1[\![\phi \sqcup \psi]\!]$.
Base case: rule (B.2).
Let $(h, o) \in \mathcal{O}_1[\![\phi]\!]$. Then, by premiss and definition 44, $\phi \sqcup \psi, h \rightsquigarrow o$. It follows that, $(h, o) \in \mathcal{O}_1[\![\phi \sqcup \psi]\!]$.
Case: rule (B.3).
Let $u \in \mathcal{O}_1[\![\psi]\!]$. Let $(h, h')t \in \mathcal{O}_1[\![\phi]\!]$. Then, by the premiss, there exists ϕ', such that $\phi, h \rightsquigarrow \phi', h'$ and $t \in \mathcal{O}_1[\![\phi']\!]$. By the induction hypothesis, $t \in \mathcal{O}_1[\![\phi' \sqcup \psi]\!]$. From definition 44, let $\phi \sqcup \psi, h \rightsquigarrow \phi', h'$. It follows that, $(h, h')t \in \mathcal{O}_1[\![\phi \sqcup \psi]\!]$.
Proof of b). As in a). □

Lemma 23. $\bigcup_v \mathcal{O}_1[\![\phi\,[v/x]]\!] \subseteq \mathcal{O}_1[\![\exists x.\,\phi]\!]$.

Proof. We prove that if $t \in \bigcup_v \mathcal{O}_1[\![\phi\,[v/x]]\!]$, then $t \in \mathcal{O}_1[\![\exists x.\,\phi]\!]$, by induction on the length of t.
Let $v \in \textsc{Val}$.
Base case: $(\{\!|,|\!\})$, trivial.
Base case: $(h, o) \in \mathcal{O}_1[\![\phi\,[v/x]]\!]$.
By rule (B.2), $\phi\,[v/x], h \rightsquigarrow o$. Then, by definition 44, $\exists x.\,\phi, h \rightsquigarrow o$. It follows that, $(h, o) \in \mathcal{O}_1[\![\exists x.\,\phi]\!]$.
Case: $(h, h')t \in \mathcal{O}_1[\![\phi\,[v/x]]\!]$.
By rule (B.3), there exists ψ such that $\phi\,[v/x], \phi \rightsquigarrow \psi, h'$ and $t \in \mathcal{O}_1[\![\psi]\!]$. By definition 44, $\exists x.\,\phi, h \rightsquigarrow \psi, h'$. By rule (B.3), $(h, h')t \in \mathcal{O}_1[\![\exists x.\,\phi]\!]$.

□

Lemma 24. $\mathcal{O}_1[\![\phi\,[F/f]]\!] \subseteq \mathcal{O}_1[\![\textbf{let } f = F \textbf{ in } \phi]\!]$.

Proof. By induction on the length of $t \in \mathcal{O}_1[\![\phi\,[F/f]]\!]$.

□

Lemma 25. $\mathcal{O}_1[\![\phi\left[[\![e]\!]^{\emptyset}/x\right]]\!] \subseteq \mathcal{O}_1[\![(\lambda x.\,\phi)\,e]\!]$.

Proof. Similarly, to lemma 24. □

Lemma 26. $\mathcal{O}_1[\![\phi\,[\mu A.\,\lambda x.\,\phi/A]\,[[\![e]\!]/x]]\!] \subseteq \mathcal{O}_1[\![(\mu A.\,\lambda x.\,\phi)\,e]\!]$.

Proof. Similarly to lemma 25.

□

The following lemma is a direct consequence of the fact that recursion is semantically the least fixpoint.

Lemma 27. $\mathcal{R}[\![\lambda x.\,\phi]\!]^{\rho[A \mapsto \mathcal{R}[\![\mu A.\lambda x.\phi]\!]^{\rho}]} = \mathcal{R}[\![\mu A.\,\lambda x.\,\phi]\!]^{\rho}$

Proof. By induction on ϕ.
Base case : Ae.

$$
\begin{aligned}
&\mathcal{R}[\![\lambda x.\,Ae]\!]^{\rho[A \mapsto \mathcal{R}[\![\mu A.\lambda x.Ae]\!]^{\rho}]} \\
&= \lambda v.\,\mathcal{R}[\![Ae]\!]^{\rho[A \mapsto \mathcal{R}[\![\mu A.\lambda x.Ae]\!]^{\rho}][x \mapsto v]} \\
&= \lambda v.\,\left(\mathcal{R}[\![A]\!]^{\rho[A \mapsto \mathcal{R}[\![\mu A.\lambda x.Ae]\!]^{\rho}][x \mapsto v]}\right)[\![e]\!]^{\rho[A \mapsto \mathcal{R}[\![\mu A.\lambda x.Ae]\!]^{\rho}][x \mapsto v]} \\
&= \lambda v.\,(\mathcal{R}[\![\mu A.\,\lambda x.\,Ae]\!]^{\rho})\,[\![e]\!]^{\rho[x \mapsto v]} \\
&= \text{by definition 61, lemma 11 and Kleene's fixpoint theorem} \\
&\quad \lambda v.\,\left(\bigcup \left\{ T_f \in \textsc{Val} \to \mathcal{P}(\textsc{Trace}) \;\middle|\; n \in \mathbb{N} \wedge T_f = (\mathcal{R}[\![\lambda x.\,Ae]\!]^{\rho[A \mapsto \emptyset]})^n \right\}\right)[\![e]\!]^{\rho[x \mapsto v]} \\
&= \lambda v.\,\left(\bigcup \left\{ T_f \in \textsc{Val} \to \mathcal{P}(\textsc{Trace}) \;\middle|\; n \in \mathbb{N} \wedge T_f = (\lambda v.\,\left(\mathcal{R}[\![A]\!]^{\rho[A \mapsto \emptyset][x \mapsto v]}\right)[\![e]\!]^{\rho[x \mapsto v]})^n \right\}\right)[\![e]\!]^{\rho[x \mapsto v]} \\
&= \bigcup \left\{ T_f \in \textsc{Val} \to \mathcal{P}(\textsc{Trace}) \;\middle|\; n \in \mathbb{N} \wedge T_f = (\lambda v.\,\left(\mathcal{R}[\![A]\!]^{\rho[A \mapsto \emptyset][x \mapsto v]}\right)[\![e]\!]^{\rho[x \mapsto v]})^{n+1} \right\} \\
&= \text{by Kleene's fixpoint theorem} \\
&\quad \mathcal{R}[\![\mu A.\,\lambda x.\,\phi]\!]^{\rho}
\end{aligned}
$$

All other cases follow directly from the inductive hypothesis. □

We now establish the reverse of corollary 1: $\mathcal{R}[\![\phi]\!]^{\emptyset} \subseteq \mathcal{O}_1[\![\phi]\!]$. This is difficult to prove directly by induction over ϕ. Specifically, substructures of ϕ extend the variable environment, for example $\mathcal{R}[\![\exists x.\, \phi]\!]^{\emptyset} = \bigcup_v \mathcal{R}[\![\phi]\!]^{\emptyset[x \mapsto v]}$, and thus we cannot directly apply the inductive hypothesis. The solution is to generalise the property for arbitrary variable stores. Then, the property we wish to prove takes the form $\mathcal{R}[\![\phi]\!]^{\rho} \subseteq \mathcal{O}_1[\![C[\phi]]\!]$, where C is a context that closes ϕ according to the bindings in ρ. In order to precisely state this property, we first formally define closing contexts induced by variable stores.

Definition 63 (Closing contexts)**.**

$$
\begin{array}{ll}
\textit{Syntactic environments:} & \eta ::= \emptyset \mid x \mapsto e : \eta \mid f \mapsto F : \eta \mid A \mapsto \phi \\
\textit{Syntactic environment application:} & \theta(\emptyset\phi) \triangleq \phi \\
& \theta((x \mapsto e : \eta)\phi) \triangleq \theta(\eta(\phi \left[[\![e]\!]^{\omega(\eta)} / x \right])) \\
& \theta((f \mapsto F : \eta)\phi) \triangleq \theta(\eta(\phi\,[F/f])) \\
& \theta((A \mapsto \psi : \eta)\phi) \triangleq \theta(\eta(\phi\,[\mu A.\, \lambda x.\, \psi / A])) \\
\textit{Syntactic environment erasure:} & \omega(\emptyset) \triangleq \emptyset \\
& \omega(x \mapsto e : \eta) \triangleq \omega(\eta)[x \mapsto [\![e]\!]^{\omega(\eta)}] \\
& \omega(f \mapsto F : \eta) \triangleq \omega(\eta)[f \mapsto \mathcal{R}[\![F]\!]^{\omega(\eta)}] \\
& \omega(A \mapsto \psi : \eta) \triangleq \omega(\eta)[A \mapsto \mathcal{R}[\![\mu A.\, \lambda x.\, \psi]\!]^{\omega(\eta)}]
\end{array}
$$

Intuitively, a syntactic environment, η, represents the closing context. We use η, to list the variable bindings that we need to introduce in ϕ. The syntactic environment application, $theta(\eta\ \phi)$, applies the syntactic environment η to ϕ by substituting the variables in ϕ with their bound values. The syntactic environment erasure, $\omega(\eta)$, erases η to a variable store that binds variables according to η.

Given definition 63, the reverse of corollary 1 is: $\mathcal{R}[\![\phi]\!]^{\omega(\eta)} \subseteq \mathcal{O}_1[\![\theta(\eta\phi)]\!]$.

Lemma 28. $\mathcal{R}[\![\phi]\!]^{\omega(\eta)} \subseteq \mathcal{O}_1[\![\theta(\eta\phi)]\!]$.

Proof. By induction on ϕ.
Base case: $a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_k$.

$$
\begin{aligned}
\mathcal{R}\left[\!\left[a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_k\right]\!\right]^{\omega(\eta)} &= \text{by definition 61} \\
&\quad \left\{ (h, h') \in \textsc{Move} \;\middle|\; \vec{v} \in \overrightarrow{\textsc{Val}} \wedge h' \in \mathsf{a}\left((\!|P|\!)^{\omega(\eta)[\vec{x} \mapsto \vec{v}]}_{\mathcal{A}}, (\!|Q|\!)^{\omega(\eta)[\vec{x} \mapsto \vec{v}]}_{\mathcal{A}} \right)^{\mathcal{A}}_k (h) \right\} \\
&\quad \cup \left\{ (h, \lightning) \in \textsc{Heap}^{\lightning} \;\middle|\; \begin{array}{l} \vec{v} \in \overrightarrow{\textsc{Val}} \wedge \mathsf{a}\left((\!|P|\!)^{\omega(\eta)[\vec{x} \mapsto \vec{v}]}_{\mathcal{A}}, (\!|Q|\!)^{\omega(\eta)[\vec{x} \mapsto \vec{v}]}_{\mathcal{A}} \right)^{\mathcal{A}}_k (h) = \emptyset \\ \quad \wedge\ (\!|Q|\!)^{\omega(\eta)[\vec{x} \mapsto \vec{v}]}_{\neq} \emptyset \end{array} \right\} \\
&\quad \cup \left\{ (\lightning, \lightning) \right\} \\
&= \text{by induction on } \eta \text{ and definition 62} \\
&\quad \mathcal{O}_1[\![\theta(\eta(a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_k))]\!]
\end{aligned}
$$

Case: $\phi; \psi$.

$$
\begin{aligned}
\mathcal{R}[\![\phi; \psi]\!]^{\omega(\eta)} &= \text{by definition 61} \\
&\quad \mathcal{R}[\![\phi]\!]^{\omega(\eta)} ; \mathcal{R}[\![\psi]\!]^{\omega(\eta)} \\
&\subseteq \text{by induction hypothesis} \\
&\quad \mathcal{O}_1[\![\theta(\eta\phi)]\!]; \mathcal{O}_1[\![\theta(\eta\psi)]\!] \\
&\subseteq \text{by induction over } \eta \text{ and lemma 20} \\
&\quad \mathcal{O}_1[\![\theta(\eta(\phi; \psi))]\!]
\end{aligned}
$$

Case: $\phi \parallel \psi$.

$$
\begin{aligned}
\mathcal{R}[\![\phi \parallel \psi]\!]^{\omega(\eta)} &= \text{by definition 61} \\
&\quad \mathcal{R}[\![\phi]\!]^{\omega(\eta)} \parallel \mathcal{R}[\![\psi]\!]^{\omega(\eta)} \\
&\subseteq \text{by induction hypothesis} \\
&\quad \mathcal{O}_1[\![\theta(\eta\phi)]\!] \parallel \mathcal{O}_1[\![\theta(\eta\psi)]\!] \\
&\subseteq \text{by induction over } \eta \text{ and lemma 21} \\
&\quad \mathcal{O}_1[\![\theta(\eta(\phi \parallel \psi))]\!]
\end{aligned}
$$

Case: $\phi \sqcup \psi$.

$$
\begin{aligned}
\mathcal{R}[\![\phi \sqcup \psi]\!]^{\omega(\eta)} &= \text{by definition 61} \\
&\quad \mathcal{R}[\![\phi]\!]^{\omega(\eta)} \cup \mathcal{R}[\![\psi]\!]^{\omega(eta)} \\
&\subseteq \text{by induction hypothesis} \\
&\quad \mathcal{O}_1[\![\theta(\eta\phi)]\!] \cup \mathcal{O}_1[\![\theta(\eta\psi)]\!] \\
&\subseteq \text{by induction over } \eta \text{ and lemma 22} \\
&\quad \mathcal{O}_1[\![\phi \sqcup \psi]\!]
\end{aligned}
$$

Case: $\phi \sqcap \psi$, similarly to previous.

Case: $\exists x.\, \phi$.

$$
\begin{aligned}
\mathcal{R}[\![\exists x.\, \phi]\!]^{\omega(\eta)} &= \text{by definition 61} \\
&\quad \bigcup_v \mathcal{R}[\![\phi]\!]^{\omega(\eta)[x \mapsto v]} \\
&= \bigcup_v \mathcal{R}[\![\phi]\!]^{\omega(x \mapsto v : \eta)} \\
&\subseteq \text{by induction hypothesis} \\
&\quad \bigcup_v \mathcal{O}_1[\![\theta((x \mapsto v : \eta)\phi)]\!]
\end{aligned}
$$

$$
\begin{aligned}
&= \text{by syntactic environment application (definition 63)} \\
&\quad \bigcup_v \mathcal{O}_1[\![\theta(\eta(\phi\,[v/x]))]\!] \\
&\subseteq \text{by induction over } \eta \text{ and lemma 23} \\
&\quad \mathcal{O}_1[\![\theta(\eta(\exists x.\,\phi))]\!]
\end{aligned}
$$

Case: $\mathbf{let}\ f = F\ \mathbf{in}\ \phi$

$$
\begin{aligned}
\mathcal{R}[\![\mathbf{let}\ f = F\ \mathbf{in}\ \phi]\!]^{\omega(\eta)} &= \text{by definition 61} \\
&\quad \mathcal{R}[\![\phi]\!]^{\omega(\eta)[f \mapsto \mathcal{R}[\![F]\!]^{\emptyset}]} \\
&= \mathcal{R}[\![\phi]\!]^{\omega(f \mapsto F : \eta)} \\
&\subseteq \text{by induction hypothesis} \\
&\quad \mathcal{O}_1[\![\theta((f \mapsto F : \eta)\phi)]\!] \\
&= \text{by syntactic environment application (definition 63)} \\
&\quad \mathcal{O}_1[\![\theta(\eta(\phi\,[F/f]))]\!] \\
&\subseteq \text{by induction over } \eta \text{ and lemma 24} \\
&\quad \mathcal{O}_1[\![\theta(\eta(\mathbf{let}\ f = F\ \mathbf{in}\ \phi))]\!]
\end{aligned}
$$

Case: $(\lambda x.\,\phi)\,e$.

$$
\begin{aligned}
\mathcal{R}[\![(\lambda x.\,\phi)\,e]\!]^{\omega(\eta)} &= \text{by definition 61} \\
&\quad \left(\mathcal{R}[\![\lambda x.\,\phi]\!]^{\omega(\eta)}\right)[\![e]\!]^{\omega(\eta)} \\
&= \left(\lambda v.\,\mathcal{R}[\![\phi]\!]^{\omega(\eta)[x \mapsto v]}\right)[\![e]\!]^{\omega(\eta)} \\
&= \mathcal{R}[\![\phi]\!]^{\omega(\eta)[x \mapsto [\![e]\!]^{\omega(\eta)}]} \\
&= \mathcal{R}[\![\phi]\!]^{\omega(x \mapsto [\![e]\!]^{\omega(\eta)} : \eta)} \\
&\subseteq \text{by the induction hypothesis} \\
&\quad \mathcal{O}_1[\![\theta((x \mapsto [\![e]\!]^{\omega(\eta)} : \eta)\phi)]\!] \\
&= \text{by syntactic environment application (definition 63)} \\
&\quad \mathcal{O}_1[\![\theta(\eta(\phi\,\left[[\![e]\!]^{\omega(\eta)}/x\right]))]\!] \\
&\subseteq \text{by induction over } \eta \text{ and lemma 25} \\
&\quad \mathcal{O}_1[\![\theta(\eta((\lambda x.\,\phi)\,e))]\!]
\end{aligned}
$$

Case: $(\mu A.\,\lambda x.\,\phi)\,e$.

$$
\begin{aligned}
\mathcal{R}[\![(\mu A.\,\lambda x.\,\phi)\,e]\!]^{\omega(\eta)} &= \text{by lemma 27} \\
&\quad \mathcal{R}[\![(\lambda x.\,\phi)\,e]\!]^{\omega(\eta)[A\mapsto\mathcal{R}[\![\mu A.\lambda x.\phi]\!]^{\omega(\eta)}]} \\
&= \mathcal{R}[\![\phi]\!]^{\omega(\eta)[A\mapsto\mathcal{R}[\![\mu A.\lambda x.\phi]\!]^{\omega(\eta)}][x\mapsto[\![e]\!]^{\omega(\eta)}]} \\
&= \mathcal{R}[\![\phi]\!]^{\omega(x\mapsto[\![e]\!]^{\omega(\eta)}:A\mapsto\phi:\eta)} \\
&\subseteq \text{by the induction hypothesis} \\
&\quad \mathcal{O}_1[\![\theta((x\mapsto[\![e]\!]^{\omega(\eta)} : A\mapsto\phi:\eta)\phi)]\!] \\
&= \text{by syntactic environment application (definition 63)} \\
&\quad \mathcal{O}_1[\![\theta(\eta(\phi\,[\mu A.\,\lambda x.\,\phi/A]\,\left[[\![e]\!]^{\omega(\eta)}\,/x\right]))]\!] \\
&\subseteq \text{by induction over } \eta \text{ and lemma 26} \\
&\quad \mathcal{O}_1[\![\theta(\eta((\mu A.\,\lambda x.\,\phi)\,e))]\!]
\end{aligned}
$$

Case: Ae. By induction on η.
Case: fe. By induction on η. □

The following lemma reflects the fact that $\mathcal{O}[\![\phi]\!]$ are obtained from the reflexive, transitive closure of a single step in the operational semantics.

Lemma 29. *If $t \in \mathcal{O}_1[\![\phi]\!]$, then $t \in \mathcal{O}[\![\phi]\!]$.*

Proof. By induction on the derivation of $t \in \mathcal{O}_1[\![\phi]\!]$.
Base case: rule (B.1).
Trivial; $(\wr,\wr) \in \mathcal{O}_1[\![\phi]\!]$ and $(\wr,\wr) \in \mathcal{O}[\![\phi]\!]$.
Base case: rule (B.2).
Let $(h, o) \in \mathcal{O}_1[\![\phi]\!]$. By the premiss, $\phi, h \rightsquigarrow o$, from which it follows that $\phi, h \rightsquigarrow^* o$. Then, by rule (7.17), $(h, o) \in \mathcal{O}[\![\phi]\!]$.
Case: rule (B.3).
Let $(h, h')t \in \mathcal{O}_1[\![\phi]\!]$. By the premiss, there exists ψ, such that $\phi, h \rightsquigarrow \psi, h'$ and $t \in \mathcal{O}_1[\![\psi]\!]$. It follows that $\phi, h \rightsquigarrow^* \psi, h'$. From the induction hypothesis, $t \in \mathcal{O}[\![\psi]\!]$. Then, by rule (7.18), $(h, h')t \in \mathcal{O}[\![\phi]\!]$. □

Corollary 3. *If $t \in \mathcal{O}_1[\![\phi]\!]$, then $t \in (\mathcal{O}[\![\phi]\!])^\dagger$.*

Proof. By lemma 29, $t \in \mathcal{O}_1[\![\phi]\!] \Rightarrow t \in \mathcal{O}[\![\phi]\!]$. By rule (7.23), $t \in \mathcal{O}[\![\phi]\!] \Rightarrow t \in (\mathcal{O}[\![\phi]\!])^\dagger$. Then, by transitivity $t \in \mathcal{O}_1[\![\phi]\!] \Rightarrow t \in (\mathcal{O}[\![\phi]\!])^\dagger$. □

Corollary 4. *If ϕ is closed, then $[\![\phi]\!]^\emptyset \subseteq (\mathcal{O}[\![\phi]\!])^\dagger$.*

Proof.

$$
\begin{aligned}
[\![\phi]\!]^\emptyset &= \text{by lemma 16} \\
&\quad \left(\mathcal{R}[\![\phi]\!]^\emptyset\right)^\dagger \\
&\subseteq \text{by lemma 28 and lemma 12 (increasing property)} \\
&\quad (\mathcal{O}_1[\![\phi]\!])^\dagger \\
&\subseteq \text{by lemma 29 and lemma 12 (increasing property)} \\
&\quad (\mathcal{O}[\![\phi]\!])^\dagger
\end{aligned}
$$

□

Corollary 5. *If ϕ is closed, then $[\![\phi]\!]^{\emptyset} = (\mathcal{O}[\![\phi]\!])^{\dagger}$.*

Proof. From corollary 2, $\mathcal{O}[\![\phi]\!] \subseteq [\![\phi]\!]^{\emptyset}$.
By lemma 12 (increasing), $(\mathcal{O}[\![\phi]\!])^{\dagger} \subseteq \left([\![\phi]\!]^{\emptyset}\right)^{\dagger}$.
By lemma 15, $(\mathcal{O}[\![\phi]\!])^{\dagger} \subseteq [\![\phi]\!]^{\emptyset}$.
Then, by corollary 4, $[\![\phi]\!]^{\emptyset} = (\mathcal{O}[\![\phi]\!])^{\dagger}$. □

B.2. Proofs of General Refinement Laws

Lemma 30 (Refl)**.** $\phi \sqsubseteq \phi$

Proof. Immediate, by definition 52 and reflexivity of $\subseteq$. □

Lemma 31 (Trans)**.** *If $\phi \sqsubseteq \psi'$, and $\psi' \sqsubseteq \psi$, then $\phi \sqsubseteq \psi$.*

Proof. Immediate, by definition 52 and transitivity of $\subseteq$. □

Lemma 32 (AntiSymm)**.** *If $\phi \sqsubseteq \psi$, and $\psi \sqsubseteq \phi$, then $\phi \equiv \psi$.*

Proof. Immediate, by definition 52 and anti-symmetricity of $\subseteq$. □

Lemma 33 (Skip)**.** $\texttt{skip}; \phi \equiv \phi \equiv \phi; \texttt{skip}$

Proof. Let ρ such that it closes ϕ.
By definition 53, $\mathcal{R}[\![\texttt{skip}]\!]^{\rho} = \mathcal{R}[\![a(\mathsf{true}, \mathsf{true})^{\epsilon}_{k}]\!]^{\rho}$.
Then by definitions 61 and 42,
$\mathcal{R}[\![a(\forall \vec{x}.\, \mathsf{true}, \mathsf{true})^{\epsilon}_{k}]\!]^{\rho} = \{(h, h) \mid h \in \textsc{Heap}\} \cup \{(\lightning, \lightning)\}$.
By rules CLStutter and CLMumble, $(\mathcal{R}[\![\texttt{skip}]\!]^{\rho}\,;\mathcal{R}[\![\phi]\!]^{\rho})^{\dagger} = (\mathcal{R}[\![\phi]\!]^{\rho})^{\dagger} = (\mathcal{R}[\![\phi]\!]^{\rho}\,;\mathcal{R}[\![\texttt{skip}]\!]^{\rho})^{\dagger}$.
By lemma 16, $[\![\texttt{skip}; \phi]\!]^{\rho} = [\![\phi]\!]^{\rho} = [\![\phi; \texttt{skip}]\!]^{\rho}$.
By definition 52, $\texttt{skip}; \phi \equiv \phi \equiv \phi; \texttt{skip}$. □

Lemma 34 (Assoc)**.** $\phi; (\psi_1; \psi_2) \equiv (\phi; \psi_1); \psi_2$

Proof. Let ρ such that it closes ϕ, ψ_1 and ψ_2.
By definitions 61 and 47, $\mathcal{R}[\![\phi; (\psi_1; \psi_2)]\!]^{\rho} = \mathcal{R}[\![(\phi; \psi_1); \psi_2]\!]^{\rho}$.
By lemma 12, $(\mathcal{R}[\![\phi; (\psi_1; \psi_2)]\!]^{\rho})^{\dagger} = (\mathcal{R}[\![(\phi; \psi_1); \psi_2]\!]^{\rho})^{\dagger}$.
By lemma 16, $[\![\phi; (\psi_1; \psi_2)]\!]^{\rho} = [\![(\phi; \psi_1); \psi_2]\!]^{\rho}$. □

Lemma 35. $\texttt{miracle} \sqsubseteq \phi$

Proof. Let ρ such that it closes ϕ.
By definition 53, $\mathcal{R}[\![\texttt{miracle}]\!]^{\rho} = \mathcal{R}[\![a(\mathsf{true}, \mathsf{false})^{\epsilon}_{k}]\!]^{\rho}$.
By definitions 61 and 42, $\mathcal{R}[\![a(\mathsf{true}, \mathsf{false})^{\epsilon}_{k}]\!]^{\rho} = \{(\lightning, \lightning)\}$.
By lemma 8, $(\lightning, \lightning) \in \mathcal{R}[\![\phi]\!]^{\rho}$. Therefore $\mathcal{R}[\![\texttt{miracle}]\!]^{\rho} \subseteq \mathcal{R}[\![\phi]\!]^{\rho}$.
By lemma 12, $(\mathcal{R}[\![\texttt{miracle}]\!]^{\rho})^{\dagger} \subseteq (\mathcal{R}[\![\phi]\!]^{\rho})^{\dagger}$.
By lemma 16, $[\![\texttt{miracle}]\!]^{\rho} \subseteq [\![\phi]\!]^{\rho}$.
Thus, by definition 52, $\texttt{miracle} \sqsubseteq \phi$. □

Lemma 36. $\phi \sqsubseteq \texttt{abort}$

Proof. Let ρ such that it closes ϕ.
By definition 53, $\mathcal{R}[\![\texttt{abort}]\!]^\rho = \mathcal{R}[\![a(\mathsf{false},\ \mathsf{true})^\epsilon_k]\!]^\rho$.
By definitions 61 and 42, $\mathcal{R}[\![a(\mathsf{false},\ \mathsf{true})^\epsilon_k]\!]^\rho = (\textsc{Heap} \times \{\lightning\}) \cup \{(\lightning, \lightning)\}$.
By definition 50, and specifically rule (7.25),
$(\mathcal{R}[\![a(\mathsf{false},\ \mathsf{true})^\epsilon_k]\!]^\rho)^\dagger = \textsc{Move}^*; \textsc{Fault}^?$,
Thus, by lemma 16, $[\![a(\mathsf{false},\ \mathsf{true})^\epsilon_k]\!]^\rho = \textsc{Trace}$, is the set of all possible traces; the top element of the refinement lattice.
Therefore, $[\![\phi]\!]^\rho \subseteq [\![\texttt{abort}]\!]^\rho$, and by definition 52, $\phi \sqsubseteq \texttt{abort}$. □

Corollary 6 (MinMax)**.** $\texttt{miracle} \sqsubseteq \phi \sqsubseteq \texttt{abort}$

Proof. By lemma 35 and lemma 36. □

Lemma 37 (EElim)**.** $\phi\,[e/x] \sqsubseteq \exists x.\, \phi$

Proof. Let ρ such that it closes ϕ.
By definition 61, $\mathcal{R}[\![\exists x.\, \phi]\!]^\rho = \bigcup_v \mathcal{R}[\![\phi]\!]^{\rho[x \mapsto v]}$.
By lemma 9, $\mathcal{R}[\![\phi\,[e/x]]\!] = \mathcal{R}[\![\phi]\!]^{\rho[x \mapsto [\![e]\!]^\rho]}$.
Let $v' = [\![e]\!]^\rho$.
Then, $\mathcal{R}[\![\phi\,[e/x]]\!]^\rho \subseteq \mathcal{R}[\![\exists x.\, \phi]\!]^\rho$.
By lemma 12, $(\mathcal{R}[\![\phi\,[e/x]]\!]^\rho)^\dagger \subseteq (\mathcal{R}[\![\exists x.\, \phi]\!]^\rho)^\dagger$.
By lemma 16, $[\![\phi\,[e/x]]\!]^\rho \subseteq [\![\exists x.\, \phi]\!]^\rho$.
Thus, by definition 52, $\phi\,[e/x] \sqsubseteq \exists x.\, \phi$. □

Lemma 38 (EIntro)**.** *If* $x \notin free(\phi)$*, then* $\exists x.\, \phi \sqsubseteq \phi$

Proof. Let ρ such that it closes ϕ.
Directly by definition 61 and $x \notin free(\phi)$, $\mathcal{R}[\![\exists x.\, \phi]\!]^\rho \subseteq \mathcal{R}[\![\phi]\!]^\rho$.
By lemma 12, $(\mathcal{R}[\![\exists x.\, \phi]\!]^\rho)^\dagger \subseteq (\mathcal{R}[\![\phi]\!]^\rho)^\dagger$.
By lemma 16, $[\![\exists x.\, \phi]\!]^\rho \subseteq [\![\phi]\!]^\rho$.
Thus, by definition 52, $\exists x.\, \phi \sqsubseteq \phi$. □

Lemma 39 (AChoiceEq)**.** $\phi \sqcup \phi \equiv \phi$

Proof. Let ρ such that it closes ϕ.
$\mathcal{R}[\![\phi]\!]^\rho \cup \mathcal{R}[\![\phi]\!]^\rho = \mathcal{R}[\![\phi]\!]^\rho$.
By lemma 12, $(\mathcal{R}[\![\phi]\!]^\rho \cup \mathcal{R}[\![\phi]\!]^\rho)^\dagger = (\mathcal{R}[\![\phi]\!]^\rho)^\dagger \cup (\mathcal{R}[\![\phi]\!]^\rho)^\dagger = (\mathcal{R}[\![\phi]\!]^\rho)^\dagger$.
Then, by lemma 16, $[\![\phi \sqcup \phi]\!]^\rho = [\![\phi]\!]^\rho$.
Thus, by definition 52, $\phi \sqcup \phi \equiv \phi$. □

Lemma 40 (AChoiceId)**.** $\phi \sqcup \texttt{miracle} \equiv \phi$

Proof. Let ρ such that it closes ϕ.
By definitions, $\mathcal{R}[\![\texttt{miracle}]\!]^\rho = \{(\lightning, \lightning)\}$.
By lemma 8, $(\lightning, \lightning) \in \mathcal{R}[\![\phi]\!]^\rho$.
Therefore, $\mathcal{R}[\![\phi]\!]^\rho \cup \mathcal{R}[\![\texttt{miracle}]\!]^\rho = \mathcal{R}[\![\phi]\!]^\rho$.
Thus, $\mathcal{R}[\![\phi \sqcup \texttt{miracle}]\!]^\rho = \mathcal{R}[\![\phi]\!]^\rho$.
Then, by lemma 12, $(\mathcal{R}[\![\phi \sqcup \texttt{miracle}]\!]^\rho)^\dagger = (\mathcal{R}[\![\phi]\!]^\rho)^\dagger$.
Then, by lemma 16, $[\![\phi \sqcup \texttt{miracle}]\!]^\rho = [\![\phi]\!]^\rho$.
Thus, by definition 52, $\phi \sqcup \texttt{miracle} \equiv \phi$. □

Lemma 41 (AChoiceAssoc)**.** $\phi \sqcup (\psi_1 \sqcup \psi_2) \equiv (\phi \sqcup \psi_1) \sqcup \psi_2$

Proof. Let ρ such that it closes ϕ, ψ_1 and ψ_2.
By associativity of set union, $\mathcal{R}[\![\phi]\!]^\rho \cup (\mathcal{R}[\![\psi_1]\!]^\rho \cup \mathcal{R}[\![\psi_2]\!]^\rho) = (\mathcal{R}[\![\phi]\!]^\rho \cup \mathcal{R}[\![\psi_1]\!]^\rho) \cup \mathcal{R}[\![\psi_2]\!]^\rho$.
Thus, $\mathcal{R}[\![\phi \sqcup (\psi_1 \sqcup \psi_2)]\!]^\rho = \mathcal{R}[\![(\phi \sqcup \psi_1) \sqcup \psi_2]\!]^\rho$.
Then, by lemma 12 $(\mathcal{R}[\![\phi \sqcup (\psi_1 \sqcup \psi_2)]\!]^\rho)^\dagger = (\mathcal{R}[\![(\phi \sqcup \psi_1) \sqcup \psi_2]\!]^\rho)^\dagger$.
Then, by lemma 16, $[\![\phi \sqcup (\psi_1 \sqcup \psi_2)]\!]^\rho = [\![(\phi \sqcup \psi_1) \sqcup \psi_2]\!]^\rho$.
Thus, by definition 52, $\phi \sqcup (\psi_1 \sqcup \psi_2) \equiv (\phi \sqcup \psi_1) \sqcup \psi_2$. □

Lemma 42 (AChoiceComm)**.** $\phi \sqcup \psi \equiv \psi \sqcup \phi$

Proof. Let ρ such that it closes ϕ and ψ.
By commutativity of set union, $\mathcal{R}[\![\phi]\!]^\rho \cup \mathcal{R}[\![\psi]\!]^\rho = \mathcal{R}[\![\psi]\!]^\rho \cup \mathcal{R}[\![\phi]\!]^\rho$.
Therefore, $\mathcal{R}[\![\phi \sqcup \psi]\!]^\rho = \mathcal{R}[\![\psi \sqcup \phi]\!]^\rho$.
Then, by lemma 12 $(\mathcal{R}[\![\phi \sqcup \psi]\!]^\rho)^\dagger = (\mathcal{R}[\![\psi \sqcup \phi]\!]^\rho)^\dagger$.
Then, by lemma 16, $[\![\phi \sqcup \psi]\!]^\rho = [\![\psi \sqcup \phi]\!]^\rho$.
Thus, by definition 52, $\phi \sqcup \psi \equiv \psi \sqcup \phi$. □

Lemma 43 (AChoiceElim)**.** $\phi \sqsubseteq \phi \sqcup \psi$

Proof. Let ρ such that it closes ϕ and ψ.
By set theory, $\mathcal{R}[\![\phi]\!]^\rho \subseteq \mathcal{R}[\![\phi]\!]^\rho \cup \mathcal{R}[\![\psi]\!]^\rho$.
Therefore, $\mathcal{R}[\![\phi]\!]^\rho \subseteq \mathcal{R}[\![\phi \sqcup \psi]\!]^\rho$.
Thus, by lemma 12, $(\mathcal{R}[\![\phi]\!]^\rho)^\dagger \subseteq (\mathcal{R}[\![\phi \sqcup \psi]\!]^\rho)^\dagger$.
Thus, by lemma 16, $[\![\phi]\!]^\rho \subseteq [\![\phi \sqcup \psi]\!]^\rho$.
Thus, by definition 52, $\phi \sqsubseteq \phi \sqcup \psi$. □

Lemma 44. $(T_1 \cup T_2); T_3 = (T_1; T_3) \cup (T_2; T_3)$

Proof. By definitions, $(T_1 \cup T_2); T_3 = \{st \mid s \in T_1 \cup T_2, t \in T_3\}$.
Also, $T_1; T_3 = \{st \mid s \in T_1, t \in T_3\}$.
Also, $T_2; T_3 = \{st \mid s \in T_2, t \in T_3\}$.
Then, $(T_1; T_3) \cup (T_2; T_3) = \{st \mid s \in T_1 \cup T_2, t \in T_3\}$ □

Corollary 7 (AChoiceDstR)**.** $(\phi_1 \sqcup \phi_2); \psi \equiv (\phi_1; \psi) \sqcup (\phi_2; \psi)$

Proof. Let ρ such that it closes ϕ and ψ.
By lemma 44, $(\mathcal{R}[\![\phi_1]\!]^\rho \cup \mathcal{R}[\![\phi_2]\!]^\rho) ; \mathcal{R}[\![\psi]\!]^\rho = (\mathcal{R}[\![\phi_1]\!]^\rho ; \mathcal{R}[\![\psi]\!]^\rho) \cup (\mathcal{R}[\![\phi_2]\!]^\rho ; \mathcal{R}[\![\psi]\!]^\rho)$.
Thus, $((\mathcal{R}[\![\phi_1]\!]^\rho \cup \mathcal{R}[\![\phi_2]\!]^\rho) ; \mathcal{R}[\![\psi]\!]^\rho)^\dagger = ((\mathcal{R}[\![\phi_1]\!]^\rho ; \mathcal{R}[\![\psi]\!]^\rho) \cup (\mathcal{R}[\![\phi_2]\!]^\rho ; \mathcal{R}[\![\psi]\!]^\rho))^\dagger$.
By definition 61, $(\mathcal{R}[\![(\phi_1 \sqcup \phi_1); \psi]\!]^\rho)^\dagger = (\mathcal{R}[\![(\phi_1; \psi) \sqcup (\phi_2; \psi)]\!]^\rho)^\dagger$.
By lemma 16, $[\![(\phi_1 \sqcup \phi_1); \psi]\!]^\rho = [\![(\phi_1; \psi) \sqcup (\phi_2; \psi)]\!]^\rho$.
Thus, by definition 52, $(\phi_1 \sqcup \phi_1); \psi \equiv (\phi_1; \psi) \sqcup (\phi_2; \psi)$. □

Lemma 45. $T_1; (T_2 \cup T_3) = (T_1; T_2) \cup (T_1; T_3)$

Proof. By definition, $T_1; (T_2 \cup T_3) = \{st \mid s \in T_1, t \in T_2 \cup T_3\}$.
Also, $T_1; T_2 = \{st \mid s \in T_1, t \in T_2\}$.
Also, $T_1; T_3 = \{st \mid s \in T_1, t \in T_3\}$.
Then, $(T_1; T_2) \cup (T_1; T_3) = \{st \mid s \in T_1, t \in T_2 \cup T_3\}$. □

Corollary 8 (AChoiceDstL)**.** $\psi; (\phi_1 \sqcup \phi_2) \equiv (\psi; \phi_1) \sqcup (\psi; \phi_2)$

Proof. Let ρ such that it closes ϕ_1, ϕ_2 and ψ.
By lemma 45, $\mathcal{R}[\![\psi]\!]^\rho ; (\mathcal{R}[\![\phi_1]\!]^\rho \cup \mathcal{R}[\![\phi_2]\!]^\rho) = (\mathcal{R}[\![\psi]\!]^\rho ; \mathcal{R}[\![\phi_1]\!]^\rho) \cup (\mathcal{R}[\![\psi]\!]^\rho ; \mathcal{R}[\![\phi_2]\!]^\rho)$.
Thus, $(\mathcal{R}[\![\psi]\!]^\rho ; (\mathcal{R}[\![\phi_1]\!]^\rho \cup \mathcal{R}[\![\phi_2]\!]^\rho))^\dagger = ((\mathcal{R}[\![\psi]\!]^\rho ; \mathcal{R}[\![\phi_1]\!]^\rho) \cup (\mathcal{R}[\![\psi]\!]^\rho ; \mathcal{R}[\![\phi_2]\!]^\rho))^\dagger$.
By definition 61, $(\mathcal{R}[\![\psi; (\phi_1 \sqcup \phi_2)]\!]^\rho)^\dagger = (\mathcal{R}[\![(\psi; \phi_1) \sqcup (\psi; \phi_2)]\!]^\rho)^\dagger$.
By lemma 16, $[\![\psi; (\phi_1 \sqcup \phi_2)]\!]^\rho = [\![(\psi; \phi_1) \sqcup (\psi; \phi_2)]\!]^\rho$.
Thus, by definition 52, $\psi; (\phi_1 \sqcup \phi_2) \equiv (\psi; \phi_1) \sqcup (\psi; \phi_2)$. □

Lemma 46 (DChoiceEq)**.** $\phi \sqcap \phi \equiv \phi$

Proof. Let ρ such that it closes ϕ.
By set intersection, $\mathcal{R}[\![\phi]\!]^\rho \cap \mathcal{R}[\![\phi]\!]^\rho = \mathcal{R}[\![\phi]\!]^\rho$.
Therefore, $\mathcal{R}[\![\phi \sqcap \phi]\!]^\rho = \mathcal{R}[\![\phi]\!]^\rho$.
Then, by lemma 12, $(\mathcal{R}[\![\phi \sqcap \psi]\!]^\rho)^\dagger = (\mathcal{R}[\![\phi]\!]^\rho)^\dagger$.
Then, by lemma 16, $[\![\phi \sqcap \psi]\!]^\rho = [\![\phi]\!]^\rho$.
Thus, by definition 52, $\phi \sqcap \phi \equiv \phi$. □

Lemma 47 (DChoiceId)**.** $\phi \sqcap \texttt{abort} \equiv \phi$

Proof. Let ρ such that it closes ϕ.
By definitions, $[\![\texttt{abort}]\!]^\rho = \textsc{Trace}$.
Then, by the fact that Trace is the top element in the $\mathcal{P}(\textsc{Trace})$ lattice, and by set intersection, $[\![\phi]\!]^\rho \cap [\![\texttt{abort}]\!]^\rho = [\![\phi]\!]^\rho$.
Then, by lemma 12, $([\![\phi]\!]^\rho \cap [\![\texttt{abort}]\!]^\rho)^\dagger = ([\![\phi]\!]^\rho)^\dagger$.
Then, by lemma 15 and definition 51, $[\![\phi \sqcap \texttt{abort}]\!]^\rho = [\![\phi]\!]^\rho$.
Thus, by definition 52, $\phi \sqcap \texttt{abort} \equiv \phi$. □

Lemma 48 (DChoiceAssoc)**.** $\phi \sqcap (\psi_1 \sqcap \psi_2) \equiv (\phi \sqcap \psi_1) \sqcap \psi_2$

Proof. Let ρ such that it closes ϕ, ψ_1 and ψ_2.
By associativity of set intersection, $\mathcal{R}[\![\phi]\!]^\rho \cap (\mathcal{R}[\![\psi_1]\!]^\rho \cap \mathcal{R}[\![\psi_2]\!]^\rho) = (\mathcal{R}[\![\phi]\!]^\rho \cap \mathcal{R}[\![\psi_1]\!]^\rho) \cap \mathcal{R}[\![\psi_2]\!]^\rho$.
Thus, $\mathcal{R}[\![\phi \sqcap (\psi_1 \sqcap \psi_2)]\!]^\rho = \mathcal{R}[\![(\phi \sqcap \psi_1) \sqcap \psi_2]\!]^\rho$.
Then, by lemma 12 $(\mathcal{R}[\![\phi \sqcap (\psi_1 \sqcap \psi_2)]\!]^\rho)^\dagger = (\mathcal{R}[\![(\phi \sqcap \psi_1) \sqcap \psi_2]\!]^\rho)^\dagger$.
Then, by lemma 16, $[\![\phi \sqcap (\psi_1 \sqcap \psi_2)]\!]^\rho = [\![(\phi \sqcap \psi_1) \sqcap \psi_2]\!]^\rho$.
Thus, by definition 52, $\phi \sqcap (\psi_1 \sqcap \psi_2) \equiv (\phi \sqcap \psi_1) \sqcap \psi_2$. □

Lemma 49 (DChoiceComm)**.** $\phi \sqcap \psi \equiv \psi \sqcap \phi$

Proof. Let ρ such that it closes ϕ and ψ.
By commutativity of set intersection, $\mathcal{R}[\![\phi]\!]^\rho \cap \mathcal{R}[\![\psi]\!]^\rho = \mathcal{R}[\![\psi]\!]^\rho \cap \mathcal{R}[\![\phi]\!]^\rho$.
Therefore, $\mathcal{R}[\![\phi \sqcap \psi]\!]^\rho = \mathcal{R}[\![\psi \sqcap \phi]\!]^\rho$.
Then, by lemma 12 $(\mathcal{R}[\![\phi \sqcap \psi]\!]^\rho)^\dagger = (\mathcal{R}[\![\psi \sqcap \phi]\!]^\rho)^\dagger$.
Then, by lemma 16, $[\![\phi \sqcap \psi]\!]^\rho = [\![\psi \sqcap \phi]\!]^\rho$.
Thus, by definition 52, $\phi \sqcap \psi \equiv \psi \sqcap \phi$. □

Lemma 50 (DChoiceElim)**.** *If* $\phi \sqsubseteq \psi_1$ *and* $\phi \sqsubseteq \psi_2$*, then* $\phi \sqsubseteq \psi_1 \sqcap \psi_2$.

Proof. Assume premisses hold. Then,

$$
\begin{aligned}
\psi_1 \sqcap \psi_2 &\sqsupseteq \text{ by first premiss and CMono} \\
&\quad \phi \sqcap \psi_2 \\
&\sqsupseteq \text{ by second premiss and CMono} \\
&\quad \phi \sqcap \phi \\
&\equiv \text{ by DChoiceEq} \\
&\quad \phi
\end{aligned}
$$

□

Lemma 51 (DChoiceIntro)**.** $\phi \sqcap \psi \sqsubseteq \phi$

Proof. Let ρ such that it closes ϕ and ψ.
By set intersection, $\mathcal{R}[\![\phi]\!]^\rho \cap \mathcal{R}[\![\psi]\!]^\rho \subseteq \mathcal{R}[\![\phi]\!]^\rho$.
Then, by definition 61, $\mathcal{R}[\![\phi \sqcap \psi]\!]^\rho \subseteq \mathcal{R}[\![\phi]\!]^\rho$.
Then, by lemma 12, $(\mathcal{R}[\![\phi \sqcap \psi]\!]^\rho)^\dagger \subseteq (\mathcal{R}[\![\phi]\!]^\rho)^\dagger$.
Then, by lemma 16, $[\![\phi \sqcap \psi]\!]^\rho \subseteq [\![\phi]\!]^\rho$.
Thus, by definition 52, $\phi \sqcap \psi \sqsubseteq \phi$. □

Lemma 52. $(T_1 \cap T_2); T_3 = (T_1; T_3) \cap (T_2; T_3)$

Proof. By definitions, $(T_1 \cap T_2); T_3 = \{st \mid s \in T_1 \cap T_2, t \in T_3\}$.
Also, $T_1; T_3 = \{st \mid s \in T_1, t \in T_3\}$.
Also, $T_2; T_3 = \{st \mid s \in T_2, t \in T_3\}$.
Then, $(T_1; T_3) \cap (T_2; T_3) = \{st \mid s \in T_1 \cap T_2, t \in T_3\}$ □

Corollary 9 (DChoiceDstR)**.** $(\phi_1 \sqcap \phi_2); \psi \equiv (\phi_1; \psi) \sqcap (\phi_2; \psi)$.

Proof. Let ρ such that it closes ϕ_1, ϕ_2 and ψ.
By lemma 52, $\mathcal{R}[\![\psi]\!]^\rho ; (\mathcal{R}[\![\phi_1]\!]^\rho \cap \mathcal{R}[\![\phi_2]\!]^\rho) = (\mathcal{R}[\![\psi]\!]^\rho ; \mathcal{R}[\![\phi_1]\!]^\rho) \cap (\mathcal{R}[\![\psi]\!]^\rho ; \mathcal{R}[\![\phi_2]\!]^\rho)$.
Thus, $(\mathcal{R}[\![\psi]\!]^\rho ; (\mathcal{R}[\![\phi_1]\!]^\rho \cap \mathcal{R}[\![\phi_2]\!]^\rho))^\dagger = ((\mathcal{R}[\![\psi]\!]^\rho ; \mathcal{R}[\![\phi_1]\!]^\rho) \cap (\mathcal{R}[\![\psi]\!]^\rho ; \mathcal{R}[\![\phi_2]\!]^\rho))^\dagger$.
By definition 61, $(\mathcal{R}[\![\psi; (\phi_1 \sqcap \phi_2)]\!]^\rho)^\dagger = (\mathcal{R}[\![(\psi; \phi_1) \sqcap (\psi; \phi_2)]\!]^\rho)^\dagger$.
By lemma 16, $[\![\psi; (\phi_1 \sqcap \phi_2)]\!]^\rho = [\![(\psi; \phi_1) \sqcap (\psi; \phi_2)]\!]^\rho$.
Thus, by definition 52, $\psi; (\phi_1 \sqcap \phi_2) \equiv (\psi; \phi_1) \sqcap (\psi; \phi_2)$. □

Lemma 53. $T_1; (T_2 \cap T_3) = (T_1; T_2) \cap (T_1; T_3)$

Proof. By definition, $T_1; (T_2 \cap T_3) = \{st \mid s \in T_1, t \in T_2 \cap T_3\}$.
Also, $T_1; T_2 = \{st \mid s \in T_1, t \in T_2\}$.
Also, $T_1; T_3 = \{st \mid s \in T_1, t \in T_3\}$.
Then, $(T_1; T_2) \cap (T_1; T_3) = \{st \mid s \in T_1, t \in T_2 \cap T_3\}$. □

Corollary 10 (DChoiceDstL)**.** $\psi; (\phi_1 \sqcap \phi_2) \equiv (\psi; \phi_1) \sqcap (\psi; \phi_2)$

Proof. Let ρ such that it closes ϕ_1, ϕ_2 and ψ.
By lemma 45, $\mathcal{R}[\![\psi]\!]^\rho ; (\mathcal{R}[\![\phi_1]\!]^\rho \cap \mathcal{R}[\![\phi_2]\!]^\rho) = (\mathcal{R}[\![\psi]\!]^\rho ; \mathcal{R}[\![\phi_1]\!]^\rho) \cap (\mathcal{R}[\![\psi]\!]^\rho ; \mathcal{R}[\![\phi_2]\!]^\rho)$.
Thus, $(\mathcal{R}[\![\psi]\!]^\rho ; (\mathcal{R}[\![\phi_1]\!]^\rho \cap \mathcal{R}[\![\phi_2]\!]^\rho))^\dagger = ((\mathcal{R}[\![\psi]\!]^\rho ; \mathcal{R}[\![\phi_1]\!]^\rho) \cap (\mathcal{R}[\![\psi]\!]^\rho ; \mathcal{R}[\![\phi_2]\!]^\rho))^\dagger$.
By definition 61, $(\mathcal{R}[\![\psi; (\phi_1 \sqcap \phi_2)]\!]^\rho)^\dagger = (\mathcal{R}[\![(\psi; \phi_1) \sqcap (\psi; \phi_2)]\!]^\rho)^\dagger$.
By lemma 16, $[\![\psi; (\phi_1 \sqcap \phi_2)]\!]^\rho = [\![(\psi; \phi_1) \sqcap (\psi; \phi_2)]\!]^\rho$.
Thus, by definition 52, $\psi; (\phi_1 \sqcap \phi_2) \equiv (\psi; \phi_1) \sqcap (\psi; \phi_2)$. □

Lemma 54 (AChoiceDstD)**.** $\phi \sqcup (\psi_1 \sqcap \psi_2) \equiv (\phi \sqcup \psi_1) \sqcap (\phi \sqcup \psi_2)$

Proof. Let ρ such that it closes ϕ, ψ_1 and ψ_2.
By distributivity of set union over set intersection,

$$\mathcal{R}[\![\phi]\!]^\rho \cup (\mathcal{R}[\![\psi_1]\!]^\rho \cap \mathcal{R}[\![\psi_2]\!]^\rho) = (\mathcal{R}[\![\phi]\!]^\rho \cup \mathcal{R}[\![\psi_1]\!]^\rho) \cap (\mathcal{R}[\![\phi]\!]^\rho \cup \mathcal{R}[\![\psi_2]\!]^\rho)$$

Therefore, by definition 61, $\mathcal{R}[\![\phi \sqcup (\psi_1 \sqcap \psi_2)]\!]^\rho = \mathcal{R}[\![(\phi \sqcup \psi_1) \sqcap (\phi \sqcup \psi_2)]\!]^\rho$.
Then, by lemma 12, $(\mathcal{R}[\![\phi \sqcup (\psi_1 \sqcap \psi_2)]\!]^\rho)^\dagger = (\mathcal{R}[\![(\phi \sqcup \psi_1) \sqcap (\phi \sqcup \psi_2)]\!]^\rho)^\dagger$.
Then, by lemma 16, $[\![\phi \sqcup (\psi_1 \sqcap \psi_2)]\!]^\rho = [\![(\phi \sqcup \psi_1) \sqcap (\phi \sqcup \psi_2)]\!]^\rho$.
Thus, by definition 52, $\phi \sqcup (\psi_1 \sqcap \psi_2) \equiv (\phi \sqcup \psi_1) \sqcap (\phi \sqcup \psi_2)$. □

Lemma 55 (DChoiceDstA)**.** $\phi \sqcap (\psi_1 \sqcup \psi_2) \equiv (\phi \sqcap \psi_1) \sqcup (\phi \sqcap \psi_2)$

Proof. Let ρ such that it closes ϕ, ψ_1 and ψ_2.
By distributivity of set intersection over set union,

$$\mathcal{R}[\![\phi]\!]^\rho \cap (\mathcal{R}[\![\psi_1]\!]^\rho \cup \mathcal{R}[\![\psi_2]\!]^\rho) = (\mathcal{R}[\![\phi]\!]^\rho \cap \mathcal{R}[\![\psi_1]\!]^\rho) \cup (\mathcal{R}[\![\phi]\!]^\rho \cap \mathcal{R}[\![\psi_2]\!]^\rho)$$

Therefore, by definition 61, $\mathcal{R}[\![\phi \sqcap (\psi_1 \sqcup \psi_2)]\!]^\rho = \mathcal{R}[\![(\phi \sqcap \psi_1) \sqcup (\phi \sqcap \psi_2)]\!]^\rho$.
Then, by lemma 12, $(\mathcal{R}[\![\phi \sqcap (\psi_1 \sqcup \psi_2)]\!]^\rho)^\dagger = (\mathcal{R}[\![(\phi \sqcap \psi_1) \sqcup (\phi \sqcap \psi_2)]\!]^\rho)^\dagger$.
Then, by lemma 16, $[\![\phi \sqcap (\psi_1 \sqcup \psi_2)]\!]^\rho = [\![(\phi \sqcap \psi_1) \sqcup (\phi \sqcap \psi_2)]\!]^\rho$.
Thus, by definition 52, $\phi \sqcap (\psi_1 \sqcup \psi_2) \equiv (\phi \sqcap \psi_1) \sqcup (\phi \sqcap \psi_2)$. □

Lemma 56 (Absorb). $\phi \sqcup (\phi \sqcap \psi) \equiv \phi \equiv \phi \sqcap (\phi \sqcup \psi)$.

Proof. Let ρ such that it closes ϕ and ψ.
By the absorption property in the lattice of powersets,

$$\mathcal{R}[\![\phi]\!]^\rho \cup (\mathcal{R}[\![\phi]\!]^\rho \cap \mathcal{R}[\![\psi]\!]^\rho) = \mathcal{R}[\![\phi]\!]^\rho = \mathcal{R}[\![\phi]\!]^\rho \cap (\mathcal{R}[\![\phi]\!]^\rho \cup \mathcal{R}[\![\psi]\!]^\rho)$$

Therefore, by definition 61, $\mathcal{R}[\![\phi \sqcup (\phi \sqcap \psi)]\!]^\rho = \mathcal{R}[\![\phi]\!]^\rho = \mathcal{R}[\![\phi \sqcap (\phi \sqcup \psi)]\!]^\rho$.
Then, by lemma 12, $(\mathcal{R}[\![\phi \sqcup (\phi \sqcap \psi)]\!]^\rho)^\dagger = (\mathcal{R}[\![\phi]\!]^\rho)^\dagger = (\mathcal{R}[\![\phi \sqcap (\phi \sqcup \psi)]\!]^\rho)^\dagger$.
Then, by lemma 16, $[\![\phi \sqcup (\phi \sqcap \psi)]\!]^\rho = [\![\phi]\!]^\rho = [\![\phi \sqcap (\phi \sqcup \psi)]\!]^\rho$.
Thus, by definition 52, $\phi \sqcup (\phi \sqcap \psi) \equiv \phi \equiv \phi \sqcap (\phi \sqcup \psi)$. □

Lemma 57 (Demonise). $\phi \sqcap \psi \sqsubseteq \phi \sqcup \psi$

Proof.

$$\begin{aligned} \phi \sqcap \psi \sqsubseteq & \text{ by DChoiceIntro} \\ & \phi \\ \sqsubseteq & \text{ by AChoiceElim} \\ & \phi \sqcup \psi \end{aligned}$$

□

Lemma 58 (ParSkip). $\phi \parallel \texttt{skip} \equiv \phi$

Proof. By definitions, and CLStutter and CLMumble rules. □

Lemma 59 (ParAssoc). $\phi \parallel (\psi_1 \parallel \psi_2) \equiv (\phi \parallel \psi_1) \parallel \psi_2$

Proof. Let ρ such that it closes ϕ, ψ_1 and ψ_2.
By lemma 17, $\mathcal{R}[\![\phi]\!]^\rho \parallel (\mathcal{R}[\![\psi_1]\!]^\rho \parallel \mathcal{R}[\![\psi_2]\!]^\rho) = (\mathcal{R}[\![\phi]\!]^\rho \parallel \mathcal{R}[\![\psi_1]\!]^\rho) \parallel \mathcal{R}[\![\psi_2]\!]^\rho$.
Therefore, by definition 61, $\mathcal{R}[\![\phi \parallel (\psi_1 \parallel \psi_2)]\!]^\rho = \mathcal{R}[\![(\phi \parallel \psi_1) \parallel \psi_2]\!]^\rho$.
Then, by lemma 12, $(\mathcal{R}[\![\phi \parallel (\psi_1 \parallel \psi_2)]\!]^\rho)^\dagger = (\mathcal{R}[\![(\phi \parallel \psi_1) \parallel \psi_2]\!]^\rho)^\dagger$.
Then, by lemma 16, $[\![\phi \parallel (\psi_1 \parallel \psi_2)]\!]^\rho = [\![(\phi \parallel \psi_1) \parallel \psi_2]\!]^\rho$.
Thus, by definition 52, $\phi \parallel (\psi_1 \parallel \psi_2) \equiv (\phi \parallel \psi_1) \parallel \psi_2$. □

Lemma 60 (ParComm). $\phi \parallel \psi \equiv \psi \parallel \phi$

Proof. Let ρ such that it closes ϕ and ψ.
By definition 51, $[\![\phi \parallel \psi]\!]^\rho = ([\![\phi]\!]^\rho \parallel [\![\psi]\!]^\rho)^\dagger$.
By lemma 17, $([\![\phi]\!]^\rho \parallel [\![\psi]\!]^\rho)^\dagger = ([\![\psi]\!]^\rho \parallel [\![\phi]\!]^\rho)^\dagger$.
Thus, by definition 51, $[\![\phi \parallel \psi]\!]^\rho = [\![\psi \parallel \phi]\!]^\rho$.
Therefore, by definition 52, $\phi \parallel \psi \equiv \psi \parallel \phi$. □

Lemma 61 (Exchange). $(\phi \parallel \psi); (\phi' \parallel \psi') \sqsubseteq (\phi; \phi') \parallel (\psi; \psi')$

Proof. Let ρ such that is closes ϕ, ϕ', ψ, ψ'.
By definition 61, $\mathcal{R}[\![(\phi \parallel \psi); (\phi' \parallel \psi')]\!]^\rho = (\mathcal{R}[\![\phi]\!]^\rho \parallel \mathcal{R}[\![\psi]\!]^\rho); (\mathcal{R}[\![\phi']\!]^\rho \parallel \mathcal{R}[\![\psi']\!]^\rho)$.
Let $t_1 \in \mathcal{R}[\![\phi]\!]^\rho$, $u_1 \in \mathcal{R}[\![\psi]\!]^\rho$, $t_2 \in \mathcal{R}[\![\phi']\!]^\rho$, $u_2 \in \mathcal{R}[\![\psi']\!]^\rho$.
Let $s \in t_1 \parallel u_1$ and $w \in t_2 \parallel u_2$.
We prove that $sw \in t_1 t_2 \parallel u_1 u_2$ by induction on the derivation of $s \in t \parallel u$.
Base case: $(h, \{\!\}) \in (h, \{\!\}) \parallel u_1 u_2$, directly by rule (7.22).
Base case: $(h, h') u_1 u_2 \in (h, h') \parallel u_1 u_2$, directly by rule (7.21).
Case $(h, h') sw \in (h, h') t_1 t_2 \parallel u_1 u_2$:
By rule (7.20), $(h, h') s \in (h, h') t_1 \parallel u_1$.
Then, by the induction hypothesis: $(h, h') sw \in (h, h') t_1 t_2 \parallel u_1 u_2$.
Case $sw \in u_1 u_2 \parallel t_1 t_2$:
By rule (7.19), $s \in u_1 \parallel t_1$ and $w \in u_2 \parallel t_2$.
Then, by the induction hypothesis: $sw \in u_1 u_2 \parallel t_1 t_2$.
Therefore, $(\mathcal{R}[\![\phi]\!]^\rho \parallel \mathcal{R}[\![\psi]\!]^\rho); (\mathcal{R}[\![\phi']\!]^\rho \parallel \mathcal{R}[\![\psi']\!]^\rho) \subseteq (\mathcal{R}[\![\phi]\!]^\rho ; \mathcal{R}[\![\phi']\!]^\rho) \parallel (\mathcal{R}[\![\psi]\!]^\rho ; \mathcal{R}[\![\psi']\!]^\rho)$.
Thus, by definition 61, $\mathcal{R}[\![(\phi \parallel \psi); (\phi' \parallel \psi')]\!]^\rho \subseteq \mathcal{R}[\![(\phi; \phi') \parallel (\psi; \psi')]\!]^\rho$.
Then, by lemma 12, $(\mathcal{R}[\![(\phi \parallel \psi); (\phi' \parallel \psi')]\!]^\rho)^\dagger \subseteq (\mathcal{R}[\![(\phi; \phi') \parallel (\psi; \psi')]\!]^\rho)^\dagger$.
By lemma 16, $[\![(\phi \parallel \psi); (\phi' \parallel \psi')]\!]^\rho \subseteq [\![(\phi; \phi') \parallel (\psi; \psi')]\!]^\rho$.
Therefore, by definition 52: $(\phi \parallel \psi); (\phi' \parallel \psi') \sqsubseteq (\phi; \phi') \parallel (\psi; \psi')$. □

Lemma 62. $(S \parallel T) \cup (S' \parallel T') \subseteq (S \cup S') \parallel (T \cup T')$

Proof. Immediate by definition 49. □

Corollary 11 (AChoiceExchange)**.** $(\phi \parallel \psi) \sqcup (\phi' \parallel \psi') \sqsubseteq (\phi \sqcup \phi') \parallel (\psi \sqcup \psi')$

Proof. Let ρ such that it closes ϕ, ϕ', ψ and ψ'.
By lemma 62, $(\mathcal{R}[\![\phi]\!]^\rho \parallel \mathcal{R}[\![\psi]\!]^\rho) \cup (\mathcal{R}[\![\phi']\!]^\rho \parallel \mathcal{R}[\![\psi']\!]^\rho) \subseteq (\mathcal{R}[\![\phi]\!]^\rho \sqcup \mathcal{R}[\![\phi']\!]^\rho) \parallel (\mathcal{R}[\![\psi]\!]^\rho \sqcup \mathcal{R}[\![\psi']\!]^\rho)$.
Therefore, by definition 61, $\mathcal{R}[\![(\phi \parallel \psi) \sqcup (\phi' \parallel \psi')]\!]^\rho \subseteq \mathcal{R}[\![(\phi \sqcup \phi') \parallel (\psi \sqcup \psi')]\!]^\rho$.
Then, by lemma 12, $(\mathcal{R}[\![(\phi \parallel \psi) \sqcup (\phi' \parallel \psi')]\!]^\rho)^\dagger \subseteq (\mathcal{R}[\![(\phi \sqcup \phi') \parallel (\psi \sqcup \psi')]\!]^\rho)^\dagger$.
Then, by lemma 16, $[\![(\phi \parallel \psi) \sqcup (\phi' \parallel \psi')]\!]^\rho \subseteq [\![(\phi \sqcup \phi') \parallel (\psi \sqcup \psi')]\!]^\rho$.
Thus, by definition 52, $(\phi \parallel \psi) \sqcup (\phi' \parallel \psi') \sqsubseteq (\phi \sqcup \phi') \parallel (\psi \sqcup \psi')$. □

Lemma 63 (SeqPar)**.** $\phi; \psi \sqsubseteq \phi \parallel \psi$

Proof.

$$
\begin{aligned}
\phi; \psi \equiv & \text{ by ParSkip and CMono} \\
& (\phi \parallel \texttt{skip}); (\psi \parallel \texttt{skip}) \\
\sqsubseteq & \text{ by Exchange and CMono rules} \\
& (\phi; \texttt{skip}) \parallel (\psi; \texttt{skip}) \\
\equiv & \text{ by Skip and CMono} \\
& \phi \parallel \psi
\end{aligned}
$$

□

Lemma 64 (ParDstLR). $\phi;(\psi_1 \parallel \psi_2) \sqsubseteq (\phi;\psi_1) \parallel \psi_2$

Proof.

$$\begin{aligned}
\phi;(\psi_1 \parallel \psi_2) &\equiv \text{by ParSkip and CMono} \\
&\quad (\phi \parallel \texttt{skip});(\psi_1 \parallel \psi_2) \\
&\sqsubseteq \text{by Exchange} \\
&\quad (\phi;\psi_1) \parallel (\texttt{skip};\psi_2) \\
&\equiv \text{by Skip and CMono} \\
&\quad (\phi;\psi_1) \parallel \psi_2
\end{aligned}$$

□

Lemma 65 (ParDstLL). $\phi;(\psi_1 \parallel \psi_2) \sqsubseteq \psi_1 \parallel (\phi;\psi_2)$

Proof.

$$\begin{aligned}
\phi;(\psi_1 \parallel \psi_2) &\equiv \text{by ParSkip, ParComm and CMono} \\
&\quad (\texttt{skip} \parallel \phi);(\psi_1 \parallel \psi_2) \\
&\sqsubseteq \text{by Exchange} \\
&\quad (\texttt{skip};\psi_1) \parallel (\phi;\psi_2) \\
&\equiv \text{by Skip and CMono} \\
&\quad \psi_1 \parallel (\phi;\psi_2)
\end{aligned}$$

□

Lemma 66 (ParDstRL). $(\phi \parallel \psi_1);\psi_2 \sqsubseteq \phi \parallel (\psi_1;\psi_2)$

Proof.

$$\begin{aligned}
(\phi \parallel \psi_1);\psi_2 &\equiv \text{by ParSkip, ParComm and CMono} \\
&\quad (\phi \parallel \psi_1);(\texttt{skip} \parallel \psi_2) \\
&\sqsubseteq \text{by Exchange} \\
&\quad (\phi;\texttt{skip}) \parallel (\psi_1;\psi_2) \\
&\equiv \text{by Skip and CMono} \\
&\quad \phi \parallel (\psi_1;\psi_2)
\end{aligned}$$

□

Lemma 67 (ParDstRR). $(\phi \parallel \psi_1);\psi_2 \sqsubseteq (\phi;\psi_2) \parallel \psi_1$

Proof.

$$
\begin{aligned}
(\phi \parallel \psi_1); \psi_2 \equiv & \text{ by } \textsc{Skip} \text{ and } \textsc{CMono} \\
& (\phi \parallel \psi_1); (\psi_2 \parallel \texttt{skip}) \\
\sqsubseteq & \text{ by } \textsc{Exchange} \\
& (\phi; \psi_2) \parallel (\psi_1; \texttt{skip}) \\
\equiv & \text{ by } \textsc{Skip} \text{ and } \textsc{CMono} \\
& (\phi; \psi_2) \parallel \psi_1
\end{aligned}
$$

□

Lemma 68 (AChoiceEq)**.** $\exists x.\, \phi \equiv \bigsqcup_{v \in \text{VAL}} \phi\,[v/x]$

Proof. Let ρ such that it closes ϕ.
By definition 61, $\mathcal{R}[\![\exists x.\, \phi]\!]^\rho = \bigcup_{v \in \text{VAL}} \mathcal{R}[\![\phi]\!]^{\rho[x \mapsto v]}$.
By lemma 10, $\mathcal{R}[\![\exists x.\, \phi]\!]^\rho = \bigcup_{v \in \text{VAL}} \mathcal{R}[\![\phi\,[v/x]]\!]^\rho$.
Then, by lemma 12, $(\mathcal{R}[\![\exists x.\, \phi]\!]^\rho)^\dagger = \left(\bigcup_{v \in \text{VAL}} \mathcal{R}[\![\phi\,[v/x]]\!]^\rho\right)^\dagger$.
Therefore, by definition 61, $(\mathcal{R}[\![\exists x.\, \phi]\!]^\rho)^\dagger = \left(\mathcal{R}\left[\!\left[\bigsqcup_{v \in \text{VAL}} \phi\,[v/x]\right]\!\right]^\rho\right)^\dagger$.
Then, by lemma 16, $[\![\exists x.\, \phi]\!]^\rho = \left[\!\left[\bigsqcup_{v \in \text{VAL}} \phi\,[v/x]\right]\!\right]^\rho$.
Thus, by definition 52, $\exists x.\, \phi \equiv \bigsqcup_{v \in \text{VAL}} \phi\,[v/x]$. □

Lemma 69 (ESeqExt)**.** *If* $x \notin free(\phi)$*, then* $\exists x.\, \phi; \psi \equiv \phi; \exists x.\, \psi$.

Proof. Let ρ such that it closes ϕ and ψ.
By definition 61, $\mathcal{R}[\![\exists x.\, \phi; \psi]\!]^\rho = \bigcup_v \mathcal{R}[\![\phi; \psi]\!]^{\rho[x \mapsto v]} = \bigcup_v \left(\mathcal{R}[\![\phi]\!]^{\rho[x \mapsto v]}; \mathcal{R}[\![\psi]\!]^{\rho[x \mapsto v]}\right)$.
By the premiss, $\mathcal{R}[\![\phi]\!]^{\rho[x \mapsto v]} = \mathcal{R}[\![\phi]\!]^\rho$.
Therefore, $\bigcup_v \left(\mathcal{R}[\![\phi]\!]^{\rho[x \mapsto v]}; \mathcal{R}[\![\psi]\!]^{\rho[x \mapsto v]}\right) = \mathcal{R}[\![\phi]\!]^\rho; \bigcup_v \mathcal{R}[\![\psi]\!]^{\rho[x \mapsto v]}$.
Thus, by definition 61, $\mathcal{R}[\![\exists x.\, \phi; \psi]\!]^\rho = \mathcal{R}[\![\phi; \exists x.\, \psi]\!]^\rho$.
By lemma 12, $(\mathcal{R}[\![\exists x.\, \phi; \psi]\!]^\rho)^\dagger = (\mathcal{R}[\![\phi; \exists x.\, \psi]\!]^\rho)^\dagger$.
By lemma 16, $[\![\exists x.\, \phi; \psi]\!]^\rho = [\![\phi; \exists x.\, \psi]\!]^\rho$.
Thus, by definition 52, $\exists x.\, \phi; \psi \equiv \phi; \exists x.\, \psi$. □

Lemma 70 (ESeqDst)**.** $\exists x.\, \phi; \psi \sqsubseteq (\exists x.\, \phi); (\exists x.\, \psi)$.

Proof. Let ρ such that it closes ϕ and ψ modulo x.
By definitions, $\bigcup_{v \in \text{VAL}} \mathcal{R}[\![\phi]\!]^{\rho[x \mapsto v]}; \mathcal{R}[\![\psi]\!]^{\rho[x \mapsto v]} \subseteq \left(\bigcup_{v \in \text{VAL}} \mathcal{R}[\![\phi]\!]^{\rho[x \mapsto v]}\right); \left(\bigcup_{v \in \text{VAL}} \mathcal{R}[\![\psi]\!]^{\rho[x \mapsto v]}\right)$.
By definition 61, $\mathcal{R}[\![\exists x.\, \phi; \psi]\!]^\rho \subseteq \mathcal{R}[\![(\exists x.\, \phi); (\exists x.\, \psi)]\!]^\rho$.
Then, by lemma 12, $(\mathcal{R}[\![\exists x.\, \phi; \psi]\!]^\rho)^\dagger \subseteq (\mathcal{R}[\![(\exists x.\, \phi); (\exists x.\, \psi)]\!]^\rho)^\dagger$.
Then, by lemma 16, $[\![\exists x.\, \phi; \psi]\!]^\rho \subseteq [\![(\exists x.\, \phi); (\exists x.\, \psi)]\!]^\rho$.
Thus, by definition 52, $\exists x.\, \phi; \psi \sqsubseteq (\exists x.\, \phi); (\exists x.\, \psi)$. □

Lemma 71 (EAChoiceDst)**.** $\exists x.\, \phi \sqcup \psi \equiv (\exists x.\, \phi) \sqcup (\exists x.\, \psi)$

Proof. Direct, via EAChoiceEq and AChoiceAssoc. □

Lemma 72 (EParDst)**.** $\exists x.\, \phi \parallel \psi \sqsubseteq (\exists x.\, \phi) \parallel (\exists x.\, \psi)$

Proof. Direct, via EAChoiceEq and AChoiceExchange. □

Lemma 73 (CMono)**.** *Let C be a specification context. If $\phi \sqsubseteq \psi$, then $C[\phi] \sqsubseteq C[\psi]$.*

Proof. By straightforward induction on $C[-]$.

□

Lemma 74 (FApplyElim)**.** $\phi\,[e/x] \equiv (\lambda x.\,\phi)\,e$

Proof. Let ρ such that it closes ϕ, modulo x.
By definition 51, $[\![(\lambda x.\,\phi)\,e]\!]^\rho = [\![\phi]\!]^{\rho[x \mapsto [\![e]\!]^\rho]}$.
Then, by lemma 10, $[\![(\lambda x.\,\phi)\,e]\!]^\rho = [\![\phi\,[e/x]]\!]^\rho$.
Thus, by definition 52, $\phi\,[e/x] \equiv (\lambda x.\,\phi)\,e$. □

Lemma 75 (FApplyElimRec)**.** $\phi\,[(\mu A.\,\lambda x.\,\phi)\,/A]\,[e/x] \equiv (\mu A.\,\lambda x.\,\phi)\,e$

Proof. Let ρ such that it closes ϕ, modulo x and A, and e.
By definition 61, $\mathcal{R}[\![(\mu A.\,\lambda x.\,\phi)\,e]\!]^\rho = (\mathcal{R}[\![\mu A.\,\lambda x.\,\phi]\!]^\rho)\,[\![e]\!]^\rho$.
Then, by lemma 27, $\mathcal{R}[\![(\mu A.\,\lambda x.\,\phi)\,e]\!]^\rho = \left(\mathcal{R}[\![\lambda x.\,\phi]\!]^{\rho[A \mapsto \mathcal{R}[\![\mu A.\lambda x.\phi]\!]^\rho]}\right)[\![e]\!]^\rho$.
By definition 61, $\mathcal{R}[\![(\mu A.\,\lambda x.\,\phi)\,e]\!]^\rho = \mathcal{R}[\![\phi]\!]^{\rho[A \mapsto \mathcal{R}[\![\mu A.\lambda x.\phi]\!]^\rho][x \mapsto [\![e]\!]^\rho]}$.
Then, by lemma 9, $\mathcal{R}[\![(\mu A.\,\lambda x.\,\phi)\,e]\!]^\rho = \mathcal{R}[\![\phi\,[(\mu A.\,\lambda x.\,\phi)\,/A]]\!]^{\rho[x \mapsto [\![e]\!]^\rho]}$.
Then, by lemma 10, $\mathcal{R}[\![(\mu A.\,\lambda x.\,\phi)\,e]\!]^\rho = \mathcal{R}[\![\phi\,[(\mu A.\,\lambda x.\,\phi)\,/A]\,[e/x]]\!]^\rho$.
Then, by lemma 12, $(\mathcal{R}[\![(\mu A.\,\lambda x.\,\phi)\,e]\!]^\rho)^\dagger = (\mathcal{R}[\![\phi\,[(\mu A.\,\lambda x.\,\phi)\,/A]\,[e/x]]\!]^\rho)^\dagger$.
Then, by lemma 16, $[\![(\mu A.\,\lambda x.\,\phi)\,e]\!]^\rho = [\![\phi\,[(\mu A.\,\lambda x.\,\phi)\,/A]\,[e/x]]\!]^\rho$.
Thus, by definition 52, $\phi\,[(\mu A.\,\lambda x.\,\phi)\,/A]\,[e/x] \equiv (\mu A.\,\lambda x.\,\phi)\,e$. □

Lemma 76 (FElim)**.** $F_l \equiv \lambda x.\,F_l x$

Proof. Let ρ such that it closes F_l.
Case analysis on F_l.
Case $\lambda x.\,\phi$, immediate by definition 51.
Case $\mu A.\,\lambda x.\,\phi$:
By definition 61, $\mathcal{R}[\![\lambda x.\,(\mu A.\,\lambda x.\,\phi)\,x]\!]^\rho = \lambda v.\,(\mathcal{R}[\![\mu A.\,\lambda x.\,\phi]\!]^\rho)\,v$.
By lemma 27, $\mathcal{R}[\![\lambda x.\,(\mu A.\,\lambda x.\,\phi)\,x]\!]^\rho = \lambda v.\,\left(\mathcal{R}[\![\lambda x.\,\phi]\!]^{\rho[A \mapsto \mathcal{R}[\![\mu A.\lambda x.\phi]\!]^\rho]}\right)v$.
Then, by definition 61, $\mathcal{R}[\![\lambda x.\,(\mu A.\,\lambda x.\,\phi)\,x]\!]^\rho = \mathcal{R}[\![\lambda x.\,\phi]\!]^{\rho[A \mapsto \mathcal{R}[\![\mu A.\lambda x.\phi]\!]^\rho]}$
Therefore, by lemma 27, $\mathcal{R}[\![\lambda x.\,(\mu A.\,\lambda x.\,\phi)\,x]\!]^\rho = \mathcal{R}[\![\mu A.\,\lambda x.\,\phi]\!]^\rho$.
Then, by lemma 12, $(\mathcal{R}[\![\lambda x.\,(\mu A.\,\lambda x.\,\phi)\,x]\!]^\rho)^\dagger = (\mathcal{R}[\![\mu A.\,\lambda x.\,\phi]\!]^\rho)^\dagger$.
Then, by lemma 16, $[\![\lambda x.\,(\mu A.\,\lambda x.\,\phi)\,x]\!]^\rho = [\![\mu A.\,\lambda x.\,\phi]\!]^\rho$.
Thus, by definition 52, $F_l \equiv \lambda x.\,F_l x$. □

Lemma 77 (FRename)**.** *If $\phi\,[e_1/x] \sqsubseteq \phi\,[e_2/x]$, then $(\lambda x.\,\phi)\,e_1 \sqsubseteq (\lambda x.\,\phi)\,e_2$.*

Proof. By FApplyElim and CMono. □

Lemma 78 (FRenameRec)**.** *If $\phi\,[(\mu A.\,\lambda x.\,\phi)\,/A]\,[e_1/x] \sqsubseteq \phi\,[(\mu A.\,\lambda x.\,\phi)\,/A]\,[e_2/x]$, then $(\mu A.\,\lambda x.\,\phi)\,e_1 \sqsubseteq (\mu A.\,\lambda x.\,\phi)\,e_2$.*

Proof. By FApplyElimRec and CMono. □

Lemma 79 (FuncIntro). *If $x \notin free(\phi)$, then $(\lambda x.\, \phi)\, () \equiv \phi$.*

Proof. Immediate by definition 51. □

Lemma 80 (Inline). $\phi\,[F/f] \equiv \textbf{let}\ f = F\ \textbf{in}\ \phi$

Proof. Immediate by definition 51 and lemma 9. □

Lemma 81 (Ind). *If $\lambda x.\, \phi\,[\lambda x.\, \psi/A] \sqsubseteq \lambda x.\, \psi$, then $\mu A.\, \lambda x.\, \phi \sqsubseteq \lambda x.\, \psi$.*

Proof. By lemma 4, the function $[\![\lambda x.\, \phi]\!]^{\rho[A \mapsto -]} : (\textsc{Val} \to \mathcal{P}(\textsc{Trace})) \to (\textsc{Val} \to \mathcal{P}(\textsc{Trace}))$ is monotonic. Furthermore, the function space $\textsc{Val} \to \mathcal{P}(\textsc{Trace})$ is a complete lattice.
By the premiss and definition 52, for all closing ρ, $[\![\lambda x.\, \phi\,[\lambda x.\, \psi/A]]\!]^{\rho} \subseteq [\![\lambda x.\, \psi]\!]^{\rho}$.
By lemma 9, $[\![\lambda x.\, \phi]\!]^{\rho[A \mapsto [\![\lambda x.\, \psi]\!]^{\rho}]} \subseteq [\![\lambda x.\, \psi]\!]^{\rho}$.
Then, by the fixpoint induction theorem, $\mu\, [\![\lambda x.\, \phi]\!]^{\rho[A \mapsto -]} \subseteq [\![\lambda x.\, \psi]\!]^{\rho}$.
Thus, by definition 51, $[\![\mu A.\, \lambda x.\, \phi]\!]^{\rho} \sqsubseteq [\![\lambda x.\, \psi]\!]^{\rho}$.
Therefore, by definition 52, $\mu A.\, \lambda x.\, \phi \sqsubseteq \lambda x.\, \psi$. □

Lemma 82 (UnrollR). *If $A \notin free(\phi;) \cup free(\psi)$, then*
$(\mu A.\, \lambda x.\, \psi \sqcup \phi; Ae')\, e \equiv \psi\,[e/x] \sqcup \phi\,[e/x]\,; (\mu A.\, \lambda x.\, \psi \sqcup \phi; Ae'')\, e'$.

Proof. By FApplyElimRec. □

Lemma 83 (UnrollL). *If $A \notin free(\phi;) \cup free(\psi)$, then*
$(\mu A.\, \lambda x.\, \psi \sqcup Ae'; \phi)\, e \equiv \psi\,[e/x] \sqcup \phi\,[e/x]\,; (\mu A.\, \lambda x.\, \psi \sqcup Ae''; \phi)\, e'$.

Proof. By FApplyElimRec. □

Lemma 84 (RecSeq). *If $A \notin free(\phi) \cup free(\psi_1) \cup free(\psi_2)$, then*

$$\left(\mu A.\, \lambda x.\, \psi_1 \sqcup \phi; Ae'\right)\, e; \psi_2\,[e/x] \equiv \left(\mu A.\, \lambda x.\, \psi_1; \psi_2 \sqcup \phi; Ae'\right)\, e$$

Proof. First, we have the following:

$$\begin{aligned}
&\lambda x.\, \psi_1; \psi_2 \sqcup \phi; \left(\lambda x.\, \left(\mu A.\, \lambda x.\, \psi_1 \sqcup \phi; Ae'\right)\, x; \psi_2\,[e/x]\right)\, e' \\
\equiv\ &\text{by FApplyElim} \\
&\lambda x.\, \psi_1; \psi_2 \sqcup \phi; \left(\mu A.\, \lambda x.\, \psi_1 \sqcup \phi; Ae'\right)\, e'; \psi_2\,[e/x] \\
\equiv\ &\text{by AChoiceDstR} \\
&\lambda x.\, \left(\psi_1 \sqcup \phi; \left(\mu A.\, \lambda x.\, \psi_1 \sqcup \phi; Ae'\right)\, e'\right)\, ; \psi_2\,[e/x] \\
\equiv\ &\text{by UnrollR and CMono} \\
&\lambda x.\, \left(\mu A.\, \lambda x.\, \psi_1 \sqcup \phi; Ae'\right)\, x; \psi_2\,[e/x]
\end{aligned}$$

Therefore, by Ind, $\mu A.\, \lambda x.\, \psi_1; \psi_2 \sqcup \phi; Ae' \equiv \lambda x.\, (\mu A.\, \lambda x.\, \psi_1 \sqcup \phi; Ae')\, x; \psi_2\,[e/x]$.
Then, by CMono and FApplyElim, $(\mu A.\, \lambda x.\, \psi_1; \psi_2 \sqcup \phi; Ae')\, e \equiv (\mu A.\, \lambda x.\, \psi_1 \sqcup \phi; Ae')\, e; \psi_2\,[e/x]$. □

B.3. Primitive Atomic Refinement Laws Proofs

Lemma 85 (UElim). $a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_k \sqsubseteq a(\forall y, \vec{x}.\, P,\, Q)^{\mathcal{A}}_k$

Proof. Let ρ such that it closes $a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_k$. Let $\vec{v} \in \overrightarrow{\text{Val}}$, of the same length as $\vec{x}$ and let $v \in \text{Val}$.
By pointwise extension, $(\!|P|\!)^{\rho[\vec{x}\mapsto\vec{v}]}_{\mathcal{A}} \subseteq (\!|P|\!)^{\rho[\vec{x}\mapsto\vec{v}][y\mapsto v]}_{\mathcal{A}}$, and $(\!|Q|\!)^{\rho[\vec{x}\mapsto\vec{v}]}_{\mathcal{A}} \subseteq (\!|Q|\!)^{\rho[\vec{x}\mapsto\vec{v}][y\mapsto v]}_{\mathcal{A}}$.
Therefore, by definition 61, $\mathcal{R}\left[\!\left[a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_k\right]\!\right]^{\rho} \subseteq \mathcal{R}\left[\!\left[a(\forall y, \vec{x}.\, P,\, Q)^{\mathcal{A}}_k\right]\!\right]^{\rho}$.
Then, by lemma 12, $\left(\mathcal{R}\left[\!\left[a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_k\right]\!\right]^{\rho}\right)^{\dagger} \subseteq \mathcal{R}\left[\!\left[a(\forall y, \vec{x}.\, P,\, Q)^{\mathcal{A}}_k\right]\!\right]^{\rho}$.
Then, by lemma 16, $\left[\!\left[a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_k\right]\!\right]^{\rho} \subseteq \left[\!\left[a(\forall y, \vec{x}.\, P,\, Q)^{\mathcal{A}}_k\right]\!\right]^{\rho}$.
Thus, by definition 52, $a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_k \sqsubseteq a(\forall y, \vec{x}.\, P,\, Q)^{\mathcal{A}}_k$. □

Lemma 86 (EPAtom). *If $x \notin free(P)$, then* $\exists x.\, a(\forall \vec{y}.\, P,\, Q)^{\mathcal{A}}_k \equiv a(\forall \vec{y}.\, P,\, \exists x.\, Q)^{\mathcal{A}}_k$

Proof. Let ρ such that it closes both specifications. Let $\vec{v} \in \overrightarrow{\text{Val}}$.
By premiss, $\forall v \in Values.\, (\!|P|\!)^{\rho[x\mapsto v]}_{\mathcal{A}} = (\!|P|\!)^{\rho}_{\mathcal{A}}$.
Then, by definition 42, $\bigcup_{v\in\text{Val}} \mathsf{a}\left((\!|P|\!)^{\rho[x\mapsto v][\vec{y}\mapsto\vec{v}]}_{\mathcal{A}}, (\!|Q|\!)^{\rho[x\mapsto v][\vec{y}\mapsto\vec{v}]}_{\mathcal{A}}\right)^{\mathcal{A}}_k = \mathsf{a}\left((\!|P|\!)^{\rho[\vec{y}\mapsto\vec{v}]}_{\mathcal{A}}, \bigcup_{v\in\text{Val}} (\!|Q|\!)^{\rho[x\mapsto v][\vec{y}\mapsto\vec{v}]}_{\mathcal{A}}\right)^{\mathcal{A}}_k$.
By definitions, $\mathsf{a}\left((\!|P|\!)^{\rho[\vec{y}\mapsto\vec{v}]}_{\mathcal{A}}, \bigcup_{v\in\text{Val}} (\!|Q|\!)^{\rho[x\mapsto v][\vec{y}\mapsto\vec{v}]}_{\mathcal{A}}\right)^{\mathcal{A}}_k = \mathsf{a}\left((\!|P|\!)^{\rho[\vec{y}\mapsto\vec{v}]}_{\mathcal{A}}, (\!|\exists x.\, Q|\!)^{\rho[\vec{y}\mapsto\vec{v}]}_{\mathcal{A}}\right)^{\mathcal{A}}_k$.
Thus, by definition 61, $\mathcal{R}\left[\!\left[\exists x.\, a(\forall \vec{y}.\, P,\, Q)^{\mathcal{A}}_k\right]\!\right]^{\rho} = \mathcal{R}\left[\!\left[a(\forall \vec{y}.\, P,\, \exists x.\, Q)^{\mathcal{A}}_k\right]\!\right]^{\rho}$.
Then, by lemma 12, lemma 16 and definition 52, $\exists x.\, a(\forall \vec{y}.\, P,\, Q)^{\mathcal{A}}_k \equiv a(\forall \vec{y}.\, P,\, \exists x.\, Q)^{\mathcal{A}}_k$.
□

Lemma 87 (EPElim). $a(\forall \vec{y}, x.\, P,\, Q)^{\mathcal{A}}_k \sqsubseteq a(\forall \vec{y}.\, \exists x.\, P,\, \exists x.\, Q)^{\mathcal{A}}_k$

Proof. Let ρ such that it closes both specifications. Let $\vec{v} \in \overrightarrow{\text{Val}}$ and $v \in \text{Val}$.
By definitions 42 and 38, $\mathsf{a}\left((\!|P|\!)^{\rho[\vec{y}\mapsto\vec{v}]}_{\mathcal{A}}, (\!|Q|\!)^{\rho[\vec{y}\mapsto\vec{v}]}_{\mathcal{A}}\right)^{\mathcal{A}}_k \subseteq \mathsf{a}\left((\!|\exists x.\, P|\!)^{\rho[\vec{y}\mapsto\vec{v}]}_{\mathcal{A}}, (\!|\exists x.\, Q|\!)^{\rho[\vec{y}\mapsto\vec{v}]}_{\mathcal{A}}\right)^{\mathcal{A}}_k$.
Therefore, by definition 61, $\mathcal{R}\left[\!\left[a(\forall \vec{y}, x.\, P,\, Q)^{\mathcal{A}}_k\right]\!\right]^{\rho} \subseteq \mathcal{R}\left[\!\left[a(\forall \vec{y}.\, \exists x.\, P,\, \exists x.\, Q)^{\mathcal{A}}_k\right]\!\right]^{\rho}$.
Then, by lemma 12, $\left(\mathcal{R}\left[\!\left[a(\forall \vec{y}, x.\, P,\, Q)^{\mathcal{A}}_k\right]\!\right]^{\rho}\right)^{\dagger} \subseteq \left(\mathcal{R}\left[\!\left[a(\forall \vec{y}.\, \exists x.\, P,\, \exists x.\, Q)^{\mathcal{A}}_k\right]\!\right]^{\rho}\right)^{\dagger}$.
Then, by lemma 16, $\left[\!\left[a(\forall \vec{y}, x.\, P,\, Q)^{\mathcal{A}}_k\right]\!\right]^{\rho} \subseteq \left[\!\left[a(\forall \vec{y}.\, \exists x.\, P,\, \exists x.\, Q)^{\mathcal{A}}_k\right]\!\right]^{\rho}$.
Thus, by definition 52, $a(\forall \vec{y}, x.\, P,\, Q)^{\mathcal{A}}_k \sqsubseteq a(\forall \vec{y}.\, \exists x.\, P,\, \exists x.\, Q)^{\mathcal{A}}_k$. □

Lemma 88 (PDisjunction). $a(\forall \vec{x}.\, P_1,\, Q_1)^{\mathcal{A}}_k \sqcup a(\forall \vec{x}.\, P_2,\, Q_2)^{\mathcal{A}}_k \sqsubseteq a(\forall \vec{x}.\, P_1 \vee P_2,\, Q_1 \vee Q_2)^{\mathcal{A}}_k$

Proof. Let ρ such that it closes both specifications.
By definition 42, $\mathsf{a}\left((\!|P_1|\!)^{\rho}_{\mathcal{A}}, (\!|Q_1|\!)^{\rho}_{\mathcal{A}}\right)^{\mathcal{A}}_k \cup \mathsf{a}\left((\!|P_2|\!)^{\rho}_{\mathcal{A}}, (\!|Q_2|\!)^{\rho}_{\mathcal{A}}\right)^{\mathcal{A}}_k \subseteq \mathsf{a}\left((\!|P_1 \vee P_2|\!)^{\rho}_{\mathcal{A}}, (\!|Q_1 \vee Q_2|\!)^{\rho}_{\mathcal{A}}\right)^{\mathcal{A}}_k$.
Therefore, by definition 61, $\mathcal{R}\left[\!\left[a(\forall \vec{x}.\, P_1,\, Q_1)^{\mathcal{A}}_k\right]\!\right]^{\rho} \cup \mathcal{R}\left[\!\left[a(\forall \vec{x}.\, P_2,\, Q_2)^{\mathcal{A}}_k\right]\!\right]^{\rho} \subseteq \mathcal{R}\left[\!\left[a(\forall \vec{x}.\, P_1 \vee P_2,\, Q_1 \vee Q_2)^{\mathcal{A}}_k\right]\!\right]^{\rho}$.
Thus, $\mathcal{R}\left[\!\left[a(\forall \vec{x}.\, P_1,\, Q_1)^{\mathcal{A}}_k \sqcup a(\forall \vec{x}.\, P_2,\, Q_2)^{\mathcal{A}}_k\right]\!\right]^{\rho} \subseteq \mathcal{R}\left[\!\left[a(\forall \vec{x}.\, P_1 \vee P_2,\, Q_1 \vee Q_2)^{\mathcal{A}}_k\right]\!\right]^{\rho}$.
Then, by lemma 12, $\left(\mathcal{R}\left[\!\left[a(\forall \vec{x}.\, P_1,\, Q_1)^{\mathcal{A}}_k \sqcup a(\forall \vec{x}.\, P_2,\, Q_2)^{\mathcal{A}}_k\right]\!\right]^{\rho}\right)^{\dagger} \subseteq \left(\mathcal{R}\left[\!\left[a(\forall \vec{x}.\, P_1 \vee P_2,\, Q_1 \vee Q_2)^{\mathcal{A}}_k\right]\!\right]^{\rho}\right)^{\dagger}$.
Then, by lemma 16, $\left[\!\left[a(\forall \vec{x}.\, P_1,\, Q_1)^{\mathcal{A}}_k \sqcup a(\forall \vec{x}.\, P_2,\, Q_2)^{\mathcal{A}}_k\right]\!\right]^{\rho} \subseteq \left[\!\left[a(\forall \vec{x}.\, P_1 \vee P_2,\, Q_1 \vee Q_2)^{\mathcal{A}}_k\right]\!\right]^{\rho}$.
Thus, by definition 52, $a(\forall \vec{x}.\, P_1,\, Q_1)^{\mathcal{A}}_k \sqcup a(\forall \vec{x}.\, P_2,\, Q_2)^{\mathcal{A}}_k \sqsubseteq a(\forall \vec{x}.\, P_1 \vee P_2,\, Q_1 \vee Q_2)^{\mathcal{A}}_k$. □

Lemma 89 (PConjunction). $a(\forall \vec{x}.\, P_1,\, Q_1)^{\mathcal{A}}_k \sqcap a(\forall \vec{x}.\, P_2,\, Q_2)^{\mathcal{A}}_k \sqsubseteq a(\forall \vec{x}.\, P_1 \wedge P_2,\, Q_1 \wedge Q_2)^{\mathcal{A}}_k$

Proof. Let ρ such that it closes both specifications.
By definition 42, $\mathsf{a}\big((\!|P_1|\!)^\rho_{\mathcal{A}}, (\!|Q_1|\!)^\rho_{\mathcal{A}}\big)^{\mathcal{A}}_k \cap \mathsf{a}\big((\!|P_2|\!)^\rho_{\mathcal{A}}, (\!|Q_2|\!)^\rho_{\mathcal{A}}\big)^{\mathcal{A}}_k \subseteq \mathsf{a}\big((\!|P_1 \wedge P_2|\!)^\rho_{\mathcal{A}}, (\!|Q_1 \wedge Q_2|\!)^\rho_{\mathcal{A}}\big)^{\mathcal{A}}_k$.
Therefore, by definition 61, $\mathcal{R}\left[\!\left[a(\forall \vec{x}.\, P_1,\, Q_1)^{\mathcal{A}}_k\right]\!\right]^\rho \cap \mathcal{R}\left[\!\left[a(\forall \vec{x}.\, P_2,\, Q_2)^{\mathcal{A}}_k\right]\!\right]^\rho \subseteq \mathcal{R}\left[\!\left[a(\forall \vec{x}.\, P_1 \wedge P_2,\, Q_1 \wedge Q_2)^{\mathcal{A}}_k\right]\!\right]^\rho$.
Thus, $\mathcal{R}\left[\!\left[a(\forall \vec{x}.\, P_1,\, Q_1)^{\mathcal{A}}_k \sqcap a(\forall \vec{x}.\, P_2,\, Q_2)^{\mathcal{A}}_k\right]\!\right]^\rho \subseteq \mathcal{R}\left[\!\left[a(\forall \vec{x}.\, P_1 \wedge P_2,\, Q_1 \wedge Q_2)^{\mathcal{A}}_k\right]\!\right]^\rho$.
Then, by lemma 12, $\left(\mathcal{R}\left[\!\left[a(\forall \vec{x}.\, P_1,\, Q_1)^{\mathcal{A}}_k \sqcap a(\forall \vec{x}.\, P_2,\, Q_2)^{\mathcal{A}}_k\right]\!\right]^\rho\right)^\dagger \subseteq \left(\mathcal{R}\left[\!\left[a(\forall \vec{x}.\, P_1 \wedge P_2,\, Q_1 \wedge Q_2)^{\mathcal{A}}_k\right]\!\right]^\rho\right)^\dagger$.
Then, by lemma 16, $\left[\!\left[a(\forall \vec{x}.\, P_1,\, Q_1)^{\mathcal{A}}_k \sqcap a(\forall \vec{x}.\, P_2,\, Q_2)^{\mathcal{A}}_k\right]\!\right]^\rho \subseteq \left[\!\left[a(\forall \vec{x}.\, P_1 \wedge P_2,\, Q_1 \wedge Q_2)^{\mathcal{A}}_k\right]\!\right]^\rho$.
Thus, by definition 52, $a(\forall \vec{x}.\, P_1,\, Q_1)^{\mathcal{A}}_k \sqcap a(\forall \vec{x}.\, P_2,\, Q_2)^{\mathcal{A}}_k \sqsubseteq a(\forall \vec{x}.\, P_1 \wedge P_2,\, Q_1 \wedge Q_2)^{\mathcal{A}}_k$. □

Lemma 90 (Frame)**.** $a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_k \sqsubseteq a(\forall \vec{x}.\, P * R,\, Q * R)^{\mathcal{A}}_k$

Proof. Let ρ such that it closes both specifications.
Let $p, r \in \textsc{View}_{\mathcal{A}}$. By definition 35, $p \leq p * r$.
Therefore, by definition 42:

$$
\begin{aligned}
\mathsf{a}\big((\!|P|\!)^\rho_{\mathcal{A}}, (\!|Q|\!)^\rho_{\mathcal{A}}\big)^{\mathcal{A}}_k(h) &= \left\{ h' \in \textsc{Heap} \;\middle|\; \begin{array}{l} \forall r \in \textsc{View}_{\mathcal{A}}.\, \forall w \in (\!|P|\!)^\rho_{\mathcal{A}} * r.\, h \in \llbracket w \rrbracket_{k;\mathcal{A}} \\ \wedge\, \exists w'.\, w\ G_{k;\mathcal{A}}\ w' \wedge h' \in \llbracket w' \rrbracket_{k;\mathcal{A}} \wedge w' \in (\!|Q|\!)^\rho_{\mathcal{A}} * r \end{array} \right\} \\
&\subseteq \left\{ h' \in \textsc{Heap} \;\middle|\; \begin{array}{l} \forall r \in \textsc{View}_{\mathcal{A}}.\, \forall w \in (\!|P|\!)^\rho_{\mathcal{A}} * (\!|R|\!)^\rho_{\mathcal{A}} * r.\, h \in \llbracket w \rrbracket_{k;\mathcal{A}} \\ \wedge\, \exists w'.\, w\ G_{k;\mathcal{A}}\ w' \wedge h' \in \llbracket w' \rrbracket_{k;\mathcal{A}} \wedge w' \in (\!|Q|\!)^\rho_{\mathcal{A}} * (\!|R|\!)^\rho_{\mathcal{A}} * r \end{array} \right\} \\
&= \text{by definition 38} \\
&\quad \left\{ h' \in \textsc{Heap} \;\middle|\; \begin{array}{l} \forall r \in \textsc{View}_{\mathcal{A}}.\, \forall w \in (\!|P * R|\!)^\rho_{\mathcal{A}} * r.\, h \in \llbracket w \rrbracket_{k;\mathcal{A}} \\ \wedge\, \exists w'.\, w\ G_{k;\mathcal{A}}\ w' \wedge h' \in \llbracket w' \rrbracket_{k;\mathcal{A}} \wedge w' \in (\!|Q * R|\!)^\rho_{\mathcal{A}} * r \end{array} \right\} \\
&= \mathsf{a}\big((\!|P * R|\!)^\rho_{\mathcal{A}}, (\!|Q * R|\!)^\rho_{\mathcal{A}}\big)^{\mathcal{A}}_k(h)
\end{aligned}
$$

Therefore, by definition 61, $\mathcal{R}\left[\!\left[a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_k\right]\!\right]^\rho \subseteq \mathcal{R}\left[\!\left[a(\forall \vec{x}.\, P * R,\, Q * R)^{\mathcal{A}}_k\right]\!\right]^\rho$.
By lemma 12, $\left(\mathcal{R}\left[\!\left[a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_k\right]\!\right]^\rho\right)^\dagger \subseteq \left(\mathcal{R}\left[\!\left[a(\forall \vec{x}.\, P * R,\, Q * R)^{\mathcal{A}}_k\right]\!\right]^\rho\right)^\dagger$.
Then, by lemma 16, $\left[\!\left[a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_k\right]\!\right]^\rho \subseteq \left[\!\left[a(\forall \vec{x}.\, P * R,\, Q * R)^{\mathcal{A}}_k\right]\!\right]^\rho$.
Thus, by definition 52, $a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_k \sqsubseteq a(\forall \vec{x}.\, P * R,\, Q * R)^{\mathcal{A}}_k$. □

Lemma 91 (Stutter)**.** $a(\forall \vec{x}.\, P,\, P)^{\mathcal{A}}_k; a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_k \sqsubseteq a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_k$

Proof. Let ρ such that it closes $a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_k$.
By definition 50, and specifically the CLStutter rule,
$\left(\mathcal{R}\left[\!\left[a(\forall \vec{x}.\, P,\, P)^{\mathcal{A}}_k; a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_k\right]\!\right]^\rho\right)^\dagger \subseteq \left(\mathcal{R}\left[\!\left[a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_k\right]\!\right]^\rho\right)^\dagger$.
Then, by lemma 16, $\left[\!\left[a(\forall \vec{x}.\, P,\, P)^{\mathcal{A}}_k; a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_k\right]\!\right]^\rho \subseteq \left[\!\left[a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_k\right]\!\right]^\rho$.
Thus, by definition 52, $a(\forall \vec{x}.\, P,\, P)^{\mathcal{A}}_k; a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_k \sqsubseteq a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_k$. □

Lemma 92 (Mumble)**.** $a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_k \sqsubseteq a(\forall \vec{x}.\, P,\, P')^{\mathcal{A}}_k; a(\forall \vec{x}.\, P',\, Q)^{\mathcal{A}}_k$

Proof. Let ρ such that it closes both specifications.
By definition 50, and specifically the CLMumble rule,
$\left(\mathcal{R}\left[\!\left[a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_k\right]\!\right]^\rho\right)^\dagger \subseteq \left(\mathcal{R}\left[\!\left[a(\forall \vec{x}.\, P,\, P')^{\mathcal{A}}_k; a(\forall \vec{x}.\, P',\, Q)^{\mathcal{A}}_k\right]\!\right]^\rho\right)^\dagger$.
Then, by lemma 16, $\left[\!\left[a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_k\right]\!\right]^\rho \subseteq \left[\!\left[a(\forall \vec{x}.\, P,\, P')^{\mathcal{A}}_k; a(\forall \vec{x}.\, P',\, Q)^{\mathcal{A}}_k\right]\!\right]^\rho$.
Thus, by definition 52, $a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_k \sqsubseteq a(\forall \vec{x}.\, P,\, P')^{\mathcal{A}}_k; a(\forall \vec{x}.\, P',\, Q)^{\mathcal{A}}_k$. □

Lemma 93 (Interleave)**.**

$$\begin{aligned} & a(\forall \vec{x}.\, P_1,\, Q_1)^{\mathcal{A}}_k \parallel a(\forall \vec{x}.\, P_2,\, Q_2)^{\mathcal{A}}_k \\ \equiv & \left(a(\forall \vec{x}.\, P_1,\, Q_1)^{\mathcal{A}}_k ; a(\forall \vec{x}.\, P_2,\, Q_2)^{\mathcal{A}}_k\right) \sqcup \left(a(\forall \vec{x}.\, P_2,\, Q_2)^{\mathcal{A}}_k ; a(\forall \vec{x}.\, P_1,\, Q_1)^{\mathcal{A}}_k\right) \end{aligned}$$

Proof. Let ρ such that it closes both specifications. Let $m_1, m_2 \in (\textsc{Heap} \times \textsc{Heap}^{\lightning}) \cup \{\lightning, \lightning\}$. Then, by definition 49, $m_1 \parallel m_2 = \{m_1 m_2, m_2 m_1\}$.

Therefore, by definition 61,

$$\begin{aligned} & \mathcal{R}\left[\!\left[a(\forall \vec{x}.\, P_1,\, Q_1)^{\mathcal{A}}_k\right]\!\right]^{\rho} \parallel \mathcal{R}\left[\!\left[a(\forall \vec{x}.\, P_2,\, Q_2)^{\mathcal{A}}_k\right]\!\right]^{\rho} \\ = & \left(\mathcal{R}\left[\!\left[a(\forall \vec{x}.\, P_1,\, Q_1)^{\mathcal{A}}_k\right]\!\right]^{\rho} ; \mathcal{R}\left[\!\left[a(\forall \vec{x}.\, P_2,\, Q_2)^{\mathcal{A}}_k\right]\!\right]^{\rho}\right) \cup \left(\mathcal{R}\left[\!\left[a(\forall \vec{x}.\, P_2,\, Q_2)^{\mathcal{A}}_k\right]\!\right]^{\rho} ; \mathcal{R}\left[\!\left[a(\forall \vec{x}.\, P_1,\, Q_1)^{\mathcal{A}}_k\right]\!\right]^{\rho}\right) \end{aligned}$$

Then, the result is established by lemma 12, lemma 16 and definition 52. □

Lemma 94 (PParallel)**.**

$$a(\forall \vec{x}.\, P_1,\, Q_1)^{\mathcal{A}}_k \parallel a(\forall \vec{x}.\, P_2,\, Q_2)^{\mathcal{A}}_k \sqsubseteq a(\forall \vec{x}.\, P_1 * P_2,\, Q_1 * Q_2)^{\mathcal{A}}_k$$

Proof. By Interleave, Frame and AChoiceEq. □

Lemma 95 (Cons)**.** *If $P \Rightarrow P'$ and $Q' \Rightarrow Q$, then $a(\forall \vec{x}.\, P',\, Q')^{\mathcal{A}}_k \sqsubseteq a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_k$.*

Proof. Let ρ such that it closes both specifications.

From the first premiss, when P is satisfied, then $(\!|P|\!)^{\rho}_{\mathcal{A}} \subseteq (\!|P'|\!)^{\rho}_{\mathcal{A}}$.

From the second premiss, when Q' is satisfied, then $(\!|Q'|\!)^{\rho}_{\mathcal{A}} \subseteq (\!|Q'|\!)^{\rho}_{\mathcal{A}}$.

Then, from definition 42, it follows that for all $h \in \textsc{Heap}$, $\mathsf{a}\left((\!|P'|\!)^{\rho}_{\mathcal{A}}, (\!|Q'|\!)^{\rho}_{\mathcal{A}}\right)^{\mathcal{A}}_k(h) \subseteq \mathsf{a}\left((\!|P|\!)^{\rho}_{\mathcal{A}}, (\!|Q|\!)^{\rho}_{\mathcal{A}}\right)^{\mathcal{A}}_k(h)$.

Therefore, $\mathcal{R}\left[\!\left[a(\forall \vec{x}.\, P',\, Q')^{\mathcal{A}}_k\right]\!\right]^{\rho} \subseteq \mathcal{R}\left[\!\left[a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_k\right]\!\right]^{\rho}$.

Then, by lemma 12, $\left(\mathcal{R}\left[\!\left[a(\forall \vec{x}.\, P',\, Q')^{\mathcal{A}}_k\right]\!\right]^{\rho}\right)^{\dagger} \subseteq \left(\mathcal{R}\left[\!\left[a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_k\right]\!\right]^{\rho}\right)^{\dagger}$.

Thus, by lemma 16, and definition 52, $a(\forall \vec{x}.\, P',\, Q')^{\mathcal{A}}_k \sqsubseteq a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_k$. □

Lemma 96 (PAChoice)**.** $a(\forall \vec{x}.\, P,\, Q \vee Q')^{\mathcal{A}}_k \sqsubseteq a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_k \sqcup a(\forall \vec{x}.\, P,\, Q')^{\mathcal{A}}_k$

Proof. Let ρ such that it closes both specifications.

By definition 38 and definition 42,
$\mathsf{a}\left((\!|P|\!)^{\rho}_{\mathcal{A}}, (\!|Q \vee Q'|\!)^{\rho}_{\mathcal{A}}\right)^{\mathcal{A}}_k = \mathsf{a}\left((\!|P|\!)^{\rho}_{\mathcal{A}}, (\!|Q|\!)^{\rho}_{\mathcal{A}} \cup (\!|Q'|\!)^{\rho}_{\mathcal{A}}\right)^{\mathcal{A}}_k = \mathsf{a}\left((\!|P|\!)^{\rho}_{\mathcal{A}}, (\!|Q|\!)^{\rho}_{\mathcal{A}}\right)^{\mathcal{A}}_k \cup \mathsf{a}\left((\!|P|\!)^{\rho}_{\mathcal{A}}, (\!|Q'|\!)^{\rho}_{\mathcal{A}}\right)^{\mathcal{A}}_k$.

By definition 61,

$$
\begin{aligned}
&\mathcal{R}\left[\!\left[a(\forall \vec{x}.\, P,\, Q \vee Q')^{\mathcal{A}}_k\right]\!\right]^{\rho} = \\
&\qquad \left\{(h,h') \in \textsc{Move} \;\middle|\; \vec{v} \in \overrightarrow{\textsc{Val}} \wedge h' \in \mathsf{a}\left((\!|P|\!)^{\rho[\vec{x}\mapsto\vec{v}]}_{\mathcal{A}}, (\!|Q \vee Q'|\!)^{\rho[\vec{x}\mapsto\vec{v}]}_{\mathcal{A}}\right)^{\mathcal{A}}_k(h)\right\} \\
&\qquad \cup \left\{(h,\lightning) \in \textsc{Heap}^{\lightning} \;\middle|\; \begin{array}{l} \vec{v} \in \overrightarrow{\textsc{Val}} \wedge \mathsf{a}\left((\!|P|\!)^{\rho[\vec{x}\mapsto\vec{v}]}_{\mathcal{A}}, (\!|Q \vee Q'|\!)^{\rho[\vec{x}\mapsto\vec{v}]}_{\mathcal{A}}\right)^{\mathcal{A}}_k(h) = \emptyset \\ \wedge\, (\!|Q \vee Q'|\!)^{\rho[\vec{x}\mapsto\vec{v}]}_{\neq}\emptyset \end{array}\right\} \\
&\qquad \cup \left\{(\lightning,\lightning)\right\} \\
&\quad = \text{by definition 38} \\
&\qquad \left\{(h,h') \in \textsc{Move} \;\middle|\; \vec{v} \in \overrightarrow{\textsc{Val}} \wedge h' \in \mathsf{a}\left((\!|P|\!)^{\rho[\vec{x}\mapsto\vec{v}]}_{\mathcal{A}}, (\!|Q|\!)^{\rho[\vec{x}\mapsto\vec{v}]}_{\mathcal{A}} \cup (\!|Q'|\!)^{\rho[\vec{x}\mapsto\vec{v}]}_{\mathcal{A}}\right)^{\mathcal{A}}_k(h)\right\} \\
&\qquad \cup \left\{(h,\lightning) \in \textsc{Heap}^{\lightning} \;\middle|\; \begin{array}{l} \vec{v} \in \overrightarrow{\textsc{Val}} \wedge \mathsf{a}\left((\!|P|\!)^{\rho[\vec{x}\mapsto\vec{v}]}_{\mathcal{A}}, (\!|Q|\!)^{\rho[\vec{x}\mapsto\vec{v}]}_{\mathcal{A}} \cup (\!|Q'|\!)^{\rho[\vec{x}\mapsto\vec{v}]}_{\mathcal{A}}\right)^{\mathcal{A}}_k(h) = \emptyset \\ \wedge\, (\!|Q|\!)^{\rho[\vec{x}\mapsto\vec{v}]}_{\mathcal{A}} \cup (\!|Q'|\!)^{\rho[\vec{x}\mapsto\vec{v}]}_{\mathcal{A}} \neq \emptyset \end{array}\right\} \\
&\qquad \cup \left\{(\lightning,\lightning)\right\} \\
&\quad = \quad \left\{(h,h') \in \textsc{Move} \;\middle|\; \vec{v} \in \overrightarrow{\textsc{Val}} \wedge h' \in \mathsf{a}\left((\!|P|\!)^{\rho[\vec{x}\mapsto\vec{v}]}_{\mathcal{A}}, (\!|Q|\!)^{\rho[\vec{x}\mapsto\vec{v}]}_{\mathcal{A}}\right)^{\mathcal{A}}_k(h)\right\} \\
&\qquad \cup \left\{(h,h') \in \textsc{Move} \;\middle|\; \vec{v} \in \overrightarrow{\textsc{Val}} \wedge h' \in \mathsf{a}\left((\!|P|\!)^{\rho[\vec{x}\mapsto\vec{v}]}_{\mathcal{A}}, (\!|Q'|\!)^{\rho[\vec{x}\mapsto\vec{v}]}_{\mathcal{A}}\right)^{\mathcal{A}}_k(h)\right\} \\
&\qquad \cup \left\{(h,\lightning) \in \textsc{Heap}^{\lightning} \;\middle|\; \begin{array}{l} \vec{v} \in \overrightarrow{\textsc{Val}} \wedge \mathsf{a}\left((\!|P|\!)^{\rho[\vec{x}\mapsto\vec{v}]}_{\mathcal{A}}, (\!|Q|\!)^{\rho[\vec{x}\mapsto\vec{v}]}_{\mathcal{A}} \cup (\!|Q'|\!)^{\rho[\vec{x}\mapsto\vec{v}]}_{\mathcal{A}}\right)^{\mathcal{A}}_k(h) = \emptyset \\ \wedge\, (\!|Q|\!)^{\rho[\vec{x}\mapsto\vec{v}]}_{\mathcal{A}} \cup (\!|Q'|\!)^{\rho[\vec{x}\mapsto\vec{v}]}_{\mathcal{A}} \neq \emptyset \end{array}\right\} \\
&\qquad \cup \left\{(\lightning,\lightning)\right\} \cup \left\{(\lightning,\lightning)\right\} \\
&\quad \subseteq \quad \left\{(h,h') \in \textsc{Move} \;\middle|\; \vec{v} \in \overrightarrow{\textsc{Val}} \wedge h' \in \mathsf{a}\left((\!|P|\!)^{\rho[\vec{x}\mapsto\vec{v}]}_{\mathcal{A}}, (\!|Q|\!)^{\rho[\vec{x}\mapsto\vec{v}]}_{\mathcal{A}}\right)^{\mathcal{A}}_k(h)\right\} \\
&\qquad \cup \left\{(h,h') \in \textsc{Move} \;\middle|\; \vec{v} \in \overrightarrow{\textsc{Val}} \wedge h' \in \mathsf{a}\left((\!|P|\!)^{\rho[\vec{x}\mapsto\vec{v}]}_{\mathcal{A}}, (\!|Q'|\!)^{\rho[\vec{x}\mapsto\vec{v}]}_{\mathcal{A}}\right)^{\mathcal{A}}_k(h)\right\} \\
&\qquad \cup \left\{(h,\lightning) \in \textsc{Heap}^{\lightning} \;\middle|\; \begin{array}{l} \vec{v} \in \overrightarrow{\textsc{Val}} \wedge \mathsf{a}\left((\!|P|\!)^{\rho[\vec{x}\mapsto\vec{v}]}_{\mathcal{A}}, (\!|Q|\!)^{\rho[\vec{x}\mapsto\vec{v}]}_{\mathcal{A}}\right)^{\mathcal{A}}_k(h) = \emptyset \\ \wedge\, (\!|Q|\!)^{\rho[\vec{x}\mapsto\vec{v}]}_{\mathcal{A}} \neq \emptyset \end{array}\right\} \\
&\qquad \cup \left\{(h,\lightning) \in \textsc{Heap}^{\lightning} \;\middle|\; \begin{array}{l} \vec{v} \in \overrightarrow{\textsc{Val}} \wedge \mathsf{a}\left((\!|P|\!)^{\rho[\vec{x}\mapsto\vec{v}]}_{\mathcal{A}}, (\!|Q'|\!)^{\rho[\vec{x}\mapsto\vec{v}]}_{\mathcal{A}}\right)^{\mathcal{A}}_k(h) = \emptyset \\ \wedge\, (\!|Q'|\!)^{\rho[\vec{x}\mapsto\vec{v}]}_{\mathcal{A}} \neq \emptyset \end{array}\right\} \\
&\qquad \cup \left\{(\lightning,\lightning)\right\} \cup \left\{(\lightning,\lightning)\right\} \\
&\quad = \text{by definition 61} \\
&\qquad \mathcal{R}\left[\!\left[a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_k\right]\!\right]^{\rho} \sqcup \mathcal{R}\left[\!\left[a(\forall \vec{x}.\, P,\, Q')^{\mathcal{A}}_k\right]\!\right]^{\rho}
\end{aligned}
$$

Then, by lemma 12,
$\left(\mathcal{R}\left[\!\left[a(\forall \vec{x}.\, P,\, Q \vee Q')^{\mathcal{A}}_k\right]\!\right]^{\rho}\right)^{\dagger} \subseteq \left(\mathcal{R}\left[\!\left[a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_k \sqcup a(\forall \vec{x}.\, P,\, Q')^{\mathcal{A}}_k\right]\!\right]^{\rho}\right)^{\dagger}$.
By lemma 16, $\left[\!\left[a(\forall \vec{x}.\, P,\, Q \vee Q')^{\mathcal{A}}_k\right]\!\right]^{\rho} \subseteq \left[\!\left[a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_k \sqcup a(\forall \vec{x}.\, P,\, Q')^{\mathcal{A}}_k\right]\!\right]^{\rho}$.
Thus, by definition 52, $a(\forall \vec{x}.\, P,\, Q \vee Q')^{\mathcal{A}}_k \sqsubseteq a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_k \sqcup a(\forall \vec{x}.\, P,\, Q')^{\mathcal{A}}_k$ □

Lemma 97 (RLevel)**.** *If $k_1 \leq k_2$, then $a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_{k_1} \sqsubseteq a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_{k_2}$*

Proof. Let ρ that closes $a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_{-}$.

Fix $h \in \textsc{Heap}$. By definition 42,

$$\begin{aligned}
\mathsf{a}\big((\!|P|\!)^{\rho}_{\mathcal{A}}, (\!|Q|\!)^{\rho}_{\mathcal{A}}\big)^{\mathcal{A}}_{k_1}(h) &= \left\{ h' \in \textsc{Heap} \;\middle|\; \begin{array}{l} \forall r \in \textsc{View}_{\mathcal{A}}.\, \forall w \in (\!|P|\!)^{\rho}_{\mathcal{A}} * r.\, h \in [\![w]\!]_{k_1;\mathcal{A}} \\ \wedge\, \exists w'.\, w\ G_{k_1;\mathcal{A}}\ w' \wedge h' \in [\![w']\!]_{k_1;\wedge} w' \in (\!|Q|\!)^{\rho}_{\mathcal{A}} * r \end{array} \right\} \\
&= \text{by } [\![w]\!]_{k_1;\mathcal{A}} = [\![w]\!]_{k_2;\mathcal{A}}, [\![w']\!]_{k_1;\mathcal{A}} = [\![w']\!]_{k_2;\mathcal{A}} \text{ as all regions are opened} \\
&\quad \left\{ h' \in \textsc{Heap} \;\middle|\; \begin{array}{l} \forall r \in \textsc{View}_{\mathcal{A}}.\, \forall w \in (\!|P|\!)^{\rho}_{\mathcal{A}} * r.\, h \in [\![w]\!]_{k_2;\mathcal{A}} \\ \wedge\, \exists w'.\, w\ G_{k_1;\mathcal{A}}\ w' \wedge h' \in [\![w']\!]_{k_2;\wedge} w' \in (\!|Q|\!)^{\rho}_{\mathcal{A}} * r \end{array} \right\} \\
&\subseteq \text{by } \ G_{k_1;\mathcal{A}} \subseteq G_{k_2;\mathcal{A}} \\
&\quad \left\{ h' \in \textsc{Heap} \;\middle|\; \begin{array}{l} \forall r \in \textsc{View}_{\mathcal{A}}.\, \forall w \in (\!|P|\!)^{\rho}_{\mathcal{A}} * r.\, h \in [\![w]\!]_{k_2;\mathcal{A}} \\ \wedge\, \exists w'.\, w\ G_{k_2;\mathcal{A}}\ w' \wedge h' \in [\![w']\!]_{k_2;\wedge} w' \in (\!|Q|\!)^{\rho}_{\mathcal{A}} * r \end{array} \right\} \\
&= \mathsf{a}\big((\!|P|\!)^{\rho}_{\mathcal{A}}, (\!|Q|\!)^{\rho}_{\mathcal{A}}\big)^{\mathcal{A}}_{k_2}(h)
\end{aligned}$$

From this and definition 61, $\mathcal{R}\left[\!\left[a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_{k_1}\right]\!\right]^{\rho} \subseteq \mathcal{R}\left[\!\left[a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_{k_2}\right]\!\right]^{\rho}$.

By lemma 12, $\left(\mathcal{R}\left[\!\left[a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_{k_1}\right]\!\right]^{\rho}\right)^{\dagger} \subseteq \left(\mathcal{R}\left[\!\left[a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_{k_2}\right]\!\right]^{\rho}\right)^{\dagger}$.

Then, by lemma 16, $\left[\!\left[a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_{k_1}\right]\!\right]^{\rho} \subseteq \left[\!\left[a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_{k_2}\right]\!\right]^{\rho}$.

Thus, by definition 52, $a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_{k_1} \sqsubseteq a(\forall \vec{x}.\, P,\, Q)^{\mathcal{A}}_{k_2}$. □

Lemma 98 (RIEq proof)**.**

$$a\Big(\forall x \in X.\, I_r(\mathbf{t}^k_\alpha(x)) * P,\, I_r(\mathbf{t}^k_\alpha(x)) * Q\Big)^{\mathcal{A}}_{k} \equiv a\Big(\forall x \in X.\, \mathbf{t}^k_\alpha(x) * P,\, \mathbf{t}^k_\alpha(x) * Q\Big)^{\mathcal{A}}_{k+1}$$

Proof. Fix ρ such that it closes both specifications. Fix $v \in X$.

Let $P' = I_r(\mathbf{t}^k_\alpha(x)) * P$ and $Q' = I_r(\mathbf{t}^k_\alpha(x)) * Q$.

$$\mathsf{a}\Big((\!|P'|\!)^{\rho[x\mapsto v]}_{\mathcal{A}}, (\!|Q'|\!)^{\rho[x\mapsto v]}_{\mathcal{A}}\Big)^{\mathcal{A}}_{k}(h) =$$
$$\left\{ h' \in \textsc{Heap} \;\middle|\; \begin{array}{l} \forall r \in \textsc{View}_{\mathcal{A}}.\, \forall w \in (\!|P'|\!)^{\rho[x\mapsto v]}_{\mathcal{A}} * r.\, h \in [\![w]\!]_{k;\mathcal{A}} \\ \wedge\, \exists w'.\, w\ G_{k;\mathcal{A}}\ w' \wedge h' \in [\![w']\!]_{k;\mathcal{A}} \wedge w' \in (\!|Q'|\!)^{\rho[x\mapsto v]}_{\mathcal{A}} * r \end{array} \right\}$$

Let $\overline{w} \in (\!|\mathbf{t}^k_\alpha(x) * P|\!)^{\rho}_{\mathcal{A}}$. Then, $[\![w]\!]_k = [\![\overline{w}]\!]_{k+1}$.

Let $\overline{w'} \in (\!|\mathbf{t}^k_\alpha(x) * Q|\!)^{\rho}_{\mathcal{A}}$. From the guarantee, $\overline{w}\ G_{k+1;\mathcal{A}}\ \overline{w'}$, and we have that $[\![w']\!]_k = [\![\overline{w'}]\!]_{k+1}$.

Therefore,

$$\begin{aligned}
&\left\{ h' \in \textsc{Heap} \;\middle|\; \begin{array}{l} \forall r \in \textsc{View}_{\mathcal{A}}.\, \forall w \in (\!|P'|\!)^{\rho[x\mapsto v]}_{\mathcal{A}} * r.\, h \in [\![w]\!]_{k;\mathcal{A}} \\ \wedge\, \exists w'.\, w\ G_{k;\mathcal{A}}\ w' \wedge h' \in [\![w']\!]_{k;\wedge} w' \in (\!|Q'|\!)^{\rho[x\mapsto v]}_{\mathcal{A}} * r \end{array} \right\} \\
=&\left\{ h' \in \textsc{Heap} \;\middle|\; \begin{array}{l} \forall r \in \textsc{View}_{\mathcal{A}}.\, \forall \overline{w} \in (\!|\overline{P'}|\!)^{\rho[x\mapsto v]}_{\mathcal{A}} * r.\, h \in [\![\overline{w}]\!]_{k+1;\mathcal{A}} \\ \wedge\, \exists \overline{w'}.\, \overline{w}\ G_{k+1;\mathcal{A}}\ \overline{w'} \wedge h' \in [\![\overline{w'}]\!]_{k+1;\wedge} \overline{w'} \in (\!|\overline{Q'}|\!)^{\rho[x\mapsto v]}_{\mathcal{A}} * r \end{array} \right\} \\
=&\mathsf{a}\Big((\!|\overline{P'}|\!)^{\rho[x\mapsto v]}_{\mathcal{A}}, (\!|\overline{Q'}|\!)^{\rho[x\mapsto v]}_{\mathcal{A}}\Big)^{\mathcal{A}}_{k+1}(h)
\end{aligned}$$

where $\overline{P'} = \mathbf{t}^k_\alpha(x) * P$ and $\overline{Q'} = \mathbf{t}^k_\alpha(x) * Q$.

Thus, from definition 51,

$$\left[\!\left[a\Big(\forall x \in X.\, I_r(\mathbf{t}^k_\alpha(x)) * P,\ I_r(\mathbf{t}^k_\alpha(x)) * Q\Big)^{\mathcal{A}}_k \right]\!\right]^\rho = \left[\!\left[a\Big(\forall x \in X.\, \mathbf{t}^k_\alpha(x) * P,\ \mathbf{t}^k_\alpha(x) * Q\Big)^{\mathcal{A}}_{k+1} \right]\!\right]^\rho$$

Then, from definition 52,

$$a\Big(\forall x \in X.\, I_r(\mathbf{t}^k_\alpha(x)) * P,\ I_r(\mathbf{t}^k_\alpha(x)) * Q\Big)^{\mathcal{A}}_k \equiv a\Big(\forall x \in X.\, \mathbf{t}^k_\alpha(x) * P,\ \mathbf{t}^k_\alpha(x) * Q\Big)^{\mathcal{A}}_{k+1}$$

□

Lemma 99 (RUEq)**.** *If $\alpha \notin \mathcal{A}$ and $\forall x \in X.\, (x, f(x)) \in \mathcal{T}_{\mathbf{t}}(G)^*$, then*

$$a\Big(\forall x \in X.\, I_r(\mathbf{t}^k_\alpha(x)) * P * [\mathsf{G}]_\alpha,\ I_r(\mathbf{t}^k_\alpha(f(x))) * Q\Big)^{\mathcal{A}}_k \equiv a\Big(\forall x \in X.\, \mathbf{t}^k_\alpha(x) * P * [\mathsf{G}]_\alpha,\ \mathbf{t}^k_\alpha(f(x)) * Q\Big)^{\mathcal{A}}_{k+1}$$

Proof. Assume the premiss.

Fix ρ such that it closes both specifications. Fix $v \in X$.

Let $P' = I_r(\mathbf{t}^k_\alpha(x)) * P * [\mathsf{G}]_\alpha$ and $Q' = I_r(\mathbf{t}^k_\alpha(f(x))) * Q$.

$$\mathsf{a}\Big((\!|P'|\!)^{\rho[x\mapsto v]}_{\mathcal{A}}, (\!|Q'|\!)^{\rho[x\mapsto v]}_{\mathcal{A}}\Big)^{\mathcal{A}}_k(h) = \left\{ h' \in \textsc{Heap} \;\middle|\; \begin{array}{l} \forall r \in \textsc{View}_{\mathcal{A}}.\, \forall w \in (\!|P'|\!)^{\rho[x\mapsto v]}_{\mathcal{A}} * r.\, h \in \llbracket w \rrbracket_{k;\mathcal{A}} \\ \quad \wedge\, \exists w'.\, w\ G_{k;\mathcal{A}}\ w' \wedge h' \in \llbracket w' \rrbracket_{k;\wedge} w' \in (\!|Q'|\!)^{\rho[x\mapsto v]}_{\mathcal{A}} * r \end{array} \right\}$$

Let $\overline{w} \in (\!|\mathbf{t}^k_\alpha(x) * P * [\mathsf{G}]_\alpha|\!)^\rho_{\mathcal{A}}$. Then, $\llbracket w \rrbracket_k = \llbracket \overline{w} \rrbracket_{k+1}$.

Let $\overline{w'} \in (\!|\mathbf{t}^k_\alpha(x) * Q|\!)^\rho_{\mathcal{A}}$. From the guarantee, $\overline{w}\ G_{k+1;\mathcal{A}}\ \overline{w'}$, and we have that $\llbracket w' \rrbracket_k = \llbracket \overline{w'} \rrbracket_{k+1}$.

Therefore,

$$\begin{aligned} &\left\{ h' \in \textsc{Heap} \;\middle|\; \begin{array}{l} \forall r \in \textsc{View}_{\mathcal{A}}.\, \forall w \in (\!|P'|\!)^{\rho[x\mapsto v]}_{\mathcal{A}} * r.\, h \in \llbracket w \rrbracket_{k;\mathcal{A}} \\ \quad \wedge\, \exists w'.\, w\ G_{k;\mathcal{A}}\ w' \wedge h' \in \llbracket w' \rrbracket_{k;\mathcal{A}} \wedge w' \in (\!|Q'|\!)^{\rho[x\mapsto v]}_{\mathcal{A}} * r \end{array} \right\} \\ =&\left\{ h' \in \textsc{Heap} \;\middle|\; \begin{array}{l} \forall r \in \textsc{View}_{\mathcal{A}}.\, \forall \overline{w} \in (\!|\overline{P'}|\!)^{\rho[x\mapsto v]}_{\mathcal{A}} * r.\, h \in \llbracket \overline{w} \rrbracket_{k+1;\mathcal{A}} \\ \quad \wedge\, \exists \overline{w'}.\, \overline{w}\ G_{k+1;\mathcal{A}}\ \overline{w'} \wedge h' \in \llbracket \overline{w'} \rrbracket_{k+1;\mathcal{A}} \wedge \overline{w'} \in (\!|\overline{Q'}|\!)^{\rho[x\mapsto v]}_{\mathcal{A}} * r \end{array} \right\} \\ =&\mathsf{a}\Big((\!|\overline{P'}|\!)^{\rho[x\mapsto v]}_{\mathcal{A}}, (\!|\overline{Q'}|\!)^{\rho[x\mapsto v]}_{\mathcal{A}}\Big)^{\mathcal{A}}_{k+1}(h) \end{aligned}$$

where $\overline{P'} = \mathbf{t}^k_\alpha(x) * P * [\mathsf{G}]_\alpha$ and $\overline{Q'} = \mathbf{t}^k_\alpha(f(x)) * Q$.

Thus, from definition 51,

$$\begin{aligned} &\left[\!\left[a\big(\forall x \in X.\, I_r(\mathbf{t}^k_\alpha(x)) * P * [\mathsf{G}]_\alpha,\ I_r(\mathbf{t}^k_\alpha(f(x))) * Q\big)^{\mathcal{A}}_k \right]\!\right]^\rho = \\ &\quad \left[\!\left[a\big(\forall x \in X.\, \mathbf{t}^k_\alpha(x) * P * [\mathsf{G}]_\alpha,\ \mathbf{t}^k_\alpha(f(x)) * Q\big)^{\mathcal{A}}_{k+1} \right]\!\right]^\rho \end{aligned}$$

Then, from definition 52,

$$a\Big(\forall x \in X.\, I_r(\mathbf{t}^k_\alpha(x)) * P * [\mathsf{G}]_\alpha,\ I_r(\mathbf{t}^k_\alpha(f(x))) * Q\Big)^{\mathcal{A}}_k \equiv a\Big(\forall x \in X.\, \mathbf{t}^k_\alpha(x) * P * [\mathsf{G}]_\alpha,\ \mathbf{t}^k_\alpha(f(x)) * Q\Big)^{\mathcal{A}}_{k+1}$$

□

B.4. Proofs of Abstract Atomic Refinement Laws

Lemma 100.

$$\exists \vec{x} \in \overrightarrow{X}, \vec{y} \in \overrightarrow{Y}.\, a\big(P_p * P(\vec{x}),\ P'_p(\vec{x},\vec{y}) * Q(\vec{x},\vec{y})\big)^{\mathcal{A}}_k; \big\langle P'_p(\vec{x},\vec{y}) \mid Q(\vec{x},\vec{y}),\ Q_p(\vec{x},\vec{y}) \mid Q(\vec{x},\vec{y})\big\rangle^{\mathcal{A}}_k$$
$$\sqsubseteq \forall \vec{x} \in \overrightarrow{X}. \left\langle P_p \mid P(\vec{x}),\ \exists \vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x},\vec{y}) \mid Q(\vec{x},\vec{y}) \right\rangle^{\mathcal{A}}_k$$

Proof. By AWeaken2, lemma 7, AChoiceElim and UnrollR. □

Lemma 101.

$$a\Big(\forall \vec{x} \in \overrightarrow{X}.\, P_p * P(\vec{x}),\ P'_p * P(\vec{x})\Big)^{\mathcal{A}}_k; \forall \vec{x} \in \overrightarrow{X}. \left\langle P'_p \mid P(\vec{x}),\ \exists \vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x},\vec{y}) \mid Q(\vec{x},\vec{y}) \right\rangle^{\mathcal{A}}_k$$
$$\sqsubseteq \forall \vec{x} \in \overrightarrow{X}. \left\langle P_p \mid P(\vec{x}),\ \exists \vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x},\vec{y}) \mid Q(\vec{x},\vec{y}) \right\rangle^{\mathcal{A}}_k$$

Proof.

$$a\Big(\forall \vec{x} \in \overrightarrow{X}.\, P_p * P(\vec{x}),\ P'_p * P(\vec{x})\Big)^{\mathcal{A}}_k; \forall \vec{x} \in \overrightarrow{X}. \left\langle P'_p \mid P(\vec{x}),\ \exists \vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x},\vec{y}) \mid Q(\vec{x},\vec{y}) \right\rangle^{\mathcal{A}}_k$$

$\sqsubseteq$ by lemma 6, Cons, EPAtom and FApplyElimRec

$$\exists p'_p.\, a\Big(\forall \vec{x} \in \overrightarrow{X}.\, P_p * P(\vec{x}),\ P'_p \wedge p'_p * P(\vec{x})\Big)^{\mathcal{A}}_k;$$
$$\mu A.\, \lambda p_p.\ \exists p'_p.\, a\Big(\forall \vec{x} \in \overrightarrow{X}.\, p_p * P(\vec{x}),\ p'_p * P(\vec{x})\Big)^{\mathcal{A}}_k; A p'_p$$
$$\sqcup\ \exists \vec{x} \in \overrightarrow{X}, \vec{y} \in \overrightarrow{Y}.\, \exists p''_p.\ a\big(p_p * P(\vec{x}),\ p''_p * Q(\vec{x},\vec{y})\big)^{\mathcal{A}}_k;$$
$$\mu B.\, \lambda p''_p.\ \exists p'''_p.\, a\big(p''_p,\ p'''_p\big)^{\mathcal{A}}_k; B p'''_p$$
$$\sqcup\ a\big(p''_p,\ Q_p(\vec{x},\vec{y})\big)^{\mathcal{A}}_k$$
$$\cdot p''_p$$
$$\cdot\, p'_p$$

$\sqsubseteq$ by Stutter, Cons, EPAtom and CMono

$$\exists p_p.\, a\Big(\forall \vec{x} \in \overrightarrow{X}.\, P_p * P(\vec{x}),\ P_p \wedge p_p * P(\vec{x})\Big)^{\mathcal{A}}_k;$$
$$\exists p'_p.\, a\Big(\forall \vec{x} \in \overrightarrow{X}.\, p_p * P(\vec{x}),\ P'_p \wedge p'_p * P(\vec{x})\Big)^{\mathcal{A}}_k;$$
$$\mu A.\, \lambda p_p.\ \exists p'_p.\, a\Big(\forall \vec{x} \in \overrightarrow{X}.\, p_p * P(\vec{x}),\ p'_p * P(\vec{x})\Big)^{\mathcal{A}}_k; A p'_p$$
$$\sqcup\ \exists \vec{x} \in \overrightarrow{X}, \vec{y} \in \overrightarrow{Y}.\, \exists p''_p.\ a\big(p_p * P(\vec{x}),\ p''_p * Q(\vec{x},\vec{y})\big)^{\mathcal{A}}_k;$$
$$\mu B.\, \lambda p''_p.\ \exists p'''_p.\, a\big(p''_p,\ p'''_p\big)^{\mathcal{A}}_k; B p'''_p$$
$$\sqcup\ a\big(p''_p,\ Q_p(\vec{x},\vec{y})\big)^{\mathcal{A}}_k$$
$$\cdot p''_p$$
$$\cdot\, p'_p$$

$\sqsubseteq$ by AChoiceElim, AChoiceComm, UnrollR and CMono

$$\exists p_p.\, a\Big(\forall \vec{x} \in \overrightarrow{X}.\, P_p * P(\vec{x}),\ P_p \wedge p_p * P(\vec{x})\Big)^{\mathcal{A}}_k;$$

$$\mu A.\, \lambda p_p.\ \exists p'_p.\, a\Big(\forall \vec{x} \in \overrightarrow{X}.\, p_p * P(\vec{x}),\ p'_p * P(\vec{x})\Big)^{\mathcal{A}}_{k}; Ap'_p$$
$$\sqcup\ \exists \vec{x} \in \overrightarrow{X}, \vec{y} \in \overrightarrow{Y}.\ \exists p''_p.\ a\big(p_p * P(\vec{x}),\ p''_p * Q(\vec{x},\vec{y})\big)^{\mathcal{A}}_{k};$$
$$\mu B.\, \lambda p''_p.\ \exists p'''_p.\, a\big(p''_p,\ p'''_p\big)^{\mathcal{A}}_{k}; Bp'''_p$$
$$\sqcup\ a\big(p''_p,\ Q_p(\vec{x},\vec{y})\big)^{\mathcal{A}}_{k}$$
$$\cdot p''_p$$
$$\cdot\, p_p$$

$\equiv$ by definition 56

$$\mathbb{V}\vec{x} \in \overrightarrow{X}.\, \Big\langle P_p \mid P(\vec{x}),\ \mathbb{I}\vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x},\vec{y}) \mid Q(\vec{x},\vec{y}) \Big\rangle^{\mathcal{A}}_{k}$$

□

Lemma 102 (MakeAtomic Proof)**.** *If $\alpha \notin \mathcal{A}$ and $\{(x,y) \mid x \in X, y \in Y(x)\} \subseteq \mathcal{T}_{\mathbf{t}}(\mathsf{G})^*$, then*

$$\exists \vec{x} \in \overrightarrow{X}.\, I(\vec{x}) \vdash \big\{P_p * \exists x \in X.\, \mathbf{t}^k_\alpha(\vec{e},x) * \alpha \Mapsto \blacklozenge,\ \exists x \in X, y \in Y(x).\, Q_p(x,\vec{x},y) * \alpha \Mapsto (x,y)\big\}^{\alpha : x \in X \rightsquigarrow Y(x), \mathcal{A}}_{k'}$$
$$\sqsubseteq \mathbb{V}x \in X, \vec{x} \in \overrightarrow{X}.\, \big\langle P_p \mid \mathbf{t}^k_\alpha(\vec{e},x) * [\mathsf{G}(\vec{e}')]_\alpha * I(\vec{x}),\ \mathbb{I}y \in Y(x).\, Q_p(x,\vec{x},y) \mid \mathbf{t}^k_\alpha(\vec{e},y) * [\mathsf{G}(\vec{e}')]_\alpha * I(\vec{x}) \big\rangle^{\mathcal{A}}_{k'}$$

Proof. Assume the premisses hold.
Let $\mathcal{A} = \alpha : x \in X \rightsquigarrow Y(x), \mathcal{A}'$.

$$\exists \vec{x} \in \overrightarrow{X}.\, I(\vec{x}) \vdash \big\{P_p * \exists x \in X.\, \mathbf{t}^k_\alpha(\vec{e},x) * \alpha \Mapsto \blacklozenge,\ \exists x \in X, y \in Y(x).\, Q_p(x,\vec{x},y) * \alpha \Mapsto (x,y)\big\}^{\mathcal{A}}_{k'}$$
$\equiv$ by definition 57
$$\mathbb{V}\vec{x} \in \overrightarrow{X}.\, \big\langle P_p * \exists x \in X.\, \mathbf{t}^k_\alpha(\vec{e},x) * \alpha \Mapsto \blacklozenge \mid I(\vec{x}),\ \mathbb{I}z \in \mathbf{1}.\, \exists x \in X, y \in Y(x).\, Q_p(x,\vec{x},y) * \alpha \Mapsto (x,y) \mid I(\vec{x}) \big\rangle^{\mathcal{A}}_{k'}$$
$\sqsubseteq$ by definition 56, Frame, and similarly to lemma 7 (in the $\sqsubseteq$ direction)
$$\exists p.\ a\Big(\forall \vec{x} \in \overrightarrow{X}.\, P_p * \exists x \in X.\, \mathbf{t}^{k'}_\alpha(\vec{e},x) * \alpha \Mapsto \blacklozenge * I(\vec{x}),\ (P_p * \exists x \in X.\, \mathbf{t}^{k'}_\alpha(x) * \alpha \Mapsto \blacklozenge) \wedge p * I(\vec{x})\Big)^{\mathcal{A}}_{k'}$$
$$\mu A.\lambda p.\ \exists p'.\, a\Big(\forall \vec{x} \in \overrightarrow{X}.\, p * I(\vec{x}),\ p' * I(\vec{x})\Big)^{\mathcal{A}}_{k'}; Ap'$$
$$\sqcup\ \exists \vec{x} \in \overrightarrow{X}.\, a(p * I(\vec{x}),\ \exists x \in X, y \in Y(x).\, Q_p(x,\vec{x},y) * \alpha \Mapsto (x,y) * I(\vec{x}))^{\mathcal{A}}_{k'}$$
$$\cdot p$$
$\sqsubseteq$ by lemma 6
$$\exists p'.\, a\Big(\forall \vec{x} \in \overrightarrow{X}.\, P_p * \exists x \in X.\, \mathbf{t}^k_\alpha(\vec{e},x) * \alpha \Mapsto \blacklozenge * I(\vec{x}),\ p' * I(\vec{x})\Big)^{\mathcal{A}}_{k'}; Ap'$$
$$\sqcup\ \exists \vec{x} \in \overrightarrow{X}.\, a\big(P_p * \exists x \in X.\, \mathbf{t}^k_\alpha(\vec{e},x) * \alpha \Mapsto \blacklozenge * I(\vec{x}),\ \exists x \in X, y \in Y(x).\, Q_p(x,\vec{x},y) * \alpha \Mapsto (x,y) * I(\vec{x})\big)^{\mathcal{A}}_{k'}$$

where

$$A = \mu A.\lambda p.\ \exists p'.\, a\Big(\forall \vec{x} \in \overrightarrow{X}.\, p * I(\vec{x}),\ p' * I(\vec{x})\Big)^{\mathcal{A}}_{k'}; Ap'$$
$$\sqcup\ \exists \vec{x} \in \overrightarrow{X}.\, a(p * I(\vec{x}),\ \exists x \in X, y \in Y(x).\, Q_p(x,\vec{x},y) * \alpha \Mapsto (x,y) * I(\vec{x}))^{\mathcal{A}}_{k'}$$

Let ρ such that it closes both specifications.
Consider $\mathsf{a}\big(\langle\!|P_p * \exists x \in X.\, \mathbf{t}^k_\alpha(\vec{e},x) * \alpha \Mapsto \blacklozenge * I(\vec{x})|\!\rangle^\rho_{\mathcal{A}}, \langle\!|p' * I(\vec{x})|\!\rangle^\rho_{\mathcal{A}}\big)^{\mathcal{A}}_{k'}$

$$\left\{ h' \in \textsc{Heap} \;\middle|\; \begin{array}{l} \mathsf{a}\big(\langle\!|P_p * \exists x \in X.\, \mathbf{t}^k_\alpha(\vec{e},x) * \alpha \Mapsto \blacklozenge * I(\vec{x})|\!\rangle^\rho_{\mathcal{A}}, \langle\!|p' * I(\vec{x})|\!\rangle^\rho_{\mathcal{A}}\big)^{\mathcal{A}}_{k'}(h) = \\ \forall \overline{r} \in \textsc{View}_{\mathcal{A}}.\, \forall \overline{w} \in \langle\!|P_p * \exists x \in X.\, \mathbf{t}^k_\alpha(\vec{e},x) * \alpha \Mapsto \blacklozenge * I(\vec{x})|\!\rangle^\rho_{\mathcal{A}} * \overline{r}.\, h \in [\![\overline{w}]\!]_{k';\mathcal{A}} \\ \wedge\ \exists \overline{w'}.\, \overline{w}\ G_{k';\mathcal{A}}\ \overline{w'} \wedge h' \in [\![\overline{w'}]\!]_{k';\mathcal{A}} \wedge \overline{w'} \in \langle\!|p' * I(\vec{x})|\!\rangle^\rho_{\mathcal{A}} * \overline{r} \end{array} \right\}$$

Let $v_x \in X$ and $v_y \in Y(v_x)$. Fix $r \in \textsc{View}_{\mathcal{A}'}$. Fix $w \in \left(\!\left|P_p * \mathbf{t}^k_\alpha(\vec{e}, x) * \left[\mathsf{G}(\vec{e}')\right]_\alpha * I(\vec{x})\right|\!\right)^\rho_{\mathcal{A}'} * r\big)$. Let $v = \left(\!\left|p' * I(\vec{x})\right|\!\right)^\rho_{\mathcal{A}}$.
Let $v' = \left\{w \in \textsc{AWorld}_{\mathrm{dom}(\mathcal{A}')} \mid (w \circ \alpha \mapsto \blacklozenge) \in v\right\}$.
Let $v'' = \left\{w \in \textsc{AWorld}_{\mathrm{dom}(\mathcal{A}')} \mid (w \circ \alpha \mapsto (x, y)) \in v\right\}$.
Let $\overline{r} = r * \left(\!\left|\left[\mathsf{G}(\vec{e}')\right]_\alpha * \alpha \Rrightarrow -\right|\!\right)^\rho_{\mathcal{A}}$.
$\overline{r}$ is stable with respect to $\mathcal{A}$ because the additional interference is $\alpha : x \in X \rightsquigarrow Y(x)$ and the subset of $\overline{r}$ that is compatible with $\left[\mathsf{G}(\vec{e}')\right]_\alpha$ must be closed under this.
Let $\overline{w} = w \circ \alpha \mapsto \blacklozenge$.
Then, by construction: $\left\|w\right\|_{k';\mathcal{A}'} = \left\|\overline{w}\right\|_{k';\mathcal{A}}$.
We have $\overline{w} \in \left(\!\left|P_p * \exists x \in X.\, \mathbf{t}^k_\alpha(\vec{e}, x) * \alpha \Rrightarrow \blacklozenge * I(\vec{x})\right|\!\right)^\rho_{\mathcal{A}} * \overline{r}$.
There exists $\overline{w'}$ such that:

i). $\overline{w}\ G_{k';\mathcal{A}}\ \overline{w'}$

ii). $h' \in \left\|\overline{w'}\right\|_{k';\mathcal{A}}$

iii). $\overline{w'} \in v' * \overline{r}$

From i) and $d_{\overline{w}} = \blacklozenge$ and $\beta_{\overline{w}} = x$ we know that either $d_{\overline{w'}} = \blacklozenge$ or $d_{\overline{w'}} = (x, y)$ for some $y \in Y(x)$. This means that $d_{\overline{w'}} \neq \diamondsuit$.
Let w' such that $w = w' \circ \alpha \mapsto -$.
Then by iii),
$w' \in \left(\!\left|p'_p * \mathbf{t}^k_\alpha(\vec{e}, x) * \left[\mathsf{G}(\vec{e}')\right]_\alpha * I(\vec{x})\right|\!\right)^{\rho[x\mapsto v_x][y\mapsto v_y][p'_p\mapsto v'][p''_p\mapsto v'']}_{\mathcal{A}'} \cup \left(\!\left|p''_p * \mathbf{t}^k_\alpha(\vec{e}, y) * \left[\mathsf{G}(\vec{e}')\right]_\alpha * I(\vec{x})\right|\!\right)^{\rho[x\mapsto v_x][y\mapsto v_y][p'_p\mapsto v'][p''_p\mapsto v'']}_{\mathcal{A}'}$.
By i) and definitions, we get $w\ G_{k';\mathcal{A}}\ w'$.
By construction, $\left\|w'\right\|_{k';\mathcal{A}'} = \left\|w\right\|_{k';\mathcal{A}}$. Thus, $h' \in \left\|w'\right\|_{k';\mathcal{A}'}$ by ii).
Therefore, from the above:

$$\mathsf{a}\left(\left(\!\left|P_p * \exists x \in X.\, \mathbf{t}^k_\alpha(\vec{e}, x) * \alpha \Rrightarrow \blacklozenge * I(\vec{x})\right|\!\right)^\rho_{\mathcal{A}}, \left(\!\left|p' * I(\vec{x})\right|\!\right)^\rho_{\mathcal{A}}\right)^{\mathcal{A}}_{k'} =$$
$$\mathsf{a}\left(\begin{array}{l}\left(\!\left|P_p * \mathbf{t}^k_\alpha(\vec{e}, x) * \left[\mathsf{G}(\vec{e}')\right]_\alpha * I(\vec{x})\right|\!\right)^{\rho[x\mapsto v_x][y\mapsto v_y][p'_p\mapsto v'][p''_p\mapsto v'']}_{\mathcal{A}'}, \\ \left(\!\left|\left(p'_p * \mathbf{t}^k_\alpha(\vec{e}, x) * \left[\mathsf{G}(\vec{e}')\right]_\alpha * I(\vec{x})\right) \vee \left(p''_p * \mathbf{t}^k_\alpha(\vec{e}, y) * \left[\mathsf{G}(\vec{e}')\right]_\alpha * I(\vec{x})\right)\right|\!\right)^{\rho[x\mapsto v_x][y\mapsto v_y][p'_p\mapsto v'][p''_p\mapsto v'']}_{\mathcal{A}'}\end{array}\right)^{\mathcal{A}'}_{k'}$$

Then by definition 61,

$$\mathcal{R}\left[\!\left[\exists p'.\, a\Big(\forall \vec{x} \in \overrightarrow{X}.\, P_p * \exists x \in X.\, \mathbf{t}^k_\alpha(x) * \alpha \Rrightarrow \blacklozenge * I(\vec{x}),\ p' * I(\vec{x})\Big)^{\mathcal{A}}_{k'}\right]\!\right]^\rho =$$
$$\mathcal{R}\left[\!\left[\begin{array}{l}\exists x \in X, y \in Y(x), p'_p, p''_p. \\ a\left(\begin{array}{l}\forall \vec{x} \in \overrightarrow{X}. P_p * \mathbf{t}^k_\alpha(\vec{e}, x) * \left[\mathsf{G}(\vec{e}')\right]_\alpha * I(\vec{x}), \\ \qquad \left(p'_p * \mathbf{t}^k_\alpha(\vec{e}, x) * \left[\mathsf{G}(\vec{e}')\right]_\alpha\right) \vee \left(P''_p * \mathbf{t}^k_\alpha(\vec{e}, y) * \left[\mathsf{G}(\vec{e}')\right]_\alpha\right) * I(\vec{x})\end{array}\right)^{\mathcal{A}'}_{k'}\end{array}\right]\!\right]^\rho \tag{L1}$$

Then, by lemma 12, lemma 16 and definition 52,

$$\exists p'.\, a\Big(\forall \vec{x} \in \overrightarrow{X}.\, P_p * \exists x \in X.\, \mathbf{t}^k_\alpha(\vec{e}, x) * \alpha \Rrightarrow \blacklozenge * I(\vec{x}),\ p' * I(\vec{x})\Big)^{\mathcal{A}}_{k'} \equiv$$
$$\exists x \in X, y \in Y(x), p'_p, p''_p.\, a\left(\forall \vec{x} \in \overrightarrow{X}.\, P_p * \mathbf{t}^k_\alpha(\vec{e}, x) * \left[\mathsf{G}(\vec{e}')\right]_\alpha * I(\vec{x}),\ \begin{array}{l}\left(p'_p * \mathbf{t}^k_\alpha(\vec{e}, x) * \left[\mathsf{G}(\vec{e}')\right]_\alpha\right) \\ \vee \left(p''_p * \mathbf{t}^k_\alpha(\vec{e}, y) * \left[\mathsf{G}(\vec{e}')\right]_\alpha\right) * I(\vec{x})\end{array}\right)^{\mathcal{A}'}_{k'}$$

Then by PAChoice,

$$a\Big(\forall \vec{x} \in \overrightarrow{X}.\, P_p * \mathbf{t}^k_\alpha(\vec{e}, x) * [\mathsf{G}(\vec{e}')]_\alpha * I(\vec{x}),\ \big(p'_p * \mathbf{t}^k_\alpha(\vec{e}, x) * [\mathsf{G}(\vec{e}')]_\alpha\big) \vee \big(p''_p * \mathbf{t}^k_\alpha(\vec{e}, y) * [\mathsf{G}(\vec{e}')]_\alpha\big) * I(\vec{x})\Big)^{\mathcal{A}'}_{k'}$$
$$\sqsubseteq$$
$$a\Big(\forall \vec{x} \in \overrightarrow{X}.\, P_p * \mathbf{t}^k_\alpha(\vec{e}, x) * [\mathsf{G}(\vec{e}')]_\alpha * I(\vec{x}),\ p'_p * \mathbf{t}^k_\alpha(\vec{e}, x) * [\mathsf{G}(\vec{e}')]_\alpha * I(\vec{x})\Big)^{\mathcal{A}'}_{k'}$$
$$\sqcup\, a\Big(\forall \vec{x} \in \overrightarrow{X}.\, P_p * \mathbf{t}^k_\alpha(\vec{e}, x) * [\mathsf{G}(\vec{e}')]_\alpha * I(\vec{x}),\ p''_p * \mathbf{t}^k_\alpha(\vec{e}, y) * [\mathsf{G}(\vec{e}')]_\alpha * I(\vec{x})\Big)^{\mathcal{A}'}_{k'}$$

Therefore, by the above, CMono and AChoiceDstR:

$$\begin{aligned}
&\exists p'.\, a\Big(\forall \vec{x} \in \overrightarrow{X}.\, P_p * \exists x \in X.\, \mathbf{t}^k_\alpha(\vec{e}, x) * \alpha \Rrightarrow \blacklozenge * I(\vec{x}),\ p' * I(\vec{x})\Big)^{\mathcal{A}}_{k}; Ap' \\
\sqcup\ & a\Big(\forall \vec{x} \in \overrightarrow{X}.\, P_p * \exists x \in X.\, \mathbf{t}^k_\alpha(\vec{e}, x) * \alpha \Rrightarrow \blacklozenge * I(\vec{x}),\ \exists x \in X, y \in Q(x).\, Q_p(x, \vec{x}, y) * \alpha \Rrightarrow (x, y) * I(\vec{x})\Big)^{\mathcal{A}}_{k'} \\
\sqsubseteq\ & \exists x \in X, y \in Y(x), p'_p, p''_p. \\
& a\Big(\forall \vec{x} \in \overrightarrow{X}.\, P_p * \mathbf{t}^k_\alpha(\vec{e}, x) * [\mathsf{G}(\vec{e}')]_\alpha * I(\vec{x}),\ p'_p * \mathbf{t}^k_\alpha(\vec{e}, x) * [\mathsf{G}(\vec{e}')]_\alpha * I(\vec{x})\Big)^{\mathcal{A}'}_{k'}; Ap' \\
\sqcup\ & a\Big(\forall \vec{x} \in \overrightarrow{X}.\, P_p * \mathbf{t}^k_\alpha(\vec{e}, x) * [\mathsf{G}(\vec{e}')]_\alpha * I(\vec{x}),\ p''_p * \mathbf{t}^k_\alpha(\vec{e}, y) * [\mathsf{G}(\vec{e}')]_\alpha * I(\vec{x})\Big)^{\mathcal{A}'}_{k'}; Ap' \\
\sqcup\ & a\Big(\forall \vec{x} \in \overrightarrow{X}.\, P_p * \exists x \in X.\, \mathbf{t}^k_\alpha(\vec{e}, x) * \alpha \Rrightarrow \blacklozenge * I(\vec{x}),\ \exists x \in X, y \in Q(x).\, Q_p(x, \vec{x}, y) * \alpha \Rrightarrow (x, y) * I(\vec{x})\Big)^{\mathcal{A}}_{k'}
\end{aligned}$$

Now, we have the following:

$$\begin{aligned}
& \exists x \in X, y \in Y(x), p'_p, p''_p. \\
& a\Big(\forall \vec{x} \in \overrightarrow{X}.\, P_p * \mathbf{t}^k_\alpha(\vec{e}, x) * [\mathsf{G}(\vec{e}')]_\alpha * I(\vec{x}),\ p'_p * \mathbf{t}^k_\alpha(\vec{e}, x) * [\mathsf{G}(\vec{e}')]_\alpha * I(\vec{x})\Big)^{\mathcal{A}'}_{k'}; \\
& \quad \forall x \in X, \vec{x} \in \overrightarrow{X}.\, \big\langle p'_p \mid \mathbf{t}^k_\alpha(\vec{e}, x) * [\mathsf{G}(\vec{e}')]_\alpha * I(\vec{x}),\ \exists y \in Y(x).\, Q_p(x, \vec{x}, y) \mid \mathbf{t}^k_\alpha(\vec{e}, y) * [\mathsf{G}(\vec{e}')]_\alpha * I(\vec{x}) \big\rangle^{\mathcal{A}'}_{k'} \\
\sqcup\ & a\Big(\forall \vec{x} \in \overrightarrow{X}.\, P_p * \mathbf{t}^k_\alpha(\vec{e}, x) * [\mathsf{G}(\vec{e}')]_\alpha * I(\vec{x}),\ p''_p * \mathbf{t}^k_\alpha(\vec{e}, y) * [\mathsf{G}(\vec{e}')]_\alpha * I(\vec{x})\Big)^{\mathcal{A}'}_{k'}; \\
& \quad \forall x \in X, \vec{x} \in \overrightarrow{X}.\, \big\langle p''_p \mid \mathbf{t}^k_\alpha(\vec{e}, y) * [\mathsf{G}(\vec{e}')]_\alpha * I(\vec{x}),\ \exists y \in Y(x).\, Q_p(x, \vec{x}, y) \mid \mathbf{t}^k_\alpha(\vec{e}, y) * [\mathsf{G}(\vec{e}')]_\alpha * I(\vec{x}) \big\rangle^{\mathcal{A}'}_{k'} \\
\sqcup\ & a\Big(\forall \vec{x} \in \overrightarrow{X}.\, P_p * \exists x \in X.\, \mathbf{t}^k_\alpha(\vec{e}, x) * \alpha \Rrightarrow \blacklozenge * I(\vec{x}),\ \exists x \in X, y \in Q(x).\, Q_p(x, \vec{x}y) * \alpha \Rrightarrow (x, y) * I(\vec{x})\Big)^{\mathcal{A}}_{k'} \\
\sqsubseteq\ & \text{by AChoiceComm, UnrollR, AChoiceElim, lemma 100 and CMono} \\
& \forall x \in X, \vec{x} \in \overrightarrow{X}.\, \big\langle P_p \mid \mathbf{t}^k_\alpha(\vec{e}, x) * [\mathsf{G}(\vec{e}')]_\alpha * I(\vec{x}),\ \exists y \in Y(x).\, Q_p(x, \vec{x}, y) \mid \mathbf{t}^k_\alpha(\vec{e}, y) * [\mathsf{G}(\vec{e}')]_\alpha * I(\vec{x}) \big\rangle^{\mathcal{A}'}_{k'} \\
\sqcup\ & \forall x \in X, \vec{x} \in \overrightarrow{X}.\, \big\langle P_p \mid \mathbf{t}^k_\alpha(\vec{e}, x) * [\mathsf{G}(\vec{e}')]_\alpha * I(\vec{x}),\ \exists y \in Y(x).\, Q_p(x, \vec{x}, y) \mid \mathbf{t}^k_\alpha(\vec{e}, y) * [\mathsf{G}(\vec{e}')]_\alpha * I(\vec{x}) \big\rangle^{\mathcal{A}'}_{k'} \\
\sqcup\ & a\Big(\forall \vec{x} \in \overrightarrow{X}.\, P_p * \exists x \in X.\, \mathbf{t}^k_\alpha(\vec{e}, x) * \alpha \Rrightarrow \blacklozenge * I(\vec{x}),\ \exists x \in X, y \in Q(x).\, Q_p(x, y) * \alpha \Rrightarrow (x, y) * I(\vec{x})\Big)^{\mathcal{A}}_{k'} \\
\sqsubseteq\ & \text{by similar reasoning to (L1) followed by Skip, UnrollR as before} \\
& \forall x \in X, \vec{x} \in \overrightarrow{X}.\, \big\langle P_p \mid \mathbf{t}^k_\alpha(\vec{e}, x) * [\mathsf{G}(\vec{e}')]_\alpha * I(\vec{x}),\ \exists y \in Y(x).\, Q_p(x, \vec{x}, y) \mid \mathbf{t}^k_\alpha(\vec{e}, y) * [\mathsf{G}(\vec{e}')]_\alpha * I(\vec{x}) \big\rangle^{\mathcal{A}'}_{k'} \\
\sqcup\ & \forall x \in X, \vec{x} \in \overrightarrow{X}.\, \big\langle P_p \mid \mathbf{t}^k_\alpha(\vec{e}, x) * [\mathsf{G}(\vec{e}')]_\alpha * I(\vec{x}),\ \exists y \in Y(x).\, Q_p(x, \vec{x}, y) \mid \mathbf{t}^k_\alpha(\vec{e}, y) * [\mathsf{G}(\vec{e}')]_\alpha * I(\vec{x}) \big\rangle^{\mathcal{A}'}_{k'} \\
\sqcup\ & \forall x \in X, \vec{x} \in \overrightarrow{X}.\, \big\langle P_p \mid \mathbf{t}^k_\alpha(\vec{e}, x) * [\mathsf{G}(\vec{e}')]_\alpha * I(\vec{x}),\ \exists y \in Y(x).\, Q_p(x, \vec{x}, y) \mid \mathbf{t}^k_\alpha(\vec{e}, y) * [\mathsf{G}(\vec{e}')]_\alpha * I(\vec{x}) \big\rangle^{\mathcal{A}'}_{k'} \\
\sqsubseteq\ & \text{by AChoiceEq} \\
& \forall x \in X, \vec{x} \in \overrightarrow{X}.\, \big\langle P_p \mid \mathbf{t}^k_\alpha(\vec{e}, x) * [\mathsf{G}(\vec{e}')]_\alpha * I(\vec{x}),\ \exists y \in Y(x).\, Q_p(x, \vec{x}, y) \mid \mathbf{t}^k_\alpha(\vec{e}, y) * [\mathsf{G}(\vec{e}')]_\alpha * I(\vec{x}) \big\rangle^{\mathcal{A}'}_{k'}
\end{aligned}$$

Thus we have established the premiss of Ind, the conclusion of which proves this lemma. □

Lemma 103.

$$a\Big(\forall x \in X, \vec{x} \in \overrightarrow{X}.\, P_p * I_r(\mathbf{t}^k_\alpha(\vec{e}, x)) * P(x, \vec{x}),\ P'_p * I_r(\mathbf{t}^k_\alpha(\vec{e}, x)) * P(x, \vec{x})\Big)^{\mathcal{A}}_k$$
$$\sqsubseteq a\Big(\forall x \in X, \vec{x} \in \overrightarrow{X}.\, P_p * \mathbf{t}^k_\alpha(\vec{e}, x) * P(x, \vec{x}) * \alpha \Mapsto \blacklozenge,\ P'_p * \mathbf{t}^k_\alpha(\vec{e}, x) * P(x, \vec{x}) * \alpha \Mapsto \blacklozenge\Big)^{\alpha : x \in X \rightsquigarrow Y(x), \mathcal{A}}_{k+1}$$

Proof. Fix $x \in X, \vec{x} \in \overrightarrow{X}$.

$$\mathsf{a}\big(\langle\!|P_p * I_r(\mathbf{t}^k_\alpha(\vec{e}, x)) * P(x, \vec{x})|\!\rangle^\rho_{\mathcal{A}}, \langle\!|P'_p * I_r(\mathbf{t}^k_\alpha(\vec{e}, x)) * P(x, \vec{x})|\!\rangle^\rho_{\mathcal{A}}\big)^{\mathcal{A}}_k(\overline{h}) =$$
$$\left\{ \overline{h}' \in \textsc{Heap} \;\middle|\; \begin{array}{l} \forall \overline{r} \in \textsc{View}_{\mathcal{A}}. \forall \overline{w} \in \langle\!|P_p * I_r(\mathbf{t}^k_\alpha(\vec{e}, x)) * P(x, \vec{x})|\!\rangle^\rho_{\mathcal{A}} * \overline{r} \wedge \overline{h} \in \llbracket \overline{w} \rrbracket_{k;\mathcal{A}} \\ \quad \wedge \exists \overline{w}'. \overline{w}\ G_{k;\mathcal{A}}\ \overline{w}' \wedge \overline{h}' \in \llbracket \overline{w}' \rrbracket_{k;\mathcal{A}} \wedge \overline{w}' \in \langle\!|P'_p * I_r(\mathbf{t}^k_\alpha(x)) * P(x, \vec{x})|\!\rangle^\rho_{\mathcal{A}} * \overline{r} \end{array} \right\}$$

Let ρ such that it closes both specifications. Let $\mathcal{A}' = \alpha : x \in X \rightsquigarrow Y(x), \mathcal{A}$. Fix $r \in \textsc{View}_{\mathcal{A}'}$, $w \in (\langle\!|P_p * I_r(\mathbf{t}^k_\alpha(\vec{e}, x)) * P(x, \vec{x}) * \alpha \Mapsto \blacklozenge|\!\rangle^\rho_{\mathcal{A}} * r)$, $h \in \llbracket w \rrbracket_{k+1;\mathcal{A}'}$ and $h' \in \llbracket w' \rrbracket_{k+1;\mathcal{A}'}$.

Let $\overline{r} \in \textsc{View}_{\mathcal{A}}$ such that we open all regions at level k (except α) with their states as given by w and remove their atomicity tracking component:

$$\overline{r} = \mathsf{removedone}_\alpha \left(r * \circledast_{\substack{\alpha' \in \mathrm{RID} \\ \alpha' \neq \alpha \\ r_w(\alpha') = (k, -, -)}} \langle\!|I_r(r_w(\alpha'), \alpha', \beta_w(\alpha'))|\!\rangle^\rho_{\mathcal{A}} \right)$$

There is some $\overline{w} \in (\langle\!|P_p * I_r(\mathbf{t}^k_\alpha(\vec{e}, x)) * P(x, \vec{x})|\!\rangle^\rho_{\mathcal{A}} * \overline{r})$, with $r_w = r_{\overline{w}}$, $\beta_w = \beta_{\overline{w}}$ and $\llbracket \overline{w} \rrbracket_{k;\mathcal{A}} = \llbracket w \rrbracket_{k+1;\mathcal{A}'}$ and thus $h \in \llbracket \overline{w} \rrbracket_{k;\mathcal{A}}$.

Thus, there is some $\overline{w}'$ such that $\overline{w}\ G_{k;\mathcal{A}}\ \overline{w}'$ and $h' \in \llbracket \overline{w}' \rrbracket_{k;\mathcal{A}}$ and $\overline{w}' \in (\langle\!|P'_p * I_r(\mathbf{t}^k_\alpha(\vec{e}, x)) * P(x, \vec{x})|\!\rangle^\rho_{\mathcal{A}} * \overline{r})$, with $\overline{w}' = w'' \circ \overline{\overline{w}'}$, where

$$\overline{\overline{w}'} \in \left(\langle\!|I_r(\mathbf{t}^k_\alpha(\vec{e}, x))|\!\rangle^\rho_{\mathcal{A}} * \circledast_{\substack{\alpha' \in \mathrm{RID} \\ \alpha' \neq \alpha \\ r_w(\alpha') = (k, -, -)}} \langle\!|I_r(r_w(\alpha'), \alpha', \beta_w(\alpha'))|\!\rangle^\rho_{\mathcal{A}} \right)$$

and $w'' \in (\langle\!|P'_p * P(x, \vec{x})|\!\rangle^\rho_{\mathcal{A}} * r)$.

Let $w' = (r_{w''}, h_{w''}, b_{w''}, \gamma_{w''}, \beta_{w''}, d_{w''}[\alpha \mapsto \blacklozenge])$.

Hence, by the guarantee $w' \in (\langle\!|P_p * \mathbf{t}^k_\alpha(\vec{e}, x) * P(x, \vec{x})|\!\rangle^\rho_{\mathcal{A}'} * r)$ and by construction $\llbracket w' \rrbracket_{k+1;\mathcal{A}'} = \llbracket \overline{w}' \rrbracket_{k;\mathcal{A}}$ and $w\ G_{k+1;\mathcal{A}'}\ w'$.

Therefore,

$$\mathsf{a}\big(\langle\!|P_p * I_r(\mathbf{t}^k_\alpha(\vec{e}, x)) * P(x, \vec{x})|\!\rangle^\rho_{\mathcal{A}}, \langle\!|P'_p * I_r(\mathbf{t}^k_\alpha(\vec{e}, x)) * P(x, \vec{x})|\!\rangle^\rho_{\mathcal{A}}\big)^{\mathcal{A}}_k(\overline{h}) =$$
$$\left\{ \overline{h}' \in \textsc{Heap} \;\middle|\; \begin{array}{l} \forall \overline{r} \in \textsc{View}_{\mathcal{A}}. \forall \overline{w} \in \langle\!|P_p * I_r(\mathbf{t}^k_\alpha(\vec{e}, x)) * P(x, \vec{x})|\!\rangle^\rho_{\mathcal{A}} * \overline{r} \wedge \overline{h} \in \llbracket \overline{w} \rrbracket_{k;\mathcal{A}} \\ \quad \wedge \exists \overline{w}'. \overline{w}\ G_{k;\mathcal{A}}\ \overline{w}' \wedge \overline{h}' \in \llbracket \overline{w}' \rrbracket_{k;\mathcal{A}} \wedge \overline{w}' \in \langle\!|P'_p(\rho) * I_r(\mathbf{t}^k_\alpha(x)) * P(x, \vec{x})(\rho)|\!\rangle^\rho_{\mathcal{A}} * \overline{r} \end{array} \right\}$$
$$\subseteq \left\{ h' \in \textsc{Heap} \;\middle|\; \begin{array}{l} \forall r \in \textsc{View}_{\mathcal{A}'}. \forall w \in (\langle\!|P_p * \mathbf{t}^k_\alpha(\vec{e}, x) * P(x, \vec{x}) * \alpha \Mapsto \blacklozenge|\!\rangle^\rho_{\mathcal{A}'} * r) \wedge h \in \llbracket w \rrbracket_{k+1;\mathcal{A}'} \wedge \\ \exists w'. w\ G_{k+1;\mathcal{A}'}\ w' \wedge h' \in \llbracket w' \rrbracket_{k+1;\mathcal{A}'} \wedge w' \in (\langle\!|P'_p * \mathbf{t}^k_\alpha(\vec{e}, x) * P(x, \vec{x}) * \alpha \Mapsto \blacklozenge|\!\rangle^\rho_{\mathcal{A}'} * r) \end{array} \right\}$$
$$= \mathsf{a}\big(\langle\!|P_p * \mathbf{t}^k_\alpha(\vec{e}, x) * P(x, \vec{x}) * \alpha \Mapsto \blacklozenge|\!\rangle^\rho_{\mathcal{A}'}, \langle\!|P'_p * \mathbf{t}^k_\alpha(\vec{e}, x) * P(x, \vec{x}) * \alpha \Mapsto \blacklozenge|\!\rangle^\rho_{\mathcal{A}'}\big)^{\mathcal{A}'}_{k+1}(h)$$

Therefore by definition 61,

$$\mathcal{R}\left[\!\left[a\Big(\forall x \in X, \vec{x} \in \overrightarrow{X}.\, P_p * I_r(\mathbf{t}_\alpha^k(\vec{e}, x)) * P(x, \vec{x}),\ P'_p * I_r(\mathbf{t}_\alpha^k(\vec{e}, x)) * P(x, \vec{x})\Big)_k^{\mathcal{A}}\right]\!\right]^\rho$$
$$\subseteq \mathcal{R}\left[\!\left[a\Big(\forall x \in X, \vec{x} \in \overrightarrow{X}.\, P_p * \mathbf{t}_\alpha^k(\vec{e}, x) * P(x, \vec{x}) * \alpha \Mapsto \blacklozenge,\ P'_p * \mathbf{t}_\alpha^k(\vec{e}, x) * P(x, \vec{x}) * \alpha \Mapsto \blacklozenge\Big)_{k+1}^{\alpha : x \in X \rightsquigarrow Y(x), \mathcal{A}}\right]\!\right]^\rho$$

By lemma 12,

$$\left(\mathcal{R}\left[\!\left[a\Big(\forall x \in X, \vec{x} \in \overrightarrow{X}.\, P_p * I_r(\mathbf{t}_\alpha^k(\vec{e}, x)) * P(x, \vec{x}),\ P'_p * I_r(\mathbf{t}_\alpha^k(\vec{e}, x)) * P(x, \vec{x})\Big)_k^{\mathcal{A}}\right]\!\right]^\rho\right)^\dagger$$
$$\subseteq \left(\mathcal{R}\left[\!\left[a\Big(\forall x \in X, \vec{x} \in \overrightarrow{X}.\, P_p * \mathbf{t}_\alpha^k(\vec{e}, x) * P(x, \vec{x}) * \alpha \Mapsto \blacklozenge,\ P'_p * \mathbf{t}_\alpha^k(\vec{e}, x) * P(x, \vec{x}) * \alpha \Mapsto \blacklozenge\Big)_{k+1}^{\alpha : x \in X \rightsquigarrow Y(x), \mathcal{A}}\right]\!\right]^\rho\right)^\dagger$$

By lemma 16,

$$\left[\!\left[a\Big(\forall x \in X, \vec{x} \in \overrightarrow{X}.\, P_p * I_r(\mathbf{t}_\alpha^k(\vec{e}, x)) * P(x, \vec{x}),\ P'_p * I_r(\mathbf{t}_\alpha^k(\vec{e}, x)) * P(x, \vec{x})\Big)_k^{\mathcal{A}}\right]\!\right]^\rho$$
$$\subseteq \left[\!\left[a\Big(\forall x \in X, \vec{x} \in \overrightarrow{X}.\, P_p * \mathbf{t}_\alpha^k(\vec{e}, x) * P(x, \vec{x}) * \alpha \Mapsto \blacklozenge,\ P'_p * \mathbf{t}_\alpha^k(\vec{e}, x) * P(x, \vec{x}) * \alpha \Mapsto \blacklozenge\Big)_{k+1}^{\alpha : x \in X \rightsquigarrow Y(x), \mathcal{A}}\right]\!\right]^\rho$$

Thus, by definition 52

$$a\Big(\forall x \in X, \vec{x} \in \overrightarrow{X}.\, P_p * I_r(\mathbf{t}_\alpha^k(\vec{e}, x)) * P(x, \vec{x}),\ P'_p * I_r(\mathbf{t}_\alpha^k(\vec{e}, x)) * P(x, \vec{x})\Big)_k^{\mathcal{A}}$$
$$\sqsubseteq a\Big(\forall x \in X, \vec{x} \in \overrightarrow{X}.\, P_p * \mathbf{t}_\alpha^k(\vec{e}, x) * P(x, \vec{x}) * \alpha \Mapsto \blacklozenge,\ P'_p * \mathbf{t}_\alpha^k(\vec{e}, x) * P(x, \vec{x}) * \alpha \Mapsto \blacklozenge\Big)_{k+1}^{\alpha : x \in X \rightsquigarrow Y(x), \mathcal{A}}$$

□

Lemma 104.

$$\exists z \in Z.\, a\left(\begin{array}{l}\forall x \in X, \vec{x} \in \overrightarrow{X}. P_p * I_r(\mathbf{t}_\alpha^k(\vec{e}, x)) * P(x, \vec{x}), \\ \qquad \exists y \in Y(x).\, P''_p(x, \vec{x}, z, y) * \big(I_r(\mathbf{t}_\alpha^k(\vec{e}, y)) * Q_1(x, \vec{x}, z, y)\big) \\ \qquad \vee \big(I_r(\mathbf{t}_\alpha^k(\vec{e}, x)) * Q_2(x, \vec{x}, z)\big)\end{array}\right)_k^{\mathcal{A}}$$
$$\sqsubseteq \exists z \in Z.\, a\left(\begin{array}{l}\forall x \in X, \vec{x} \in \overrightarrow{X}. P_p * \mathbf{t}_\alpha^k(\vec{e}, x) * P(x, \vec{x}) * \alpha \Mapsto \blacklozenge, \\ \qquad \exists y \in Y(x).\, P''_p(x, \vec{x}, z, y) * \big(\mathbf{t}_\alpha^k(y) * Q_1(x, \vec{x}, z, y) * \alpha \Mapsto (x, y)\big) \\ \qquad \vee \big(\mathbf{t}_\alpha^k(\vec{e}, x) * Q_2(x, \vec{x}, z) * \alpha \Mapsto \blacklozenge\big)\end{array}\right)_{k+1}^{\alpha : x \in X \rightsquigarrow Y(x)}$$

Proof. Let ρ such that it closes both specifications. Fix $x \in X, \vec{x} \in \overrightarrow{X}$. Let $\mathcal{A}' = \alpha : x \in X \rightsquigarrow Y(x), \mathcal{A}$. Fix $z \in Z$.

$$\mathsf{a}\left(\begin{array}{l}(\!|P_p * I_r(\mathbf{t}_\alpha^k(\vec{e}, x)) * P(x, \vec{x})|\!)^\rho_{\mathcal{A}}, \\ (\!|\exists y \in Y(x).\, P''_p(x, \vec{x}, z, y) * \big(I_r(\mathbf{t}_\alpha^k(\vec{e}, y)) * Q_1(x, \vec{x}, z, y)\big) \vee \big(I_r(\mathbf{t}_\alpha^k(\vec{e}, x)) * Q_2(x, \vec{x}, z)\big)|\!)^\rho_{\mathcal{A}}\end{array}\right)_k^{\mathcal{A}}(\overline{h}) =$$
$$\left\{\overline{h}' \in \textsc{Heap} \;\middle|\; \begin{array}{l}\forall \overline{r} \in \textsc{View}_{\mathcal{A}}. \forall \overline{w} \in ((\!|P_p * I_r(\mathbf{t}_\alpha^k(\vec{e}, x)) * P(x, \vec{x})|\!)^\rho_{\mathcal{A}} * \overline{r}) \wedge \overline{h} \in \|\overline{w}\|_{k;\mathcal{A}} \wedge \exists \overline{w}.\, \overline{w}\ G_{k;\mathcal{A}}\ \overline{w}' \\ \quad \wedge\ \overline{h}' \in \|\overline{w}'\|_{k;\mathcal{A}} \wedge \overline{w}' \in ((\!|\exists y \in Y(x).\, P''_p(x, \vec{x}, z, y) * (I_r(\mathbf{t}_\alpha^k(\vec{e}, y)) * Q_1(x, \vec{x}, z, y)|\!)^\rho_{\mathcal{A}}) \\ \quad \cup\ ((\!|I_r(\mathbf{t}_\alpha^k(\vec{e}, x)) * Q_2(x, \vec{x}, z)|\!)^\rho_{\mathcal{A}}) * \overline{r})\end{array}\right\}$$

Fix $r \in \textsc{View}_{\mathcal{A}'}$, $w \in ((\!|P_p * I_r(\mathbf{t}_\alpha^k(\vec{e}, x)) * P(x, \vec{x}) * \alpha \Mapsto \blacklozenge|\!)^\rho_{\mathcal{A}'} * r)$, $h \in \|w\|_{k+1;\mathcal{A}'}$ and $h' \in \|w'\|_{k+1;\mathcal{A}'}$.

Let $\overline{r} \in \textsc{View}_{\mathcal{A}}$ such that we open all regions at level k (except α) with their states as given by w and remove their atomicity tracking component.

$$\overline{R} = \mathsf{removedone}_{\alpha}\left(r * \circledast_{\substack{\alpha' \in \mathrm{RID} \\ \alpha' \neq \alpha \\ r_w(\alpha') = (k,-,-)}} (\!| I_r(r_w(\alpha'), \alpha', \beta_w(\alpha')) |\!)^{\rho}_{\mathcal{A}'}\right)$$

There is some $\overline{w} \in ((\!| P_p * I_r(\mathbf{t}^k_\alpha(\vec{e}, x)) * P(x, \vec{x}) |\!)^{\rho}_{\mathcal{A}} * \overline{r})$, with $r_w = r_{\overline{w}}$, $\beta_w = \beta_{\overline{w}}$ and $\lVert \overline{w} \rVert_{k;\mathcal{A}} = \lVert w \rVert_{k+1;\mathcal{A}}$ and thus $h \in \lVert \overline{w} \rVert_{k;\mathcal{A}}$.

Thus, there is some $\overline{w}'$ such that $\overline{w}\ G_{k;\mathcal{A}}\ \overline{w}'$ and $h' \in \lVert \overline{w}' \rVert_{k;\mathcal{A}}$ and

$$\overline{w}' \in \left((\!| \exists y \in Y(x).\, P''_p(x, \vec{x}, z, y) * \left(\mathbf{t}^k_\alpha(\vec{e}, y) * Q_1(x, \vec{x}, z, y) |\!)^{\rho}_{\mathcal{A}}\right) \cup \left((\!| \mathbf{t}^k_\alpha(\vec{e}, x) * Q_2(x, \vec{x}, z) |\!)^{\rho}_{\mathcal{A}}\right) * \overline{r}\right)$$

We have the following cases for $\overline{w}'$:

- $\overline{w}' \in ((\!| P''_p(x, \vec{x}, z, y) * I_r(\mathbf{t}^k_\alpha(\vec{e}, y)) * Q_1(x, \vec{x}, z, y) |\!)^{\rho}_{\mathcal{A}} * \overline{r})$ for some $y \in Y(x)$ and $z \in Z$. Then, $\overline{w}' = w'' \circ \overline{w'}$ where

  $$\overline{w'} \in \left((\!| I_r(\mathbf{t}^k_\alpha(\vec{e}, y)) |\!)^{\rho}_{\mathcal{A}} * \circledast_{\substack{\alpha' \in \mathrm{RID} \\ \alpha' \neq \alpha \\ r_w(\alpha') = (k,-,-)}} (\!| I_r(r_w(\alpha'), \alpha', \beta_w(\alpha')) |\!)^{\rho}_{\mathcal{A}}\right)$$

  and $w'' \in ((\!| P''_p(x, \vec{x}, z, y) * Q_1(x, \vec{x}, z, y) |\!)^{\rho}_{\mathcal{A}'} * r)$.

  Let $w' = (r_{w''}, h_{w''}, b_{w''}, \gamma_{w''}, \beta_{w''}[\alpha \mapsto z], d_{w''}[\alpha \mapsto (x, z)])$.

  Then, by the guarantee, $w' \in ((\!| P''_p(x, z, y) * \mathbf{t}^k_\alpha(\vec{e}, y) * Q_1(x, \vec{x}, z, y) |\!)^{\rho}_{\mathcal{A}'} * r)$ and by construction $\lVert w' \rVert_{k+1;\mathcal{A}'} = \lVert \overline{w}' \rVert_{k;\mathcal{A}}$.

- $\overline{w}' \in ((\!| P''_p(x, \vec{x}, z, y) * I_r(\mathbf{t}^k_\alpha(\vec{e}, x)) * Q_2(x, \vec{x}, z) |\!)^{\rho}_{\mathcal{A}} * \overline{r})$ for some $y \in Y(x)$ and $z \in Z$. Then, $\overline{w}' = w'' \circ \overline{w'}$ where

  $$\overline{w'} \in \left((\!| I_r(\mathbf{t}^k_\alpha(\vec{e}, x)) |\!)^{\rho}_{\mathcal{A}} * \circledast_{\substack{\alpha' \in \mathrm{RID} \\ \alpha' \neq \alpha \\ r_{w'}(\alpha') = (k,-,-)}} (\!| I_r(r_w(\alpha'), \alpha', \beta_w \alpha') |\!)^{\rho}_{\mathcal{A}}\right)$$

  and $w'' \in ((\!| P''_p(x, \vec{x}, z, y) * Q_2(x, \vec{x}, z) |\!)^{\rho}_{\mathcal{A}'} * r)$.

  Let $w' = (r_{w''}, h_{w''}, b_{w''}, \gamma_{w''}, \beta_{w''}, d_{w''}[\alpha \mapsto \blacklozenge])$.

  Then, by the guarantee, $w' \in ((\!| P''_p(x, \vec{x}, z, y) * \mathbf{t}^k_\alpha(\vec{e}, x) * Q_2(x, \vec{x}, z) |\!)^{\rho}_{\mathcal{A}'} * r)$ and by construction $\lVert w' \rVert_{k+1;\mathcal{A}'} = \lVert \overline{w}' \rVert_{k;\mathcal{A}}$.

By both cases we have that:

$$
\begin{aligned}
&\mathsf{a}\left(\begin{array}{l}(\!|P_p * I_r(\mathbf{t}^k_\alpha(\vec{e},x)) * P(x,\vec{x})|\!)^\rho_{\mathcal{A}},\\ (\!|\exists y \in Y(x).\, P''_p(x,\vec{x},z,y) * \big(I_r(\mathbf{t}^k_\alpha(\vec{e},y)) * Q_1(x,\vec{x},z,y)\big) \vee \big(I_r(\mathbf{t}^k_\alpha(\vec{e},x)) * Q_2(x,\vec{x},z)\big)|\!)^\rho_{\mathcal{A}}\end{array}\right)^{\mathcal{A}}_k(\overline{h}) =\\
&\left\{\overline{h}' \in \textsc{Heap} \;\middle|\; \begin{array}{l}\forall \overline{r} \in \textsc{View}_{\mathcal{A}}.\, \forall \overline{w} \in ((\!|P_p * I_r(\mathbf{t}^k_\alpha(\vec{e},x)) * P(x,\vec{x})|\!)^\rho_{\mathcal{A}} * \overline{r}) \wedge \overline{h} \in \lfloor\!\lfloor \overline{w} \rfloor\!\rfloor_{k;\mathcal{A}} \wedge \exists \overline{w}.\, \overline{w}\; G_{k;\mathcal{A}}\; \overline{w}'\\ \quad \wedge \overline{h}' \in \lfloor\!\lfloor \overline{w}' \rfloor\!\rfloor_{k;\mathcal{A}} \wedge \overline{w}' \in (\exists y \in Y(x).\, (\!|P''_p(x,\vec{x},z,y) * (I_r(\mathbf{t}^k_\alpha(\vec{e},y)) * Q_1(x,\vec{x},z,y)|\!)^\rho_{\mathcal{A}})\\ \quad \cup ((\!|I_r(\mathbf{t}^k_\alpha(\vec{e},x)) * Q_2(x,\vec{x},z)|\!)^\rho_{\mathcal{A}}) * \overline{r})\end{array}\right\}\\
&\subseteq \left\{h' \in \textsc{Heap} \;\middle|\; \begin{array}{l}\forall r \in \textsc{View}_{\mathcal{A}'}.\, \forall w \in ((\!|P_p * \mathbf{t}^k_\alpha(\vec{e},x) * P(x,\vec{x}) * \alpha \Mapsto \blacklozenge|\!)^\rho_{\mathcal{A}'} * r) \wedge h \in \lfloor\!\lfloor w \rfloor\!\rfloor_{k+1;\mathcal{A}'}\\ \quad \wedge \exists w.\, w\; G_{k+1;\mathcal{A}'}\; w' \wedge h' \in \lfloor\!\lfloor w' \rfloor\!\rfloor_{k+1;\mathcal{A}'}\\ \quad \wedge w' \in ((\!|\exists y \in Y(x).\, P''_p(x,\vec{x},z,y) * (\mathbf{t}^k_\alpha(\vec{e},z) * Q_1(x,\vec{x},z,y) * \alpha \Mapsto (x,y)|\!)^\rho_{\mathcal{A}'})\\ \quad \vee ((\!|\mathbf{t}^k_\alpha(\vec{e},x) * Q_2(x,\vec{x},z) * \alpha \Mapsto \blacklozenge|\!)^\rho_{\mathcal{A}'}) * r)\end{array}\right\}\\
&= \mathsf{a}\left(\begin{array}{l}(\!|P_p * \mathbf{t}^k_\alpha(\vec{e},x) * P(x,\vec{x}) * \alpha \Mapsto \blacklozenge|\!)^\rho_{\mathcal{A}'},\\ (\!|\exists y \in Y(x).\, P''_p(x,\vec{x},z,y) * \begin{array}{l}\big(\mathbf{t}^k_\alpha(y) * Q_1(x,\vec{x},z,y) * \alpha \Mapsto (x,y)\big)\\ \vee \big(\mathbf{t}^k_\alpha(\vec{e},x) * Q_2(x,\vec{x},z) * \alpha \Mapsto \blacklozenge\big)\end{array}|\!)^\rho_{\mathcal{A}'}\end{array}\right)^{\mathcal{A}'}_{k+1}(h)
\end{aligned}
$$

Then by definition 61,

$$
\begin{aligned}
&\mathcal{R}\left\llbracket a\left(\begin{array}{r}\forall x \in X, \vec{x} \in \overrightarrow{X}.P_p * I_r(\mathbf{t}^k_\alpha(\vec{e},x)) * P(x,\vec{x}),\\ \exists y \in Y(x).\, P''_p(x,\vec{x},z,y) * \big(I_r(\mathbf{t}^k_\alpha(\vec{e},y)) * Q_1(x,\vec{x},z,y)\big)\\ \vee \big(I_r(\mathbf{t}^k_\alpha(\vec{e},x)) * Q_2(x,\vec{x},z)\big)\end{array}\right)^{\mathcal{A}}_k\right\rrbracket^\rho\\
&\subseteq \mathcal{R}\left\llbracket a\left(\begin{array}{r}\forall x \in X, \vec{x} \in \overrightarrow{X}.P_p * \mathbf{t}^k_\alpha(\vec{e},x) * P(x,\vec{x}) * \alpha \Mapsto \blacklozenge,\\ \exists y \in Y(x).\, P''_p(x,\vec{x},z,y) * \big(\mathbf{t}^k_\alpha(y) * Q_1(x,\vec{x},z,y) * \alpha \Mapsto (x,y)\big)\\ \vee \big(\mathbf{t}^k_\alpha(\vec{e},x) * Q_2(x,\vec{x},z) * \alpha \Mapsto \blacklozenge\big)\end{array}\right)^{\alpha:x\in X\rightsquigarrow Y(x)}_{k+1}\right\rrbracket^\rho
\end{aligned}
$$

By lemma 12,

$$
\begin{aligned}
&\left(\mathcal{R}\left\llbracket a\left(\begin{array}{r}\forall x \in X, \vec{x} \in \overrightarrow{X}.P_p * I_r(\mathbf{t}^k_\alpha(\vec{e},x)) * P(x,\vec{x}),\\ \exists y \in Y(x).\, P''_p(x,\vec{x},z) * \big(I_r(\mathbf{t}^k_\alpha(\vec{e},y)) * Q_1(x,\vec{x},z,y)\big)\\ \vee \big(I_r(\mathbf{t}^k_\alpha(\vec{e},x)) * Q_2(x,\vec{x},z)\big)\end{array}\right)^{\mathcal{A}}_k\right\rrbracket^\rho\right)^\dagger\\
&\subseteq \left(\mathcal{R}\left\llbracket a\left(\begin{array}{r}\forall x \in X, \vec{x} \in \overrightarrow{X}.P_p * \mathbf{t}^k_\alpha(\vec{e},x) * P(x,\vec{x}) * \alpha \Mapsto \blacklozenge,\\ \exists y \in Y(x).\, P''_p(x,\vec{x},z,y) * \big(\mathbf{t}^k_\alpha(y) * Q_1(x,\vec{x},z,y) * \alpha \Mapsto (x,y)\big)\\ \vee \big(\mathbf{t}^k_\alpha(\vec{e},x) * Q_2(x,\vec{x},z) * \alpha \Mapsto \blacklozenge\big)\end{array}\right)^{\alpha:x\in X\rightsquigarrow Y(x)}_{k+1}\right\rrbracket^\rho\right)^\dagger
\end{aligned}
$$

Then, by lemma 16,

$$
\begin{aligned}
&\left\llbracket a\left(\begin{array}{r}\forall x \in X, \vec{x} \in \overrightarrow{X}.P_p * I_r(\mathbf{t}^k_\alpha(\vec{e},x)) * P(x,\vec{x}),\\ \exists y \in Y(x).\, P''_p(x,\vec{x},z,y) * \big(I_r(\mathbf{t}^k_\alpha(\vec{e},y)) * Q_1(x,\vec{x},z,y)\big)\\ \vee \big(I_r(\mathbf{t}^k_\alpha(\vec{e},x)) * Q_2(x,\vec{x},z)\big)\end{array}\right)^{\mathcal{A}}_k\right\rrbracket^\rho\\
&\subseteq \left\llbracket a\left(\begin{array}{r}\forall x \in X, \vec{x} \in \overrightarrow{X}.P_p * \mathbf{t}^k_\alpha(\vec{e},x) * P(x,\vec{x}) * \alpha \Mapsto \blacklozenge,\\ \exists y \in Y(x).\, P''_p(x,\vec{x},z,y) * \big(\mathbf{t}^k_\alpha(y) * Q_1(x,\vec{x},z,y) * \alpha \Mapsto (x,y)\big)\\ \vee \big(\mathbf{t}^k_\alpha(\vec{e},x) * Q_2(x,\vec{x},z) * \alpha \Mapsto \blacklozenge\big)\end{array}\right)^{\alpha:x\in X\rightsquigarrow Y(x)}_{k+1}\right\rrbracket^\rho
\end{aligned}
$$

Thus, by definition 52 and CMono,

$$\exists z \in Z.\, a \left(\begin{array}{r} \forall x \in X, \vec{x} \in \overrightarrow{X}. P_p * I_r(\mathbf{t}_\alpha^k(\vec{e}, x)) * P(x, \vec{x}), \\ \exists y \in Y(x).\, P_p''(x, \vec{x}, z, y) * \left(I_r(\mathbf{t}_\alpha^k(\vec{e}, y)) * Q_1(x, \vec{x}, z, y)\right) \\ \vee \left(I_r(\mathbf{t}_\alpha^k(\vec{e}, x)) * Q_2(x, \vec{x}, z)\right) \end{array} \right)_k^{\mathcal{A}}$$

$$\sqsubseteq \exists z \in Z.\, a \left(\begin{array}{r} \forall x \in X, \vec{x} \in \overrightarrow{X}. P_p * \mathbf{t}_\alpha^k(\vec{e}, x) * P(x, \vec{x}) * \alpha \Rrightarrow \blacklozenge, \\ \exists y \in Y(x).\, P_p''(x, \vec{x}, z, y) * \left(\mathbf{t}_\alpha^k(y) * Q_1(x, \vec{x}, z, y) * \alpha \Rrightarrow (x, y)\right) \\ \vee \left(\mathbf{t}_\alpha^k(\vec{e}, x) * Q_2(x, \vec{x}, z) * \alpha \Rrightarrow \blacklozenge\right) \end{array} \right)_{k+1}^{\alpha : x \in X \rightsquigarrow Y(x)}$$

□

Lemma 105 (UpdateRegion)**.**

$$\mathbb{V} x \in X, \vec{x} \in \overrightarrow{X}. \left\langle \begin{array}{l} P_p \mid I_r(\mathbf{t}_\alpha^k(\vec{e}, x)) * P(x, \vec{x}), \\ \mathbb{I}(y, z) \in Y(x) \times Z.\, Q_p(x, \vec{x}, z, y) \mid \left(I_r(\mathbf{t}_\alpha^k(\vec{e}, y)) * Q_1(x, \vec{x}, z, y)\right) \\ \qquad\qquad \vee \left(I_r(\mathbf{t}_\alpha^k(\vec{e}, x)) * Q_2(x, \vec{x}, z)\right) \end{array} \right\rangle_k^{\mathcal{A}}$$

$$\sqsubseteq \mathbb{V} x \in X, \vec{x} \in \overrightarrow{X}. \left\langle \begin{array}{l} P_p \mid \mathbf{t}_\alpha^k(\vec{e}, x) * P(x, \vec{x}) * \alpha \Rrightarrow \blacklozenge, \\ \mathbb{I}(y, z) \in Y(x) \times Z.\, Q_p(x, \vec{x}, z, y) \mid \begin{array}{l} \left(\mathbf{t}_\alpha^k(\vec{e}, y) * Q_1(x, \vec{x}, z, y) * \alpha \Rrightarrow (x, y)\right) \\ \vee \left(\mathbf{t}_\alpha^k(\vec{e}, x) * Q_2(x, \vec{x}, z) * \alpha \Rrightarrow \blacklozenge\right) \end{array} \end{array} \right\rangle_{k+1}^{k\ \alpha : x \in X \rightsquigarrow Y(x), \mathcal{A}}$$

Proof.

$$\mathbb{V} x \in X, \vec{x} \in \overrightarrow{X}. \left\langle \begin{array}{l} P_p \mid I_r(\mathbf{t}_\alpha^k(\vec{e}, x)) * P(x, \vec{x}), \\ \mathbb{I}(y, z) \in Y(x) \times Z.\, Q_p(x, \vec{x}, z, y) \mid \left(I_r(\mathbf{t}_\alpha^k(\vec{e}, y)) * Q_1(x, \vec{x}, z, y)\right) \\ \qquad\qquad \vee \left(I_r(\mathbf{t}_\alpha^k(\vec{e}, x)) * Q_2(x, \vec{x}, z)\right) \end{array} \right\rangle_k^{\mathcal{A}}$$

≡ by definition 56

$$\exists p_p.\, a\Big(\forall x \in X, \vec{x} \in \overrightarrow{X}.\, P_p * I_r(\mathbf{t}_\alpha^k(\vec{e}, x)) * P(x, \vec{x}),\ P_p \wedge p_p * I_r(\mathbf{t}_\alpha^k(\vec{e}, x)) * P(x, \vec{x})\Big)_k^{\mathcal{A}};$$

$$\mu A.\, \lambda p_p.\, \exists p_p'.\, a\Big(\forall x \in X, \vec{x} \in \overrightarrow{X}.\, p_p * I_r(\mathbf{t}_\alpha^k(\vec{e}, x)) * P(x, \vec{x}),\ p_p' * I_r(\mathbf{t}_\alpha^k(\vec{e}, x)) * P(x, \vec{x})\Big)_k^{\mathcal{A}}; A p_p'$$

$$\sqcup \exists x \in X, (y, z) \in Y(x) \times Z, p_p''.\, a \left(\begin{array}{l} p_p * I_r(\mathbf{t}_\alpha^k(\vec{e}, x)) * P(x, \vec{x}), \\ p_p'' * \left(I_r(\mathbf{t}_\alpha^k(\vec{e}, y)) * Q_1(x, \vec{x}, z, y)\right) \\ \qquad \vee \left(I_r(\mathbf{t}_\alpha^k(\vec{e}, x)) * Q_2(x, \vec{x}, z)\right) \end{array} \right)_k^{\mathcal{A}};$$

$$\mu B.\, \lambda p_p''.\, \exists p_p'''.\, a\big(p_p'',\, p_p'''\big)_k^{\mathcal{A}}; B p_p'''$$

$$\sqcup\ a\big(p_p'',\, Q_p(x, \vec{x}, z, y)\big)_k^{\mathcal{A}}$$

$$\cdot p_p''$$

$$\cdot p_p$$

⊑ by lemma 103, lemma 104, Cons, RLevel, AContext, CMono

for the recursive function, then RIEq,AContext,Frame,Cons and CMono, for the first step, with $\mathcal{A}' = \alpha : x \in X \rightsquigarrow Q(x), \mathcal{A}$

$$
\begin{aligned}
&\exists p_p.\, a\Big(\forall x \in X, \vec{x} \in \overrightarrow{X}.\, P_p * \mathbf{t}^k_\alpha(\vec{e}, x) * P(x,\vec{x}) * \alpha \Rrightarrow \blacklozenge,\ P_p \wedge p_p * \mathbf{t}^k_\alpha(\vec{e}, x) * P(x,\vec{x}) * \alpha \Rrightarrow \blacklozenge\Big)^{\mathcal{A}'}_{k+1};\\
&\mu A.\, \lambda P_p.\ \exists p'_p.\, a\Big(\forall x \in X, \vec{x} \in \overrightarrow{X}.\, p_p * \mathbf{t}^k_\alpha(\vec{e}, x) * P(x,\vec{x}) * \alpha \Rrightarrow \blacklozenge,\ p'_p * I_r(\mathbf{t}^k_\alpha(\vec{e}, x)) * P(x,\vec{x}) * \alpha \Rrightarrow \blacklozenge\Big)^{\mathcal{A}'}_{k+1}; A p'_p\\
&\qquad \sqcup\ \exists x \in X, (y,z) \in Y(x) \times Z, p''_p.\\
&\qquad\quad a\begin{pmatrix} p_p * I_r(\mathbf{t}^k_\alpha(\vec{e}, x)) * P(x,\vec{x}), \\ p''_p * \big(\mathbf{t}^k_\alpha(\vec{e}, z) * Q_1(x,\vec{x},z,y) * \alpha \Rrightarrow (x,y)\big) \\ \qquad \vee \big(\mathbf{t}^k_\alpha(\vec{e}, x) * Q_2(x,\vec{x},z) * \alpha \Rrightarrow \blacklozenge\big) \end{pmatrix}^{\mathcal{A}'}_{k+1};\\
&\qquad\quad \mu B.\, \lambda p''_p.\ \exists p'''_p.\, a\big(p''_p,\ p'''_p\big)^{\mathcal{A}'}_{k+1}; B p'''_p\\
&\qquad\qquad\quad \sqcup\, a\big(p''_p,\ Q_p(x,\vec{x},z,y)\big)^{\mathcal{A}'}_{k+1}\\
&\qquad\quad \cdot p''_p\\
&\quad \cdot p_p\\
\equiv\ &\text{by definition 56}\\
&\quad \forall x \in X, \vec{x} \in \overrightarrow{X}.\\
&\quad \left\langle \begin{array}{l} P_p \mid \mathbf{t}^k_\alpha(\vec{e}, x) * P(x,\vec{x}) * \alpha \Rrightarrow \blacklozenge, \\ \exists (y,z) \in Y(x) \times Z.\, Q_p(x,\vec{x},z,y) \mid \big(\mathbf{t}^k_\alpha(\vec{e}, y) * Q_1(x,\vec{x},z,y) * \alpha \Rrightarrow (x,y)\big) \\ \qquad\qquad\qquad\qquad \vee \big(\mathbf{t}^k_\alpha(\vec{e}, x) * Q_2(x,\vec{x}z) * \alpha \Rrightarrow \blacklozenge\big) \end{array} \right\rangle^{\alpha : x \in X \rightsquigarrow Y(x), \mathcal{A}}_{k+1}
\end{aligned}
$$

□

Lemma 106 (OpenRegion)**.**

$$
\begin{aligned}
&\forall x \in X, \vec{x} \in \overrightarrow{X}.\, \Big\langle P_p \mid I_r(\mathbf{t}^k_\alpha(\vec{e}, x)) * P(x,\vec{x}),\ \exists \vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x},\vec{y}) \mid I_r(\mathbf{t}^k_\alpha(\vec{e}, x)) * Q(x,\vec{x},\vec{y}) \Big\rangle^{\mathcal{A}}_{k}\\
&\quad \equiv \forall x \in X, \vec{x} \in \overrightarrow{X}.\, \Big\langle P_p \mid \mathbf{t}^k_\alpha(\vec{e}, x) * P(x,\vec{x}),\ \exists \vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x},\vec{y}) \mid \mathbf{t}^k_\alpha(\vec{e}, x) * Q(x,\vec{x},\vec{y}) \Big\rangle^{\mathcal{A}}_{k+1}
\end{aligned}
$$

Proof. By definition 56, RIEq, RLevel and CMono. □

Lemma 107 (UseAtomic)**.**

$$
\begin{aligned}
&\forall x \in X, \vec{x} \in \overrightarrow{X}.\, \Big\langle P_p \mid I_r(\mathbf{t}^k_\alpha(\vec{e}, x)) * P(x,\vec{x}) * [\mathsf{G}(\vec{e}')]_\alpha,\ \exists \vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x},\vec{y}) \mid I_r(\mathbf{t}^k_\alpha(\vec{e}, f(x))) * Q(x,\vec{x},\vec{y}) \Big\rangle^{\mathcal{A}}_{k}\\
&\quad \equiv \forall x \in X, \vec{x} \in \overrightarrow{X}.\, \Big\langle P_p \mid \mathbf{t}^k_\alpha(\vec{e}, x) * P(x,\vec{x}) * [\mathsf{G}(\vec{e}')]_\alpha,\ \exists \vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x},\vec{y}) \mid \mathbf{t}^k_\alpha(\vec{e}, f(x)) * Q(x,\vec{x},\vec{y}) \Big\rangle^{\mathcal{A}}_{k+1}
\end{aligned}
$$

Proof. By definition 56, RIEq,Frame for the first step, and RUEq, RIEq, RLevel for the recursive function, and CMono. □

Lemma 108 (AFrame)**.**

$$
\begin{aligned}
&\forall \vec{x} \in \overrightarrow{X}.\, \Big\langle P_p \mid P(\vec{x}),\ \exists \vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x},\vec{y}) \mid Q(\vec{x},\vec{y}) \Big\rangle^{\mathcal{A}}_{k}\\
&\sqsubseteq \forall \vec{x} \in \overrightarrow{X}.\, \Big\langle P_p * R' \mid P(\vec{x}) * R(\vec{x}),\ \exists \vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x},\vec{y}) * R' \mid Q(\vec{x},\vec{y}) * R(\vec{x},\vec{y}) \Big\rangle^{\mathcal{A}}_{k}
\end{aligned}
$$

Proof. By definition 56, Frame and CMono. □

Lemma 109 (AWeaken1)**.**

$$\begin{aligned}&\mathbb{V}\vec{x} \in \overrightarrow{X}.\left\langle P_p \mid P' * P(\vec{x}),\ \mathbb{I}\vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x},\vec{y}) \mid Q'(\vec{x},\vec{y}) * Q(\vec{x},\vec{y})\right\rangle_k^{\mathcal{A}}\\ \sqsubseteq\ &\mathbb{V}\vec{x} \in \overrightarrow{X}.\left\langle P_p * P' \mid P(\vec{x}),\ \mathbb{I}\vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x},\vec{y}) * Q'(\vec{x},\vec{y}) \mid Q(\vec{x},\vec{y})\right\rangle_k^{\mathcal{A}}\end{aligned}$$

Proof. By definition 56, Frame and CMono. □

Lemma 110 (Primitive)**.**

$$\begin{aligned}&a\Big(\forall\vec{x} \in \overrightarrow{X}.\, P_p * P(\vec{x}),\ \exists\vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x},\vec{y}) * Q(\vec{x},\vec{y})\Big)_k^{\mathcal{A}}\\ \sqsubseteq\ &\mathbb{V}\vec{x} \in \overrightarrow{X}.\left\langle P_p \mid P(\vec{x}),\ \mathbb{I}\vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x},\vec{y}) \mid Q(\vec{x},\vec{y})\right\rangle_k^{\mathcal{A}}\end{aligned}$$

Proof. Trivial, using Mumble and Ind. □

Lemma 111 (AEElim)**.**

$$\begin{aligned}&\mathbb{V}x \in X, \vec{x} \in \overrightarrow{X}.\left\langle P_p \mid P(x,\vec{x}),\ \mathbb{I}\vec{y} \in \overrightarrow{Y}.\, P_p(x,\vec{x},\vec{y}) \mid Q(x,\vec{x},\vec{y})\right\rangle_k^{\mathcal{A}}\\ \sqsubseteq\ &\mathbb{V}\vec{x} \in \overrightarrow{X}.\left\langle P_p \mid \exists x \in X.\, P(x,\vec{x}),\ \mathbb{I}\vec{y} \in \overrightarrow{Y}.\, \exists x \in X.\, P_p(x,\vec{x},\vec{y}) \mid \exists x \in X.\, Q(x,\vec{x},\vec{y})\right\rangle_k^{\mathcal{A}}\end{aligned}$$

Proof. By EPElim and CMono. □

Lemma 112 (EAAtom)**.** *If* $x \notin \mathtt{free}(P_p) \cup \mathtt{free}(P)$ *and* $y \notin \mathtt{free}(P_p) \cup \mathtt{free}(P)$*, then*

$$\begin{aligned}&\exists y.\, \exists x.\, \mathbb{V}\vec{x} \in \overrightarrow{X}.\left\langle P_p \mid P(\vec{x}),\ \mathbb{I}\vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x},\vec{y}) \mid Q(\vec{x},\vec{y})\right\rangle_k^{\mathcal{A}}\\ \equiv\ &\mathbb{V}\vec{x} \in \overrightarrow{X}.\left\langle P_p \mid P(\vec{x}),\ \mathbb{I}y,\vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x},y,\vec{y}) \mid \exists x.\, Q(\vec{x},y,\vec{y})\right\rangle_k^{\mathcal{A}}\end{aligned}$$

Proof. By definition 56, EPAtom and CMono. □

Lemma 113 (AStutter)**.**

$$\begin{aligned}&\mathbb{V}\vec{x} \in \overrightarrow{X}.\left\langle P_p \mid P(\vec{x}),\ P'_p \mid P(\vec{x})\right\rangle_k^{\mathcal{A}}; \mathbb{V}\vec{x} \in \overrightarrow{X}.\left\langle P'_p \mid P(\vec{x}),\ \mathbb{I}\vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x},\vec{y}) \mid Q(\vec{x},\vec{y})\right\rangle_k^{\mathcal{A}}\\ \sqsubseteq\ &\mathbb{V}\vec{x} \in \overrightarrow{X}.\left\langle P_p \mid P(\vec{x}),\ \mathbb{I}\vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x},\vec{y}) \mid Q(\vec{x},\vec{y})\right\rangle_k^{\mathcal{A}}\end{aligned}$$

Proof. First,

$$\begin{aligned}&\mathbb{V}\vec{x} \in \overrightarrow{X}.\left\langle P_p \mid P(\vec{x}),\ P'_p \mid P(\vec{x})\right\rangle_k^{\mathcal{A}};\\ \equiv\ &\text{by definition 56}\\ &\exists p_p.\, a\Big(\forall\vec{x} \in \overrightarrow{X}.\, P_p * P(\vec{x}),\ P_p \wedge p_p * P(\vec{x})\Big)_k^{\mathcal{A}};\\ &\quad \mu A.\, \lambda p_p.\ \exists p'_p.\, a\Big(\forall\vec{x} \in \overrightarrow{X}.\, p_p * P(\vec{x}),\ p'_p * P(\vec{x})\Big)_k^{\mathcal{A}}; Ap'_p\\ &\qquad \sqcup\ \exists\vec{x} \in \overrightarrow{X}.\, \exists p''_p.\ a\big(p_p * P(\vec{x}),\ p''_p * P(\vec{x})\big)_k^{\mathcal{A}};\\ &\qquad\qquad \mu B.\, \lambda p''_p.\ \exists p'''_p.\, a\big(p''_p,\ p'''_p\big)_k^{\mathcal{A}}; Bp'''_p\\ &\qquad\qquad\qquad \sqcup\ a\big(p''_p,\ P'_p\big)_k^{\mathcal{A}}\\ &\qquad\qquad \cdot p''_p\end{aligned}$$

$\cdot\, p_p$

$\sqsubseteq$ by Frame, UElim, EElim and CMono

$\exists p_p.\, a\Big(\forall \vec{x} \in \overrightarrow{X}.\, P_p * P(\vec{x}),\ P_p \wedge p_p * P(\vec{x})\Big)^{\mathcal{A}}_k;$

$\quad \mu A.\, \lambda p_p.\ \exists p'_p.\, a\Big(\forall \vec{x} \in \overrightarrow{X}.\, p_p * P(\vec{x}),\ p'_p * P(\vec{x})\Big)^{\mathcal{A}}_k; Ap'_p$

$\quad\quad \sqcup\ \exists \vec{x} \in \overrightarrow{X}.\, \exists p''_p.\, a\big(p_p * P(\vec{x}),\ p''_p * P(\vec{x})\big)^{\mathcal{A}}_k;$

$\quad\quad\quad \mu B.\, \lambda p''_p.\ \exists \vec{x} \in \overrightarrow{X}.\, \exists p'''_p.\, a\big(\forall \vec{x}.\, p''_p * P(\vec{x}),\ p'''_p * P(\vec{x})\big)^{\mathcal{A}}_k; Bp'''_p$

$\quad\quad\quad\quad \sqcup\ a\Big(\forall \vec{x} \in \overrightarrow{X}.\, p''_p * P(\vec{x}),\ P'_p * P(\vec{x})\Big)^{\mathcal{A}}_k$

$\quad\quad\quad \cdot p''_p$

$\quad\quad \cdot\, p_p$

$\sqsubseteq$ by AChoiceComm, AChoiceElim, UnrollR and CMono

$\exists p_p.\, a\Big(\forall \vec{x} \in \overrightarrow{X}.\, P_p * P(\vec{x}),\ P_p \wedge p_p * P(\vec{x})\Big)^{\mathcal{A}}_k;$

$\quad \mu A.\, \lambda p_p.\ \exists p'_p.\, a\Big(\forall \vec{x} \in \overrightarrow{X}.\, p_p * P(\vec{x}),\ p'_p * P(\vec{x})\Big)^{\mathcal{A}}_k; Ap'_p$

$\quad\quad \sqcup\ \mu B.\, \lambda p''_p.\ \exists \vec{x} \in \overrightarrow{X}.\, \exists p'''_p.\, a\big(p''_p * P(\vec{x}),\ p'''_p * P(\vec{x})\big)^{\mathcal{A}}_k; Bp'''_p$

$\quad\quad\quad \sqcup\ a\Big(\forall \vec{x} \in \overrightarrow{X}.\, p''_p * P(\vec{x}),\ P'_p * P(\vec{x})\Big)^{\mathcal{A}}_k$

$\quad\quad \cdot p''_p$

$\quad \cdot\, p_p$

$\sqsubseteq$ by Ind, for premiss: α-conversion, EPElim, EElim, AChoiceElim and UnrollR

$\exists p_p.\, a\Big(\forall \vec{x} \in \overrightarrow{X}.\, P_p * P(\vec{x}),\ P_p \wedge p_p * P(\vec{x})\Big)^{\mathcal{A}}_k;$

$\quad \mu A.\, \lambda p_p.\ \exists \vec{x} \in \overrightarrow{X}.\, \exists p'_p.\, a\big(p_p * P(\vec{x}),\ p'_p * P(\vec{x})\big)^{\mathcal{A}}_k; Ap'_p$

$\quad\quad \sqcup\ a\Big(\forall \vec{x} \in \overrightarrow{X}.\, p_p * P(\vec{x}),\ P'_p * P(\vec{x})\Big)^{\mathcal{A}}_k$

$\quad \cdot\, p_p$

Then,

$\mathbb{V}\vec{x} \in \overrightarrow{X}.\, \langle P_p \mid P(\vec{x}),\ P'_p \mid P(\vec{x})\rangle^{\mathcal{A}}_k; \mathbb{V}\vec{x} \in \overrightarrow{X}.\, \Big\langle P'_p \mid P(\vec{x}),\ \mathbb{I}\vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x},\vec{y}) \mid Q(\vec{x},\vec{y})\Big\rangle^{\mathcal{A}}_k$

$\sqsubseteq$ by CMono

$\exists p.\, a\Big(\forall \vec{x} \in \overrightarrow{X}.\, P_p * P(\vec{x}),\ P_p \wedge p_p * P(\vec{x})\Big)^{\mathcal{A}}_k;$

$\quad \mu A.\, \lambda p_p.\ \exists \vec{x} \in \overrightarrow{X}.\, \exists p'_p.\, a\big(p_p * P(\vec{x}),\ p'_p * P(\vec{x})\big)^{\mathcal{A}}_k; Ap'_p$

$\quad\quad \sqcup\ a\Big(\forall \vec{x} \in \overrightarrow{X}.\, p_p * P(\vec{x}),\ P'_p * P(\vec{x})\Big)^{\mathcal{A}}_k$

$\quad \cdot\, p_p;$

$\quad \mathbb{V}\vec{x} \in \overrightarrow{X}.\, \Big\langle P'_p \mid P(\vec{x}),\ \mathbb{I}\vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x},\vec{y}) \mid Q(\vec{x},\vec{y})\Big\rangle^{\mathcal{A}}_k$

$\equiv$ by AChoiceComm and RecSeq

$\exists p.\, a\Big(\forall \vec{x} \in \overrightarrow{X}.\, P_p * P(\vec{x}),\ P_p \wedge p_p * P(\vec{x})\Big)^{\mathcal{A}}_k;$

$$\mu A.\, \lambda p_p.\ \exists \vec{x} \in \overrightarrow{X}.\, \exists p'_p.\, a\big(p_p * P(\vec{x}),\ p'_p * P(\vec{x})\big)^{\mathcal{A}}_k; Ap'_p$$
$$\sqcup\ a\Big(\forall \vec{x} \in \overrightarrow{X}.\, p_p * P(\vec{x}),\ P'_p * P(\vec{x})\Big)^{\mathcal{A}}_k;$$
$$\mathbb{V}\vec{x} \in \overrightarrow{X}.\, \Big\langle P'_p \mid P(\vec{x}),\ \mathbb{I}\vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x},\vec{y}) \mid Q(\vec{x},\vec{y}) \Big\rangle^{\mathcal{A}}_k$$
$$\cdot\, p_p$$

$\sqsubseteq$ by lemma 101 and CMono

$$\exists p.\, a\Big(\forall \vec{x} \in \overrightarrow{X}.\, P_p * P(\vec{x}),\ P_p \wedge p_p * P(\vec{x})\Big)^{\mathcal{A}}_k;$$
$$\mu A.\, \lambda p_p.\ \exists \vec{x} \in \overrightarrow{X}.\, \exists p'_p.\, a\big(p_p * P(\vec{x}),\ p'_p * P(\vec{x})\big)^{\mathcal{A}}_k; Ap'_p$$
$$\sqcup\ \mathbb{V}\vec{x} \in \overrightarrow{X}.\, \Big\langle p_p \mid P(\vec{x}),\ \mathbb{I}\vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x},\vec{y}) \mid Q(\vec{x},\vec{y}) \Big\rangle^{\mathcal{A}}_k$$
$$\cdot\, p_p$$

$\sqsubseteq$ by Ind, for premiss: lemma 101, CMono and AChoiceEq, Stutter and definition 56

$$\mathbb{V}\vec{x} \in \overrightarrow{X}.\, \Big\langle P_p \mid P(\vec{x}),\ \mathbb{I}\vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x},\vec{y}) \mid Q(\vec{x},\vec{y}) \Big\rangle^{\mathcal{A}}_k$$

□

Lemma 114 (AMumble)**.**

$$\mathbb{V}\vec{x} \in \overrightarrow{X}.\, \Big\langle P_p \mid P(\vec{x}),\ \mathbb{I}\vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x},\vec{y}) \mid Q(\vec{x},\vec{y}) \Big\rangle^{\mathcal{A}}_k$$
$$\sqsubseteq \mathbb{V}\vec{x} \in \overrightarrow{X}.\, \big\langle P_p \mid P(\vec{x}),\ P'_p \mid P'(\vec{x}) \big\rangle^{\mathcal{A}}_k; \mathbb{V}\vec{x} \in \overrightarrow{X}.\, \Big\langle P'_p \mid P'(\vec{x}),\ \mathbb{I}\vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x},\vec{y}) \mid Q(\vec{x},\vec{y}) \Big\rangle^{\mathcal{A}}_k$$

Proof. By Mumble steps in definition 56, create the recursive function for $\mathbb{V}\vec{x} \in \overrightarrow{X}.\, \Big\langle P'_p \mid P'(\vec{x}),\ \mathbb{I}\vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x},\vec{y}) \mid Q(\vec{x},\vec{y}) \Big\rangle^{\mathcal{A}}_k$. Then, apply RecSeq. □

Lemma 115 (ADisj)**.**

$$\mathbb{V}\vec{x} \in \overrightarrow{X}.\, \Big\langle P_p \mid P(\vec{x}),\ \mathbb{I}\vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x},\vec{y}) \mid Q(\vec{x},\vec{y}) \Big\rangle^{\mathcal{A}}_k \sqcup \mathbb{V}\vec{x} \in \overrightarrow{X}.\, \Big\langle P'_p \mid P'(\vec{x}),\ \mathbb{I}\vec{y} \in \overrightarrow{Y}.\, Q'_p(\vec{x},\vec{y}) \mid Q'(\vec{x},\vec{y}) \Big\rangle^{\mathcal{A}}_k$$
$$\sqsubseteq \mathbb{V}\vec{x} \in \overrightarrow{X}.\, \Big\langle P_p \vee P'_p \mid P(\vec{x}) \vee P'(\vec{x}),\ \mathbb{I}\vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x},\vec{y}) \vee Q'_p(\vec{x},\vec{y}) \mid Q(\vec{x},\vec{y}) \vee Q'(\vec{x},\vec{y}) \Big\rangle^{\mathcal{A}}_k$$

Proof. First we observe the following:

$$\mathbb{V}\vec{x} \in \overrightarrow{X}.\, \Big\langle P_p \mid P(\vec{x}),\ \mathbb{I}\vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x},\vec{y}) \mid Q(\vec{x},\vec{y}) \Big\rangle^{\mathcal{A}}_k \sqsubseteq$$
$$\exists p_p, p''_p.\, a\Big(\forall \vec{x} \in \overrightarrow{X}.\, P_p * P(\vec{x}),\ P_p \wedge p_p * P(\vec{x})\Big)^{\mathcal{A}}_k;$$
$$\mu A.\, \lambda p_p.\ \exists p'_p.\, a\Big(\forall \vec{x} \in \overrightarrow{X}.\, p_p * P(\vec{x}),\ p'_p * P(\vec{x})\Big)^{\mathcal{A}}_k; Ap'_p$$
$$\sqcup\ \exists \vec{x} \in \overrightarrow{X}, \vec{y} \in \overrightarrow{Y}.\, a\big(p_p * P(\vec{x}),\ p''_p * Q(\vec{x},\vec{y})\big)^{\mathcal{A}}_k$$
$$\cdot p_p;$$
$$\mu B.\, \lambda p''_p.\ \exists p'''_p.\, a\big(p''_p,\ p'''_p\big)^{\mathcal{A}}_k; Bp'''_p$$
$$\sqcup\ a\big(p''_p,\ Q_p(\vec{x},\vec{y})\big)^{\mathcal{A}}_k$$
$$\cdot p''_p$$

Each recursive function above is structurally similar to the recursive function of a Hoare specification statement according to lemma 7. Thus we proceed as in lemma 126. □

Lemma 116 (AConj)**.**

$$\forall \vec{x} \in \overrightarrow{X}. \left\langle P_p \mid P(\vec{x}),\ \exists \vec{y} \in \overrightarrow{Y}. Q_p(\vec{x},\vec{y}) \mid Q(\vec{x},\vec{y}) \right\rangle_k^{\mathcal{A}} \sqcap \forall \vec{x} \in \overrightarrow{X}. \left\langle P'_p \mid P'(\vec{x}),\ \exists \vec{y} \in \overrightarrow{Y}. Q'_p(\vec{x},\vec{y}) \mid Q'(\vec{x},\vec{y}) \right\rangle_k^{\mathcal{A}}$$
$$\sqsubseteq \forall \vec{x} \in \overrightarrow{X}. \left\langle P_p \wedge P'_p \mid P(\vec{x}) \wedge P'(\vec{x}),\ \exists \vec{y} \in \overrightarrow{Y}. Q_p(\vec{x},\vec{y}) \wedge Q'_p(\vec{x},\vec{y}) \mid Q(\vec{x},\vec{y}) \wedge Q'(\vec{x},\vec{y}) \right\rangle_k^{\mathcal{A}}$$

Proof. Similarly to lemma 115. □

Lemma 117 (ACons)**.** *If* $P_p \Rightarrow P'_p$*, and* $\forall \vec{x} \in \overrightarrow{X}. P(\vec{x}) \Rightarrow P'(\vec{x})$*, and* $\forall \vec{x} \in \overrightarrow{X}, \vec{y} \in \overrightarrow{Y}. Q'_p(\vec{x},\vec{y}) \Rightarrow Q_p(\vec{x},\vec{y})$*, and* $\forall \vec{x} \in \overrightarrow{X}. Q'(\vec{x},\vec{y}) \Rightarrow Q(\vec{x},\vec{y})$*, then*

$$\forall \vec{x} \in \overrightarrow{X}. \left\langle P'_p \mid P'(\vec{x}),\ \exists \vec{y} \in \overrightarrow{Y}. Q'_p(\vec{x},\vec{y}) \mid Q'(\vec{x},\vec{y}) \right\rangle_k^{\mathcal{A}}$$
$$\sqsubseteq \forall \vec{x} \in \overrightarrow{X}. \left\langle P_p \mid P(\vec{x}),\ \exists \vec{y} \in \overrightarrow{Y}. Q_p(\vec{x},\vec{y}) \mid Q(\vec{x},\vec{y}) \right\rangle_k^{\mathcal{A}}$$

Proof. By definition 56, Cons and CMono. □

Lemma 118 (Subst1)**.** *If* $f : X' \to X$*, then*

$$\forall x \in X, \vec{x} \in \overrightarrow{X}. \left\langle P_p \mid P(\vec{x}),\ \exists y \in Y(x), \vec{y} \in \overrightarrow{Y}. Q_p(\vec{x},\vec{y}) \mid Q(\vec{x},\vec{y}) \right\rangle_k^{\mathcal{A}}$$
$$\sqsubseteq \forall x' \in X', \vec{x} \in \overrightarrow{X}. \left\langle P_p \mid P(f(x'),\vec{x}),\ \exists y \in Y(f(x')), \vec{y} \in \overrightarrow{Y}. Q_p(f(x'),y,\vec{y}) \mid Q(f(x'),y,\vec{y}) \right\rangle_k^{\mathcal{A}}$$

Proof. Directly from the premiss and definition 56. □

Lemma 119 (Subst2)**.** *If* $f_x : Y'(x) \to Y(x)$*, then*

$$\forall \vec{x} \in \overrightarrow{X}. \left\langle P_p \mid P(\vec{x}),\ \exists y' \in \overrightarrow{Y'(x)}, \vec{y} \in \overrightarrow{Y}. Q_p(\vec{x}, f_x(y'), \vec{y}) \mid Q(\vec{x}, f_x(y'), \vec{y}) \right\rangle_k^{\mathcal{A}}$$
$$\sqsubseteq \forall \vec{x} \in \overrightarrow{X}. \left\langle P_p \mid P(\vec{x}),\ \exists y \in Y(x), \vec{y} \in \overrightarrow{Y}. Q_p(\vec{x},y,\vec{y}) \mid Q(\vec{x},y,\vec{y}) \right\rangle_k^{\mathcal{A}}$$

Proof. Directly from the premiss and definition 56. □

Lemma 120 (ARLevel)**.** *If* $k_1 \le k_2$*, then*

$$\forall \vec{x} \in \overrightarrow{X}. \left\langle P_p \mid P(\vec{x}),\ \exists \vec{y} \in \overrightarrow{Y}. Q_p(\vec{x},\vec{y}) \mid Q(\vec{x},\vec{y}) \right\rangle_{k_1}^{\mathcal{A}}$$
$$\sqsubseteq \forall \vec{x} \in \overrightarrow{X}. \left\langle P_p \mid P(\vec{x}),\ \exists \vec{y} \in \overrightarrow{Y}. Q_p(\vec{x},\vec{y}) \mid Q(\vec{x},\vec{y}) \right\rangle_{k_2}^{\mathcal{A}}$$

Proof. By RLevel on definition 56. □

Lemma 121 (AAContext)**.** *If* $\alpha \notin \mathcal{A}$*, then*

$$\forall \vec{x} \in \overrightarrow{X}. \left\langle P_p \mid P(\vec{x}),\ \exists \vec{y} \in \overrightarrow{Y}. Q_p(\vec{x},\vec{y}) \mid Q(\vec{x},\vec{y}) \right\rangle_k^{\mathcal{A}}$$
$$\sqsubseteq \forall \vec{x} \in \overrightarrow{X}. \left\langle P_p \mid P(\vec{x}),\ \exists \vec{y} \in \overrightarrow{Y}. Q_p(\vec{x},\vec{y}) \mid Q(\vec{x},\vec{y}) \right\rangle_k^{\alpha:x\in\overrightarrow{X}\rightsquigarrow Y(x),\mathcal{A}}$$

Proof. By AContext on definition 56. □

Lemma 122 (EElimHoare Proof)**.** *The* EElimHoare *refinement law holds.*

Proof. By definition 57, AChoiceComm, CMono, UnrollR, EElim, EAChoiceDst. □

B.5. Proofs of Hoare-Statement Refinement Laws

Lemma 123 (Seq)**.** *If* $\phi \sqsubseteq I \vdash \{P,\ R\}^{\mathcal{A}}_k$ *and* $\psi \sqsubseteq I \vdash \{R,\ Q\}^{\mathcal{A}}_k$*, then* $\phi;\psi \sqsubseteq I \vdash \{P,\ Q\}^{\mathcal{A}}_k$*.*

Proof. Follows directly from the premisses, definition 57, AStutter and CMono. □

Lemma 124.

$$I \vdash \{P,\ Q\}^{\mathcal{A}}_k \equiv \exists p.\, a(P * I,\ P \wedge p * I)^{\mathcal{A}}_k;\ \mu A.\lambda p.\ \exists p'.\, a(p * I,\ p' * I)^{\mathcal{A}}_k; Ap' \sqcup a(p * I,\ Q * I)^{\mathcal{A}}_k \cdot p$$

Proof. Similarly to lemma 7. □

Lemma 125 (Disjunction)**.** *If* $\phi \sqsubseteq I \vdash \{P_1,\ Q_1\}^{\mathcal{A}}_k$ *and* $\psi \sqsubseteq I \vdash \{P_2,\ Q_2\}^{\mathcal{A}}_k$*, then* $\phi \sqcup \psi \sqsubseteq I \vdash \{P_1 \vee P_2,\ Q_1 \vee Q_2\}^{\mathcal{A}}_k$*.*

Proof.

$\phi \sqcup \psi \sqsubseteq$ by CMono

$I \vdash \{P_1,\ Q_1\}^{\mathcal{A}}_k \sqcup I \vdash \{P_2,\ Q_2\}^{\mathcal{A}}_k$

$\sqsubseteq$ by lemma 124, lemma 6, FApplyElimRec, AChoiceComm and CMono

$$\begin{pmatrix} a(P_1 * I,\ Q_1 * I)^{\mathcal{A}}_k \\ \sqcup\ \exists p'.\, a(P_1 * I,\ p' * I)^{\mathcal{A}}_k; I \vdash \{p',\ Q_1\}^{\mathcal{A}}_k \end{pmatrix} \sqcup \begin{pmatrix} a(P_2 * I,\ Q_2 * I)^{\mathcal{A}}_k \\ \sqcup\ \exists p'.\, a(P_2 * I,\ p' * I)^{\mathcal{A}}_k; I \vdash \{p',\ Q_2\}^{\mathcal{A}}_k \end{pmatrix}$$

$\sqsubseteq$ by AChoiceAssoc, EAChoiceDst and CMono

$$a(P_1 * I,\ Q_1 * I)^{\mathcal{A}}_k \sqcup a(P_2 * I,\ Q_2 * I)^{\mathcal{A}}_k \sqcup \exists p'.\, \left(a(P_1 * I,\ p' * I)^{\mathcal{A}}_k; I \vdash \{p',\ Q_1\}^{\mathcal{A}}_k\right) \sqcup \left(a(P_2 * I,\ p' * I)^{\mathcal{A}}_k; I \vdash \{p',\ Q_2\}^{\mathcal{A}}_k\right)$$

$\sqsubseteq$ by PDisjunction, Cons and CMono

$$a(P_1 \vee P_2 * I,\ Q_1 \vee Q_2 * I)^{\mathcal{A}}_k \sqcup \exists p'.\, \left(a(P_1 * I,\ p' * I)^{\mathcal{A}}_k; I \vdash \{p',\ Q_1\}^{\mathcal{A}}_k\right) \sqcup \left(a(P_2 * I,\ p' * I)^{\mathcal{A}}_k; I \vdash \{p',\ Q_2\}^{\mathcal{A}}_k\right)$$

$\sqsubseteq$ by HCons and CMono

$$a(P_1 \vee P_2 * I,\ Q_1 \vee Q_2 * I)^{\mathcal{A}}_k \sqcup \exists p'.\, \left(a(P_1 * I,\ p' * I)^{\mathcal{A}}_k; I \vdash \{p',\ Q_1 \vee Q_2\}^{\mathcal{A}}_k\right) \sqcup \left(a(P_2 * I,\ p' * I)^{\mathcal{A}}_k; I \vdash \{p',\ Q_1 \vee Q_2\}^{\mathcal{A}}_k\right)$$

$\sqsubseteq$ by AChoiceDstR and CMono

$$a(P_1 \vee P_2 * I,\ Q_1 \vee Q_2 * I)^{\mathcal{A}}_k \sqcup \exists p'.\, \left(a(P_1 * I,\ p' * I)^{\mathcal{A}}_k \sqcup a(P_2 * I,\ p' * I)^{\mathcal{A}}_k\right); I \vdash \{p',\ Q_1 \vee Q_2\}^{\mathcal{A}}_k$$

$\sqsubseteq$ by PDisjunction, Cons and CMono

$$a(P_1 \vee P_2 * I,\ Q_1 \vee Q_2 * I)^{\mathcal{A}}_k \sqcup \exists p'.\, a(P_1 \vee P_2 * I,\ p' * I)^{\mathcal{A}}_k; I \vdash \{p',\ Q_1 \vee Q_2\}^{\mathcal{A}}_k$$

$\sqsubseteq$ by AChoiceComm, FApplyElimRec, lemma 6 and lemma 124

$I \vdash \{P_1 \vee P_2,\ Q_1 \vee Q_2\}_k^{\mathcal{A}}$

□

Lemma 126 (Conjunction). *If* $\phi \sqsubseteq I \vdash \{P_1,\ Q_1\}_k^{\mathcal{A}}$ *and* $\psi \sqsubseteq I \vdash \{P_2,\ Q_2\}_k^{\mathcal{A}}$*, then* $\phi \sqcap \psi \sqsubseteq I \vdash \{P_1 \wedge P_2,\ Q_1 \wedge Q_2\}_k^{\mathcal{A}}$.

Proof. Similarly to lemma 127.

□

Lemma 127 (Parallel). *If* $\phi \sqsubseteq I \vdash \{P_1,\ Q_1\}_k^{\mathcal{A}}$ *and* $\psi \sqsubseteq I \vdash \{P_2,\ Q_2\}_k^{\mathcal{A}}$*, then* $\phi \parallel \psi \sqsubseteq I \vdash \{P_1 * P_2,\ Q_1 * Q_2\}_k^{\mathcal{A}}$.

Proof. First we show the following:

$$
\begin{aligned}
& I \vdash \{P_1 * P_2,\ Q_1 * Q_2\}_k^{\mathcal{A}} \\
\equiv\ & \text{by lemma 124 and lemma 6} \\
& a(P_1 * P_2 * I,\ Q_1 * Q_2 * I)_k^{\mathcal{A}} \\
& \sqcup\ \exists p'.\, a(P_1 * P_2 * I,\ p' * I)_k^{\mathcal{A}}; I \vdash \{p',\ Q_1 * Q_2\}_k^{\mathcal{A}} \\
\sqsupseteq\ & \text{by AChoiceElim} \\
& a(P_1 * P_2 * I,\ Q_1 * Q_2 * I)_k^{\mathcal{A}} \\
\equiv\ & \text{by Stutter and Ind} \\
& \exists p.\ a(P_1 * P_2 * I,\ (P_1 * P_2) \wedge p * I)_k^{\mathcal{A}}; \\
& \quad \left(\mu A.\, \lambda p.\, a(p * I,\ Q_1 * Q_2 * I)_k^{\mathcal{A}} \sqcup Ap\right) p
\end{aligned}
$$

Next, for the recursive function derived above:

$$
\begin{aligned}
& a(P_1 * P_2 * I,\ Q_1 * Q_2 * I)_k^{\mathcal{A}} \sqcup \left(I \vdash \{P_1,\ Q_1\}_k^{\mathcal{A}} \parallel I \vdash \{P_1,\ Q_2\}_k^{\mathcal{A}}\right) \\
\sqsupseteq\ & \text{by AChoiceElim} \\
& I \vdash \{P_1,\ Q_1\}_k^{\mathcal{A}} \parallel I \vdash \{P_1,\ Q_2\}_k^{\mathcal{A}}
\end{aligned}
$$

Thus, by Ind and Trans:

$$I \vdash \{P_1,\ Q_1\}_k^{\mathcal{A}} \parallel I \vdash \{P_1,\ Q_2\}_k^{\mathcal{A}} \sqsubseteq I \vdash \{P_1 * P_2,\ Q_1 * Q_2\}_k^{\mathcal{A}}$$

The result follows by the premisses, CMono and Trans.

□

Lemma 128 (HFrame). $I \vdash \{P,\ Q\}_k^{\mathcal{A}} \sqsubseteq I \vdash \{P * R,\ Q * R\}_k^{\mathcal{A}}$

Proof. By definition 57, AFrame and CMono. □

Lemma 129 (EElimHoare)**.**

$$\frac{}{I \vdash \{P,\ Q\}_k^{\mathcal{A}} \sqsubseteq I \vdash \{\exists y.\, P,\ \exists y.\, Q\}_k^{\mathcal{A}}}\ \textsc{EElimHoare}$$

Proof. By definitions 57 and 56, EPElim and CMono. □

Lemma 130 (EHAtom)**.** *If* $x \notin \mathtt{free}(P) \cup \mathtt{free}(I)$*, then*

$$\exists x.\, I \vdash \{P,\ Q\}_k^{\mathcal{A}} \equiv I \vdash \{P,\ \exists x.\, Q\}_k^{\mathcal{A}}$$

Proof. By definition 57 and EAAtom. □

Lemma 131 (AWeaken2 Proof)**.**

$$\begin{aligned}&\mathbb{V}\vec{x} \in \overrightarrow{X}.\left\langle P_p \mid P(\vec{x}) * I(\vec{x}),\ \exists\vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x},\vec{y}) \mid Q(\vec{x},\vec{y}) * I(\vec{x})\right\rangle_k^{\mathcal{A}}\\ &\sqsubseteq \exists\vec{x} \in \overrightarrow{X}.\, I(\vec{x}) \vdash \left\{P_p * P(\vec{x}),\ \exists\vec{y} \in \overrightarrow{Y}.\, Q_p(\vec{x},\vec{y}) * Q(\vec{x},\vec{y})\right\}_k^{\mathcal{A}}\end{aligned}$$

Proof. Direct, by definition 57 and AWeaken1.

□

Lemma 132 (HCons)**.** *If* $P \Rightarrow P'$ *and* $Q' \Rightarrow Q$*, then* $I \vdash \{P',\ Q'\}_k^{\mathcal{A}} \sqsubseteq I \vdash \{P,\ Q\}_k^{\mathcal{A}}$*.*

Proof. By definition 57 and ACons. □

Lemma 133 (HRLevel)**.** *If* $k_1 \leq k_2$*, then* $\{P,\ Q\}_{k_1}^{\mathcal{A}} \sqsubseteq \{P,\ Q\}_{k_2}^{\mathcal{A}}$*.*

Proof. By definition 57 and ARLevel. □

Lemma 134 (HAContext)**.** *If* $\alpha \notin \mathcal{A}$*, then* $\{P,\ Q\}_k^{\mathcal{A}} \sqsubseteq \{P,\ Q\}_k^{\alpha:x\in X\rightsquigarrow Y(x),\mathcal{A}}$*.*

Proof. By definition 57 and AAContext. □

C. Fault-Tolerant Views Framework

C.1. General Framework and Soundness

We construct a general reasoning framework about host failures on top of the Views framework [35]. The framework encodes program logic judgements with fault-conditions, $s \vdash \{p\}\,\mathbb{C}\,\{q\}$, into judgements of the Views' framework program logic. Then, soundness of our framework depends on soundness of the encoded logic and on properties required by the encoding itself.

The framework's construction follows the same structure as that of Views, even sharing exactly the same parameters in some cases. We will be explicit as to with which parameters we instantiate the underlying Views framework.

Parameter 1 (Atomic Commands)**.** *A set of (syntactic) atomic commands* Atom*, ranged over by* a*.*

This is directly passed into Views as the same parameter (parameter A).

Here, the word "atomic" is used in the sense of "primitive". These commands inhibit a primitive programming language, which includes parallel composition, $\mathbb{C}_1||\mathbb{C}_2$, non-deterministic choice, $\mathbb{C}_1+\mathbb{C}_2$, iteration, $\mathbb{C}^*$, and sequential composition $\mathbb{C}_1;\mathbb{C}_2$. Traditional control flow structures of imperative programming languages, such as $\texttt{if} - \texttt{then} - \texttt{else}$ and while loops, can be encoded into this basic language by combining the primitives with additional atomic commands.

Parameter 2 (Volatile Machine States)**.** *Assume a set of volatile machine states,* Volatile*, ranged over by* v*. There is an exceptional* host-failed state*,* $\lightning \in$ Volatile*, which represents the effect of a host failure on the volatile state.*

Parameter 3 (Durable Machine States)**.** *Assume a set of durable machine states,* Durable*, ranged over by* d*.*

Definition 64 (Machine States)**.** *The set of machine states,* S*, ranged over by* s*, is defined as:*

$$S = \textsc{Volatile} \times \textsc{Durable}$$

This is passed into Views as the "machine states" parameter (parameter B).

Parameter 4 (Interpretation of Atomic Commands)**.** *Assume a function* $[\![-]\!] : \textsc{Atom} \to S \to \mathcal{P}(S)$ *that associates each atomic command with a non-determistic state transformer. (Where necessary, we lift non-deterministic state transformers to sets of states.)*

This is directly passed into Views as the same parameter (parameter C).

The following two properties, are properties of our host failure reasoning framework and not of Views. They are essential to prove soundness of the logic's inference rules based on the rules of the Views program logic.

Property 1 (Host Failure). *For every* $a \in \textsc{Atom}$, $(v, d) \in \mathrm{dom}(\llbracket a \rrbracket)$ *with* $v \neq \lightning$

$$\exists d' \in \textsc{Durable}.\ (\lightning, d) \in \llbracket a \rrbracket((v, d))$$

This property ensures that every atomic command will (potentially) host-fail if starting in a non-host-failed state. Thus, any program made up of such atomic commands will non-deterministically experience a host-failure at any point in time.

Property 2 (Host Failure Propagation). *For every* $a \in \textsc{Atom}$, $d \in \textsc{Durable}$

$$\llbracket a \rrbracket((\lightning, d)) = \{(\lightning, d)\}$$

This property ensures that once a host-failure occurs, any subsequent atomic command is forced to remain in a host-failed state.

Parameter 5 (Volatile and Durable Views). *Assume a volatile view monoid* $(\textsc{View}_v, *_v, u_v)$ *and a durable view monoid* $(\textsc{View}_d, *_d, u_d)$, *where* $\lightning \notin \textsc{View}_v$. *The view monoid* $(\textsc{View}, *, u)$ *is defined as the product lifting of the volatile and durable view monoids.*

Definition 65 (View Monoid Encoding). *Let* $(\textsc{View}_{\lightning,v}, *_{\lightning,v}, u_v)$ *be the view monoid where,* $\textsc{View}_{\lightning,v} = \textsc{View}_v \cup \{\lightning\}$ *and for every* $p, q \in \textsc{View}_{\lightning,v}$

$$p *_{\lightning,v} q = \begin{cases} \lightning & \text{if } p = \lightning \vee q = \lightning \\ p *_v q & \text{otherwise} \end{cases}$$

The view monoid $(\textsc{View}_{\lightning}, *_{\lightning}, u_{\lightning})$ *is defined as the product lifting of the view monoids* $\textsc{View}_{\lightning,v}$ *and* $\textsc{View}_d$.

This is passed on to Views as the "views commutative semi-group" parameter (parameter D). To avoid confusion from this point forwards, we refer to *views* as elements of the carrier set of the monoid, and to *Views* as the Views framework.

Parameter 6 (Volatile and Durable Reification). *Assume the volatile view reification function* $\llparenthesis - \rrparenthesis_v : \textsc{View}_v \to \mathcal{P}(\textsc{Volatile})$. *Assume the durable view reification function* $\llparenthesis - \rrparenthesis_d : \textsc{View}_d \to \mathcal{P}(\textsc{Durable})$.

Definition 66 (Reification Encoding). *The reification function* $\llparenthesis - \rrparenthesis_{\lightning} : \textsc{View}_{\lightning} \to \mathcal{P}(S)$ *is defined by:*

$$\llparenthesis (p_v, p_d) \rrparenthesis_{\lightning} = \begin{cases} \llparenthesis p_v \rrparenthesis_v \times \llparenthesis p_d \rrparenthesis_d & \text{if } p_v \neq \lightning \\ \{\lightning\} \times \llparenthesis p_d \rrparenthesis_d & \text{otherwise} \end{cases}$$

Definition 66 is passed on to Views as the "reification" parameter (parameter F).

In order to encode the concept that a command may execute normally *or* may experience a host-failure, we require a notion of disjunction for the views of definition 65. Following is the formal definition as a parameter, and the associated properties required by Views.

Parameter 7 (Disjunction). *Assume a function* $\bigvee : (I \to \textsc{View}) \to \textsc{View}$ *satisfying the following properties:*

- *Join Distributivity:* $p * \bigvee_{i \in I} q_i = \bigvee_{i \in I} (p * q_i)$
- *Join Morphism:* $[\![\bigvee_{i \in I} p_i]\!] = \bigcup_{i \in I} [\![p_i]\!]$

Definition 67 (Disjunction Encoding)**.** *The function* $\bigvee_{\updownarrow} : (I \to \textsc{View}_{\updownarrow}) \to \textsc{View}_{\updownarrow}$ *that extends* $\bigvee$ *s.t. for every* $i \in I, j \in J$ *with* $p_i = \updownarrow$ *and* $p_j \neq \updownarrow$

$$\bigvee_{\updownarrow, k \in I \cup J} p_k = \{\updownarrow\} \cup \bigvee_{j \in J} p_j$$

By the properties of set union and join distributivity and morphism of $\bigvee$*,* $\bigvee_{\updownarrow}$ *also satisfies join distributivity and morphism.*

This is passed on to Views as the "view combination" parameter (parameter N).

We associate axioms with each atomic command.

Parameter 8 (Axiomatisation)**.** *Assume a set of axioms* $\textsc{Axioms} \subseteq \textsc{View}_d \times \textsc{View} \times \textsc{Atom} \times \textsc{View}$*.*

Definition 68 (Axiomatisation Encoding)**.** *For every* $(s, p, a, q) \in \textsc{Axioms}$*,* $(p, a, q \vee (\updownarrow, s)) \in \textsc{Axioms}_{\updownarrow}$*.*

This is passed on to View as the "axiomatisation" parameter (parameter E).

Definition 69 (Entailment)**.** *The entailment relation* $\vDash \subseteq \textsc{View} \times \textsc{View}$*, is defined by:*

$$p \vDash q \iff^{def} \exists s.\ (s, p, \mathsf{id}, q) \in \textsc{Axioms}$$

The entailment relation $\vDash_d \subseteq \textsc{View}_d \times \textsc{View}_d$*, is defined by:*

$$d \vDash d' \iff^{def} \exists s, v, v'.\ (s, (v, d), \mathsf{id}, (v', d')) \in \textsc{Axioms}$$

Parameter 9 (Recovery programs)**.** *A function* **recovers** : $\textsc{Comm} \to \textsc{Comm}$ *associating programs to recovery programs. The function must be such that:* $\forall \mathbb{C}_R \in \mathrm{codom}(\textsc{Comm}).\ \mathbf{recovers}(\mathbb{C}_R) = \mathbb{C}_R$*. That is, the recovery of a recovery is the same recovery.*

Lemma 135 (Host Failure Propagation Axiom)**.** *The following axiom holds in the program logic of* $\textsc{View}_{\updownarrow}$*:*

$$\{(\updownarrow, s)\}\, \mathbb{C}\, \{(\updownarrow, s)\}$$

Proof. From property 2, for all $a \in \textsc{Atom}$, $((\updownarrow, s), a, (\updownarrow, s)) \in \textsc{Axioms}_{\updownarrow}$. The conclusion follows by induction on $\mathbb{C}$. □

Next we define how judgements of our program logic are encoded into Views judgements, define the rules of our logic and justify them by using the encoding and Views proof rules. We distinguish between the rules of our logic and rules of Views by pre-pending V in the name of a Views proof rule.

Definition 70 (Program Logic)**.** *Judgements are of the form:* $s \vdash \{p\}\, \mathbb{C}\, \{q\}$*, where* $p, q \in \textsc{View}$*,* $s \in \textsc{View}_d$ *and* $\mathbb{C} \in \textsc{Comm}$*, and are encoded in views as* $\{p\}\, \mathbb{C}\, \{q \vee (\updownarrow, s)\}$*. The proof rules for these judgements and their justification are as follows:*

$$\frac{\textsc{Axiom} \quad (s, p, a, q) \in \textsc{Axioms}}{s \vdash \{p\}\, a\, \{q\}}$$

Proof. From the axiom encoding (def 68) and the axiom rule of the Views program logic. □

$$\textsc{Frame}\ \frac{s \vdash \{p\}\,\mathbb{C}\,\{q\}}{s *_d r_d \vdash \{p * (r_v, r_d)\}\,\mathbb{C}\,\{q * (r_v, r_d)\}}$$

Proof. Assume the premiss holds. The premiss is encoded in Views as:

$$\{p\}\,\mathbb{C}\,\{q \vee (\lightning, s)\}$$

Then, apply the frame rule of the views program logic with frame (r_v, r_d) (note $r_v \neq \lightning$):

$$\textsc{VFrame}\ \frac{\{p\}\,\mathbb{C}\,\{q \vee (\lightning, s)\}}{\left\{p *_{\lightning} (r_v, r_d)\right\}\,\mathbb{C}\,\left\{q \vee (\lightning, s) *_{\lightning} (r_v, r_d)\right\}}$$

From the conclusion we have:

$$\left\{p *_{\lightning} (r_v, r_d)\right\}\,\mathbb{C}\,\left\{q \vee (\lightning, s) *_{\lightning} (r_v, r_d)\right\}$$
$$\iff$$
$$\left\{p *_{\lightning} (r_v, r_d)\right\}\,\mathbb{C}\,\left\{(q *_{\lightning} (r_v, r_d)) \vee ((\lightning, s) *_{\lightning} (r_v, r_d))\right\}$$
$$\iff$$
$$\left\{p *_{\lightning} (r_v, r_d)\right\}\,\mathbb{C}\,\left\{(q *_{\lightning} (r_v, r_d)) \vee (\lightning *_{\lightning,v} r_v, s *_d r_d)\right\}$$
$$\iff$$
$$\left\{p *_{\lightning} (r_v, r_d)\right\}\,\mathbb{C}\,\left\{(q *_{\lightning} (r_v, r_d)) \vee (\lightning, s *_d r_d)\right\}$$
$$\iff$$
$$\{p * (r_v, r_d)\}\,\mathbb{C}\,\left\{(q * (r_v, r_d)) \vee (\lightning, s *_d r_d)\right\}$$
$$\iff$$
$$s *_d r_d \vdash \{p * (r_v, r_d)\}\,\mathbb{C}\,\{q * (r_v, r_d)\}$$

which is the conclusion of our frame rule. □

$$\frac{}{s \vdash \{p\}\ \texttt{skip}\ \{p\}}$$

Proof. From the encoding of $s \vdash \{p\}\,\texttt{skip}\,\{p\}$ and application of the Views rule for `skip`. □

$$\textsc{Choice}\ \frac{s \vdash \{p\}\,\mathbb{C}_1\,\{q\} \qquad s \vdash \{p\}\,\mathbb{C}_2\,\{q\}}{s \vdash \{p\}\,\mathbb{C}_1 + \mathbb{C}_2\,\{q\}}$$

Proof. Assume the premisses hold. The premisses are encoded into views as:

$$\{p\}\,\mathbb{C}_1\,\{q \vee (\wr, s)\} \text{ and } \{p\}\,\mathbb{C}_2\,\{q \vee (\wr, s)\}$$

Applying the Views rule for $\mathbb{C}_1 + \mathbb{C}_2$ we get the conclusion $\{p\}\,\mathbb{C}_1 + \mathbb{C}_2\,\{q \vee (\wr, s)\}$ which is the encoding of the conclusion of our rule. □

$$\textsc{Iter}\ \frac{s \vdash \{p\}\,\mathbb{C}\,\{p\}}{s \vdash \{p\}\,\mathbb{C}^*\,\{p\}}$$

Proof. Assume the premiss holds. The premiss is encoded as $\{p\}\,\mathbb{C}\,\{p \vee (\wr, s)\}$. Then,

$$\dfrac{p \vDash_{\wr} p \vee (\wr, s) \qquad \dfrac{\dfrac{\{p\}\,\mathbb{C}\,\{p \vee (\wr, s)\} \qquad \{(\wr, s)\}\,\mathbb{C}\,\{(\wr, s)\}}{\{p \vee (\wr, s)\}\,\mathbb{C}\,\{p \vee (\wr, s)\}}\ \textsc{VDisj}}{\{p \vee (\wr, s)\}\,\mathbb{C}^*\,\{p \vee (\wr, s)\}}\ \textsc{VIter}}{\{p\}\,\mathbb{C}^*\,\{p \vee (\wr, s)\}}\ \textsc{VCons}$$

The final conclusion in the derivation is the encoding of the conclusion of our rule. □

$$\textsc{Seq}\ \frac{s \vdash \{p\}\,\mathbb{C}_1\,\{r\} \qquad s \vdash \{r\}\,\mathbb{C}_2\,\{q\}}{s \vdash \{p\}\,\mathbb{C}_1; \mathbb{C}_2\,\{q\}}$$

Proof. Assume the premisses hold. They are encoded as $\{p\}\,\mathbb{C}\,\{r \vee (\wr, s)\}$ and $\{r\}\,\mathbb{C}\,\{q \vee (\wr, s)\}$. Then,

$$\dfrac{\{p\}\,\mathbb{C}_1\,\{r \vee (\wr, s)\} \qquad \textsc{VDisj}\ \dfrac{\{r\}\,\mathbb{C}_2\,\{q \vee (\wr, s)\} \qquad \{(\wr, s)\}\,\mathbb{C}_2\,\{(\wr, s)\}}{\{r \vee (\wr, s)\}\,\mathbb{C}_2\,\{q \vee (\wr, s)\}}}{\{p\}\,\mathbb{C}_1; \mathbb{C}_2\,\{q \vee (\wr, s)\}}\ \textsc{VSeq}$$

The final conclusion in the derivation is the encoding of the conclusion of our rule. □

$$\textsc{Para}\ \frac{s_1 \vdash \{p_1\}\,\mathbb{C}_1\,\{q_1\} \qquad s_2 \vdash \{p_2\}\,\mathbb{C}_2\,\{q_2\} \qquad q_1 = (v_1, d_1) \qquad q_2 = (v_2, d_2)}{(s_1 *_d s_2) \vee (s_1 *_d d_2) \vee (s_2 *_d d_1) \vdash \{p_1 * p_2\}\,\mathbb{C}_1 \parallel \mathbb{C}_2\,\{q_1 * q_2\}}$$

Proof. Assume the premisses hold. The premisses are encoded as:

$\{p_1\}\,\mathbb{C}_1\,\{q_1 \vee (\lightning, s_1)\}$ and $\{p_2\}\,\mathbb{C}_2\,\{q_2 \vee (\lightning, s_2)\}$. Then,

$$\dfrac{\dfrac{\begin{array}{c}\{p_1\}\,\mathbb{C}_1\,\{q_1 \vee (\lightning, s_1)\} \\ \{p_2\}\,\mathbb{C}_2\,\{q_2 \vee (\lightning, s_2)\}\end{array}}{\left\{\ p_1 * p_2\ \right\}}\ \textsc{VPara}}{\begin{array}{c}\mathbb{C}_1 \parallel \mathbb{C}_2 \\ \left\{\ (q_1 \vee (\lightning, s_1)) * (q_2 \vee (\lightning, s_2))\ \right\} \qquad q_1 = (v_1, d_1) \qquad q_2 = (v_2, d_2)\end{array}}{\{p_1 * p_2\}\,\mathbb{C}_1 \parallel \mathbb{C}_2\,\{(q_1 * q_2) \vee (\lightning, (s_1 *_d s_2) \vee (s_1 *_d d_2) \vee (s_2 *_d d_1))\}}\ \textsc{VCons-post}$$

The final conclusion in the derivation is the encoding of the conclusion of our rule. □

$$\textsc{Cons-pre}\quad \frac{p \vDash p' \qquad s \vdash \{p'\}\,\mathbb{C}\,\{q\}}{s \vdash \{p\}\,\mathbb{C}\,\{q\}}$$

$$\textsc{Cons-post}\quad \frac{q' \vDash q \qquad s \vdash \{p\}\,\mathbb{C}\,\{q'\}}{s \vdash \{p\}\,\mathbb{C}\,\{q\}}$$

$$\textsc{Cons-fault}\quad \frac{s' \vDash s \qquad s' \vdash \{p\}\,\mathbb{C}\,\{q\}}{s \vdash \{p\}\,\mathbb{C}\,\{q\}}$$

Proof. The first rule is justified by the Views encoding and applying precondition strengthening. The second and third rules is justified by the Views encoding and applying postcondition weakening. □

$$\textsc{Disj}\quad \frac{\forall i \in I.\, s \vdash \{p_i\}\,\mathbb{C}\,\{q\}}{s \vdash \left\{\bigvee \{p_i\}_{i \in I}\right\}\,\mathbb{C}\,\{q\}}$$

Proof. The rule is justified by encoding the premiss and conclusions and the disjunction rule of Views. □

$$\textsc{Recovery-Abstraction}\quad \frac{s \vdash \{(p_v, p_d)\}\,\mathbb{C}\,\{(q_v, q_d)\} \qquad s \vdash \{(u_v, s)\}\,\mathbb{C}_R\,\{(q'_v, r)\} \qquad \mathbb{C}_R\ \textbf{recovers}\ \mathbb{C}}{r \vdash \{(p_v, p_d)\}\,[\mathbb{C}]\,\{(q_v, q_d)\}}$$

Proof. First we encode $[\mathbb{C}]$ as follows:

$$[\mathbb{C}] \triangleq \mathbb{C}; (norm + ((pf_\emptyset; \mathbb{C}_R)^+; norm; pf_\lightning))$$

where $\mathbb{C}^+ \triangleq \mathbb{C}; \mathbb{C}^*$ and $pf_\emptyset, norm, pf_\lightning$ are commands (not available to clients) with the following

axiomatic semantics:

$$\{p \vee (\lightning, s)\}\ norm\ \{p\}$$
$$\{p \vee (\lightning, s)\}\ pf_{\emptyset}\ \{(u_v, s)\}$$
$$\{(v, d)\}\ pf_{\lightning}\ \{(\lightning, d)\}$$

Note that because the commands are not available to the client, as in they are only used within the encoding into Views, properties 1 and 2 can be ignored.

Assume the premisses hold. The first two premisses are encoded as:

$$\{(p_v, p_d)\}\ \mathbb{C}\ \{(q_v, q_d) \vee (\lightning, s)\}$$
$$\{(u_v, s)\}\ \mathbb{C}_R\ \{(q'_v, r) \vee (\lightning, s)\}$$

Then, we have the following derivations:

$$\frac{\{(q_v, q_d) \vee (\lightning, s)\}\ norm\ \{(q_v, q_d)\} \qquad (q_v, q_d) \vDash (q_v, q_d) \vee (\lightning, r)}{\{(q_v, q_d) \vee (\lightning, s)\}\ norm\ \{(q_v, q_d) \vee (\lightning, r)\}}\ \textsc{VCons-post}$$

$$\frac{\{(q'_v, r) \vee (\lightning, s)\}\ norm\ \{(q'_v, r)\} \qquad \dfrac{\{(q'_v, r)\}\ pf_{\lightning}\ \{(\lightning, r)\} \quad (\lightning, r) \vDash_{\lightning} (q_v, q_d) \vee (\lightning, r)}{\{(q'_v, r)\}\ pf_{\lightning}\ \{(q_v, q_d) \vee (\lightning, r)\}}\ \textsc{VCons-post}}{\{(q'_v, r) \vee (\lightning, s)\}\ norm; pf_{\lightning}\ \{(q_v, q_d) \vee (\lightning, r)\}}\ \textsc{VSeq}$$

$$\frac{\dfrac{\{(q'_v, r) \vee (\lightning, s)\}\ pf_{\emptyset}\ \{(u_v, s)\} \qquad \{(u_v, s)\}\ \mathbb{C}_R\ \{(q'_v, r) \vee (\lightning, s)\}}{\{(q'_v, r) \vee (\lightning, s)\}\ pf_{\emptyset}; \mathbb{C}_R\ \{(q'_v, r) \vee (\lightning, s)\}}\ \textsc{VSeq}}{\{(q'_v, r) \vee (\lightning, s)\}\ (pf_{\emptyset}; \mathbb{C}_R)^*\ \{(q'_v, r) \vee (\lightning, s)\}}\ \textsc{VIter}$$

$$\frac{\dfrac{\{(q_v, q_d) \vee (\lightning, s)\}\ pf_{\emptyset}\ \{(u_v, s)\} \quad \{(u_v, s)\}\ \mathbb{C}_R\ \{(q'_v, r) \vee (\lightning, s)\}}{\left\{\ (q_v, q_d) \vee (\lightning, s)\ \right\}\ pf_{\emptyset}; \mathbb{C}_R\ \left\{\ (q'_v, r) \vee (\lightning, s)\ \right\}}\ \textsc{VSeq} \qquad \{(q'_v, r) \vee (\lightning, s)\}\ (pf_{\emptyset}; \mathbb{C}_R)^*\ \{(q'_v, r) \vee (\lightning, s)\}}{\{(q_v, q_d) \vee (\lightning, s)\}\ (pf_{\emptyset}; \mathbb{C}_R); (pf_{\emptyset}; \mathbb{C}_R)^*\ \{(q'_v, r) \vee (\lightning, s)\}}\ \textsc{VSeq}}{\{(q_v, q_d) \vee (\lightning, s)\}\ (pf_{\emptyset}; \mathbb{C}_R)^+\ \{(q'_v, r) \vee (\lightning, s)\}}\ \mathbb{C}^+$$

$$\frac{\left\{(q_v, q_d) \vee (\lightning, s)\right\} (pf_\emptyset; \mathbb{C}_R)^+ \left\{(q'_v, r) \vee (\lightning, s)\right\} \qquad \left\{(q'_v, r) \vee (\lightning, s)\right\} norm; pf_{\lightning} \left\{(q_v, q_d) \vee (\lightning, r)\right\}}{\left\{(q_v, q_d) \vee (\lightning, s)\right\} (pf_\emptyset; \mathbb{C}_R)^+; (norm; pf_{\lightning}) \left\{(q_v, q_d) \vee (\lightning, r)\right\}} \text{ VSeq}$$

$$\frac{\begin{array}{c}\left\{(q_v, q_d) \vee (\lightning, s)\right\} norm \left\{(q_v, q_d) \vee (\lightning, r)\right\} \\ \left\{(q_v, q_d) \vee (\lightning, s)\right\} (pf_\emptyset; \mathbb{C}_R)^+; norm; pf_{\lightning} \left\{(q_v, q_d) \vee (\lightning, r)\right\}\end{array}}{\left\{(q_v, q_d) \vee (\lightning, s)\right\} norm + ((pf_\emptyset; \mathbb{C}_R)^+; norm; pf_{\lightning}) \left\{(q_v, q_d) \vee (\lightning, r)\right\}} \text{ VChoice}$$

$$\frac{\dfrac{\begin{array}{c}\{(p_v, p_d)\} \mathbb{C} \left\{(q_v, q_d) \vee (\lightning, s)\right\} \\ \left\{(q_v, q_d) \vee (\lightning, s)\right\} norm + ((pf_\emptyset; \mathbb{C}_R)^+; norm; pf_{\lightning}) \left\{(q_v, q_d) \vee (\lightning, r)\right\}\end{array}}{\{(p_v, p_d)\} \mathbb{C}; (norm + ((pf_\emptyset; \mathbb{C}_R)^+; norm; pf_{\lightning})) \left\{(q_v, q_d) \vee (\lightning, r)\right\}} \text{ VSeq}}{\{(p_v, p_d)\} [\mathbb{C}] \left\{(q_v, q_d) \vee (\lightning, r)\right\}} [\mathbb{C}]$$

The final conclusion in the derivation is the encoding of the conclusion of our rule. □

When properties 1 and 2 hold, then the soundness of our program logic depends on the soundness of its encoding in Views. This is established by the "axiom soundness" property of Views (property G) for the encoded axioms of definition 68. When all the parameters of the framework are instantiated and the aforementioned three properties established, then the following theorem holds.

Theorem 7 (Soundness)**.** *If the judgement $s \vdash \{p\}\, \mathbb{C}\, \{q\}$ is derivable in the program logic, then if we run the program $\mathbb{C}$ from state reified from view p, then $\mathbb{C}$ will either not terminate, or terminate in state reified from view q, or a host failure will occur destroying any volatile state and the remaining durable state (after potential recoveries) will reify from s.*

C.2. FTCSL

We now encode FTCSL into the general framework. The encoding is based on using pairs of volatile and durable views to account for local state and the shared resource invariant. Let Volatile be the volatile machine states (parameter 2) and Durable be the durable machine states (parameter 3). Let $(\text{View}_v, *_v, u_v)$ and $(\text{View}_d, *_d, u_d)$ be (disjoint concurrent separation logic) volatile and durable view monoids respectively.

Definition 71 (FTCSL Volatile and Durable Views)**.** *The FTCSL volatile view monoid is:*

$$((\text{View}_v \times \text{View}_v) \uplus \{\bot\}, *_{cv}, u_{cv})$$

where $u_{cv} = (u_v, u_v)$*, and*

$$(p,i) *_{cv} (q,j) = \begin{cases} (p *_v q, i) & \text{if } i = j \vee j = u_v \\ (p *_v q, j) & \text{if } i = u_v \\ \bot & \text{otherwise} \end{cases}$$

Similarly, the FTCSL durable view monoid is:

$$((\textsc{View}_d \times \textsc{View}_d) \uplus \{\bot\}, *_{cd}, u_{cd})$$

where $u_{cd} = (u_d, u_d)$*, and*

$$(p,i) *_{cd} (p,j) = \begin{cases} (p *_d q, i) & \text{if } i = j \vee j = u_d \\ (p *_d q, j) & \text{if } i = u_d \\ \bot & \text{otherwise} \end{cases}$$

We instantiate parameter 5 using the definition above.

Let $[\![-]\!]_v$ and $[\![-]\!]_d$ be volatile and durable reifications for $\textsc{View}_v$ and $\textsc{View}_d$ respectively.

Definition 72 (FTCSL Volatile and Durable Reification)**.** *FTCSL volatile reification is defined as:*

$$[\![(p,i)]\!]_{cv} = [\![p *_v i]\!]_v \qquad\qquad [\![\bot]\!]_{cv} = \emptyset$$

Similarly, FTCSL durable reification is defined as:

$$[\![(p,i)]\!]_{cd} = [\![p *_d i]\!]_d \qquad\qquad [\![\bot]\!]_{cd} = \emptyset$$

We instantiate parameter 6 with the definition above.

Definition 73 (FTCSL Program Logic)**.** *Judgements of FTCSL are of the form:*

$$\boxed{(j_v, j_d)}; s \vdash \{(p_v, p_d)\}\, \mathbb{C}\, \{(q_v, q_d)\}$$

FTCSL judgements are encoded in the general framework as:

$$(s, j_d) \vdash \{((p_v, j_v), (p_d, j_d))\}\, \mathbb{C}\, \{((q_v, j_v), (q_d, j_d))\}$$

The rules of FTCSL are given in figure 9.4 (written using assertions). Sequence, consequence, frame, parallel and recovery abstraction rules are justified directly by the respective rules of the general program logic (definition 70) and the judgement encoding.

The atomic rule*, written using views, is:*

$$\frac{\boxed{(u_v, u_d)}; (p_d *_d j_d) \vee (q_d *_d j_d) \vdash \{(p_v *_v j_v, p_d *_d j_d)\}\, \mathbb{C}\, \{(q_v *_v j_v, q_d *_d j_d)\}}{\boxed{(j_v, j_d)}; p_d \vee q_d \vdash \{(p_v, p_d)\}\, \langle \mathbb{C} \rangle\, \{(q_v, q_d)\}}$$

Proof. Assume the premiss holds. The premiss is encoded into the general judgement as:

$$((p_d *_d j_d) \vee (q_d *_d j_d), u_d) \vdash \{(((p_v *_v j_v), u_v), ((p_d *_d j_d), u_d))\}\, \mathbb{C}\, \{(((q_v *_v j_v), u_v), ((q_d *_d j_d), u_d))\}$$

By the reifications of definition 72, it is clear that:

$$[\![(p_v *_v j_v, i_v)]\!]_{cv} = [\![(p_v, i_v *_v j_v)]\!]_{cv}$$
$$[\![(p_d *_d j_d, i_d)]\!]_{cd} = [\![(p_d, i_d *_d j_d)]\!]_{cd}$$

This means, that by moving views between the resource invariant and the local state does not change the underlying machine states. We use this to justify the following axioms:

$$((p_d *_d j_d), i_d) \vdash \{(((p_v *_v j_v), i_v), ((p_d *_d j_d), i_d))\}\, \mathsf{id}\, \{((p_v, (i_v *_v j_v)), (p_d, (i_d *_d j_d)))\}$$

$$(p_d, (i_d *_d j_d)) \vdash \{((p_v, (i_v * j_v)), (p_d, (i_d *_d j_d)))\}\, \mathsf{id}\, \{(((p_v *_v j_v), i_v), ((p_d *_d j_d), i_d))\}$$

$$(j_d, p_d \vee q_d) \vdash \{(u_v, (u_d, (p_d *_d q_d) \vee (q_d *_d j_d)))\}\, \mathsf{id}\, \{(u_v, (j_d, p_d \vee q_d))\}$$

The axioms justify the following entailments:

$$((p_v, j_v), (p_d, j_d)) \vDash (((p_v *_v j_v), u_v), ((p_d *_d j_d), u_d))$$
$$(((q_v *_v j_v), u_v), ((q_d *_d j_d), u_d)) \vDash ((q_v, j_v), (q_d, j_d))$$
$$(u_d, (p_d *_d j_d) \vee (q_d *_d j_d)) \vDash_d (p_d \vee q_d, j_d)$$

Then, we apply the consequence rules:

$$\frac{\begin{array}{c}(u_d, (p_d *_d j_d) \vee (q_d *_d j_d)) \vdash \left\{\ (((p_v *_v j_v), u_v), ((q_d *_d j_d), u_d))\ \right\} \\ \mathbb{C} \\ \left\{\ (((q_v *_v j_v), u_v), ((q_d *_d j_d), u_d))\ \right\} \\ ((p_v, j_v), (p_d, j_d)) \vDash (((p_v *_v j_v), u_v), ((p_d *_d j_d), u_d)) \\ (((q_v *_v j_v), u_v), ((q_d *_d j_d), u_d)) \vDash ((q_v, j_v), (q_d, j_d)) \\ (u_d, (p_d *_d j_d) \vee (q_d *_d j_d)) \vDash_d (p_d \vee q_d, j_d)\end{array}}{(p_d \vee q_d, j_d) \vdash \{((p_v, j_v), (p_d, j_d))\}\, \langle \mathbb{C} \rangle\, \{((q_v, j_v), (q_d, j_d))\}}\ \textsc{Cons-pre,post,fault}$$

The conclusion of the above derivation is the encoding of the atomic rule's conclusion. □

Next, the share rule*, written using views, is:*

$$\frac{\boxed{(j_v *_v r_v, j_d *_d r_d)}; s \vdash \{(p_v, p_d)\}\, \mathbb{C}\, \{(q_v, q_d)\}}{\boxed{(j_v, j_d)}; s \vdash \{(p_v *_v r_v, p_d *_d r_d)\}\, \mathbb{C}\, \{(q_v *_v r_v, q_d *_d r_d)\}}$$

Proof. Assume the premiss holds. The premiss is encoded into FTV as:

$$(s, j_d *_d r_d) \vdash \{((p_v, j_v *_v r_v), (p_d, j_d *_d r_d))\}\, \mathbb{C}\, \{((q_v, j_v *_v r_v), (q_d, j_d *_d r_d))\}$$

By the same argument as in the atomic rule, the following entailments hold:

$$((p_v *_v r_v, j_v), (p_d *_d r_d, j_d)) \vDash ((p_v, j_v *_v r_v), (p_d, j_d *_d r_d))$$
$$((q_v, j_v *_v r_v), (q_d, j_d *_d r_d)) \vDash ((q_v *_v r_v, j_v), (q_d *_d r_d, j_d))$$
$$(s, j_d *_d r_d) \vDash_d (s *_d r_d, j_d)$$

Then, by application of the consequence rules:

$$\frac{\begin{array}{c}(s, j_d *_d r_d) \vdash \{((p_v, j_v *_v r_v), (p_d, j_d *_d r_d))\}\, \mathbb{C}\, \{((q_v, j_v *_v r_v), (q_d, j_d *_d r_d))\} \\ ((p_v *_v r_v, j_v), (p_d *_d r_d, j_d)) \vDash ((p_v, j_v *_v r_v), (p_d, j_d *_d r_d)) \\ ((q_v, j_v *_v r_v), (q_d, j_d *_d r_d)) \vDash ((q_v *_v r_v, j_v), (q_d *_d r_d, j_d))(s, j_d *_d r_d) \vDash_d (s *_d r_d, j_d)\end{array}}{(s *_d r_d, j_d) \vdash \{((p_v *_v r_v, j_v), (p_d *_d r_d, j_d))\}\, \mathbb{C}\, \{((q_v *_v r_v, j_v), (q_d *_d r_d, j_d))\}}\ \textsc{Cons-pre,post}$$

The conclusion of the above derivation is the encoding of the atomic rule's conclusion. □

Given the parameters to the general framework described here, including disjunction (parameter 7) and recovery programs 9, and with the required properties established, we justify soundness of FTCSL by the soundness of the general framework.

Theorem 8 (FTCSL Soundness)**.** *If the judgement* $\boxed{(j_v, j_d)}; s \vdash \{(p_v, p_d)\}\, \mathbb{C}\, \{(q_v, q_d)\}$ *is derivable in the program logic, then if we run the program* $\mathbb{C}$ *from an initial state that is reified from view* $(p_v *_v j_v, p_d *_d j_d)$*, then* $\mathbb{C}$ *will either not terminate, or it will terminate in a state reified from view* $(q_v *_v j_v, q_d *_d j_d)$*, or a host failure will occur destroying any volatile state and remaining durable state (after potential recoveries) will reify from* $s *_d j_d$*. The view* (j_v, j_d) *is preserved by every step of* $\mathbb{C}$*.*