Skeletal Semantics and Their Interpretations*

MARTIN BODIN, Imperial College London, United Kingdom
PHILIPPA GARDNER, Imperial College London, United Kingdom
THOMAS JENSEN, Inria, Univ Rennes, IRISA, France
ALAN SCHMITT, Inria, Univ Rennes, IRISA, France

The development of mechanised language specification based on structured operational semantics, with applications to verified compilers and sound program analysis, requires huge effort. General theory and frameworks have been proposed to help with this effort. However, none of this work provides a systematic way of developing concrete and abstract semantics, connected together by a general consistency result. We introduce a *skeletal semantics* of a language, where each skeleton describes the complete semantic behaviour of a language construct. We define a general notion of *interpretation*, which provides a systematic and language-independent way of deriving semantic judgements from the skeletal semantics. We explore four generic interpretations: a simple well-formedness interpretation; a concrete interpretation; an abstract interpretation; and a constraint generator for flow-sensitive analysis. We prove general *consistency results* between interpretations, depending only on simple language-dependent lemmas. We illustrate our ideas using a simple WHILE language.

CCS Concepts: • **Theory of computation → Program semantics**;

Additional Key Words and Phrases: programming language, semantics, abstract interpretation

ACM Reference Format:
Martin Bodin, Philippa Gardner, Thomas Jensen, and Alan Schmitt. 2019. Skeletal Semantics and Their Interpretations. *Proc. ACM Program. Lang.* 3, POPL, Article 44 (January 2019), 31 pages. https://doi.org/10.1145/3290357

1 INTRODUCTION

Plotkin's Structural Operational Semantics [Plotkin 1981] provides a methodology for formally describing a programming language using a collection of inference rules. It has been widely used to provide, for example, mechanised language specifications of substantial parts of ML [Owens 2008], C [Blazy and Leroy 2009; Norrish 1998] and JavaScript [Bodin et al. 2014]. These specifications have, in turn, been used to build verified compilers [Kumar et al. 2014; Leroy 2006] and to develop sound program analysis [Cachera et al. 2005; Jourdan et al. 2015; Klein and Nipkow 2002]. Such language specifications and their applications require huge effort, stretching the fundamental theory and tools to their limits. Researchers have therefore spent considerable thought developing general theories and frameworks where some of this effort can be unified for a wide class of languages.

Abstract interpretation [Cousot and Cousot 1977] is a well-known general theory for analysing programs. It provides general definitions for describing when an abstract semantics is consistent

*This research has been partially supported by the ANR projects AJACS ANR-14-CE28-0008 and CISC ANR-17-CE25-0014-01. Bodin and Gardner were partially supported by the EPSRC programme grant 'REMS: Rigorous Engineering of Mainstream Systems', EP/K008528/1.

Authors' addresses: Martin Bodin, Imperial College London, United Kingdom; Philippa Gardner, Imperial College London, United Kingdom; Thomas Jensen, Inria, Univ Rennes, IRISA, France; Alan Schmitt, Inria, Univ Rennes, IRISA, France.

2475-1421/2019/1-ART44
https://doi.org/10.1145/3290357

(sound) with respect to a concrete semantics, and even suggests a methodology for how to construct consistent abstract semantics from concrete semantics [Cousot 1999; Midtgaard and Jensen 2008; Van Horn and Might 2011]. We focus on abstract semantics arising from concrete operational semantics. A prominent example can be found in the Verasco project [Jourdan et al. 2015] which provides a Coq-certified static analyser based on abstract interpretation, specifically targeting CompCert's mechanised C specification. Schmidt [Schmidt 1995, 1997a] has demonstrated how to build abstract derivations from concrete derivations arising from an operational semantics, illustrating a close connection between the abstract and concrete semantics. The concepts are general, but the work does not attempt to be systematic. Inspired by Schmidt, Bodin *et al.* [Bodin et al. 2015] have identified a general rule format that can be systematically instantiated to both concrete and abstract semantics, with a general consistency result. However, their general rule format is based on a non-standard style of operational semantics, called *pretty-big-step operational semantics* [Charguéraud 2013], introduced to provide a Coq-mechanised specification of JavaScript [Bodin et al. 2014]. It does not provide a general systematic approach for constructing an abstract semantics from a standard operational semantics.

A general framework provides a unifying meta-language for writing operational inference rules, in order to develop general environments for analysis [Harper et al. 1987; Jung et al. 2017; Pfenning and Schürmann 1999; Roşu and Şerbănuţă 2010]. Much of the work on frameworks does not aim to describe abstract analysis. One notable exception is the Iris framework [Jung et al. 2017] for reasoning about concurrent programs. Iris provides a systematic method for building a concurrent program logic from concrete operational semantics, proving a general consistency result. It starts from a concrete operational semantics and generically builds the program logic. Consequently, the general consistency result relies on language-dependent lemmas which require an induction over the possibly complex constructs of the language. It does not work with abstract semantics in general, and the lemmas associated with the general consistency result are difficult to prove.

We introduce a new approach. We have developed a meta-language, which we call a *skeletal semantics*, from which it is possible to construct systematically both concrete and abstract semantics, and prove a general consistency result. Our skeletal semantics comprises:

- *skeletons*, where each skeleton describes the complete behaviour of one language construct;
- generic *interpretations*, which systematically derive semantic judgements from the skeletons: for example, a generic concrete interpretation built using the usual concrete judgements of an operational semantics, parameterised by an input state, command and output state, and a generic abstract interpretation built from more abstract judgements over abstract domains;
- a general *consistency result* between interpretations, which depends on simple language-dependent lemmas.

Our definitions of skeletal semantics and interpretations have been mechanised in the Coq theorem prover, and the consistency result proved.

Skeletal semantics can be used to describe languages specified using big-step operational semantics and languages specified using an English standard such as the ECMAScript standard. In this introductory paper, we focus on a simple While language as the illustrative example; the lambda calculus is given in the Coq artefact. Consider the usual *if* command of a While language, whose behaviour is typically defined in an operational semantics using two standard rules for the true and false case. Instead, in our skeletal semantics, the behaviour of the *if* command is given by one skeleton comprising: a semantic *judgement*, in this case parameterised by input state, expression and value, and instantiated via the interpretations with, for example, the usual concrete and abstract judgements for evaluating expressions; then a *branch* of two paths guarded by *filters* for determining the true and false case, followed by judgements for the appropriate subcommands.

$$\frac{\sigma, e \Downarrow \mathit{true} \qquad \sigma, t_1 \Downarrow x_o}{\sigma, \mathit{if}\ e\ t_1\ t_2 \Downarrow x_o} \qquad\qquad \frac{\sigma, e \Downarrow \mathit{false} \qquad \sigma, t_2 \Downarrow x_o}{\sigma, \mathit{if}\ e\ t_1\ t_2 \Downarrow x_o}$$

Fig. 1. Usual concrete rules for the *if* construct

Our *if* skeleton thus describes the information given in the two normal *if* rules, collected together under one syntactic construct.

Skeletons provide all the information necessary to give systematically both concrete and abstract interpretations. Intuitively, our generic concrete interpretation picks one path from each branching/merging of the skeleton, whereas our generic abstract interpretation merges all the appropriate paths. In fact, our interpretations span many different types of analysis. The paper contains a simple well-formedness interpretation for simple sorts, suggesting that we can give many forms of standard well-formedness result associated with states and types. We also give an interpretation building a constraint generator for flow-sensitive analysis. We discuss other forms of analyses in future work.

We have proved general consistency results between interpretations, which depend on simple language-dependent filter lemmas. These filter lemmas only describe properties of the filters of a language, which are functions on the language values. The complexity of proving these filter lemmas thus only depends on the complexity the filters, which are simple in comparison with the complexity of the whole language. We explore the instantiation of our consistency result for our While language, demonstrating the consistency of the abstract interpretation with respect to the concrete interpretation for a selection of domains, as well as the consistency between the constraint generation and the abstract interpretation.

In summary, we have come a long way to answering the challenge of developing a language-independent framework for relating concrete and abstract semantics. The real test will come when we move from the simple languages explored in this paper to real-world languages such as OCaml and JavaScript, discussed in the future work.

EXAMPLE: THE WHILE LANGUAGE

We demonstrate our skeletal semantics in action using the simple conditional statement from the While language. Consider the usual concrete rules associated with the conditional statement in Figure 1, and the abstract rules in Figure 2, supposing that the Booleans are abstracted by the usual four-valued lattice given by $\{\mathit{true}^{\#}, \mathit{false}^{\#}, \top_{\mathit{bool}}, \bot_{\mathit{bool}}\}$. These abstract rules are intuitively correct, but they are first built in an ad hoc way and then shown to be related to the concrete rules using a Galois connection. More generally, the systematic construction of abstract rules from concrete rules requires a deep understanding of how the analysed programming language evaluates expressions: in a case like a vanilla While language, this is quite straightforward; for a complex language such as JavaScript [Bodin et al. 2014; ECMA 2018; Maffeis et al. 2008], the relationship between the concrete and abstract semantics can be difficult to get right.

We define the skeletal semantics in Section 2, which provides a general meta-theory for defining language semantics. Figure 3 shows the skeleton associated with the *if* construct, with generic subterms denoted by x_{t_1}, x_{t_2}, and x_{t_3}, input state x_σ and output state x_o. Judgements of the form $H(-, x_{t_i}, -)$ identify the required subcomputations associated with the subterms $x_{t_1}, x_{t_2}, x_{t_3}$. The skeleton stitches these judgements together, using the input and output states, the internal symbolic variable x_{f_1}, and the branching which identifies paths through the skeleton using the filters `isTrue` and `isFalse`, resulting in the output state x_o. Such a skeleton thus explicitly describes both the data

$$\frac{\sigma, e \Downarrow \mathit{true}^{\#} \qquad \sigma, t_1 \Downarrow x_o}{\sigma, \mathit{if}\ e\ t_1\ t_2 \Downarrow x_o} \qquad \frac{\sigma, e \Downarrow \mathit{false}^{\#} \qquad \sigma, t_2 \Downarrow x_o}{\sigma, \mathit{if}\ e\ t_1\ t_2 \Downarrow x_o}$$

$$\frac{\sigma, e \Downarrow \top_{\mathit{bool}} \qquad \sigma, t_1 \Downarrow x_o \qquad \sigma, t_2 \Downarrow x_o}{\sigma, \mathit{if}\ e\ t_1\ t_2 \Downarrow x_o} \qquad \frac{\sigma, e \Downarrow \bot_{\mathit{bool}}}{\sigma, \mathit{if}\ e\ t_1\ t_2 \Downarrow \bot}$$

Fig. 2. Usual abstract rules for the *if* construct

$$\textsc{If}\left(\mathit{if}\ x_{t_1}\ x_{t_2}\ x_{t_3}\right) \coloneqq \left[H\left(x_\sigma, x_{t_1}, x_{f_1}\right); \begin{pmatrix} \texttt{isTrue}\left(x_{f_1}\right); H\left(x_\sigma, x_{t_2}, x_o\right) \\ \texttt{isFalse}\left(x_{f_1}\right); H\left(x_\sigma, x_{t_3}, x_o\right) \end{pmatrix}_{\{x_o\}} \right]$$

Fig. 3. Skeleton for the *if* construct

flow and the control flow associated with a language construct, identifying the common pattern underlying the concrete and abstract rules.

We provide a general definition of interpretation for our skeletal semantics in Section 3 and study four generic interpretations:

- A simple well-formedness interpretation (Section 3), which states that the stitching of the skeleton in Figure 3 respects the sorting of the basic constructs.
- The concrete interpretation (Section 4), which intuitively picks one path from each branching of the skeleton, corresponding to the two rules of Figure 1.
- The abstract interpretation (Section 5), whose complex definition (Figure 10) boils down to the intuitive description given by the rule of Figure 4: a rule with optional branches, considering all paths compatible with the return value of the expression e. This rule naturally subsumes the four rules of Figure 2.
- A constraint generator for flow-sensitive static analysis (Section 7). Although these constraints are different in nature to the abstract semantics, they are expressed in our meta-theory using the same mechanism: that is, an interpretation of the skeletal semantics. This provides a strong connection between them.

We also provide general definitions of *consistency* between interpretations (Section 3.2), with general consistency proofs based on filter lemmas (Section 3.3). The shared structure of our different interpretations greatly eases the proof process. We use our consistency definitions to show that the abstract interpretation is correct with respect to the concrete interpretation, and that any solution to the constraints given by our constraint generator must give rise to a correct abstract semantics.

Throughout the paper, we instantiate our definitions and results to the While language as a way of introducing our ideas and demonstrating how classic proof techniques based on an abstract interpretation of While can be captured with our approach (Section 6). We however emphasise that skeletons and interpretations, as well as their consistency proofs, are generic and can be applied to any programming language. To begin to illustrate this, we extend our While language with exceptions, input/output and a heap in Section 8.

The definitions and proofs of Sections 2 to 5 have been formalised in Coq; those of Sections 6 and 7 have been proven on paper. They are all available from the companion website[1].

[1] http://skeletons.inria.fr

$$\frac{\sigma^{\#}, e \Downarrow v^{\#} \qquad \left([\![\texttt{isTrue}]\!]^{\#}\left(v^{\#}\right) \implies \sigma^{\#}, t_1 \Downarrow \sigma_o^{\#}\right) \qquad \left([\![\texttt{isFalse}]\!]^{\#}\left(v^{\#}\right) \implies \sigma^{\#}, t_2 \Downarrow \sigma_o^{\#}\right)}{\sigma^{\#}, \mathit{if}\ e\ t_1\ t_2 \Downarrow \sigma_o^{\#}}$$

Fig. 4. The abstract interpretation of the *if* construct: intuitive description.

c	Signature
const	$\mathit{lit} \longrightarrow \mathit{expr}$
var	$\mathit{ident} \longrightarrow \mathit{expr}$
$+$	$(\mathit{expr} \times \mathit{expr}) \longrightarrow \mathit{expr}$
$=$	$(\mathit{expr} \times \mathit{expr}) \longrightarrow \mathit{expr}$
$\neg$	$\mathit{expr} \longrightarrow \mathit{expr}$

c	Signature
skip	*stat*
$:=$	$(\mathit{ident} \times \mathit{expr}) \longrightarrow \mathit{stat}$
;	$(\mathit{stat} \times \mathit{stat}) \longrightarrow \mathit{stat}$
if	$(\mathit{expr} \times \mathit{stat} \times \mathit{stat}) \longrightarrow \mathit{stat}$
while	$(\mathit{expr} \times \mathit{stat}) \longrightarrow \mathit{stat}$

Fig. 5. Constructors for While

2 SKELETAL SEMANTICS

2.1 Terms

Terms t of a skeletal semantics are built using base terms, term variables, and constructors. Base terms are left unspecified and correspond to the basic blocks of the syntax, such as literals or program identifiers. They are instantiated by interpretations. We assume a countable set of term variables, ranged over by x_t, and a finite set of constructors, ranged over by c. A term is thus a base term, a term variable, or a constructor applied to terms.

We also assume a countable set of *sorts*, ranged over by s. The sorts are separated into *base sorts*, for base terms, and *program sorts*, for terms built using constructors. Any base term belongs to a single base sort. The *signature* of a constructor c, written $\mathit{sig}(c)$, is of the form $(s_1..s_n) \longrightarrow s$, where n is the arity of c, the s_i for $i = 1..n$ are sorts, and s is a program sort.

Running Example. For the While language, the base sorts are *ident* for the program variables and *lit* for the literals. Program sorts are *expr* for expressions and *stat* for statements. The signature of constructors is given in Figure 5.

Let Γ be a mapping from term variables to sorts. Sorted terms are either base terms, term variables x_t of sort $\Gamma(x_t)$, or a term $c(t_1..t_n)$ of sort s, where c has signature $\mathit{sig}(c) = (s_1..s_n) \longrightarrow s$ and the terms $t_1..t_n$ have the appropriate sort. We write $\mathit{Sort}_\Gamma(t)$ for the sort of t. Let E be a mapping from term variables to terms such that $\forall x_t \in \mathit{dom}(E), \mathit{Sort}_\Gamma(E(x_t)) = \Gamma(x_t)$. We extend it to terms as $E(c(t_1..t_n)) = c(E(t_1)..E(t_n))$ when defined. We write $\mathit{Sort}(t)$ for $\mathit{Sort}_\emptyset(t)$ and $\mathit{Tvar}(t)$ for the set of term variables in t. We say t is *closed* if $\mathit{Tvar}(t) = \emptyset$. In that case, we write $t : s$ for $\mathit{Sort}(t) = s$.

Lemma 2.1. *Let t a term, E an environment mapping term variables to closed terms, and Γ a sorting environment such that $\mathit{Tvar}(t) \subseteq \mathit{dom}(E)$, $\mathit{Tvar}(t) \subseteq \mathit{dom}(\Gamma)$, and for any $x_t \in \mathit{Tvar}(t)$ we have $\Gamma(x_t) = \mathit{Sort}(E(x_t))$. Then we have $\mathit{Sort}_\Gamma(t) = \mathit{Sort}(E(t))$.*

2.2 Skeletons

We assume a countable set of *flow variables*, ranged over by x_f, which are used in the skeleton bodies to hold semantic values (states, intermediate values, ...). Among flow variables, we distinguish two of them: x_σ holds the semantic state at the start of a skeleton, and x_o is supposed to hold the semantic result at the end of a skeleton. We let *skeletal variables*, ranged over by x or y, be the

$$
\begin{aligned}
\textsc{Lit}(\mathit{const}\,(x_t)) &\coloneqq \left[\texttt{litInt}\,(x_t)\ ?\triangleright\ x_{f_1}; \texttt{intVal}\,(x_{f_1})\ ?\triangleright\ x_o\right] \\
\textsc{Var}(\mathit{var}\,(x_t)) &\coloneqq \left[\texttt{read}\,(x_t, x_\sigma)\ ?\triangleright\ x_o\right] \\
\textsc{Add}(x_{t_1} + x_{t_2}) &\coloneqq \begin{bmatrix} H\,(x_\sigma, x_{t_1}, x_{f_1})\,; \texttt{isInt}\,(x_{f_1})\ ?\triangleright\ x_{f_{1'}}; H\,(x_\sigma, x_{t_2}, x_{f_2})\,; \\ \quad \texttt{isInt}\,(x_{f_2})\ ?\triangleright\ x_{f_{2'}}; \texttt{add}\,(x_{f_{1'}}, x_{f_{2'}})\ ?\triangleright\ x_{f_3}; \texttt{intVal}\,(x_{f_3})\ ?\triangleright\ x_o \end{bmatrix} \\
\textsc{Eq}(x_{t_1} = x_{t_2}) &\coloneqq \begin{bmatrix} H\,(x_\sigma, x_{t_1}, x_{f_1})\,; \texttt{isInt}\,(x_{f_1})\ ?\triangleright\ x_{f_{1'}}; H\,(x_\sigma, x_{t_2}, x_{f_2})\,; \\ \quad \texttt{isInt}\,(x_{f_2})\ ?\triangleright\ x_{f_{2'}}; \texttt{eq}\,(x_{f_{1'}}, x_{f_{2'}})\ ?\triangleright\ x_{f_3}; \texttt{boolVal}\,(x_{f_3})\ ?\triangleright\ x_o \end{bmatrix} \\
\textsc{Neg}(\neg x_t) &\coloneqq \left[H\,(x_\sigma, x_t, x_{f_1})\,; \texttt{isBool}\,(x_{f_1})\ ?\triangleright\ x_{f_2}; \texttt{neg}\,(x_{f_2})\ ?\triangleright\ x_{f_3}; \texttt{boolVal}\,(x_{f_3})\ ?\triangleright\ x_o\right] \\
\textsc{Skip}(\mathit{skip}) &\coloneqq \left[\texttt{id}\,(x_\sigma)\ ?\triangleright\ x_o\right] \\
\textsc{Asn}(x_{t_1} \coloneqq x_{t_2}) &\coloneqq \left[H\,(x_\sigma, x_{t_2}, x_{f_1})\,; \texttt{write}\,(x_{t_1}, x_\sigma, x_{f_1})\ ?\triangleright\ x_o\right] \\
\textsc{Seq}(x_{t_1};\ x_{t_2}) &\coloneqq \left[H\,(x_\sigma, x_{t_1}, x_{f_1})\,; H\,(x_{f_1}, x_{t_2}, x_o)\right] \\
\textsc{If}(\mathit{if}\ x_{t_1}\ x_{t_2}\ x_{t_3}) &\coloneqq \begin{bmatrix} H\,(x_\sigma, x_{t_1}, x_{f_1})\,; \texttt{isBool}\,(x_{f_1})\ ?\triangleright\ x_{f_{1'}}; \begin{pmatrix} \texttt{isTrue}\,(x_{f_{1'}})\,; H\,(x_\sigma, x_{t_2}, x_o) \\ \texttt{isFalse}\,(x_{f_{1'}})\,; H\,(x_\sigma, x_{t_3}, x_o) \end{pmatrix}_{\{x_o\}} \end{bmatrix} \\
\textsc{While}(\mathit{while}\ x_{t_1}\ x_{t_2}) &\coloneqq \begin{bmatrix} H\,(x_\sigma, x_{t_1}, x_{f_1})\,; \texttt{isBool}\,(x_{f_1})\ ?\triangleright\ x_{f_{1'}}; \\ \quad \begin{pmatrix} \texttt{isTrue}\,(x_{f_{1'}})\,; H\,(x_\sigma, x_{t_2}, x_{f_2})\,; H\,(x_{f_2}, \mathit{while}\ x_{t_1}\ x_{t_2}, x_o) \\ \texttt{isFalse}\,(x_{f_{1'}})\,; \texttt{id}\,(x_\sigma)\ ?\triangleright\ x_o \end{pmatrix}_{\{x_o\}} \end{bmatrix}
\end{aligned}
$$

Fig. 6. Skeletal semantics for While

union of term variables and flow variables. A *skeleton* has the shape $\textsc{Name}(c(x_{t_1}..x_{t_n})) \coloneqq S$, where Name is the skeleton name, c is a constructor, $x_{t_1}..x_{t_n}$ are term variables, and S is the *skeleton body*:

$$
\begin{aligned}
\textsc{Skeleton Body} \quad & S ::= [] \mid B; S \\
\textsc{Bone} \quad & B ::= H\,(x_{f_1}, t, x_{f_2}) \mid F\,(x_1..x_n)\ ?\triangleright\ (y_1..y_m) \mid (S_1..S_n)_V
\end{aligned}
$$

where $H(-,-,-)$ is the (terminal) hook constructor and F ranges over the set of filter functions.

A skeleton body is a sequence of *bones*. A bone is either a *hook judgement* $H(x_{f_1}, t, x_{f_2})$, built using the constructor $H(-,-,-)$ from an input flow variable x_{f_1}, a term t to be hooked during interpretation, and an output flow variable x_{f_2}; or a *filter* $F(x_1..x_n)\ ?\triangleright\ (y_1..y_m)$ which tests if the values bound to its input skeletal variables $(x_1..x_n)$ satisfy a condition specified by F, and in that case outputs values to be bound to $(y_1..y_m)$; or a set of *branches* $(S_1..S_n)_V$ which represent the different behavioural pathways, where V declares the skeletal variables that are shared and must be defined by all branches.

A filter with no output skeletal variables is simply written $F(x_1..x_n)$. It then acts as a predicate.

Requirement 2.2. We require that there exists exactly one skeleton for any given constructor c.

Running Example. The skeletons of our While example are given in Figure 6. Requirement 2.2 is trivially satisfied.

2.3 Flow Sorts

We extend the sorts with *flow sorts*, that are the sorts of values in interpretations. In our running example, flow sorts are *store* for the variable store, *val* for values, *int* for integers, and *bool* for Booleans. We relate flow sorts to hooks and filters as follows.

f	$fsort(f)$
litInt	$lit \longrightarrow int$
intVal	$int \longrightarrow val$
isInt	$val \longrightarrow int$
add	$(int, int) \longrightarrow int$

f	$fsort(f)$
boolVal	$bool \longrightarrow val$
isBool	$val \longrightarrow bool$
isTrue	$bool \longrightarrow ()$
isFalse	$bool \longrightarrow ()$

f	$fsort(f)$
eq	$(int, int) \longrightarrow bool$
neg	$bool \longrightarrow bool$
read	$(ident, store) \longrightarrow val$
write	$(ident, store, val) \longrightarrow store$
id	$store \longrightarrow store$

Fig. 7. Filter sorts

In a hook $H(x_{f_1}, t, x_{f_2})$, the flow variable x_{f_1} stands for an input state that fits with t, and x_{f_2} stands for a result. Given a program sort s, we define $in(s)$ as its input flow sort and $out(s)$ as its output flow sort. In our running example, the input flow sort of both expressions and statements is *store*. The output flow sort of expressions is *val* and the output flow sort of statements is *store*.

Similarly, a filter $F(x_1..x_n)\ ?\triangleright\ (y_1..y_m)$ is assigned a signature, written $fsort(F)$, of the form $(s_1..s_n) \longrightarrow (s'_1..s'_m)$. We write $()$ for the output sort of a filter if $m = 0$ and omit the enclosing parentheses when n or m is 1. Filter signatures for our running example are given in Figure 7.

We check the consistency of the hook and filters with the skeletons in our well-formedness interpretation, introduced in Section 3.1.

3 INTERPRETATIONS

An *interpretation* I specifies base terms and how to interpret the empty skeleton body, hooks, filters, and branches. It defines a set of *interpretation states*, ranged over by Σ in this section but with specific notations for each interpretation, and a set of *interpretation results*, ranged over by O in this section, as well as the following relations:

- $[\![[]]\!]^I(\Sigma) \Downarrow O$ defining the interpretation of the empty skeleton body;
- $[\![H(x_{f_1}, t, x_{f_2})]\!]^I(\Sigma) \Downarrow \Sigma'$ defining the interpretation of a hook;
- $[\![F(x_1..x_n)\ ?\triangleright\ (y_1..y_m)]\!]^I(\Sigma) \Downarrow \Sigma'$ defining the interpretation of a filter, for each filter F;
- $[\![\bigoplus_n]\!]^I_V(O, \Sigma) \Downarrow \Sigma'$ defining the merging of the interpretation of branches, where O is a partial function from $[1..n]$ to interpretation results, and where V is the set of skeletal variables defined and shared by all branches.

Given a skeleton body S and the relations above, we define the remaining cases for the interpretation of S as follows.

$$\begin{pmatrix} [\![B]\!]^I(\Sigma) \Downarrow \Sigma' \\ [\![S]\!]^I(\Sigma') \Downarrow O \end{pmatrix} \implies [\![B; S]\!]^I(\Sigma) \Downarrow O$$

$$\begin{pmatrix} \forall i \in dom(O).\ [\![S_i]\!]^I(\Sigma) \Downarrow O(i) \\ \left[\!\!\left[\bigoplus_n\right]\!\!\right]^I_V(O, \Sigma) \Downarrow \Sigma' \end{pmatrix} \implies [\![(S_1..S_n)_V]\!]^I_V(\Sigma) \Downarrow \Sigma'$$

Interpretations enable us to define the meaning of skeletons by only specifying the parts that matter. Interpretations apply to any skeletons and are thus independent of the language. The rest of the paper presents different interpretation and their relations.

3.1 Well-Formedness Interpretation

The first interpretation we consider is a *well-formedness* interpretation, to verify that every skeleton is well formed. More precisely, we verify that every skeletal variable used has been first defined, that every variable defined in a skeleton is fresh (with an exception for branches, see below), and that the sorting of filters, hooks, and branches are consistent.

Intuitively, in the hook $H(x_{f_1}, t, x_{f_2})$, flow variable x_{f_1} is *used* and flow variable x_{f_2} is *defined*. Similarly, in the filter $F(x_1..x_n)\ ?\triangleright\ (y_1..y_m)$, skeletal variables $(x_1..x_n)$ are used and skeletal variables $(y_1..y_m)$ are defined. The case for branches $(S_1..S_n)_V$ is a bit more involved. First, each branch S_i *must* define the skeletal variables in V. Second, every variable defined in the whole set of branches must be distinct, with the exception of the variables in V as they have to be defined in every branch. And third, the only variables defined by the branches that may be used in the rest of the skeleton body are those in V.

Assuming for each base sort a set of base terms, pairwise disjoint, we define the well-formedness (WF) interpretation in Figure 8. Its interpretation states and result consist of a pair of a sorting environments Γ, mapping term variables to base and program sorts, and flow variables to flow sorts, and a set $\mathcal{D}$ of skeletal variables that have been defined at that point. In this interpretation, we write $x : s$ to state that the kind of variable and sort match, namely term variables with base or program sorts, and flow variables with flow sorts.

The interpretation for the empty skeleton body is trivial, it simply returns its arguments. The interpretation of a hook $H(x_{f_1}, t, x_{f_2})$ checks that x_{f_1} is in Γ, that every term variable of t is also in Γ, and that variable x_{f_2} is fresh (i.e., not in $\mathcal{D}$). In addition, it checks that the sort for x_{f_1} is what t expects as input sort and that x_{f_2} is latter bound to an output sort of t.

The interpretation for a filter $F(x_1..x_n)\ ?\triangleright\ (y_1..y_m)$ is similar. It ensures that the input skeletal variables $(x_1..x_n)$ are in Γ, that the number and kind of both input and output variables match the signature of F, that the output variables are fresh, that the sort of the input variable corresponds to the input signature of F, and it continues binding the output variables to the output signature of F.

Finally, the interpretation of the merging of branches checks that every branch is well formed, that the variables in V are exactly those shared by the branches (neither less nor more than those), and that the sorting environments returned by the branches all agree when restricted to V. In that case, the returned sorting environment is the concatenation of the input environment and the one shared by the branches. The $n \geq 2$ constraint is to have a more concise way of stating that the variables shared by the branches are exactly those in V. It is not a restriction as an empty set of branches is useless, it prevents the skeleton from being interpreted as offering no pathway, and a singleton set of branches can be inlined.

Let $t = c(t_1..t_n)$ be a closed term such that $Sort(t) = s$, where s is a program sort. There are two ways to assign an output sort to t: directly, as $out(s)$, or using the WF interpretation of the skeleton for c to compute the associated sort x_o. If both coincide, we say the skeleton is *well formed*.

Definition 3.1. A skeleton $\textsc{Name}(c(x_{t_1}..x_{t_n})) := S$ is *well formed* iff for any closed term $t = c(t_1..t_n)$ such that $Sort(t) = s$, we have $[\![S]\!]^{\mathsf{wf}}\ (\Gamma, \mathcal{D}) \Downarrow (\Gamma', \mathcal{D}')$ and $\Gamma'(x_o) = out(s)$, with the initial sorting environment Γ being $\{x_\sigma \mapsto in(s) + x_{t_1} \mapsto Sort(t_1)..x_{t_n} \mapsto Sort(t_n)\}$, and with $\mathcal{D} = dom(\Gamma)$.

In the following we only consider well-formed skeletons. For instance, the skeletons for While are well formed.

3.2 Interpretation Consistency

We now define how to relate interpretations. Given interpretations I_1 and I_2, we assume a relation $OKst(\Sigma_1, \Sigma_2)$ between the interpretation states, and a relation $OKout(O_1, O_2)$ between their results. Intuitively, consistency is the propagation of these relations along interpretations.

$$\Longrightarrow \left[\!\left[[]\right]\!\right]^{\mathsf{wf}}(\Gamma, \mathcal{D}) \Downarrow (\Gamma, \mathcal{D})$$

$$\left(\begin{array}{r} x_{f_1} \in dom(\Gamma) \subseteq \mathcal{D} \\ Tvar(t) \subseteq dom(\Gamma) \\ \Gamma\left(x_{f_1}\right) = in(Sort_\Gamma(t)) \\ x_{f_2} \notin \mathcal{D} \\ \Gamma' = \Gamma + x_{f_2} \mapsto out(Sort_\Gamma(t)) \\ \mathcal{D}' = \mathcal{D} \cup \left\{x_{f_2}\right\} \end{array}\right) \Longrightarrow \left[\!\left[H\left(x_{f_1}, t, x_{f_2}\right)\right]\!\right]^{\mathsf{wf}}(\Gamma, \mathcal{D}) \Downarrow (\Gamma', \mathcal{D}')$$

$$\left(\begin{array}{r} (x_1..x_n) \subseteq dom(\Gamma) \subseteq \mathcal{D} \\ (x_1..x_n) : (\Gamma(x_1)..\Gamma(x_n)) \\ y_1..y_m \text{ are pairwise distinct} \\ (y_1..y_m) \cap \mathcal{D} = \emptyset \\ fsort(F) = (\Gamma(x_1)..\Gamma(x_n)) \rightarrow (s_1..s_m) \\ (y_1..y_m) : (s_1..s_m) \\ \Gamma' = \Gamma + (y_1..y_m) \mapsto (s_1..s_m) \\ \mathcal{D}' = \mathcal{D} \cup (y_1..y_m) \end{array}\right) \Longrightarrow \left[\!\left[F(x_1..x_n)\ ?\triangleright (y_1..y_m)\right]\!\right]^{\mathsf{wf}}(\Gamma, \mathcal{D}) \Downarrow (\Gamma', \mathcal{D}')$$

$$\left(\begin{array}{r} n \geq 2 \\ dom(\Gamma) \subseteq \mathcal{D} \\ \forall i \in [1..n].\mathcal{O}(i) = (\Gamma_i, \mathcal{D}_i) \\ \forall i \in [1..n].dom(\Gamma_i) \subseteq \mathcal{D}_i \\ \forall ij.i \neq j \implies (\mathcal{D}_i \setminus \mathcal{D}) \cap \left(\mathcal{D}_j \setminus \mathcal{D}\right) = V \\ \forall i \in [1..n].\Gamma + \Gamma_i|_V = \Gamma' \\ \mathcal{D}' = \bigcup_{i \in [1..n]} \mathcal{D}_i \end{array}\right) \Longrightarrow \left[\!\left[\bigoplus_n\right]\!\right]^{\mathsf{wf}}_V (\mathcal{O}, (\Gamma, \mathcal{D})) \Downarrow (\Gamma', \mathcal{D}')$$

Fig. 8. WF Interpretation

We define two kinds of consistency: one about where interpretations are defined, i.e, whether they return a result, and one about their results.

Definition 3.2. Interpretation I_1 is *existentially consistent* with interpretation I_2 if for any S, Σ_1, Σ_2, and O_1, such that $OKst(\Sigma_1, \Sigma_2)$ and $[\![S]\!]^{I_1}(\Sigma_1) \Downarrow O_1$, there exists a O_2 such that $[\![S]\!]^{I_2}(\Sigma_2) \Downarrow O_2$ and $OKout(O_1, O_2)$.

Definition 3.3. Interpretations I_1 and I_2 are *universally consistent* if for any S, Σ_1, Σ_2, O_1, and O_2, if $OKst(\Sigma_1, \Sigma_2)$, $[\![S]\!]^{I_1}(\Sigma_1) \Downarrow O_1$ and $[\![S]\!]^{I_2}(\Sigma_2) \Downarrow O_2$, then $OKout(O_1, O_2)$.

3.3 Proving Consistency

Both consistency properties can be stated at the level of the building block of interpretations. Formally, we have the following two lemmas.

Lemma 3.4. *Let I_1 and I_2 be two interpretations, OKst a relation between their input states, and OKout a relation between their output states. If for any Σ_1 and Σ_2 such that $OKst(\Sigma_1, \Sigma_2)$ we have*

(1) $[\![[]]\!]^{I_1}(\Sigma_1) \Downarrow O_1 \implies \exists O_2.\ [\![[]]\!]^{I_2}(\Sigma_2) \Downarrow O_2 \ \wedge\ OKout(O_1, O_2)$
(2) $\left[\!\left[H(x_{f_1}, t, x_{f_2})\right]\!\right]^{I_1}(\Sigma_1) \Downarrow \Sigma_1' \implies \exists \Sigma_2'.\ \left[\!\left[H(x_{f_1}, t, x_{f_2})\right]\!\right]^{I_2}(\Sigma_2) \Downarrow \Sigma_2' \ \wedge\ OKst(\Sigma_1', \Sigma_2')$
(3) $[\![F(x_1..x_n)\ ?\triangleright\ (y_1..y_m)]\!]^{I_1}(\Sigma_1) \Downarrow \Sigma_1' \implies$
$\exists \Sigma_2'.\ [\![F(x_1..x_n)\ ?\triangleright\ (y_1..y_m)]\!]^{I_2}(\Sigma_2) \Downarrow \Sigma_2' \ \wedge\ OKst(\Sigma_1', \Sigma_2')$
(4) $dom(O_1) = dom(O_2) \subseteq \{1..n\} \ \wedge\ \forall i \in dom(O_1).OKout(O_1(i), O_2(i))$
$\wedge\ \left[\!\left[\bigoplus_n\right]\!\right]_V^I(O_1, \Sigma_1) \Downarrow \Sigma_1' \implies \exists \Sigma_2'.\ \left[\!\left[\bigoplus_n\right]\!\right]_V^I(O_2, \Sigma_2) \Downarrow \Sigma_2' \wedge OKst(\Sigma_1', \Sigma_2')$

then I_1 is existentially consistent with I_2.

Lemma 3.5. *Let I_1 and I_2 be two interpretations, and OKst a relation between their input states. If for any Σ_1 and Σ_2 such that $OKst(\Sigma_1, \Sigma_2)$ we have*

(1) $[\![[]]\!]^{I_1}(\Sigma_1) \Downarrow O_1 \ \wedge\ [\![[]]\!]^{I_2}(\Sigma_2) \Downarrow O_2 \implies OKout(O_1, O_2)$
(2) $\left[\!\left[H(x_{f_1}, t, x_{f_2})\right]\!\right]^{I_1}(\Sigma_1) \Downarrow \Sigma_1' \ \wedge\ \left[\!\left[H(x_{f_1}, t, x_{f_2})\right]\!\right]^{I_2}(\Sigma_2) \Downarrow \Sigma_2' \implies OKst(\Sigma_1', \Sigma_2')$
(3) $[\![F(x_1..x_n)\ ?\triangleright\ (y_1..y_m)]\!]^{I_1}(\Sigma_1) \Downarrow \Sigma_1' \wedge [\![F(x_1..x_n)\ ?\triangleright\ (y_1..y_m)]\!]^{I_2}(\Sigma_2) \Downarrow \Sigma_2' \implies OKst(\Sigma_1', \Sigma_2')$
(4) $dom(O_1) \subseteq \{1..n\} \ \wedge dom(O_2) \subseteq \{1..n\} \ \wedge\ \forall i \in dom(O_1) \cap dom(O_2).OKout(O_1(i), O_2(i))$
$\wedge\ \left[\!\left[\bigoplus_n\right]\!\right]_V^I(O_1, \Sigma_1) \Downarrow \Sigma_1' \ \wedge\ \left[\!\left[\bigoplus_n\right]\!\right]_V^I(O_2, \Sigma_2) \Downarrow \Sigma_2' \implies OKst(\Sigma_1', \Sigma_2')$

then I_1 and I_2 are universally consistent.

4 CONCRETE INTERPRETATION

We now define an interpretation used to compute a big-step evaluation semantics in the form of a *triple set*: a set of triples (also called *judgements*) of the form *(state, term, result)*. For each base sort we assume a set of base terms, pairwise disjoint, and for each flow sort a set of *values*. We write $t : s$ to state that base term t has base sort s, and $v : s$ to state that value v has flow sort s.

For each filter $F(x_1..x_n)\ ?\triangleright\ (y_1..y_m)$ such that $fsort(F) = (s_1..s_n) \to (s_1'..s_m')$, we assume an interpretation $[\![F]\!]$ which is a relation between elements of $(s_1..s_n)$ and elements of $(s_1'..s_m')$. We write $[\![F]\!](v_1..v_n) \Downarrow (v_1'..v_m')$ to state it relates $(v_1..v_n)$ to $(v_1'..v_m')$.

The input state of a concrete interpretation is a pair comprising

- an environment Σ mapping term variables to closed terms and flow variables to values,
- a set T of triples of value, closed term, and value, representing already known judgements and used to give meaning to the sub-derivations $H(x_{f_1}, t, x_{f_2})$.

The interpretation result maps term variables to closed terms and flow variables to values.

We define the concrete interpretation in Figure 9. For the empty skeleton body, it simply returns its environment. For a hook $H(x_{f_1}, t, x_{f_2})$, it looks up in the triple set a known computation for $\Sigma(x_{f_1})$ and $\Sigma(t)$ whose result is v, and it continues binding x_{f_2} to v. Note that if the language is non-deterministic, there may be several such values and one is picked. For a filter F, one uses its interpretation with the input $(\Sigma(x_1)..\Sigma(x_n))$. As filter interpretations are relations, there may be several results as well. Finally, to merge branches, the interpretation picks a branch that successfully returned a result and extends its environment accordingly.

Running Example. We instantiate the base sort *ident* with strings and *lit* with integers. We instantiate the flow sort *int* with integers, *bool* with Booleans, *val* with the disjoint union *int* + *bool*, and *store* with a partial function from strings to *val*. The concrete interpretation of the filters are the following partial functions: `litInt` is the identity on integers, `intVal` and `boolVal` inject their arguments in *int* + *bool*, `read`(id, st): applies st to id (since st is a partial function, it may not return a result), `isInt`(v): matches v in the disjoint union *int* + *bool*, returns v if it is in *int*, `add`(i_1, i_2): returns the integer addition of i_1 and i_2, `eq`(i_1, i_2): returns `true` if $i_1 = i_2$, `false` otherwise, `isBool`(v): matches v in the disjoint union *int* + *bool*, returns v if it is in *bool*, `write`(id, st, v): returns the

$$\Longrightarrow [\![[]]\!](\Sigma, T) \Downarrow \Sigma$$

$$\begin{pmatrix}(\Sigma(x_{f_1}), \Sigma(t), v) \in T \\ \Sigma' = \Sigma + x_{f_2} \mapsto v\end{pmatrix} \Longrightarrow [\![H(x_{f_1}, t, x_{f_2})]\!](\Sigma, T) \Downarrow (\Sigma', T)$$

$$\begin{pmatrix}[\![F]\!](\Sigma(x_1)..\Sigma(x_n)) \Downarrow (v_1..v_m) \\ \Sigma' = \Sigma + y_1 \mapsto v_1..y_m \mapsto v_m\end{pmatrix} \Longrightarrow [\![F(x_1..x_n)\ ?\triangleright (y_1..y_m)]\!](\Sigma, T) \Downarrow (\Sigma', T)$$

$$\begin{pmatrix}O(i) = \Sigma_i \\ V \subseteq dom(\Sigma_i) \\ \Sigma' = \Sigma + \Sigma_i|_V\end{pmatrix} \Longrightarrow \left[\!\!\left[\bigoplus_n\right]\!\!\right]_V (O, (\Sigma, T)) \Downarrow (\Sigma', T)$$

Fig. 9. Concrete Interpretation

partial function mapping id to v and any other id' to $st(id')$, $\texttt{id}(st)$: returns st, $\texttt{isTrue}(b)$: returns () if $b = \texttt{true}$, $\texttt{isFalse}(b)$: returns () if $b = \texttt{false}$. In the rest of the paper, we directly write x for $var(\texttt{x})$ and n for $const(n)$ in the examples.

4.1 Consistency of WF and Concrete Interpretations

Definition 4.1. We say a triple set T is *well formed* if all its elements are well formed, i.e., if $(\sigma, t, v) \in T$, then $t = c(t_1..t_n)$ and there is a sort s such that $Sort(t) = s$, $\sigma : in(s)$, and $v : out(s)$.

We define $OKst((\Gamma, \mathcal{D}), (\Sigma, T))$ as follows: T is well-formed, $dom(\Gamma) = dom(\Sigma)$, and for any $x \in dom(\Gamma)$ we have $\Sigma(x) : \Gamma(x)$. We define $OKout((\Gamma, \mathcal{D}), \Sigma)$ as follows: $dom(\Gamma) = dom(\Sigma)$ and for any $x \in dom(\Gamma)$ we have $\Sigma(x) : \Gamma(x)$.

Lemma 4.2. *The well-formedness and concrete interpretations are universally consistent.*

4.2 Concrete Derivations

The concrete interpretation describes how skeletons can be interpreted from a set of hooks. The *immediate consequence* $\mathcal{H}$ describes how skeletons can be assembled. It starts from a set of well-formed triples (that is, of Hoare triples) T, and derives a new set of judgements using the concrete interpretation. Intuitively, from the set of triples generated by derivations of depth at most n, it builds the set of triples generated by derivations of depth at most $n + 1$. It is defined as follows.

$$\mathcal{H}(T) = \left\{ (\sigma, t, v) \;\middle|\; \begin{array}{c} t = c(t_1..t_n) \wedge Sort(t) = s \\ \textsc{Name}(c(x_{t_1}..x_{t_n})) := S \in Rules \\ \sigma : in(s) \\ \Sigma = x_\sigma \mapsto \sigma + x_{t_1} \mapsto t_1..x_{t_n} \mapsto t_n \\ [\![S]\!](\Sigma, T) \Downarrow \Sigma' \\ \Sigma'(x_o) = v \end{array} \right\}$$

Lemma 4.3. *The functional $\mathcal{H}$ is monotone.*

Proof. This is immediate by inspecting the interpretation of skeletal bodies, as the only one where T is used is for hooks, and a bigger T does not remove results. □

Lemma 4.4. *If T is a well-formed triple set, then $\mathcal{H}(T)$ is a well-formed triple set.*

We now show that the smallest fixpoint of $\mathcal{H}$ corresponds to the set of triples generated by any finite derivation, or in other words, an inductive definition of the concrete rules.

Lemma 4.5. *$\mathcal{H}$ is continuous: for any increasing sequence of triple sets (T_i), we have $\bigcup_i \mathcal{H}(T_i) = \mathcal{H}(\bigcup_i T_i)$.*

Proof. We prove this result by double inclusion. The inclusion $\bigcup_i \mathcal{H}(T_i) \subseteq \mathcal{H}(\bigcup_i T_i)$ follows from the monotony of $\mathcal{H}$. To show that $\mathcal{H}(\bigcup_i T_i) \subseteq \bigcup_i \mathcal{H}(T_i)$, we show that for all triple set T and $(\sigma, t, o) \in \mathcal{H}(T)$, there exists a finite subset T' of T such that $(\sigma, t, o) \in \mathcal{H}(T')$. This result is immediate by induction over the structure of the skeleton S. Then, for each $(\sigma, t, o) \in \mathcal{H}(\bigcup_i T_i)$, there exists a finite subset T' of $\bigcup_i T_i$ such that $(\sigma, t, o) \in \mathcal{H}(T')$. As T' is finite and (T_i) monotone, there exists n such that $T' \subseteq T_i$. We conclude by monotonicity of $\mathcal{H}$. □

Definition 4.6. The concrete semantics $\Downarrow$ is the smallest fixpoint of $\mathcal{H}$.

Lemma 4.7. *We have $\Downarrow = \bigcup_n \mathcal{H}^n(\emptyset)$.*

Proof. The set of triple sets ordered by inclusion is a CPO, and $\mathcal{H}$ is continuous on this CPO. We conclude by Kleene fixpoint theorem. □

Lemma 4.8. *The concrete semantics $\Downarrow$ is well-formed.*

Proof. Let $(\sigma, t, v) \in \Downarrow$. By Lemma 4.7, there exists a finite number n such that $(\sigma, t, v) \in \mathcal{H}^n(\emptyset)$. We prove by induction on n that (σ, t, v) has the expected properties. It is immediate for 0, and for $n + 1$ we simply apply lemma 4.4. □

5 ABSTRACT INTERPRETATION

This section describes how a set of skeletons defining a programming language can be re-interpreted over an abstract domain of properties to obtain an abstract interpretation of the language.

5.1 Abstract Domains

An abstract interpretation of a set of skeletons must define abstract domains for all the terms and flow sorts used in the skeleton bodies, ending with abstract semantic states and abstract results.

Elements in the abstract domains represent sets of values in the corresponding concrete domain (they are related through the concretion function γ introduced below). The abstract interpretation framework is designed to be parametric in the choice of abstract domains for base values such as integers, Booleans, and program states. All we require is that each abstract domain for sort s is a partial order $\sqsubseteq$ with a least element, denoted $\bot_s$, representing the empty set. For example, the lattice of intervals can be used as an abstract domains for integers, with $\bot_{int}$ being the empty interval. Similarly, a state that maps program variables to integer values can be abstracted as a mapping from variables to intervals, or as a polyhedron that defines linear relations between program variables.

Skeletal variables can also range over terms. For each program or base sort s, we define an abstract domain by imposing a flat partial order on the set of terms of that sort (*i.e.*, we relate a term to itself and no other term) and by adding a $\bot_s$ element, smaller than all terms of that sort. Abstract base terms include every concrete base term, they may also include additional terms that denote sets of concrete base terms. To ease notation, we sometimes omit the sort in $\bot$ in an equality. In this case, $v^\# = \bot$ should be read $v^\# = \bot_{Sort(v^\#)}$ and, $v^\# \neq \bot$ should be read $v^\# \neq \bot_{Sort(v^\#)}$.

5.2 Abstract Interpretation of Skeletons

In addition to the abstract domains, an abstract interpretation must specify its input and output states, and how the empty skeleton body, hooks, filters, and the merging of branches are interpreted.

$$\implies [\![[]]\!]^{\#}\left(f, \Sigma^{\#}, T^{\#}\right) \Downarrow \left(f, \Sigma^{\#}\right)$$

$$\Sigma^{\#\prime} = \Sigma^{\#} + x_{f_2} \mapsto \bot_{out(Sort(\Sigma^{\#}(t)))} \implies [\![H\left(x_{f_1}, t, x_{f_2}\right)]\!]^{\#}\left(\bot, \Sigma^{\#}, T^{\#}\right) \Downarrow \left(\bot, \Sigma^{\#\prime}, T^{\#}\right)$$

$$\begin{pmatrix} \Sigma^{\#}\left(x_{f_1}\right) \sqsubseteq \sigma^{\#} \\ \Sigma^{\#}\left(t\right) \sqsubseteq t^{\#} \\ \left(\sigma^{\#}, t^{\#}, \bot\right) \in T^{\#} \\ \Sigma^{\#\prime} = \Sigma^{\#} + x_{f_2} \mapsto \bot_{out(Sort(\Sigma^{\#}(t)))} \end{pmatrix} \implies [\![H\left(x_{f_1}, t, x_{f_2}\right)]\!]^{\#}\left(\top, \Sigma^{\#}, T^{\#}\right) \Downarrow \left(\bot, \Sigma^{\#\prime}, T^{\#}\right)$$

$$\begin{pmatrix} \Sigma^{\#}\left(x_{f_1}\right) \sqsubseteq \sigma^{\#} \\ \Sigma^{\#}\left(t\right) \sqsubseteq t^{\#} \\ \left(\sigma^{\#}, t^{\#}, v^{\#}\right) \in T^{\#} \\ v^{\#} \sqsubseteq v^{\#\prime} \\ \Sigma^{\#\prime} = \Sigma^{\#} + x_{f_2} \mapsto v^{\#\prime} \end{pmatrix} \implies [\![H\left(x_{f_1}, t, x_{f_2}\right)]\!]^{\#}\left(\top, \Sigma^{\#}, T^{\#}\right) \Downarrow \left(\top, \Sigma^{\#\prime}, T^{\#}\right)$$

$$\begin{pmatrix} fsort\left(F\right) = \left(s_1^{\prime}..s_n^{\prime}\right) \rightarrow \left(s_1..s_m\right) \\ \Sigma^{\#\prime} = \Sigma^{\#} + y_1 \mapsto \bot_{s_1}..y_m \mapsto \bot_{s_m} \end{pmatrix} \implies [\![F\left(x_1..x_n\right) ?\triangleright \left(y_1..y_m\right)]\!]^{\#}\left(\bot, \Sigma^{\#}, T^{\#}\right) \Downarrow \left(\bot, \Sigma^{\#\prime}, T^{\#}\right)$$

$$\begin{pmatrix} \left(\Sigma^{\#}\left(x_1\right)..\Sigma^{\#}\left(x_n\right)\right) \sqsubseteq \left(v_1^{\#}..v_n^{\#}\right) \\ [\![F]\!]^{\#}\left(v_1^{\#}..v_n^{\#}\right) = \bot \\ fsort\left(F\right) = \left(s_1^{\prime}..s_n^{\prime}\right) \rightarrow \left(s_1..s_m\right) \\ \Sigma^{\#\prime} = \Sigma^{\#} + y_1 \mapsto \bot_{s_1}..y_m \mapsto \bot_{s_m} \end{pmatrix} \implies [\![F\left(x_1..x_n\right) ?\triangleright \left(y_1..y_m\right)]\!]^{\#}\left(\top, \Sigma^{\#}, T^{\#}\right) \Downarrow \left(\bot, \Sigma^{\#\prime}, T^{\#}\right)$$

$$\begin{pmatrix} \left(\Sigma^{\#}\left(x_1\right)..\Sigma^{\#}\left(x_n\right)\right) \sqsubseteq \left(v_1^{\#}..v_n^{\#}\right) \\ [\![F]\!]^{\#}\left(v_1^{\#}..v_n^{\#}\right) \sqsubseteq \left(v_1^{\#\prime}..v_m^{\#\prime}\right) \\ \Sigma^{\#\prime} = \Sigma^{\#} + y_1 \mapsto v_1^{\#\prime}..y_m \mapsto v_m^{\#\prime} \end{pmatrix} \implies [\![F\left(x_1..x_n\right) ?\triangleright \left(y_1..y_m\right)]\!]^{\#}\left(\top, \Sigma^{\#}, T^{\#}\right) \Downarrow \left(\top, \Sigma^{\#\prime}, T^{\#}\right)$$

$$\begin{pmatrix} n \geq 1 \\ \forall i \in [1..n].O\left(i\right) = \left(\bot, \Sigma_i^{\#}\right) \\ \forall i \in [1..n].V \subseteq dom\left(\Sigma_i^{\#}\right) \\ \forall i, j \in [1..n].\ \Sigma_i^{\#}|_V = \Sigma_j^{\#}|_V \\ \Sigma^{\#\prime} = \Sigma^{\#} + \Sigma_1^{\#}|_V \end{pmatrix} \implies \left[\!\!\left[\bigoplus_n \right]\!\!\right]_V^{\#}\left(f, O, \left(\Sigma^{\#}, T^{\#}\right)\right) \Downarrow \left(\bot, \Sigma^{\#\prime}, T^{\#}\right)$$

$$\begin{pmatrix} dom\left(O\right) = [1..n] \\ \mathcal{E} = \left\{\Sigma_i^{\#} \middle| O\left(i\right) = \left(\top, \Sigma_i^{\#}\right)\right\} \neq \emptyset \\ \forall \Sigma_i^{\#} \in \mathcal{E}.V \subseteq dom\left(\Sigma_i^{\#}\right) \\ \Sigma_i^{\#} \in \mathcal{E} \implies \Sigma^{\#\prime} = \Sigma^{\#} + \Sigma_i^{\#}|_V \end{pmatrix} \implies \left[\!\!\left[\bigoplus_n \right]\!\!\right]_V^{\#}\left(\top, O, \left(\Sigma^{\#}, T^{\#}\right)\right) \Downarrow \left(\top, \Sigma^{\#\prime}, T^{\#}\right)$$

Fig. 10. Abstract Interpretation

For each filter symbol F of signature $(s_1..s_n) \rightarrow (s_1^{\prime}..s_m^{\prime})$ we assume a total function $[\![F]\!]^{\#}$ from the domain corresponding to $(s_1..s_n)$ to the domain corresponding to $(s_1^{\prime}..s_m^{\prime})$. A filter interpretation may return $\bot$ to state it is not defined for that input.

The input state of an abstract interpretation is a triple $(f, \Sigma^\#, T^\#)$ comprising a *flag* f, an abstract environment $\Sigma^\#$ (mapping skeletal variables to abstract terms and values), and a set of abstract semantic triples $T^\#$ that gives semantics to hooks. A flag is either $\bot$ or $\top$ and it indicates whether it has been determined that the current skeleton does not apply ($\bot$) or that it may still apply ($\top$). The output state of an abstract interpretation is a flag and an abstract environment where skeletal variables hold the result of the abstract interpretation. Figure 10 defines the abstract semantics.

The abstract interpretation of an empty list of hypotheses just returns the flag and environment from its input. There are three cases for the interpretation of a hook $H(x_{f_1}, t, x_{f_2})$. If we have determined that the skeleton does not apply, we set x_{f_2} to $\bot$ of the correct sort. In the two other cases, we need to have a triple $(\sigma^\#, t^\#, v^\#)$ from $T^\#$ such that $\Sigma^\#(x_{f_2}) \sqsubseteq \sigma^\#$ and $\Sigma^\#(t) \sqsubseteq t^\#$. This loss of precision gives some flexibility for such a derivation. We then have two (non exclusive) cases: if $v^\# = \bot$, then we know the skeleton does not apply, and set the flag to $\bot$ and x_{f_2} to the appropriate $\bot$. For the last case, we do not restrict what $v^\#$ is (it may still be $\bot$), and we bind in the resulting environment x_{f_2} to some $v^{\#\prime}$ that may be less precise than $v^\#$, again to gain flexibility.

The abstract interpretation of a filter $F(x_1..x_n)\ ?\triangleright\ (y_1..y_m)$ also has three cases. If we know the skeleton does not apply, we just bind the output variables $(y_1..y_m)$ to the appropriate $\bot$ depending on the signature of F. Otherwise, we apply the filter interpretation to an approximation of the arguments as given by the environment. If the result is $\bot$, we know the skeleton does not apply and switch the flag to $\bot$, as well as extend the environment with $\bot$ of the correct sort. Otherwise, we keep the flag as $\top$ and extend the environment to an approximation of the result of the filter.

For the merging operator, we interpret every possible branch and collect their results in O. If all branching have the $\bot$ flag (either because the $\bot$ flag was set before their interpretation, which would then be propagated, or because they newly returned it), then the skeleton does not apply and we set the flag accordingly, extending the environment with mappings from the shared skeletal variables V to $\bot$ of the correct sort. Otherwise, we collect all branches that have a $\top$ flag. They must all return abstract environments that agree on the shared variables (which is why the approximations in the filter and hook cases are useful, to ensure this is possible), and we extend the current environment with this common environment.

The key difference between the abstract and concrete interpretations is how the different results are merged in case of branching. The concrete semantics picks one of them, whereas the abstract semantics requires all branches that provided a result to agree. This is because the goal of the abstract semantics is to infer abstract semantic triples that are valid statements about all possible resulting states, i.e., about all possible concrete choices in case of branching.

5.3 Consistency of WF and Abstract Interpretations

We define $\mathit{OKst}((\Gamma, \mathcal{D}), (f, \Sigma^\#, T^\#))$ as follows: $T^\#$ is well formed, $\mathit{dom}(\Gamma) = \mathit{dom}(\Sigma^\#)$, and for any $x \in \mathit{dom}(\Gamma)$ we have $\Sigma^\#(x) : \Gamma(x)$. We define $\mathit{OKout}((\Gamma, \mathcal{D}), (f, \Sigma^\#))$ as follows: $\mathit{dom}(\Gamma) = \mathit{dom}(\Sigma^\#)$ and for any $x \in \mathit{dom}(\Gamma)$ we have $\Sigma^\#(x) : \Gamma(x)$.

Lemma 5.1. *The well-formedness and abstract interpretations are universally consistent.*

5.4 Abstract Derivations

We define the abstract immediate consequence operator from well-formed triple sets to triple sets in Figure 11. As in the concrete case, the immediate consequence describes how to assemble skeletons.

Lemma 5.2. *The functional $\mathcal{H}^\#$ is monotonic.*

Proof. This is immediate by inspecting the interpretation of skeletal bodies, as the only one where $T^\#$ is used is for hooks, and a bigger $T^\#$ does not remove results. □

$$\mathcal{H}^{\#}\left(T^{\#}\right) = \left\{ (\sigma^{\#}, t^{\#}, v^{\#}) \;\middle|\; \begin{array}{c} t^{\#} = c\left(t_1^{\#}..t_n^{\#}\right) \wedge \mathit{Sort}\left(t^{\#}\right) = s \\ \textsc{Name}\left(c\left(x_{t_1}..x_{t_n}\right)\right) := S \in \mathit{Rules} \\ \sigma^{\#} : \mathit{in}(s) \\ \Sigma^{\#} = x_\sigma \mapsto \sigma^{\#} + x_{t_1} \mapsto t_1^{\#}..x_{t_n} \mapsto t_n^{\#} \\ [\![S]\!]^{\#}\left(\top, \Sigma^{\#}, T^{\#}\right) \Downarrow \left(f, \Sigma^{\#\prime}\right) \\ \Sigma^{\#\prime}\left(x_o\right) = v^{\#} \end{array} \right\}$$

Fig. 11. The abstract immediate consequence operator

Lemma 5.3. *If $T^{\#}$ is a well-formed triple set, then $\mathcal{H}^{\#}(T^{\#})$ is a well-formed triple set.*

An abstract semantics $\Downarrow^{\#}$ is a set of facts of the form $(\sigma^{\#}, t^{\#}, v^{\#})$ stating that from state $\sigma^{\#}$ term $t^{\#}$ evaluates to $v^{\#}$. A *correct* abstract semantics is one where such triples correspond to triples in the concrete semantics (see Section 5.5). The more facts an abstract semantics contains, the more useful it is, as it provides more information about the behaviour of terms. Hence, we choose as abstract semantics the one with most facts, i.e., the greatest fixpoint of $\mathcal{H}^{\#}$. This choice provides a proof technique: since the greatest fixpoint is the union of all sets such that $T^{\#} \subseteq \mathcal{H}^{\#}(T^{\#})$, to prove that a fact is correct, one can propose a candidate set $T^{\#}$ containing this fact, and then show that $T^{\#} \subseteq \mathcal{H}^{\#}(T^{\#})$. This amounts to proving that the facts $T^{\#}$ constitute an invariant of the semantics. If we were to translate such invariants into a derivation, the resulting derivation may be infinite.[2]

We could also define the abstract semantics $\Downarrow^{\#}$ as the smallest fixpoint of $\mathcal{H}^{\#}$. This would be sound but, having fewer facts, we would then miss valuable abstract results. More precisely, if a triple $(\sigma^{\#}, t, v^{\#})$ belongs to the smallest fixpoint of $\mathcal{H}^{\#}$, then (as abstract triple sets form a CPO ordered by inclusion and $\mathcal{H}^{\#}$ is continuous on this CPO), there exists a finite number n such that $(\sigma^{\#}, t, v^{\#}) \in \mathcal{H}^{\#n}(\emptyset)$. In other words, there exists a finite abstract derivation yielding the triple $(\sigma^{\#}, t, v^{\#})$. This implies that for all concrete state $\sigma \in \gamma(\sigma^{\#})$, the program t terminates. We would thus have lost all facts for which the abstract semantics cannot prove termination. Defining the abstract semantics as the greatest fixpoint of $\mathcal{H}^{\#}$ solves this issue.

Definition 5.4. The abstract semantics $\Downarrow^{\#}$ is the largest fixpoint of $\mathcal{H}^{\#}$ as a function from well-formed triple sets to well-formed triple sets. This restriction is well-defined by Lemma 5.3.

Lemma 5.5. *$\Downarrow^{\#}$ is well formed.*

Proof. As $\Downarrow^{\#}$ is the largest fixpoint of $\mathcal{H}^{\#}$, it is the union of all well-formed triple sets $T^{\#}$ such that $T^{\#} \subseteq \mathcal{H}^{\#}(T^{\#})$. Let $(\sigma^{\#}, t^{\#}, v^{\#}) \in \Downarrow^{\#}$, there is $T^{\#} \subseteq \mathcal{H}^{\#}(T^{\#})$ where $(\sigma^{\#}, t^{\#}, v^{\#}) \in T^{\#}$ and $T^{\#}$ is well formed. Hence $(\sigma^{\#}, t^{\#}, v^{\#})$ has the requested properties. □

5.5 Consistency of Concrete and Abstract Interpretations

We assume a *concretion function* γ for the abstract domain, from abstract terms to sets of concrete terms, and from abstract values to sets of concrete values. We impose several constraints on γ. First, γ must be compatible with $\sqsubseteq$: if $t \in \gamma(t^{\#})$ and $t^{\#} \sqsubseteq t^{\#\prime}$, then $t \in \gamma(t^{\#\prime})$, and if $v \in \gamma(v^{\#})$ and $v^{\#} \sqsubseteq v^{\#\prime}$, then $v \in \gamma(v^{\#\prime})$. Second, for any abstract term $t^{\#}$ of sort s, the set $\gamma(t^{\#})$ must only contain terms of sort s. In addition, $\gamma(c(t_1^{\#}..t_n^{\#})) = \left\{c(t_1..t_n) \middle| t_i \in \gamma(t_i^{\#})\right\}$. Conversely, for any concrete term t, we have $\gamma(t) = \{t\}$, as abstract base terms are extensions of concrete base terms.

[2] See the Figure 10 of [Schmidt 1997a] for an example of such representation.

Lemma 5.6. *Let Σ be a mapping from term variables to concrete terms, and $\Sigma^\#$ be a mapping from term variables to abstract terms. If $Tvar(t) \subseteq dom(\Sigma)$, $Tvar(t) \subseteq dom(\Sigma^\#)$, and $\forall x_t \in Tvar(t), \Sigma(x_t) \in \gamma(\Sigma^\#(x_t))$, then $\Sigma(t) \in \gamma(\Sigma^\#(t))$.*

Proof. By induction on the structure of t. If it is a base term, then the result holds by hypothesis on base terms, if it is a term variable, then the result is immediate, and otherwise we prove the property by induction on the subterms. □

Regarding values, we have similar restrictions: for any abstract value $v^\#$ of sort s, the concrete values in $\gamma(v^\#)$ all have sort s. We also require the abstract interpretation of filters to be consistent with the concrete one: if $[\![F]\!]\,(v_1..v_n) \Downarrow (v'_1..v'_m)$ and $\forall i \in [1..n].v_i \in \gamma(v_i^\#)$, then $[\![F]\!]^\#\,(v_1^\#..v_n^\#) = (v_1^{\#\prime}..v_m^{\#\prime})$ and $\forall i \in [1..n].v'_i \in \gamma(v_i^{\#\prime})$. In particular, if the concrete filter relates its input to an output, the abstract filter cannot return $\bot$.

Definition 5.7. Let T a concrete triple set and $T^\#$ an abstract triple set. We say they are *consistent* if for any $(\sigma, t, v) \in T$ and $(\sigma^\#, t^\#, v^\#) \in T^\#$, if $\sigma \in \gamma(\sigma^\#)$ and $t \in \gamma(t^\#)$, then $v \in \gamma(v^\#)$.

We define $OKst((\Sigma, T), (f, \Sigma^\#, T^\#))$ as follows: $f = \top$, $dom(\Sigma) = dom(\Sigma^\#)$, for any $x \in dom(\Sigma)$, we have $\Sigma(x) \in \gamma(\Sigma^\#(x))$, and T and $T^\#$ are well formed and consistent. We define $OKout(\Sigma, (f, \Sigma^\#))$ as follows: $f = \top$, $dom(\Sigma) = dom(\Sigma^\#)$, and for any $x \in dom(\Sigma)$, we have $\Sigma(x) \in \gamma(\Sigma^\#(x))$.

Lemma 5.8. *The concrete and abstract interpretations are universally consistent.*

Lemma 5.9. *Let T and $T^\#$ well formed and consistent triple sets, then $\mathcal{H}(T)$ and $\mathcal{H}^\#(T^\#)$ are well formed and consistent triple sets.*

We finally show that the abstract semantics is correct relative to the concrete semantics. In a nutshell, for any triple in the concrete semantics $\Downarrow$ (the smallest fixpoint of $\mathcal{H}$) and any triple in the abstract semantics $\Downarrow^\#$ (the largest fixpoint of $\mathcal{H}^\#$), if the input states and terms are related, then the output values are related. Formally, we have the following.

Definition 5.10. An abstract triple set $T^\#$ is *correct* if it is well-formed and consistent with $\Downarrow$.

Theorem 5.11. *$\Downarrow^\#$ is correct.*

Proof. We prove by induction on k that $\mathcal{H}^k(\emptyset)$ and $\Downarrow^\#$ are well formed and consistent. The check that $\Downarrow^\#$ is well formed is simply Lemma 5.5.

The result is immediate for $k = 0$ since $\emptyset$ is well formed, and there is nothing else to check.

Let $k = n + 1$, by induction we have $\mathcal{H}^n(\emptyset)$ and $\Downarrow^\#$ are well formed and consistent. By Lemma 4.4 we have $\mathcal{H}^{n+1}(\emptyset)$ and $\mathcal{H}^\#(\Downarrow^\#) = \Downarrow^\#$ are well formed and consistent, as required.

To conclude, we apply Lemma 4.7. □

Note that in the previous theorem we only use the fact that $\Downarrow^\#$ is a fixpoint: it does not have to be the greatest fixpoint.

5.6 Example: Interval analysis of While

To give a concrete example of an abstract interpretation we design a value analysis of the language While in the style of Schmidt's Abstract Interpretation of Natural Semantics [Schmidt 1995]. We have the following flow sorts in the semantic definition (cf. Figure 7): *int*, *bool*, *val*, and *store*.

We describe an analysis in which integers are approximated by intervals, ordered by inclusion. Writing $[n, m]$ for the interval of integers between n and m (with the convention that $[n, m] = \emptyset$ if $m < n$), we can define the abstract domains for each of the flow sort as follows:

$$int^\# = ([n, m] : n \in \mathbf{Z} \cup \{-\infty\} \wedge m \in \mathbf{Z} \cup \{+\infty\}) \qquad val^\# = int^\# \times bool^\#$$

$$bool^\# = \{\bot_{bool}, true^\#, false^\#, \top_{bool}\} \qquad store^\# = ident^\# \longrightarrow val^\#$$

$$[\![\texttt{litInt}]\!]^{\#} = \lambda(n).[n,n] \qquad [\![\texttt{id}]\!]^{\#} = \lambda\sigma^{\#}.\ \sigma^{\#}$$

$$[\![\texttt{intVal}]\!]^{\#} = \lambda i.\ (i, \bot_{bool}) \qquad [\![\texttt{boolVal}]\!]^{\#} = \lambda b.\ (\bot_{int}, b)$$

$$[\![\texttt{isInt}]\!]^{\#} = \lambda(i,b).\ i \qquad [\![\texttt{isBool}]\!]^{\#} = \lambda(i,b).\ b$$

$$[\![\texttt{add}]\!]^{\#} = \lambda([l_1,u_1],[l_2,u_2]).[l_1+l_2, u_1+u_2]$$

$$[\![\texttt{eq}]\!]^{\#} = \lambda(i_1,i_2).\ \begin{cases} true^{\#} & \text{if } i_1 = [n,n] = i_2 \\ false^{\#} & \text{if } i_1 \cap i_2 = \emptyset \\ \top_{bool} & \text{otherwise} \end{cases} \qquad [\![\texttt{neg}]\!]^{\#} = \lambda b.\ \begin{cases} false^{\#} & \text{if } b = true^{\#} \\ true^{\#} & \text{if } b = false^{\#} \\ b & \text{otherwise} \end{cases}$$

$$[\![\texttt{read}]\!]^{\#} = \lambda\left(\sigma^{\#}, \mathsf{x}\right).\ \sigma^{\#}(\mathsf{x}) \qquad [\![\texttt{write}]\!]^{\#} = \lambda(\mathsf{x}, \sigma^{\#}, v^{\#}).\ \sigma^{\#}\left[\mathsf{x} \leftarrow v^{\#}\right]$$

$$[\![\texttt{isTrue}]\!]^{\#} = \lambda b.\ \begin{cases} \bot & \text{if } b \in \left\{\bot_{bool}, false^{\#}\right\} \\ () & \text{otherwise} \end{cases} \qquad [\![\texttt{isFalse}]\!]^{\#} = \lambda b.\ \begin{cases} \bot & \text{if } b \in \left\{\bot_{bool}, true^{\#}\right\} \\ () & \text{otherwise} \end{cases}$$

Fig. 12. Abstract interpretation of filters

Abstract base terms are concrete base terms. We abstract identifiers by themselves, $ident^{\#} = ident$, with only the trivial (reflexive) ordering. The abstract domain of Booleans is (isomorphic to) the set of subsets of Booleans, ordered by inclusion. The abstract domain of values is the defined as the Cartesian product, ordered component-wise, of the abstract domain of integers and Booleans, where each component gives an approximation of the concrete value, *provided* that the value is of the corresponding sort. Stores are mappings from identifiers to values, ordered pointwise. Undefined identifiers are mapped to the undefined value $\bot_{val^{\#}}$. The concretisation function γ from abstract domains to concrete domains formalises the relation between concrete and abstract values.

$$\gamma([n,m]) = \{i \mid n \le i \le m\} \qquad \gamma(i,b) = \gamma(i) \cup \gamma(b) \qquad \gamma(\top_{bool}) = \{true, false\}$$

$$\gamma(\bot_{bool}) = \emptyset \qquad \gamma\left(true^{\#}\right) = \{true\} \qquad \gamma\left(false^{\#}\right) = \{false\}$$

$$\gamma\left(\sigma^{\#}\right) = \left\{\Sigma \,\middle|\, \forall \mathsf{x}.\Sigma(\mathsf{x}) \in \gamma\left(\sigma^{\#}(\mathsf{x})\right)\right\}$$

The abstraction of the basic filters used in the definition of While is given in Figure 12. Notice that the abstract interpretation of the filters `isInt` and `isBool` return an abstract integer and an abstract Boolean, respectively, instead of a Boolean stating whether their argument can be an integer and a Boolean. This is correct because an abstract value that is only an integer has the shape $(i, \bot_{bool})$, and applying `isBool` to it returns $\bot_{bool}$, indicating it contains no Boolean.

Lemma 5.12. *The abstract filters are consistent with the concrete filters.*

Lemma 5.13. *The abstract semantics of While is correct.*

6 DERIVING PROOF TECHNIQUES FROM AN ABSTRACT SEMANTICS

This section presents several proof techniques derived from an abstract semantics and instantiated in our While language.

6.1 Abstract rules for analysing While

Given the instantiation of the filters used in the abstract semantic of While, we can now derive an abstract interpretation of While programs. The result of an abstract interpretation of a program is a set of abstract triples that correctly describes the program behaviour. We shall present the analysis through a set of syntax-directed inference rules for inferring such triples. For a given term

$c(t_1..t_n)$, we take the corresponding skeleton $\textsc{Name}(c(x_{t_1}..x_{t_n})) \coloneqq S$ in the semantics and apply the general abstract interpretation to the skeleton body S. This results in a series of conditions for a triple to be valid that will form the hypotheses of the inference rules.

Rule for addition. As a first example, we derive a rule for analysing arithmetic expressions such as $t_1 + t_2$. A triple $(\sigma^\#, t_1 + t_2, v^\#)$ is valid if it belongs to a fixpoint $T^\#$ of $\mathcal{H}^\#$. Unfolding definitions,

$$\begin{aligned}
& \left(\sigma^\#, t_1 + t_2, v^\#\right) \in T^\# = \mathcal{H}^\#\left(T^\#\right) \\
\Leftrightarrow \quad & \left[\!\!\left[\begin{array}{l} H\left(x_\sigma, x_{t_1}, x_{f_1}\right); \texttt{isInt}\left(x_{f_1}\right) \,?\triangleright\, x_{f_{1'}}; H\left(x_\sigma, x_{t_2}, x_{f_2}\right); \\ \texttt{isInt}\left(x_{f_2}\right) \,?\triangleright\, x_{f_{2'}}; \texttt{add}\left(x_{f_{1'}}, x_{f_{2'}}\right) \,?\triangleright\, x_{f_3}; \texttt{intVal}\left(x_{f_3}\right) \,?\triangleright\, x_o \end{array} \right]\!\!\right]^\# \left(\top, \Sigma_1^\#, T^\#\right) \Downarrow \left(f, \Sigma_o^\#\right) \\
& \quad \wedge \quad \Sigma_1^\# = x_\sigma \mapsto \sigma^\# + x_{t_1} \mapsto t_1 + x_{t_2} \mapsto t_2 \quad \wedge \quad v^\# = \Sigma_o^\#(x_o)
\end{aligned}$$

For simplicity, we here choose to ignore weakenings and the non-$\top$-case for the flag f. In other words, we are ignoring the possibility of short-cutting the abstract interpretation of the rule if a $\bot$ is found during the abstract execution.

$$\begin{aligned}
& \left(\sigma^\#, t_1 + t_2, v^\#\right) \in \mathcal{H}^\#\left(T^\#\right) \\
\Leftarrow \quad & \left(\Sigma_1^\#(x_\sigma), \Sigma_1^\#\left(x_{t_1}\right), v_1^\#\right) \in T^\# \\
& \quad \wedge \quad \left[\!\!\left[\begin{array}{l} \texttt{isInt}\left(x_{f_1}\right) \,?\triangleright\, x_{f_{1'}}; H\left(x_\sigma, x_{t_2}, x_{f_2}\right); \texttt{isInt}\left(x_{f_2}\right) \,?\triangleright\, x_{f_{2'}}; \\ \texttt{add}\left(x_{f_{1'}}, x_{f_{2'}}\right) \,?\triangleright\, x_{f_3}; \texttt{intVal}\left(x_{f_3}\right) \,?\triangleright\, x_o \end{array} \right]\!\!\right]^\# \left(\top, \Sigma_2^\#, T^\#\right) \Downarrow \left(\top, \Sigma_o^\#\right) \\
& \quad \wedge \quad \Sigma_1^\# = x_\sigma \mapsto \sigma^\# + x_{t_1} \mapsto t_1 + x_{t_2} \mapsto t_2 \quad \wedge \quad \Sigma_2^\# = \Sigma_1^\# + x_{f_1} \mapsto v_1^\# \quad \wedge \quad v^\# = \Sigma_o^\#(x_o)
\end{aligned}$$

Interpreting the filter `isInt` makes us consider the integer projection of the abstract value $v_1^\#$. We can thus rewrite the implication as follows.

$$\begin{aligned}
& \left(\sigma^\#, t_1 + t_2, v^\#\right) \in \mathcal{H}^\#\left(T^\#\right) \\
\Leftarrow \quad & \left(\sigma^\#, t_1, v_1^\#\right) \in T^\# \quad \wedge \quad v_1^\# = \left(i_1^\#, b_1^\#\right) \\
& \quad \wedge \quad \left[\!\!\left[\begin{array}{l} H\left(x_\sigma, x_{t_2}, x_{f_2}\right); \texttt{isInt}\left(x_{f_2}\right) \,?\triangleright\, x_{f_{2'}}; \\ \texttt{add}\left(x_{f_{1'}}, x_{f_{2'}}\right) \,?\triangleright\, x_{f_3}; \texttt{intVal}\left(x_{f_3}\right) \,?\triangleright\, x_o \end{array} \right]\!\!\right]^\# \left(\top, \Sigma_2^\#, T^\#\right) \Downarrow \left(\top, \Sigma_o^\#\right) \\
& \quad \wedge \quad \Sigma_2^\# = x_\sigma \mapsto \sigma^\# + x_{t_1} \mapsto t_1 + x_{t_2} \mapsto t_2 + x_{f_1} \mapsto v_1^\# + x_{f_{1'}} \mapsto i_1^\# \quad \wedge \quad v^\# = \Sigma_o^\#(x_o)
\end{aligned}$$

We can continue unfolding the abstract interpretation of the rule. We eventually reach the following implication:

$$\begin{aligned}
& \left(\sigma^\#, t_1 + t_2, v^\#\right) \in \mathcal{H}^\#\left(T^\#\right) \\
\Leftarrow \quad & \left(\sigma^\#, t_1, v_1^\#\right) \in T^\# \ \wedge\ \left(\sigma^\#, t_2, v_2^\#\right) \in T^\# \ \wedge\ v_1^\# = \left(i_1^\#, b_1^\#\right) \ \wedge\ v_2^\# = \left(i_2^\#, b_2^\#\right) \ \wedge\ v^\# = \Sigma_o^\#(x_o) \\
& \quad \wedge \quad \left[\!\!\left[\texttt{add}\left(x_{f_{1'}}, x_{f_{2'}}\right) \,?\triangleright\, x_{f_3}; \texttt{intVal}\left(x_{f_3}\right) \,?\triangleright\, x_o\right]\!\!\right]^\# \left(\top, \Sigma_4^\#, T^\#\right) \Downarrow \left(\top, \Sigma_o^\#\right) \\
& \quad \wedge \quad \Sigma_4^\# = x_\sigma \mapsto \sigma^\# + x_{t_1} \mapsto t_1 + x_{t_2} \mapsto t_2 + x_{f_1} \mapsto v_1^\# + x_{f_{1'}} \mapsto i_1^\# + x_{f_2} \mapsto v_2^\# + x_{f_{2'}} \mapsto i_2^\# \\
\Leftarrow \quad & \left(\sigma^\#, t_1, \left(i_1^\#, b_1^\#\right)\right) \in T^\# \ \wedge\ \left(\sigma^\#, t_2, \left(i_2^\#, b_2^\#\right)\right) \in T^\# \ \wedge\ [\![add]\!]^\#\left(i_1^\#, i_2^\#\right) = i^\# \ \wedge\ v^\# = \left(i^\#, \bot_{bool}\right)
\end{aligned}$$

By writing $\sigma^\# \vdash t : v^\#$ for $(\sigma^\#, t, v^\#) \in T^\#$ we get the familiar rule below.

$$\frac{\sigma^\# \vdash t_1 : \left(i_1^\#, b_1^\#\right) \qquad \sigma^\# \vdash t_2 : \left(i_2^\#, b_2^\#\right) \qquad [\![add]\!]^\#\left(i_1^\#, i_2^\#\right) = i^\#}{\sigma^\# \vdash t_1 + t_2 : \left(i^\#, \bot_{bool}\right)}$$

Rule for conditionals. In the case of the addition, the structure of the skeleton was linear. We have seen that we ignored some branches (the ones triggering $\bot$), but these were not very important. We now show the example of conditionals, where branches are more visible. A triple $(\sigma^\#, \mathit{if}\ t_1\ t_2\ t_3, \sigma_o^\#)$ is valid if it belongs to a fixpoint $T^\#$ of $\mathcal{H}^\#$. Unfolding definitions, and passing through the linear part of the skeleton, we get:

$$
\begin{array}{ll}
& \left(\sigma^\#, \mathit{if}\ t_1\ t_2\ t_3, \sigma_o^\#\right) \in \mathcal{H}^\#\left(T^\#\right) \\
\Leftrightarrow & \\
& \left[\!\!\left[H\left(x_\sigma, x_{t_1}, x_{f_1}\right); \texttt{isBool}\left(x_{f_1}\right)\ ?\triangleright\ x_{f_{1'}}; \begin{pmatrix} \texttt{isTrue}\left(x_{f_{1'}}\right); H\left(x_\sigma, x_{t_2}, x_o\right) \\ \texttt{isFalse}\left(x_{f_{1'}}\right); H\left(x_\sigma, x_{t_3}, x_o\right) \end{pmatrix}_{\{x_o\}} \right]\!\!\right]^\# \left(\top, \Sigma_1^\#, T^\#\right) \Downarrow \left(f, \Sigma_o^\#\right) \\
& \qquad \wedge \quad \Sigma_1^\# = x_\sigma \mapsto \sigma^\# + x_{t_1} \mapsto t_1 + x_{t_2} \mapsto t_2 + x_{t_3} \mapsto t_3 \quad \wedge \quad \sigma_o^\# = \Sigma_o^\#\left(x_o\right) \\
\Leftarrow & \quad \left(\sigma^\#, t_1, v_1^\#\right) \in T^\# \quad \wedge \quad v_1^\# = \left(i_1^\#, b_1^\#\right) \quad \wedge \\
& \quad \left[\!\!\left[\begin{pmatrix} \texttt{isTrue}\left(x_{f_{1'}}\right); H\left(x_\sigma, x_{t_2}, x_o\right) \\ \texttt{isFalse}\left(x_{f_{1'}}\right); H\left(x_\sigma, x_{t_3}, x_o\right) \end{pmatrix}_{\{x_o\}} \right]\!\!\right]^\# \left(\top, \Sigma_2^\#, T^\#\right) \Downarrow \left(f, \Sigma_o^\#\right) \\
& \qquad \wedge \quad \Sigma_1^\# = x_\sigma \mapsto \sigma^\# + x_{t_1} \mapsto t_1 + x_{t_2} \mapsto t_2 + x_{t_3} \mapsto t_3 + x_{f_1} \mapsto v_1^\# + x_{f_{1'}} \mapsto b_1^\#
\end{array}
$$

From this stage, we continue the analysis in each of the two subbranches to build a map O representing the outputs of both branches. We consider two cases, depending on the value of $b_1^\#$.

First, if $b_1^\#$ is $\top_{bool}$. We then have both `isTrue` and `isFalse` holding on $b_1^\#$. By unfolding definitions and using weakening for the results of the two hooks, we get the following implication:

$$
\begin{array}{ll}
& \left(\sigma^\#, \mathit{if}\ t_1\ t_2\ t_3, \sigma_o^\#\right) \in \mathcal{H}^\#\left(T^\#\right) \\
\Leftarrow & \left(\sigma^\#, t_1, v_1^\#\right) \in T^\# \quad \wedge \quad \left(\sigma^\#, t_2, \sigma_2^\#\right) \in T^\# \quad \wedge \quad \left(\sigma^\#, t_3, \sigma_3^\#\right) \in T^\# \\
& \qquad \wedge \quad v_1^\# = \left(i_1^\#, \top_{bool}\right) \quad \wedge \quad \sigma_2^\# \sqsubseteq \sigma_o^\# \quad \wedge \quad \sigma_3^\# \sqsubseteq \sigma_o^\#
\end{array}
$$

Using the same notations as above we can simplify this rule as below.

$$
\frac{\sigma^\# \vdash t_1 : \left(i_1^\#, \top_{bool}\right) \qquad \sigma^\# \vdash t_2 : \sigma_2^\# \qquad \sigma_2^\# \sqsubseteq \sigma_o^\# \qquad \sigma^\# \vdash t_3 : \sigma_3^\# \qquad \sigma_3^\# \sqsubseteq \sigma_o^\#}{\sigma^\# \vdash \mathit{if}\ t_1\ t_2\ t_3 : \sigma_o^\#}
$$

This rule is imprecise (we assume that we get $\top_{bool}$ when evaluating the conditional's expression), but shows how our equivalent of concrete rules are merged in the abstract interpretation. We now consider a more precise version of the rule, for the case when the conditional expression evaluates to $\mathit{true}^\#$. The other cases $\mathit{false}^\#$ and $\bot_{bool}$ are similar. In this case, the `isTrue` filter holds, but not `isFalse`: we can derive the judgement below when $\Sigma^\#(x_{f_{1'}}) = \mathit{true}^\#$.

$$
\left[\!\!\left[\texttt{isFalse}\left(x_{f_{1'}}\right); H\left(x_\sigma, x_{t_3}, x_o\right) \right]\!\!\right]^\# \left(\top, \Sigma^\#, T^\#\right) \Downarrow \left(\bot, \Sigma_o^\#\right)
$$

Following the rules for abstract interpretation (see Figure 10), this removes the second branch from the $\mathcal{E}$ set, only leaving constraints from the first branch. We thus get the following implication, where we no longer need the weakening for the result of the hook.

$$
\begin{array}{ll}
& \left(\sigma^\#, \mathit{if}\ t_1\ t_2\ t_3, \sigma_o^\#\right) \in \mathcal{H}^\#\left(T^\#\right) \\
\Leftarrow & \left(\sigma^\#, t_1, v_1^\#\right) \in T^\# \quad \wedge \quad \left(\sigma^\#, t_2, \sigma_o^\#\right) \in T^\# \quad \wedge \quad v_1^\# = \left(i_1^\#, \mathit{true}^\#\right)
\end{array}
$$

We can rewrite this implication as above into the rule

$$
\frac{\sigma^\# \vdash t_1 : \left(i_1^\#, \mathit{true}^\#\right) \qquad \sigma^\# \vdash t_2 : \sigma_2^\#}{\sigma^\# \vdash \mathit{if}\ t_1\ t_2\ t_3 : \sigma_2^\#}.
$$

Rule for loops. The skeleton for loops is close to the one for conditionals. We can similarly derive abstract rules such as the ones below.

$$\frac{\sigma^\# \vdash t_1 : (i_1^\#, \top_{bool}) \qquad \sigma^\# \vdash t_2 : \sigma_2^\# \qquad \sigma_2^\# \vdash \mathit{while}\, t_1\, t_2 : \sigma_3^\# \qquad \sigma_3^\# \sqsubseteq \sigma^\#}{\sigma^\# \vdash \mathit{while}\, t_1\, t_2 : \sigma^\#}$$

$$\frac{\sigma^\# \vdash t_1 : (i_1^\#, \mathit{true}^\#) \qquad \sigma^\# \vdash t_2 : \sigma_2^\# \qquad \sigma_2^\# \vdash \mathit{while}\, t_1\, t_2 : \sigma_3^\#}{\sigma^\# \vdash \mathit{while}\, t_1\, t_2 : \sigma_3^\#} \qquad \frac{\sigma^\# \vdash t_1 : (i_1^\#, \mathit{false}^\#)}{\sigma^\# \vdash \mathit{while}\, t_1\, t_2 : \sigma^\#}$$

We can also use the fact that any fixpoint of $\mathcal{H}^\#$ is considered valid. The following implication (which we can prove in a way similar to above) is valid for any well-formed set $T^\#$.

$$\begin{aligned} & (\sigma^\#, \mathit{while}\, t_1\, t_2, \sigma_o^\#) \in \mathcal{H}^\#(T^\#) \\ \Leftarrow \quad & (\sigma^\#, t_1, v_1^\#) \in T^\# \quad \wedge \quad v_1^\# \sqsubseteq (i_1^\#, \top_{bool}) \\ & \quad \wedge \quad (\sigma^\#, t_2, \sigma_2^\#) \in T^\# \quad \wedge \quad (\sigma_2^\#, \mathit{while}\, t_1\, t_2, \sigma_3^\#) \in T^\# \quad \wedge \quad \sigma_3^\# \sqsubseteq \sigma_o^\# \quad \wedge \quad \sigma^\# \sqsubseteq \sigma_o^\# \end{aligned}$$

In particular, as the condition $v_1^\# \sqsubseteq (i_1^\#, \top_{bool})$ is vacuously true, we can weaken this implication as follows (forcing all intermediate states to be the same).

$$(\sigma^\#, \mathit{while}\, t_1\, t_2, \sigma^\#) \in \mathcal{H}^\#(T^\#) \quad \Leftarrow \quad (\sigma^\#, t_1, v_1^\#) \in T^\# \wedge (\sigma^\#, t_2, \sigma^\#) \in T^\# \wedge (\sigma^\#, \mathit{while}\, t_1\, t_2, \sigma^\#) \in T^\#$$

This implication means that given any $T_0^\#$, such that $T_0^\# \subseteq \mathcal{H}^\#(T_0^\#)$, that associates t_1 in the state $\sigma^\#$ with a result (that is that there exists $v^\#$ such that $(\sigma^\#, t_1, v^\#) \in T_0^\#$), and such that $(\sigma^\#, t_2, \sigma^\#) \in T_0^\#$, we can extend $T_0^\#$ into $T_1^\# = T_0^\# \cup \{(\sigma^\#, \mathit{while}\, t_1\, t_2, \sigma^\#)\}$. By monotonicity of $\mathcal{H}^\#$, we get $T_0^\# \subseteq \mathcal{H}^\#(T_1^\#)$, and by the above implication, we get $(\sigma^\#, \mathit{while}\, t_1\, t_2, \sigma^\#) \in \mathcal{H}^\#(T_1^\#)$. Hence, $T_1^\# \subseteq \mathcal{H}^\#(T_1^\#)$, and every triple in $T_1^\#$ is correct in relation to $\Downarrow$. In other words, the following familiar rule is admissible.

$$\frac{\sigma^\# \vdash t_1 : v^\# \qquad \sigma^\# \vdash t_2 : \sigma^\#}{\sigma^\# \vdash \mathit{while}\, t_1\, t_2 : \sigma^\#}$$

6.2 State Splitting

As another example of the use of the abstract interpretation, we show how to extend the abstract semantics to obtain more precise results. Our motivating example is t: $\mathit{while}\, \neg(\mathrm{x} = 0)\, \mathrm{x} := \mathrm{x} - 1$ for which we want to show that the triple $(\mathrm{x} \mapsto [0, \infty], t, \mathrm{x} \mapsto 0)$ is correct (we simplify notation and write n for $([n, n], \bot_{bool})$, and $[n, m]$ for $([n, m], \bot_{bool})$). Proving this is not possible as such. To see this, observe that in the rule for While, the same state is used to run the expression and the statement, hence the return value of the expression is not reflected in the state (it may only prevent a branch from being taken). Communicating information from an expression back to a state is a non-trivial problem which depends on the language considered, but we can help the abstract interpretation by splitting the state in three parts: $\{(\mathrm{x} \mapsto 0, t, \mathrm{x} \mapsto 0), (\mathrm{x} \mapsto [1, \infty], t, \mathrm{x} \mapsto 0), (\mathrm{x} \mapsto [0, \infty], t, \mathrm{x} \mapsto 0)\}$. Let $T^\#$ be the set of triples (listed below) obtained from adding triples for every sub-expression of t. We can show that $\{(\mathrm{x} \mapsto 0, t, \mathrm{x} \mapsto 0), (\mathrm{x} \mapsto [1, \infty], t, \mathrm{x} \mapsto 0)\} \subset \mathcal{H}^\#(T^\#)$ (the second triple uses $(\mathrm{x} \mapsto [0, \infty], t, \mathrm{x} \mapsto 0)$ to evaluate the recursive while term). However there is still one of the three triples that cannot be derived, *viz.*, $(\mathrm{x} \mapsto [0, \infty], t, \mathrm{x} \mapsto 0) \in \mathcal{H}^\#(T^\#)$.

To derive this third triple, we introduce a proof technique called *state splitting* to obtain a more precise abstract semantics. The core idea of the technique is that if the state $\sigma^\#$ of a triple $(\sigma^\#, t^\#, v^\#)$ is covered by the states of some triples $(\sigma_1^\#, t^\#, v^\#)..(\sigma_n^\#, t^\#, v^\#)$, in the sense that $\gamma(\sigma^\#) \subseteq \gamma(\sigma_1^\#) \cup .. \cup \gamma(\sigma_n^\#)$, then we may use $(\sigma^\#, t^\#, v^\#)$ in the input triple set $T^\#$ of $\mathcal{H}^\#(T^\#)$ *without* having to show that $(\sigma^\#, t^\#, v^\#)$ is in the resulting triple set $\mathcal{H}^\#(T^\#)$ and still remain correct.

Formally, we first define a function *Sp* from triple sets to triple sets that adds such triples.

Definition 6.1. Let $T^\#$ an abstract triple set. We define the state splitting function $Sp(T^\#)$ as:

$$Sp\left(T^\#\right) = \left\{ \left(\sigma^\#, t^\#, v^\#\right) \middle| \begin{array}{c} \left\{\left(\sigma_1^\#, t^\#, v^\#\right) .. \left(\sigma_n^\#, t^\#, v^\#\right)\right\} \subseteq T^\# \text{ with } n \geq 1 \\ \forall i \in [1..n]. Sort\left(\sigma^\#\right) = Sort\left(\sigma_i^\#\right) \\ \gamma\left(\sigma^\#\right) \subseteq \gamma\left(\sigma_1^\#\right) \cup .. \cup \gamma\left(\sigma_n^\#\right) \end{array} \right\}$$

Lemma 6.2. *For any $T^\#$, $T^\# \subseteq Sp(T^\#)$, and Sp is monotonic.*

Lemma 6.3. *Let $T^\#$ a well-formed triple set, then $Sp(T^\#)$ is well formed.*

Proof. Let $(\sigma^\#, t^\#, v^\#) \in Sp(T^\#)$, then there is some $\sigma_1^\#, t^\#, v^\# \in T^\#$ such that $Sort(\sigma^\#) = Sort(\sigma_1^\#) = in(t^\#)$ and $Sort(v^\#) = out(t^\#)$. □

We next show that the functional $Sp(\mathcal{H}^\#(Sp(\cdot)))$ has the same consistency property as $\mathcal{H}^\#(\cdot)$.

Lemma 6.4. *Let T and $T^\#$ be well formed and consistent triple sets, then $\mathcal{H}(T)$ and $Sp(\mathcal{H}^\#(Sp(T^\#)))$ are well formed and consistent triple sets.*

We finally state that the proof technique is correct.

Lemma 6.5. *Let $T^\#$ a well-formed abstract triple set. If $T^\# \subseteq \mathcal{H}^\#(Sp(T^\#))$, then $Sp(T^\#)$ is correct.*

We turn back to our example. Consider the following triple set.

$$T^\# = \left\{ \begin{array}{c} (\mathsf{x} \mapsto 0, 0, 0), (\mathsf{x} \mapsto [1, \infty], 0, 0), (\mathsf{x} \mapsto 0, -1, -1), (\mathsf{x} \mapsto [1, \infty], -1, -1), \\ (\mathsf{x} \mapsto 0, \mathsf{x}, 0), (\mathsf{x} \mapsto [1, \infty], \mathsf{x}, [1, \infty]), \\ \left(\mathsf{x} \mapsto 0, \mathsf{x} = 0, true^\#\right), \left(\mathsf{x} \mapsto [1, \infty], \mathsf{x} = 0, false^\#\right), \\ \left(\mathsf{x} \mapsto 0, \neg(\mathsf{x} = 0), false^\#\right), \left(\mathsf{x} \mapsto [1, \infty], \neg(\mathsf{x} = 0), true^\#\right), \\ (\mathsf{x} \mapsto [1, \infty], \mathsf{x} - 1, [0, \infty]), (\mathsf{x} \mapsto [1, \infty], \mathsf{x} := \mathsf{x} - 1, \mathsf{x} \mapsto [0, \infty]), \\ (\mathsf{x} \mapsto 0, t, \mathsf{x} \mapsto 0), (\mathsf{x} \mapsto [1, \infty], t, \mathsf{x} \mapsto 0) \end{array} \right\}$$

We can show that $T^\# \subseteq Sp(\mathcal{H}^\#(Sp(T^\#)))$, hence every triple of $Sp(T^\#)$ is correct, in particular $(\mathsf{x} \mapsto [0, \infty], t, \mathsf{x} \mapsto 0)$.

Note that this proof technique does not depend on the programming language considered. The difficulty is transferred to the choice of how to split the state, but as long as the splitting is correct (the added triple is covered by the existing ones), the resulting technique is sound.

7 CONSTRAINT GENERATION

As a final interpretation, we show how the abstract interpretation can be used to construct an actual program analyser. We define the analyser as an interpretation that generates data flow constraints to analyse a given program [Nielson et al. 1999].[3] Constraint-based program analysis is a well-known technique for defining analyses. We show how this technique can be lifted and defined entirely as an interpretation, by generating constraints over all the flow variables used in a semantic definition.

We first need to formalise (and extend) the standard notion of *program point*. We take a program point pp to be a list of integers denoting a position in a term. Program points form a monoid with concatenation operator $\cdot$ and neutral element ϵ. We define a subterm operator $t@\mathsf{pp}$ as follows.

$$t@\epsilon \triangleq t \qquad c\,(t_1..t_n)\,@k{\cdot}\mathsf{pp} \triangleq \begin{cases} t_k@\mathsf{pp} & \text{if } k \in [1..n] \\ \text{undefined} & \text{otherwise} \end{cases}$$

[3]We impose the technical restriction that any hook used in a skeleton can be matched to a program point of the program (closed term) t_0 under consideration. Thus constraint-based analysis of code-generating code is not considered here.

We assume a function PP that for a given term t_0 states the set of program points for which constraints will be generated. It typically consists of the set of executable subterms of t_0. We require the program points of $PP(t_0)$ to be executable: if $\mathsf{pp} \in PP(t_0)$, then $t_0@\mathsf{pp} = c(t_1..t_n)$. Requirement 2.2 enforces the existence of a skeleton for this term.

We next define a partial operator HPP_{t_0} that associates program points to the terms occurring in the hooks of a skeleton. Formally, if $HPP_{t_0}(\mathsf{pp}, \mathrm{N}, t) = \mathsf{pp}'$, then (1) skeleton N is applicable: $\mathsf{pp} \in PP(t_0)$, $t_0@\mathsf{pp} = c(t_1..t_n)$, and N is of the form $\mathrm{N}(c(x_{t_1}..x_{t_n})) \coloneqq S$, (2) a hook $H(\_, t, \_)$ occurs in S, and (3) the resulting program point is part of the set of explored program points: $\mathsf{pp}' \in PP(t_0)$ and $t_0@\mathsf{pp}' = (x_{t_1} \mapsto t_1..x_{t_n} \mapsto t_n)(t)$.

Constraints are either of the form $[x = x']$, $[x \sqsubseteq x']$, or $[x : s]$, where x and x' are variables and s a sort. We generate variable names in constraints of the form $\mathsf{pp}\text{-}x$. The constraint generation function Gen that takes a program t_0 and returns the set of constraints generated by t_0 is defined as

$$Gen(t_0) \triangleq \bigcup C \cup \left\{ \begin{array}{l} [\mathsf{pp}\text{-}x_\sigma : in(Sort(t_0@\mathsf{pp}))], \\ [\mathsf{pp}\text{-}x_o : out(Sort(t_0@\mathsf{pp}))], \\ \forall i \in [1..n].\ [\mathsf{pp}\text{–}x_{t_i} = t_i] \end{array} \middle| \begin{array}{c} \mathsf{pp} \in PP(t_0) \wedge t_0@\mathsf{pp} = c(t_1..t_n) \\ \mathrm{N}(c(x_{t_1}..x_{t_n})) \coloneqq S \in Rules \\ [\![S]\!]^c(\mathrm{N}, \mathsf{pp}, \emptyset) \Downarrow C \\ \mathcal{D}_\mathrm{N}(\emptyset) = \{x_{t_1}..x_{t_n}, x_\sigma\} \wedge x_o \in \mathcal{D}_\mathrm{N}(C) \end{array} \right\}$$

For each skeleton N we define a function $\mathcal{D}_\mathrm{N}$ that maps sets of constraints to sets of skeletal variables. This is not necessary for the constraint generation but is used to prove consistency between constraints and the abstract semantics.

The constraint generation interpretation of skeletons $[\![S]\!]^c$ is given in Figure 13. The rule for hooks generates constraints for connecting the input state $\mathsf{pp}'\text{-}x_\sigma$ with the flow variable holding the input state in the hook $\mathsf{pp}\text{–}x_{f_1}$, and the resulting output state of the hook with the output of the hook. Each filter comes with a constraint generation function $[\![F]\!]^c$ specific to the analysis of that filter. We require that the constraints generated for that filter agree with the abstract semantics: if $\mathcal{S}$ is a solution to the constraints $[\![F]\!]^c(\mathsf{pp}\text{–}x_1..\mathsf{pp}\text{–}x_n, \mathsf{pp}\text{–}y_1..\mathsf{pp}\text{–}y_m)$, then following holds: $[\![F]\!]^\#(\mathcal{S}(\mathsf{pp}\text{–}x_1)..\mathcal{S}(\mathsf{pp}\text{–}x_n)) \sqsubseteq (\mathcal{S}(\mathsf{pp}\text{–}y_1)..\mathcal{S}(\mathsf{pp}\text{–}y_m))$. For analysing a set of branches, we generate constraints for each branch and return the union of these constraint sets.

Correctness. A *solution* $\mathcal{S}$ of a set of constraints C is a mapping from the variables in C to abstract values and terms such that every constraint in C holds.

Lemma 7.1. *Let t_0 be a term and $\mathcal{S}$ be a solution of $Gen(t_0)$. Let $T^\#$ be defined as follows:*

$$T^\# = \left\{ (\sigma^\#, t, v^\#) \middle| \begin{array}{c} \mathsf{pp} \in PP(t_0) \\ t = t_0@\mathsf{pp} \\ \mathcal{S}(\mathsf{pp}\text{-}x_\sigma) = \sigma^\# \\ \mathcal{S}(\mathsf{pp}\text{-}x_o) = v^\# \end{array} \right\}$$

Then $T^\#$ is well typed and $T^\# \subseteq \mathcal{H}^\#(T^\#)$.

Discussion. The constraints we generate are *path-insensitive*: they do not capture the fact that when a filter does not hold, the rest of the skeleton does not matter. Constraints can be path-sensitive by letting the state of the interpretation be a pair consisting of a set *Stop* of constraint sets representing pathways in the skeleton that are stopped, similar to the $\bot$ flag in the abstract interpretation, and another set *Run* of constraint sets representing all the running paths. When a filter is encountered, the *Run* sets are added to *Stop* with the additional constraint that the filter returns $\bot$. The usual constraints for the filter are added to each set in *Run*. In a nutshell, we duplicate constraints for each filter: once when it does not hold, and once when it may hold. At the end of

$$\Longrightarrow \llbracket [] \rrbracket^c (\mathrm{N}, \mathsf{pp}, C) \Downarrow C$$

$$\left(\begin{array}{r} HPP_{t_0}(\mathsf{pp}, \mathrm{N}, t) = \mathsf{pp}' \\ x_{f_1} \in \mathcal{D}_{\mathrm{N}}(C) \\ C' = C \cup \left\{ \begin{array}{l} \left[\mathsf{pp}\text{-}x_{f_1} \sqsubseteq \mathsf{pp}'\text{-}x_\sigma\right], \\ \left[\mathsf{pp}'\text{-}x_o \sqsubseteq \mathsf{pp}\text{-}x_{f_2}\right] \end{array} \right\} \\ \mathcal{D}_{\mathrm{N}}(C') = \mathcal{D}_{\mathrm{N}}(C) \cup \{x_{f_2}\} \end{array}\right) \Longrightarrow \llbracket H\left(x_{f_1}, t, x_{f_2}\right) \rrbracket^c (\mathrm{N}, \mathsf{pp}, C) \Downarrow (\mathrm{N}, \mathsf{pp}, C')$$

$$\left(\begin{array}{r} \{x_1..x_n\} \subseteq \mathcal{D}_{\mathrm{N}}(C) \\ \llbracket F \rrbracket^c \begin{pmatrix} \mathsf{pp}\text{-}x_1..\mathsf{pp}\text{-}x_n, \\ \mathsf{pp}\text{-}y_1..\mathsf{pp}\text{-}y_m \end{pmatrix} = C_f \\ C' = C \cup C_f \\ \mathcal{D}_{\mathrm{N}}(C') = \mathcal{D}_{\mathrm{N}}(C) \cup \{y_1..y_m\} \end{array}\right) \Longrightarrow \llbracket F\,(x_1..x_n)\ ?\triangleright\ (y_1..y_m) \rrbracket^c (\mathrm{N}, \mathsf{pp}, C) \Downarrow (\mathrm{N}, \mathsf{pp}, C')$$

$$\left(\begin{array}{r} i \geq 1 \\ \forall i \in [1..n].O(i) = C_i \\ \forall i \in [1..n].V \subseteq \mathcal{D}_{\mathrm{N}}(C_i) \\ C' = C \cup \bigcup_{i \in [1..n]} C_i \\ \mathcal{D}_{\mathrm{N}}(C') = \mathcal{D}_{\mathrm{N}}(C) \cup V \end{array}\right) \Longrightarrow \left[\!\!\left[\bigoplus_n \right]\!\!\right]^c_V (O, (\mathrm{N}, \mathsf{pp}, C)) \Downarrow (\mathrm{N}, \mathsf{pp}, C')$$

Fig. 13. Constraint Generation

the interpretation, the global constraint to be satisfied is the disjunction of all constraint sets in *Run* and *Stop*, each constraint set interpreted as a conjunction of its atomic constraints.

Example. Consider $t_0 = \mathit{while}\ \neg(\mathtt{x} = 0)\ \mathtt{x} := \mathtt{x} - 1$. Its executable subterms are

$$PP(t_0) = \{\epsilon, 1, 1{\cdot}1, 1{\cdot}1{\cdot}1, 1{\cdot}1{\cdot}2, 2, 2{\cdot}2, 2{\cdot}2{\cdot}1, 2{\cdot}2{\cdot}2\}\,.$$

Note that the subterm $\mathtt{x}$ appears both as program points $1{\cdot}1{\cdot}1$ and $2{\cdot}2{\cdot}1$ of t_0. For the different filters we generate symbolic constraints that will reuse abstract filters: $\llbracket \mathtt{isBool} \rrbracket^c (x, y) = \{[y = \mathtt{isBool}(x)]\}$. A mapping $\mathcal{S}$ is then a solution of such a symbolic constraint if $\mathcal{S}(y) = \llbracket \mathtt{isBool} \rrbracket^{\#} (\mathcal{S}(x))$.

The definition of $\mathit{Gen}(t_0)$ generates a large number of constraints. We focus on a selection of them: those generated by the initial program point ϵ. The associated skeleton is

$$\textsc{While}(\mathit{while}\, x_{t_1}\, x_{t_2}) \coloneqq \left[H\left(x_\sigma, x_{t_1}, x_{f_1}\right); \mathtt{isBool}\left(x_{f_1}\right)\ ?\triangleright\ x_{f_{1'}}; \ldots\right].$$

The constraint generation then produces the constraints

$$\left[\epsilon\text{-}x_\sigma : \mathit{store}\right], \qquad \left[\epsilon\text{-}x_{t_1} = \neg\left(\mathtt{x} = 0\right)\right], \qquad \left[\epsilon\text{-}x_o : \mathit{store}\right], \qquad \left[\epsilon\text{-}x_{t_2} = \mathtt{x} := \mathtt{x} - 1\right].$$

as well as the constraints given by $\left[\!\left[H(x_\sigma, x_{t_1}, x_{f_1}); \mathtt{isBool}(x_{f_1})\ ?\triangleright\ x_{f_{1'}}; \ldots\right]\!\right]^c (\mathrm{N}, \mathsf{pp}, \emptyset)$. The hook case links the variable $\epsilon\text{-}x_\sigma$ to the input of x_{t_1}, which here represents $\neg(\mathtt{x} = 0)$: $HPP_{t_0}(\epsilon, \textsc{While}, x_{t_1}) = 1$ and we thus generate the two constraints

$$\left[\epsilon\text{-}x_\sigma \sqsubseteq 1\text{-}x_\sigma\right], \left[1\text{-}x_o \sqsubseteq \epsilon\text{-}x_{f_1}\right].$$

c	Signature
in	*expr*
out	$expr \longrightarrow stat$

c	Signature
throw	*stat*
try catch	$(stat, stat) \longrightarrow stat$

c	Signature
ref	$expr \longrightarrow expr$
!	$expr \longrightarrow expr$
$\longleftarrow$	$(expr, expr) \longrightarrow stat$

Fig. 14. Additional Constructors for While

f	$fsort(f)$
`in`	$in \longrightarrow (val, in)$
`alloc`	$(heap, val) \longrightarrow (heap, loc)$
`locVal`	$loc \longrightarrow val$
`isLoc`	$val \longrightarrow loc$
`get`	$(loc, heap) \longrightarrow val$
`set`	$(loc, heap, val) \longrightarrow heap$
`out`	$(out, val) \longrightarrow out$

f	$fsort(f)$
`mkSt`	$(in, out, store, heap) \longrightarrow state$
`splitSt`	$state \longrightarrow (in, out, store, heap)$
`mkValSt`	$(val, state) \longrightarrow valState$
`getValSt`	$valState \longrightarrow (val, state)$
`mkOK`	$state \longrightarrow excState$
`mkExc`	$state \longrightarrow excState$
`isOK`	$excState \longrightarrow state$
`isExc`	$excState \longrightarrow state$

Fig. 15. Additional filters

The constraints on 1-x_σ and 1-x_o are generated when considering the program point 1, corresponding to the evaluation of $\neg(x = 0)$ (corresponding to the skeleton Neg). As stated, the set of all generated constraints is large; it is provided in the supplementary material on the companion website.

8 EXTENDING WHILE WITH EXCEPTIONS, INPUT/OUTPUT, AND A HEAP

To further illustrate the use of skeletal semantics, we extend our While language with exceptions, input/output, and a heap. We first need to define new flow sorts: *in* for input streams, *out* for output streams, *heap* for heaps, *loc* for locations in the heap, *state* for the combination of the streams with a store and a heap, *valState* for the further combination with a value, and *excState* for a *state* extended to signal whether an exception was raised. We still have two program sorts (*expr* and *stat*), but their input flow sorts are *state*, and their output flow sorts are now *valState* for expressions and *excState* for statements. Figure 14 lists the additional constructors of our language. The additional filters are defined in Figure 15, the rules for expressions in Figure 16, and the rules for statements in Figure 17. To help reading the rules, flow variables have names related to their sorts: σ for *state*, w for *valState*, v for *val*, n for *int*, i for *in*, and so on.

Instantiation of concrete interpretation. We instantiate the *in* and *out* sorts with list of values, denoted by L. We instantiate locations as integers. A heap is a pair of an integer (the next free location) and a map from integers to values. We instantiate the *state* sort as a tuple of *in*, *out*, *store*, and *heap*, the *valState* sort as a pair of *val* and *state*, and the *excState* sort as a pair of a Boolean and *state*. The *val* sort is extended to include a case for locations, as well as the `intVal`, `boolVal`, `isInt`, and `isBool` filters. The `locVal` filter injects a location in the *val* type, and the `isLoc` filter applies if the *val* argument is a location, which it then returns. The `in` filter applies if the input list is not empty, it returns its head and its tail. The `alloc` filters applied to $((n, m), v)$ returns the heap $(n + 1, m + n \mapsto v)$. The `get` filter applies if the location is in the heap, and it returns the

$$\textsc{Lit}(\mathit{const}\,(x_t)) \coloneqq \left[\texttt{litInt}\,(x_t)\ ?\triangleright\ x_{f_n};\texttt{intVal}\,(x_{f_n})\ ?\triangleright\ x_{f_v};\texttt{mkValSt}\,(x_{f_v},x_\sigma)\ ?\triangleright\ x_o\right]$$

$$\textsc{Var}(\mathit{var}\,(x_t)) \coloneqq \left[\begin{array}{l}\texttt{splitSt}\,(x_\sigma)\ ?\triangleright\ (x_{f_i},x_{f_o},x_{f_s},x_{f_h});\texttt{read}\,(x_t,x_{f_s})\ ?\triangleright\ x_{f_v};\\ \quad\texttt{mkSt}\,(x_{f_i},x_{f_o},x_{f_s},x_{f_h})\ ?\triangleright\ x_{f_\sigma};\texttt{mkValSt}\,(x_{f_v},x_{f_\sigma})\ ?\triangleright\ x_o\end{array}\right]$$

$$\textsc{In}(\mathit{in}) \coloneqq \left[\begin{array}{l}\texttt{splitSt}\,(x_\sigma)\ ?\triangleright\ (x_{f_i},x_{f_o},x_{f_s},x_{f_h});\texttt{in}\,(x_{f_i})\ ?\triangleright\ (x_{f_v},x_{f_{i'}});\\ \quad\texttt{mkSt}\,(x_{f_{i'}},x_{f_o},x_{f_s},x_{f_h})\ ?\triangleright\ x_{f_\sigma};\texttt{mkValSt}\,(x_{f_v},x_{f_\sigma})\ ?\triangleright\ x_o\end{array}\right]$$

$$\textsc{Alloc}(\mathit{ref}\,(x_t)) \coloneqq \left[\begin{array}{l}H\,(x_\sigma,x_t,x_{f_w});\texttt{getValSt}\,(x_{f_w})\ ?\triangleright\ (x_{f_v},x_{f_\sigma});\\ \texttt{splitSt}\,(x_{f_\sigma})\ ?\triangleright\ (x_{f_i},x_{f_o},x_{f_s},x_{f_h});\texttt{alloc}\,(x_{f_h},x_{f_v})\ ?\triangleright\ (x_{f_{h'}},x_{f_l});\\ \quad\texttt{locVal}\,(x_{f_l})\ ?\triangleright\ x_{f_{v'}};\texttt{mkSt}\,(x_{f_i},x_{f_o},x_{f_s},x_{f_{h'}})\ ?\triangleright\ x_{f_{\sigma'}};\\ \quad\quad\texttt{mkValSt}\,(x_{f_{v'}},x_{f_{\sigma'}})\ ?\triangleright\ x_o\end{array}\right]$$

$$\textsc{Acc}(!x_t) \coloneqq \left[\begin{array}{l}H\,(x_\sigma,x_t,x_{f_w});\texttt{getValSt}\,(x_{f_w})\ ?\triangleright\ (x_{f_v},x_{f_\sigma});\texttt{isLoc}\,(x_{f_v})\ ?\triangleright\ x_{f_l};\\ \quad\texttt{splitSt}\,(x_{f_\sigma})\ ?\triangleright\ (x_{f_i},x_{f_o},x_{f_s},x_{f_h});\texttt{get}\,(x_{f_l},x_{f_h})\ ?\triangleright\ x_{f_{v'}};\\ \quad\quad\texttt{mkSt}\,(x_{f_i},x_{f_o},x_{f_s},x_{f_h})\ ?\triangleright\ x_{f_{\sigma'}};\texttt{mkValSt}\,(x_{f_{v'}},x_{f_{\sigma'}})\ ?\triangleright\ x_o\end{array}\right]$$

$$\textsc{Add}(x_{t_1}+x_{t_2}) \coloneqq \left[\begin{array}{l}H\left(x_\sigma,x_{t_1},x_{f_{w_1}}\right);\texttt{getValSt}\left(x_{f_{w_1}}\right)\ ?\triangleright\ \left(x_{f_{v_1}},x_{f_{\sigma_1}}\right);\texttt{isInt}\left(x_{f_{v_1}}\right)\ ?\triangleright\ x_{f_{n_1}};\\ H\left(x_{f_{\sigma_1}},x_{t_2},x_{f_{w_2}}\right);\texttt{getValSt}\left(x_{f_{w_2}}\right)\ ?\triangleright\ \left(x_{f_{v_2}},x_{f_{\sigma_2}}\right);\texttt{isInt}\left(x_{f_{v_2}}\right)\ ?\triangleright\ x_{f_{n_2}};\\ \quad\texttt{add}\left(x_{f_{n_1}},x_{f_{n_2}}\right)\ ?\triangleright\ x_{f_n};\texttt{intVal}\,(x_{f_n})\ ?\triangleright\ x_{f_v};\texttt{mkValSt}\left(x_{f_v},x_{f_{\sigma_2}}\right)\ ?\triangleright\ x_o\end{array}\right]$$

$$\textsc{Eq}(x_{t_1}=x_{t_2}) \coloneqq \left[\begin{array}{l}H\left(x_\sigma,x_{t_1},x_{f_{w_1}}\right);\texttt{getValSt}\left(x_{f_{w_1}}\right)\ ?\triangleright\ \left(x_{f_{v_1}},x_{f_{\sigma_1}}\right);\texttt{isInt}\left(x_{f_{v_1}}\right)\ ?\triangleright\ x_{f_{n_1}};\\ H\left(x_{f_{\sigma_1}},x_{t_2},x_{f_{w_2}}\right);\texttt{getValSt}\left(x_{f_{w_2}}\right)\ ?\triangleright\ \left(x_{f_{v_2}},x_{f_{\sigma_2}}\right);\texttt{isInt}\left(x_{f_{v_2}}\right)\ ?\triangleright\ x_{f_{n_2}};\\ \quad\texttt{eq}\left(x_{f_{n_1}},x_{f_{n_2}}\right)\ ?\triangleright\ x_{f_b};\texttt{boolVal}\,(x_{f_b})\ ?\triangleright\ x_{f_v};\texttt{mkValSt}\left(x_{f_v},x_{f_{\sigma_2}}\right)\ ?\triangleright\ x_o\end{array}\right]$$

$$\textsc{Neg}(\neg x_t) \coloneqq \left[\begin{array}{l}H\,(x_\sigma,x_t,x_{f_w});\texttt{getValSt}\,(x_{f_w})\ ?\triangleright\ (x_{f_v},x_{f_\sigma});\texttt{isBool}\,(x_{f_v})\ ?\triangleright\ x_{f_b};\\ \quad\texttt{neg}\,(x_{f_b})\ ?\triangleright\ x_{f_{b'}};\texttt{boolVal}\,(x_{f_{b'}})\ ?\triangleright\ x_{f_{v'}};\texttt{mkValSt}\,(x_{f_{v'}},x_{f_\sigma})\ ?\triangleright\ x_o\end{array}\right]$$

Fig. 16. Skeletal semantics for extended While (Expressions)

corresponding value. The `set` filter applies if the location is in the heap, and it return the heap updated with the given value. The `out` filter always apply and adds the given value to the output list. The `mkOK` filter (resp. the `mkExc` filter) always applies and builds a pair of *true* (resp. *false*) and the given state. The `isOK` filter (resp. the `isExc` filter) applies if the Boolean is true (resp. is false), it then return the *state* component of the tuple. Other filters build or deconstruct tuples.

Instantiation of abstract interpretation. To illustrate the flexibility of our approach, we choose a coarse abstraction for the *in* and *out* sorts (they are either $\bot$ or a single abstract value), and a precise abstraction of heaps: abstract heaps are modelled similar to concrete heap as a pair $(n, m^\#)$ of an integer and a mapping from integers to abstract values. Locations are abstracted as sets of integers. Tuples are abstracted as tuples of the abstraction of their components. The tuple-manipulating abstract filters are straightforward, so we only detail the other ones in Figure 18.

Lemma 8.1. *The abstract filters are consistent with the concrete filters.*

$$\textsc{Skip}(\mathit{skip}) \coloneqq [\mathtt{mkOK}\,(x_\sigma)\ ?\triangleright\ x_o] \qquad \textsc{Throw}(\mathit{throw}) \coloneqq [\mathtt{mkExc}\,(x_\sigma)\ ?\triangleright\ x_o]$$

$$\textsc{Asn}(x_{t_1} := x_{t_2}) \coloneqq \begin{bmatrix} H\,(x_\sigma, x_{t_2}, x_{f_w})\,;\mathtt{getValSt}\,(x_{f_w})\ ?\triangleright\ (x_{f_v}, x_{f_\sigma})\,; \\ \mathtt{splitSt}\,(x_{f_\sigma})\ ?\triangleright\ (x_{f_i}, x_{f_o}, x_{f_s}, x_{f_h})\,;\mathtt{write}\,(x_{t_1}, x_{f_s}, x_{f_v})\ ?\triangleright\ x_{f_{s'}}; \\ \mathtt{mkSt}\,(x_{f_i}, x_{f_o}, x_{f_{s'}}, x_{f_h})\ ?\triangleright\ x_{f_{\sigma'}}; \mathtt{mkOK}\,(x_{f_{\sigma'}})\ ?\triangleright\ x_o \end{bmatrix}$$

$$\textsc{Set}(x_{t_1} \leftarrow x_{t_2}) \coloneqq \begin{bmatrix} H\left(x_\sigma, x_{t_1}, x_{f_{w_1}}\right)\,;\mathtt{getValSt}\left(x_{f_{w_1}}\right)\ ?\triangleright\ \left(x_{f_{v_1}}, x_{f_\sigma}\right)\,;\mathtt{isLoc}\left(x_{f_{v_1}}\right)\ ?\triangleright\ x_{f_l} \\ H\left(x_{f_\sigma}, x_{t_2}, x_{f_{w_2}}\right)\,;\mathtt{getValSt}\left(x_{f_{w_2}}\right)\ ?\triangleright\ \left(x_{f_{v_2}}, x_{f_{\sigma'}}\right)\,; \\ \mathtt{splitSt}\,(x_{f_{\sigma'}})\ ?\triangleright\ (x_{f_i}, x_{f_o}, x_{f_s}, x_{f_h})\,;\mathtt{set}\left(x_{f_l}, x_{f_h}, x_{f_{v_2}}\right)\ ?\triangleright\ x_{f_{h'}}; \\ \mathtt{mkSt}\,(x_{f_i}, x_{f_o}, x_{f_s}, x_{f_{h'}})\ ?\triangleright\ x_{f_{\sigma''}}; \mathtt{mkOK}\,(x_{f_{\sigma''}})\ ?\triangleright\ x_o \end{bmatrix}$$

$$\textsc{Out}(\mathit{out}\,(x_{t_1})) \coloneqq \begin{bmatrix} H\,(x_\sigma, x_{t_1}, x_{f_w})\,;\mathtt{getValSt}\,(x_{f_w})\ ?\triangleright\ (x_{f_v}, x_{f_\sigma})\,; \\ \mathtt{splitSt}\,(x_{f_\sigma})\ ?\triangleright\ (x_{f_i}, x_{f_o}, x_{f_s}, x_{f_h})\,;\mathtt{out}\,(x_{f_o}, x_{f_v})\ ?\triangleright\ x_{f_{o'}}; \\ \mathtt{mkSt}\,(x_{f_i}, x_{f_{o'}}, x_{f_s}, x_{f_h})\ ?\triangleright\ x_{f_{\sigma'}}; \mathtt{mkOK}\,(x_{f_{\sigma'}})\ ?\triangleright\ x_o \end{bmatrix}$$

$$\textsc{Seq}(x_{t_1};\ x_{t_2}) \coloneqq \left[H\,(x_\sigma, x_{t_1}, x_{f_e})\,; \begin{pmatrix} \mathtt{isOK}\,(x_{f_e})\ ?\triangleright\ x_{f_\sigma}; H\,(x_{f_\sigma}, x_{t_2}, x_o) \\ \mathtt{isExc}\,(x_{f_e})\ ?\triangleright\ x_{f_{\sigma'}}; \mathtt{mkExc}\,(x_{f_{\sigma'}})\ ?\triangleright\ x_o \end{pmatrix}_{\{x_o\}} \right]$$

$$\textsc{Try}(\mathit{try}\,x_{t_1}\,\mathit{catch}\,x_{t_2}) \coloneqq \left[H\,(x_\sigma, x_{t_1}, x_{f_e})\,; \begin{pmatrix} \mathtt{isOK}\,(x_{f_e})\ ?\triangleright\ x_{f_\sigma}; \mathtt{mkOK}\,(x_{f_\sigma})\ ?\triangleright\ x_o \\ \mathtt{isExc}\,(x_{f_e})\ ?\triangleright\ x_{f_{\sigma'}}; H\,(x_{f_{\sigma'}}, x_{t_2}, x_o) \end{pmatrix}_{\{x_o\}} \right]$$

$$\textsc{If}(\mathit{if}\ x_{t_1}\ x_{t_2}\ x_{t_3}) \coloneqq \begin{bmatrix} H\,(x_\sigma, x_{t_1}, x_{f_w})\,;\mathtt{getValSt}\,(x_{f_w})\ ?\triangleright\ (x_{f_v}, x_{f_\sigma})\,; \\ \mathtt{isBool}\,(x_{f_v})\ ?\triangleright\ x_{f_b}; \begin{pmatrix} \mathtt{isTrue}\,(x_{f_b})\,; H\,(x_{f_\sigma}, x_{t_2}, x_o) \\ \mathtt{isFalse}\,(x_{f_b})\,; H\,(x_{f_\sigma}, x_{t_3}, x_o) \end{pmatrix}_{\{x_o\}} \end{bmatrix}$$

$$\textsc{While}(\mathit{while}\,x_{t_1}\ x_{t_2}) \coloneqq \begin{bmatrix} H\,(x_\sigma, x_{t_1}, x_{f_w})\,;\mathtt{getValSt}\,(x_{f_w})\ ?\triangleright\ (x_{f_v}, x_{f_\sigma})\,;\mathtt{isBool}\,(x_{f_v})\ ?\triangleright\ x_{f_b}; \\ \begin{pmatrix} \mathtt{isTrue}\,(x_{f_b})\,; H\,(x_{f_\sigma}, x_{t_2}, x_{f_e})\,; \\ \begin{pmatrix} \mathtt{isOK}\,(x_{f_e})\ ?\triangleright\ x_{f_{\sigma'}}; H\,(x_{f_{\sigma'}}, \mathit{while}\,x_{t_1}\,x_{t_2}, x_o) \\ \mathtt{isExc}\,(x_{f_e})\ ?\triangleright\ x_{f_{\sigma''}}; \mathtt{mkExc}\,(x_{f_{\sigma''}})\ ?\triangleright\ x_o \end{pmatrix}_{\{x_o\}} \\ \mathtt{isFalse}\,(x_{f_b})\,; \mathtt{mkOK}\,(x_{f_\sigma})\ ?\triangleright\ x_o \end{pmatrix}_{\{x_o\}} \end{bmatrix}$$

Fig. 17. Skeletal semantics for While2 (Statements)

Lemma 8.2. *The abstract semantics of Extended While is correct.*

9 RELATED WORK

Ott [Sewell et al. 2010] is a formalism for describing language semantics and type systems. Ott proposes a meta-language with a humanly readable syntax for writing semantic definitions as inference rules, and has facilities for translating these definitions into executable interpreters and specifications in proof assistants such as Coq and HOL. Lem [Mulligan et al. 2014] offers a core functional language extended with logical features from proof assistants for writing semantic

$$\gamma\left(v^{\#}\right) = \left\{L \middle| \forall v \in L. v \in \gamma\left(v^{\#}\right)\right\} \qquad \gamma\left(\left(a^{\#}, b^{\#}, c^{\#}\right)\right) = \{(a,b,c) | a \in \gamma(a) \wedge b \in \gamma(b) \wedge c \in \gamma(c)\}$$

$$\gamma\left(l^{\#}\right) = l^{\#} \qquad \gamma\left(\left(n, m^{\#}\right)\right) = \left\{ (n,m) \middle| \begin{array}{c} dom(m) = dom\left(m^{\#}\right) \\ \forall i \in dom(m).m[i] \in \gamma\left(m^{\#}[i]\right) \end{array} \right\}$$

$$\texttt{in}^{\#}\left(v^{\#}\right) = \left(v^{\#}, v^{\#}\right) \qquad \texttt{isOK}^{\#}\left(b^{\#}, s^{\#}\right) = \begin{cases} s^{\#} & \text{if } true^{\#} \sqsubseteq b^{\#} \\ \bot_{state} & \text{otherwise} \end{cases}$$

$$\texttt{out}^{\#}\left(v_1^{\#}, v_2^{\#}\right) = v_1^{\#} \sqcup v_2^{\#} \qquad \texttt{isExc}^{\#}\left(b^{\#}, s^{\#}\right) = \begin{cases} s^{\#} & \text{if } false^{\#} \sqsubseteq b^{\#} \\ \bot_{state} & \text{otherwise} \end{cases}$$

$$\texttt{locVal}^{\#}\left(l^{\#}\right) = \left(\bot_{int}, \bot_{bool}, l^{\#}\right) \qquad \texttt{isval}^{\#}\left(n^{\#}, b^{\#}, l^{\#}\right) = l^{\#}$$

$$\texttt{get}^{\#}\left(l^{\#}, \left(n, m^{\#}\right)\right) = \bigsqcup_{l \in l^{\#}} m^{\#}[l] \qquad \texttt{alloc}^{\#}\left(\left(n, m^{\#}\right), v^{\#}\right) = \left(n+1, m^{\#} + n \mapsto v^{\#}\right)$$

$$\texttt{set}^{\#}\left(l^{\#}, \left(n, m^{\#}\right), v^{\#}\right) = \left(n, m'^{\#}\right) \text{ where } \begin{cases} m'^{\#}[l] = m^{\#}[l] \sqcup v^{\#} & \text{if } l \in l^{\#} \\ m'^{\#}[l] = m^{\#}[l] & \text{otherwise} \end{cases}$$

Fig. 18. Abstract Interpretation of Extended While

models. Ott can be used to describe static type systems but neither Ott nor Lem has been used to derive program analyses.

Action Semantics [Mosses 1992] was developed by Mosses and Watt as a modular format for writing semantics. Turi and Plotkin [Turi and Plotkin 1997] propose a generic way of defining small step operational semantics, presented in a category theoretic framework. More recently, Churchill emphet al. [Churchill et al. 2015] addresses the issue of the reusability of operational semantics. Their approach is based on structures called *fundamental constructs*, or *funcons*, which only specify the changed parts of the state for a given construct. For instance, the funcon for *if* does not mention environments, but only the Boolean part of the value which is needed. Funcons can then be combined to build a programming language. There is a connection between these funcons and our rules as they are both meant to capture the whole behaviour of a given language construct. One difference is that funcons have a certain degree of sort polymorphism, menaning that *e.g.*, conditional statements and conditional expressions can be treated by the same "if"-funcon. Skeletal semantics would treat each sort separately but would re-use the filters, so as to avoid increasing the proof effort required. To the extent of our knowledge, the work on funcons has been focused on building extendable concrete semantics, and has never been used to build an abstract semantics.

Views [Dinsdale-Young et al. 2013] has a concrete operational semantics for control flow, but is parameterised on the state model and basic commands. It proposes a program logic for this language, which is parameterised on the actions of the basic commands. They prove a general soundness result stating that it suffices to check soundness for each basic command. This corresponds in our framework to the fact that only simple properties on filters need to be checked. Similarly, Keidel *et al.* [Keidel et al. 2018] very recently proposed to capture the similarity between a concrete an abstract interpreter using a shared interpreter parameterised by *arrows* that could be instantiated to concrete and abstract versions, thus reducing the proof effort needed.

Iris [Jung et al. 2017] is a concurrent separation logic framework. It is parameterised by a small-step reduction relation and it proposes a logic to reason about resources. This logic is parameterised by a representation of resources in the form of an algebraic structure called a "camera". Cameras

come with local properties about resources that users have to check. These local constraints yield the soundness of the Iris logic. To be used in practice, Iris requires its users to provides lemmas about weakest preconditions for each language construct. These lemmas are easy to find and prove in simple examples (such as a vanilla While), but they require a deep understanding about how one reasons about the considered language. Such lemmas can be much complex to express (let alone prove) in complex languages such as JavaScript [Gardner et al. 2012]. We believe that our framework guides the proof effort by local reasoning: at each step, the abstract interpretation naturally considers every applicable branch.

The $\mathbb{K}$ framework [Roşu and Şerbănuţă 2010] proposes a formalism for writing operational semantics and for constructing program verifiers directly on top of the semantic definitions, as opposed to using an intermediate representation and/or a verification generator linked to a specific program logic. The semantic rules are given as rewriting rules over terms of semantic state. The $\mathbb{K}$ framework has been used to write semantic definitions of several real-world languages, including C, Java, and JavaScript. The program verifiers are based on matching logic [Roşu 2017], a formalism for reasoning about patterns and the set of terms that they match. A language-independent set of proof rules defines a Reachability Logic which can reason about the set of reachable states of a program. This has been instantiated to obtain program verifiers reasoning about data structures of heap-manipulating programs in C, Java, and JavaScript [Ştefănescu et al. 2016].

The $\mathbb{K}$ framework has goals similar to ours: derive verifiers from operational semantics, correct by construction. A key difference is that the semantics of the $\mathbb{K}$ specification tool is complex and not clearly documented [Li and Gunter 2018]. In this work, we have focused on crystallising a general yet simple rule format. Our format enables a general definition of when a semantics is well-defined and provides a generic correctness theorem for the derived program verifiers that can be machine-checked in the Coq proof assistant.

Schmidt initiated the abstract interpretation of big-step operational semantics [Schmidt 1995] by showing how to abstract derivation trees (using co-induction to harness infinite derivations) and derived classical data flow and control flow analyses as abstract interpretations. Other systematic derivations of static analyses have taken small-step operational semantics as starting point. Schmidt [Schmidt 1997b] discusses the general principles for such an approach and compares small-step and big-step operational semantics as foundations for abstract interpretation. Cousot [Cousot 1999] shows how to derive static analyses for an imperative language defined by a compositional transition semantics using the principles of abstract interpretation. Midtgaard and Jensen [Midtgaard and Jensen 2008] use a similar approach for calculating control-flow analyses for functional languages from operational semantics in the form of abstract machines. Van Horn and Might [Van Horn and Might 2010, 2011] show how a series of analyses for higher-order functional languages can be derived from operational semantics formulated as abstract machines. The atomic operations of the machines are given an abstract interpretation and it is shown that the "abstract abstract machines" can simulate all the transitions of the concrete abstract machine. The abstract machines used by Van Horn and Might can be expressed in our rule format: the atomic operations correspond to our filters and the simulation result corresponds to our consistency result for concrete and abstract interpretations. The two works differ slightly in scope in that we are interested in a general semantic rule format and its meta-theory whereas Van Horn and Might are concerned with giving a systematic derivation of advanced analyses for higher-order languages with state.

Inspired by Schmidt, Bodin *et al.* [Bodin et al. 2015] identify a rule format that can be systematically instantiated to both concrete and abstract semantics, with a generic consistency result. Our work generalises their approach. Their rule format is based on a non-standard style of operational semantics, called pretty-big-step operational semantics [Charguéraud 2013], which cuts up standard big-step rules into many fine-grained rules. In our work, one skeleton describes the behaviour of

one language construct. Our skeletal semantics captures many forms of traditional operational semantics, such as the traditional big-step semantics studied in this paper.

10 CONCLUSIONS AND FUTURE WORK

We have introduced a new meta-language for capturing the behaviour of programming languages, called *skeletal semantics*. A skeleton provides a simple way of describing the complete behaviour of a language construct *in a single definition*. We have given a language-independent, generic definition of *interpretation* of a skeletal semantics, systematically deriving semantic judgements from the skeletons. We have explored four such interpretations: a well-formedness interpretation; a concrete interpretation; an abstract interpretation; and a constraint generator for flow-sensitive analysis. A key advantage of skeletal semantics is that we are able to establish general, language-independent consistency results, which can then be instantiated to specific programming language by proving simple language-dependent filter lemmas.

In this paper, we have focused on proving the fundamental properties of skeletal semantics, using the simple While language and its extensions as illustrative examples. We have demonstrated that we can capture many language constructs including higher-order and object-oriented features. In future, we would like to explore how our formalism scales to real-world languages such as OCaml and JavaScript. For instance, the specification of JavaScript [ECMA 2018] is written in a style where the whole behaviour of each language construct is described in a single definition. It should be comparatively straightforward to provide a specification of JavaScript using a skeletal semantics.

A distinguishing feature of skeletal semantics is that interpretations can be used to characterise several styles of semantics, independently of the language considered. In this paper, we have focused on big-step semantics. In future, we plan to capture other forms of semantics such as small-step operational semantics, semantics for describing concurrent, distributed and interactive computation, and abstract machines, using an approach similar to [Uustalu 2013].

We have interesting proof techniques that are worth further exploration. For example, we have demonstrated how to add an abstract rule for state splitting. The proof technique used to validate this abstract rule, namely that abstract interpretation is a greatest fixpoint, is not specific to state splitting. We thus want to explore other abstract rules validated by this greatest fixpoint. In particular, we conjecture that we can use this approach to obtain a frame rule for skeletal semantics, paving the way for the integration of separation logic as an abstract interpretation. It is also possible to generate better (more precise) constraints than those given in Section 7, as well as constraints for other analyses such as control flow analysis. Based on the skeletal semantics for the λ calculus, we have reproduced the constraint generation for 0-CFA [Palsberg 1995]. We are currently studying how more advanced control flow analyses for other languages can be expressed in our framework.

Finally, we have mechanised in Coq the definitions of skeletal semantics and interpretations, and have proved the general consistency results. We have formalised the well-formedness, concrete and abstract interpretations, verifying that the abstract interpretation for the While language is correct. We have also mechanised a skeletal semantics for the λ calculus. We are currently studying how to leverage this Coq mechanisation to build a certificate checker for abstract analysis.

REFERENCES

Sandrine Blazy and Xavier Leroy. 2009. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning* 43, 3 (2009), 263–288.

Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. 2014. A Trusted Mechanised JavaScript Specification. In *Proc. of the 41st ACM Symposium on Principles of Programming Languages ((POPL 2014))*. ACM, 87–100.

Martin Bodin, Thomas Jensen, and Alan Schmitt. 2015. Certified Abstract Interpretation with Pretty-Big-Step Semantics. In *Proc. of the 2015 ACM Conference on Certified Programs and Proofs (CPP'15)*. ACM, 29–40.

David Cachera, Thomas Jensen, David Pichardie, and Vlad Rusu. 2005. Extracting a data flow analyser in constructive logic. *Theoretical Computer Science* 342, 1 (2005), 56–78.

Arthur Charguéraud. 2013. Pretty-big-step Semantics. In *European Symposium on Programming (ESOP'13)*. Springer LNCS vol. 7792, 41–60.

Martin Churchill, Peter D Mosses, Neil Sculthorpe, and Paolo Torrini. 2015. Reusable Components of Semantic Specifications. In *Transactions on Aspect-Oriented Software Development XII*. Springer, 132–179.

Patrick Cousot. 1999. The Calculational Design of a Generic Abstract Interpreter. In *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam.

Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A unified lattice model for static analysis of programs by construction of approximations of fixpoints. In *Proc. of 4th ACM Symposium on Principles of Programming Languages (POPL '77)*. ACM, 238–252.

Andrei Ştefănescu, Daejun Park, Shijiao Yuwen, Yilong Li, and Grigore Roşu. 2016. Semantics-Based Program Verifiers for All Languages. In *Proc. of the 31th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'16)*. ACM, 74–91.

Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew Parkinson, and Hongseok Yang. 2013. Views: Compositional Reasoning for Concurrent Programs. In *Proc. of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '13)*. ACM, 287–300.

ECMA. 2018. ECMAScript 2018 Language Specification (ECMA-262, 9th edition). https://www.ecma-international.org/ecma-262/9.0/index.html

Philippa Gardner, Sergio Maffeis, and Gareth Smith. 2012. Towards a Program Logic for JavaScript. *ACM SIGPLAN Notices* 47, 1 (2012), 31–44.

Robert Harper, Furio Honsell, and Gordon D. Plotkin. 1987. A Framework for Defining Logics. In *Proc. of the Symposium on Logic in Computer Science (LICS '87)*. 194–204.

Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. 2015. A Formally-Verified C Static Analyzer. In *Proc. of the 42nd ACM Symposium on Principles of Programming Languages (POPL '15)*. ACM, 247–259.

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2017. Iris from the Ground Up. *Submitted to JFP* (2017).

Sven Keidel, Casper Bach Poulsen, and Sebastian Erdweg. 2018. Compositional Soundness Proofs of Abstract Interpreters. *Proc. ACM Program. Lang.* 2, ICFP, Article 72 (July 2018), 26 pages. https://doi.org/10.1145/3236767

Gerwin Klein and Tobias Nipkow. 2002. Verified Bytecode Verifiers. *Theoretical Computer Science* 298, 3 (2002), 583–626.

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: a verified implementation of ML. In *Proc. of 41st Annual ACM Symposium on Principles of Programming Languages, POPL '14*. 179–192.

Xavier Leroy. 2006. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *Proc/ of 33rd ACM Symposium on Principles of Programming Languages (POPL'06)*. ACM, 42–54.

Liyi Li and Elsa L. Gunter. 2018. *IsaK: A Complete Semantics of* $\mathbb{K}$. Technical Report. Computer Science, Univ. of Illinois Urbana-Champaign.

Sergio Maffeis, John C. Mitchell, and Ankur Taly. 2008. An Operational Semantics for JavaScript. In *Proc. of APLAS'08 (LNCS)*, Vol. 5356. 307–325.

Jan Midtgaard and Thomas Jensen. 2008. A Calculational Approach to Control-Flow Analysis by Abstract Interpretation. In *Proc. of 15th Static Analysis Symposium (SAS'08)*. Springer LNCS vol. 5079, 347–362.

Peter D. Mosses. 1992. *Action Semantics*. Cambridge University Press.

Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. 2014. Lem: Reusable Engineering of Real-world Semantics. In *Proc. of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14)*. ACM, 175–188.

Flemming Nielson, Hanne R. Nielson, and Chris Hankin. 1999. *Principles of Program Analysis*. Springer-Verlag.

Michael Norrish. 1998. *C formalised in HOL*. Technical Report UCAM-CL-TR-453. University of Cambridge, Computer Laboratory.

Scott Owens. 2008. A Sound Semantics for OCamllight. In *Proc. of 17th European Symposium on Programming, ESOP 2008*. Springer LNCS vol. 4960, 1–15.

Jens Palsberg. 1995. Closure Analysis in Constraint Form. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 17, 1 (1995), 47–62.

Frank Pfenning and Carsten Schürmann. 1999. System Description: Twelf - A Meta-Logical Framework for Deductive Systems. In *Proc. of 16th International Conference on Automated Deduction (CADE-16)*. 202–206.

Gordon Plotkin. 1981. *A Structural Approach to Operational Semantics*. Technical Report FN-19. DAIMI, Aarhus University.

Grigore Roşu. 2017. Matching Logic. *Logical Methods in Computer Science* 13, 4 (2017), 1–61. https://doi.org/abs/1705.06312

Grigore Roşu and Traian Florin Şerbănuţă. 2010. An Overview of the $\mathbb{K}$ Semantic Framework. *Journal of Logic and Algebraic Programming* 79, 6 (2010), 397–434.

David A. Schmidt. 1995. Natural-semantics-based Abstract Interpretation (preliminary version). In *Proc. of 3rd Static Analysis Symposium (SAS'95)*. Springer LNCS vol. 983, 1–18.

David A. Schmidt. 1997a. Abstract Interpretation in the Operational Semantics Hierarchy. *BRICS Report Series* 4, 2 (1997).

David A. Schmidt. 1997b. Abstract Interpretation of Small-Step Semantics. In *Proc. 5th LOMAPS Workshop on Analysis and Verification of Multiple-Agent Languages*. Springer LNCS vol. 1192, 76–99.

Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok StrniÅąa. 2010. Ott: Effective Tool Support for the Working Semanticist. *Journal of Functional Programming* 20, 1 (2010), 71–122.

Daniele Turi and Gordon Plotkin. 1997. Towards a Mathematical Operational Semantics. In *Proc. of 12th Annual IEEE Symposium on Logic in Computer Science (LICS'97)*. IEEE, 280–291.

Tarmo Uustalu. 2013. Coinductive Big-Step Semantics for Concurrency. In *Proceedings 6th Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, PLACES 2013, Rome, Italy, 23rd March 2013. (EPTCS)*, Nobuko Yoshida and Wim Vanderbauwhede (Eds.), Vol. 137. 63–78. https://doi.org/10.4204/EPTCS.137.6

David Van Horn and Matthew Might. 2010. Abstracting Abstract Machines. In *Proc. of ACM 2010 Int. Conf. on Functional Programming (ICFP'10)*. ACM, 51–62.

David Van Horn and Matthew Might. 2011. Abstracting Abstract Machines: A Systematic Approach to Higher-order Program Analysis. *Commun. ACM* 54, 9 (2011), 101–109.