DOM: Towards a Formal Specification

Philippa Gardner Gareth Smith Mark Wheelhouse Uri Zarfaty

Imperial College

{pg,gds,mjw03,udz}@doc.ic.ac.uk

Abstract

The W3C Document Object Model (DOM) specifies an XML update library. DOM is written in English, and is therefore not compositional and not complete. We provide a first step towards a compositional specification of DOM. Unlike DOM, we are able to work with a minimal set of commands and obtain a complete reasoning for straight-line code. Our work transfers O'Hearn, Reynolds and Yang's local Hoare reasoning for analysing heaps to XML, viewing XML as an in-place memory store as does DOM. In particular, we apply recent work by Calcagno, Gardner and Zarfaty on local Hoare reasoning about a simple tree-update language to DOM, showing that our reasoning scales to DOM. Our reasoning not only formally specifies a significant subset of DOM Core Level 1, but can also be used to verify e.g. invariant properties of simple Javascript programs.

General Terms XML, DOM, local Hoare reasoning, Context Logic

Keywords XML, specification, logical reasoning, verification, locality

1. Introduction

The Document Object Model (DOM) [W3C00] specifies an XML update library, and is maintained by the World Wide Web Consortium (W3C). Its purpose is to be:

> a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents.

A DOM implementation exists in most popular high-level languages, and is used in many applications for accessing and updating XML. For example, consider a webpage with a button labelled 'today's weather'; click on the button and embedded Javascript (using an implementation of DOM) puts 'today's weather' in the tree.

DOM is written in English. It describes the behaviour of individual commands. DOM is not compositional, in the sense that a specification of a composite command cannot be determined directly from the specification of its parts. This means that DOM specifies some redundant composite commands, such as the `replaceChild` command. In this paper, we provide a concise, compositional specification of DOM. Unlike DOM, we are able to work with a minimal set of commands and obtain a complete reasoning for straight-line code. Our work transfers pioneering techniques in local Hoare reasoning for analysing heaps [ORY01] to XML, viewing XML as an in-place memory store as does DOM. In particular, we apply recent work on local Hoare reasoning about a simple tree-update language using Context Logic [CGZ05] to this DOM application, showing that the Context-logic reasoning scales to DOM's more complicated tree structure and update language. Our reasoning not only formally specifies DOM, but can also be used to verify, for example, simple Javascript programs.

The Document Object Model

The documentation for DOM is substantial [W3C05]. DOM is divided into a number of levels, of which the Level 1 is the most fundamental. The Level 1 specification is itself separated into two parts: Core, which 'provides a low-level set of fundamental interfaces that can represent any structured document'; and HTML, which 'provides additional, higher-level interfaces... to provide a more convenient view of an HTML document'. We are only interested in the fundamental interfaces in DOM Core Level 1. In Section 1.1.4 of the DOM Specification, we read:

> The DOM Core APIs present two somewhat different sets of interfaces to an XML/HTML document; one presenting an 'object-oriented' approach with a hierarchy of inheritance, and a 'simplified' view that allows all manipulation to be done via the Node interface

We work with the Node interface. We make a further simplification, concentrating on that part of DOM Core Level 1 which focuses on the XML tree structure, rather than also working with the content of the structure. The main conceptual difficulties lie with this tree structure; in DOM, the other structures (attributes, text, etc) are presented as tree nodes with simpler properties. We will extend our specification to the full DOM Core Level 1 in future.

The fact that DOM is written in English means that understanding the precise conditions under which a command applies is error prone. This is significant, since the DOM approach only works if a DOM implementation really does conform with the specification. E.g., the command `appendChild` has the DOM specification

```
appendChild Adds the node newChild to the end of the list of
            children to this node. If the newChild is already
            in the tree, it is first removed.
....
Exceptions ...
  HIERARCHY_REQUEST_ERR:   Raised if this node is of a type
  that does not allow children of type of the newChild node,
  or if the node to append is one of this node's ancestors.
```

This DOM specification first gives the intuition regarding the behaviour of the method, and then reinforces this intuition with details about when it does not work, such as when `newChild` is an ancestor of the node in question. This fundamental safety condition is buried inside one of several exceptions associated with the method, and is easy to miss.

Plan-X '08 9 January 2008, San Francisco.

We observed that this safety error condition had been missed by the Python mini-DOM implementation [Smi06][Whe07]; Jason Orendorff has recently provided a patch which corrects this error [Ore07]. Section 8.7.3 of the documentation for Python mini-DOM [Var06] states: 'DOMException is currently not supported in xml.dom.minidom. Instead, xml.dom.minidom uses standard Python exceptions such as TypeError and AttributeError'. This is a perfectly sensible design decision, especially since DOM actively encourages this approach to reporting errors: 'error conditions may be indicated using native error reporting mechanisms'. However, it meant that the programmers understandably did not pay close attention to the HIERARCHY_REQUEST_ERR above: the part of the error involving typing is covered by Python exceptions; the part stating that `newChild` cannot be an ancestor is not covered by Python exceptions and was ignored. This meant that the operation silently went ahead, creating a structure with a loop. If the loop structure was used subsequently by a program, then the program would diverge. With our style of local reasoning, this fundamental error in the basic behaviour of update would have been avoided.

Local Hoare Reasoning

We give a compositional specification of DOM, using *local* Hoare reasoning which provides a recent breakthrough in reasoning about the way programs manipulate the memory. Researchers previously used Hoare reasoning based on First-order Logic to specify how programs interacted with the *whole* memory. O'Hearn, Reynolds and Yang instead introduced local Hoare reasoning based on Separation Logic [ORY01]. The idea is to specify how programs interact with a small part of the memory touched by a program. Their work is proving to be essential for modular reasoning about large programs [BCC$^+$07], and for reasoning about concurrent, distributed programming [O'H05]. Inspired by this work, Calcagno, Gardner and Zarfaty studied local Hoare reasoning about simple tree update using Context Logic [CGZ05]. Local data update typically identifies the portion of data to be replaced, removes it, and inserts new data in the same place. Context Logic reasons about both data and this place of insertion (contexts).

Consider the command '`appendChild (parent, newChild)`', which moves the tree at `newChild` to be the last child of `parent`. The command only succeeds when the trees `parent` and `newChild` are present in the store, and when `newChild` is not an ancestor of parent. This safety property is expressible in Context Logic by

$$\exists \texttt{tag}, \texttt{tag'}, \texttt{fid}, \texttt{fid'}. \begin{array}{l}(\varnothing_{\mathrm{F}} \multimap (\mathfrak{C} \circ_{\mathrm{T}} (\texttt{tag}_{\texttt{parent}}[\texttt{f}]_{\texttt{fid}}))) \circ_{\mathrm{F}} \\ \langle \texttt{tag'}_{\texttt{newChild}}[\texttt{f'}]_{\texttt{fid'}} \rangle_{\mathrm{F}}\end{array}$$

This formula states that the data structure can be split into two disjoint parts: a subtree satisfying data formula $\langle \texttt{tag'}_{\texttt{newChild}}[\texttt{f'}]_{\texttt{fid'}} \rangle_{\mathrm{F}}$ stating that the top node is `newChild`, and a context satisfying context formula $\varnothing_{\mathrm{F}} \multimap (\mathfrak{C} \circ_{\mathrm{T}} (\texttt{tag}_{\texttt{parent}}[\texttt{f}]_{\texttt{fid}}))$ stating that, when the empty forest $\varnothing_{\mathrm{F}}$ is put in the hole, then the store can be split into a context and a subtree with top node `parent`. Thus, both `newChild` and `parent` are in the tree, and `newChild` is not an ancestor of `parent`. The corresponding post-condition is

$$\exists \texttt{tag}, \texttt{tag'}, \texttt{fid}, \texttt{fid'}. \mathfrak{C} \circ_{\mathrm{T}} (\texttt{tag}_{\texttt{parent}}[\texttt{f} \otimes \langle \texttt{tag'}_{\texttt{newChild}}[\texttt{f'}]_{\texttt{fid'}} \rangle_{\mathrm{F}}]_{\texttt{fid}})$$

This formula states that the resulting store has a changed subtree `parent`, which now has the subtree $\langle \texttt{tag'}_{\texttt{newChild}}[\texttt{f'}]_{\texttt{fid'}} \rangle_{\mathrm{F}}$ at the end of its list of children.

The fact that our specification is compositional has significant implications. We are able to focus on a minimal set of update commands, whereas DOM has to specify all the update commands for which a specification is useful. For example, the specification of the command `getPreviousSibling` is derivable in our specification, but is specified directly in DOM. We are able to show that our specification is complete for straight-line code, using a standard technique of deriving the weakest preconditions of our commands. For example, the command `insertAfter`, which inserts a subtree after a specific child node, is not specified in DOM, even though its sister command `insertBefore` is. `insertAfter` can be implemented using `insertBefore`, and hence can by specified by our reasoning. Finally, we can verify invariant properties of Javascript programs. For example, we show that a simple program for moving a person to a new address in an address book satisfies an XML-schema invariant specifying that an XML-document is an address book.

2. Minimal DOM

We describe Minimal DOM, a language which captures the essence of DOM for tree update. DOM is specified in an object-oriented manner, and hence encapsulates data and behaviour into objects. In this paper we separate the concerns, by presenting an abstract data structure and a series of commands over that structure. This approach is consistent with DOM's "simplified' view that allows all manipulation to be done via the Node interface'.

2.1 The Tree Structure

Recall from the introduction that we focus on the fundamental XML-tree structure, rather than the content of that structure (text, attributes, etc). We present an abstract data structure consisting of trees, forests and groves. Trees correspond to (part of) the Node interface in DOM. Forests correspond to the sub-collections of the NodeList interface in DOM, while complete forests with identifiers correspond directly to the NodeList interface. Groves correspond to the object store in which Nodes exist.

Definition 2.1 (Trees, forests and groves)**.** Given an infinite set TAG of node tags and ID of node identifiers, we define trees $\mathbf{t} \in \mathrm{T}$, forests $\mathbf{f} \in \mathrm{F}$ and groves $\mathbf{g} \in \mathrm{G}$ by

$$\begin{array}{rrl} \text{trees} & \mathbf{t} & ::= \mathbf{tag}_{\mathbf{id}}[\mathbf{f}]_{\mathbf{fid}} \\ \text{forests} & \mathbf{f} & ::= \varnothing_{\mathrm{F}} \mid \langle \mathbf{t} \rangle_{\mathrm{F}} \mid \mathbf{f} \otimes \mathbf{f} \\ \text{groves} & \mathbf{g} & ::= \varnothing_{\mathrm{G}} \mid \langle \mathbf{t} \rangle_{\mathrm{G}} \mid \mathbf{g} \oplus \mathbf{g} \end{array}$$

where $\mathbf{tag} \in$ TAG and $\mathbf{id}, \mathbf{fid} \in$ ID. For well-formedness, the IDs must be unique. There is also a structural congruence stating that $\otimes$ is associative with identity $\varnothing_{\mathrm{F}}$, and that $\oplus$ is associative and commutative with identity $\varnothing_{\mathrm{G}}$. We write $|\mathbf{f}|$ and $|\mathbf{g}|$ for the length of a forest or the size of a grove respectively.

Since we update our data in place (as does DOM), we must refer to subdata directly. To this end, each node and list of children has a unique identifier which may be directly referenced by a program using program variables. For example, the XML structure $\langle \texttt{html} \rangle \langle \texttt{head} \rangle \langle \texttt{/head} \rangle \langle \texttt{body} \rangle \langle \texttt{/body} \rangle \langle \texttt{/html} \rangle$ is given by the tree $\mathbf{html}_{\mathbf{id}_1}[\langle \mathbf{head}_{\mathbf{id}_2}[\varnothing_{\mathrm{F}}]_{\mathbf{fid}_2} \rangle_{\mathrm{F}} \otimes \langle \mathbf{body}_{\mathbf{id}_3}[\varnothing_{\mathrm{F}}]_{\mathbf{fid}_3} \rangle_{\mathrm{F}}]_{\mathbf{fid}_1}$. Notice that we do not give identifiers to arbitrary forests $\mathbf{f}$, only to complete lists $[\mathbf{f}]_{\mathbf{fid}}$, as in DOM.

We also define natural contexts associated with our abstract data structures. Contexts are not used in DOM. They are however useful for describing the formal operational semantics of our update commands in Section 2.2, and are essential for the context reasoning described in Section 3.

Definition 2.2 (Contexts)**.** Given a infinite set TAG of node tags and ID of node identifiers, we define tree contexts $\mathfrak{C}\mathbf{t} \in \mathrm{CT}$, forest contexts $\mathfrak{C}\mathbf{f} \in \mathrm{CF}$ and grove contexts $\mathfrak{C}\mathbf{g} \in \mathrm{CG}$ by

$$\begin{array}{rrl} \text{tree contexts} & \mathfrak{C}\mathbf{t} & ::= -_{\mathrm{T}} \mid \mathbf{tag}_{\mathbf{id}}[\mathfrak{C}\mathbf{f}]_{\mathbf{fid}} \\ \text{forest contexts} & \mathfrak{C}\mathbf{f} & ::= -_{\mathrm{F}} \mid \langle \mathfrak{C}\mathbf{t} \rangle_{\mathrm{F}} \mid \mathfrak{C}\mathbf{f} \otimes \mathbf{f} \mid \mathbf{f} \otimes \mathfrak{C}\mathbf{f} \\ \text{grove contexts} & \mathfrak{C}\mathbf{g} & ::= -_{\mathrm{G}} \mid \langle \mathfrak{C}\mathbf{t} \rangle_{\mathrm{G}} \mid \mathfrak{C}\mathbf{g} \oplus \mathbf{g} \end{array}$$

As in definition 2.1, the identifiers are unique and there is a natural structural congruence on the contexts.

Given data types $\mathrm{D}_1, \mathrm{D}_2 \in \{\mathrm{T}, \mathrm{F}, \mathrm{G}\}$, we sometimes write $\mathbf{cd}{:}\mathrm{D}_1 \to \mathrm{D}_2$ to denote a context $\mathbf{cd} \in \mathbf{C}\mathrm{D}_2$ with hole $-_{\mathrm{D}_1}$. We call $\mathrm{D}_1 \to \mathrm{D}_2$ the context type of $\mathbf{cd}$. The DOM context structure is quite complex compared with our previous work on a simple tree structure which had one hole type: tree and forest contexts have tree and forest holes, while grove contexts have holes of arbitrary type. Notice that a forest hole of a grove context must have a parent node, whereas this is not the case for a tree or grove hole. The distinction between the tree $\mathbf{t}$, the forest $\langle \mathbf{t} \rangle_{\mathrm{F}}$ and the grove $\langle \mathbf{t} \rangle_{\mathrm{G}}$ is thus important for our context reasoning.

We define the partial application function $\mathrm{ap} : (\mathrm{D}_1 \to \mathrm{D}_2) \times \mathrm{D}_1 \rightharpoonup \mathrm{D}_2$, which returns a result if there is no clash of identifiers between the arguments of the function.

Definition 2.3 (Context application)**.** Given data types $\mathrm{D}_1, \mathrm{D}_2 \in \{\mathrm{T}, \mathrm{F}, \mathrm{G}\}$, we define the partial application function $\mathrm{ap} : (\mathrm{D}_1 \to \mathrm{D}_2) \times \mathrm{D}_1 \rightharpoonup \mathrm{D}_2$, which is defined by induction on the structure of the first argument:

$$
\begin{array}{ll}
\mathrm{ap}(-_{\mathrm{T}}, \mathbf{t}) & \triangleq \mathbf{t} \\
\mathrm{ap}(\mathbf{tag}_{\mathbf{id}}[\mathbf{cf}]_{\mathbf{fid}}, \mathbf{d}_1) & \triangleq \mathbf{tag}_{\mathbf{id}}[\mathrm{ap}(\mathbf{cf}, \mathbf{d}_1)]_{\mathbf{fid}} \quad \text{if } \mathbf{id}, \mathbf{fid} \notin \mathbf{d}_1 \\
\mathrm{ap}(-_{\mathrm{F}}, \mathbf{f}) & \triangleq \mathbf{f} \\
\mathrm{ap}(\langle \mathbf{ct} \rangle_{\mathrm{F}}, \mathbf{d}_1) & \triangleq \langle \mathrm{ap}(\mathbf{ct}, \mathbf{d}_1) \rangle_{\mathrm{F}} \\
\mathrm{ap}(\mathbf{cf} \otimes \mathbf{f}, \mathbf{d}_1) & \triangleq \mathrm{ap}(\mathbf{cf}, \mathbf{d}_1) \otimes \mathbf{f} \\
\mathrm{ap}(\mathbf{f} \otimes \mathbf{cf}, \mathbf{d}_1) & \triangleq \mathbf{f} \otimes \mathrm{ap}(\mathbf{cf}, \mathbf{d}_1) \\
\mathrm{ap}(-_{\mathrm{G}}, \mathbf{g}) & \triangleq \mathbf{g} \\
\mathrm{ap}(\langle \mathbf{ct} \rangle_{\mathrm{G}}, \mathbf{d}_1) & \triangleq \langle \mathrm{ap}(\mathbf{ct}, \mathbf{d}_1) \rangle_{\mathrm{G}} \\
\mathrm{ap}(\mathbf{cg} \oplus \mathbf{g}, \mathbf{d}_1) & \triangleq \mathrm{ap}(\mathbf{cg}, \mathbf{d}_1) \oplus \mathbf{g}
\end{array}
$$

We use $\mathrm{ap}(\mathbf{cd}_2, \mathbf{d}_1)\downarrow$ to denote that $\mathrm{ap}(\mathbf{cd}_2, \mathbf{d}_1)$ is defined.

As normal for in-place update, our language depends on a variable store and expressions. The store s includes variables of type ID, TAG, $\mathbb{Z}$ and $\mathbb{B}$. The expressions consist of variables and constants, arithmetic operations on integers, and logical operations on booleans. Variables of type ID also permit a **null** value, recording the absence of a node (for example, the top node of a tree in the grove has no parent node). Expressions of type ID only include variables and the **null** value, since programs do not refer directly to identifier constants, just as with standard imperative programs which do not refer to literal heap addresses.

Definition 2.4 (Variable store)**.** The variable store s is a total function sending variables to their values. The store contains four types of variable: id variables $\mathrm{Var}_{\mathrm{ID}} = \{\mathtt{id}, \mathtt{fid}, \mathtt{node}, \mathtt{list}, \dots\}$, tag variables $\mathrm{Var}_{\mathrm{TAG}} = \{\mathtt{tag}, \dots\}$, integer variables $\mathrm{Var}_{\mathbb{Z}} = \{\mathtt{int}, \mathtt{length}, \dots\}$ and boolean variables $\mathrm{Var}_{\mathbb{B}} = \{\mathtt{bool}, \dots\}$:

$$
\begin{array}{l}
s : (\mathrm{Var}_{\mathrm{ID}} \to \mathrm{ID} \uplus \{\mathbf{null}\}) \times (\mathrm{Var}_{\mathrm{TAG}} \to \mathrm{TAG}) \times \\
\quad (\mathrm{Var}_{\mathbb{Z}} \to \mathbb{Z}) \times (\mathrm{Var}_{\mathbb{B}} \to \mathbb{B})
\end{array}
$$

The notation $\mathrm{VAR}_{\mathrm{STORE}}$ denotes the set of store variables.

Definition 2.5 (Expressions)**.** Id expressions $\mathrm{Exp}_{\mathrm{ID}} = \{\mathtt{Id}, \dots\}$, tag expressions $\mathrm{Exp}_{\mathrm{TAG}} = \{\mathtt{Tag}, \dots\}$, integer expressions $\mathrm{Exp}_{\mathbb{Z}} = \{\mathtt{Int}, \dots\}$ and boolean expressions $\mathrm{Exp}_{\mathbb{B}} = \{\mathtt{Bool}, \dots\}$ are defined by:

$$
\begin{array}{rl}
\mathtt{Id} ::= & \mathbf{null} \mid \mathtt{id} \\
\mathtt{Tag} ::= & \mathbf{tag} \mid \mathtt{tag} \\
\mathtt{Int} ::= & \mathbf{n} \mid \mathtt{int} \mid \mathtt{Int} + \mathtt{Int} \mid \mathtt{Int} - \mathtt{Int} \\
\mathtt{Bool} ::= & \mathbf{false} \mid \mathtt{bool} \mid \mathtt{Bool} \Rightarrow \mathtt{Bool} \\
& \mid \mathtt{Id} = \mathtt{Id} \mid \mathtt{Tag} = \mathtt{Tag} \mid \mathtt{Int} = \mathtt{Int} \mid \mathtt{Int} > \mathtt{Int}
\end{array}
$$

where $\mathbf{tag} \in \mathrm{TAG}$ and $\mathbf{n} \in \mathbb{Z}$, $\mathtt{int} \in \mathrm{Var}_{\mathbb{Z}}$ and $\mathtt{bool} \in \mathrm{Var}_{\mathbb{B}}$. The evaluation $[\![\mathtt{Exp}_{\mathrm{V}}]\!]s$ of an expression $\mathtt{Exp}_{\mathrm{V}}$ on a store s, for $\mathrm{V} \in \{\mathrm{ID}, \mathrm{TAG}, \mathbb{Z}, \mathbb{B}\}$, is defined as expected.

2.2 The Language

We now introduce Minimal DOM, which represents the essence of the Node interface view of the DOM API in a minimal and sufficient update language. In the spirit of presenting an imperative ('flattened') interface to the object-oriented library, we abandon object-oriented notation. Hence, we specify the methods of the Node interface as imperative commands over a shared grove: for example, the method call '$p.\mathtt{appendChild}(c)$' becomes the command '$\mathtt{appendChild}(p, c)$'. Similarly, we represent object attributes as a pair of get and set commands, with the set command omitted if the attribute is read only. As it turns out, all the relevant Node and NodeList attributes are read only: for example, the '$n.\mathtt{parentNode}$' attribute can be represented by the '$\mathtt{getParentNode}(n)$' command alone. Some attributes and methods in the Node interface are omitted from Minimal DOM since they are concerned with only the content of the tree and not the tree structure itself. Others are omitted because they are redundant, in that they may be expressed as the composition of other commands. Finally, neither the Node nor the NodeList interface provide a means of introducing new Nodes into the grove. For this functionality, we introduce the Minimal DOM command `createNode`, which performs the same function as the DOM Document method createElement, in our minimal environment.

In order to reason about programs which use the Minimal DOM *library*, we also require a Minimal DOM *language* for those programs to be written in. Our language is as simple and general as possible, consisting only of imperative sequencing, conditionals, while loops and the variables and expressions defined in Section 2.1. For convenience, we also implicitly assume procedural recursion.

Definition 2.6 (Minimal DOM)**.** The Minimal DOM commands are

$\mathtt{C} ::= \mathtt{appendChild}(\mathtt{parent}, \mathtt{newChild})$	append tree
$\mid \mathtt{removeChild}(\mathtt{parent}, \mathtt{oldChild})$	remove child
$\mid \mathtt{tag} := \mathtt{getNodeName}(\mathtt{node})$	get node name
$\mid \mathtt{id} := \mathtt{getParentNode}(\mathtt{node})$	get parent node
$\mid \mathtt{fid} := \mathtt{getChildNodes}(\mathtt{node})$	get child nodes
$\mid \mathtt{node} := \mathtt{createNode}(\mathtt{Tag})$	create node
$\mid \mathtt{node} := \mathtt{item}(\mathtt{list}, \mathtt{Int})$	get forest node
$\mid \mathtt{id} := \mathtt{Id} \mid \mathtt{tag} := \mathtt{Tag} \mid \mathtt{int} := \mathtt{Int} \mid \mathtt{bool} := \mathtt{Bool}$	assignment
$\mid \mathtt{C}\ ;\ \mathtt{C}$	sequencing
$\mid \mathtt{if\ Bool\ then\ C\ else\ C}$	if-then-else
$\mid \mathtt{while\ Bool\ do\ C}$	while-do
$\mid \mathtt{skip}$	skip

The DOM commands have the following behaviour:

`appendChild(parent, newChild)` moves tree `newChild` from its current position to the end of `parent`'s child list. Requires that `parent` exists and that `newChild` exists and is not an ancestor of `parent`.

`removeChild(parent, oldChild)` removes the tree `oldChild` from the tree `parent`'s child forest and re-inserts it at the root of the grove. Requires that `parent` exists and `oldChild` is a child of `parent`.

`name := getNodeName(node)` assigns to the variable `name` the nodeName value of `node`.

`id := getParentNode(node)` assigns to the variable `id` the id of the parent of `node`, if it exists, and **null** otherwise.

`fid := getChildNodes(node)` assigns to the variable `fid` the id of the child forest of the element `node`.

`node := createNode(Tag)` creates a new element, with fresh `id` and `fid`, at the root of the grove, with a name equal to `Tag`, and records its id in the variable `node`.

$$\frac{\mathbf{g} \equiv \mathrm{ap}(\mathbf{g'}, \langle \mathbf{tag'}_{s(\texttt{newChild})}[\mathbf{f'}]_{\mathbf{fid'}} \rangle_{\mathrm{G}}) \quad \mathrm{ap}(\mathbf{g'}, \varnothing_{\mathrm{G}}) \equiv \mathrm{ap}(\mathbf{g}, \mathbf{tag}_{s(\texttt{parent})}[\mathbf{f}]_{\mathbf{fid}}) \quad \mathbf{g'} \equiv \mathrm{ap}(\mathbf{g}, \mathbf{tag}_{s(\texttt{parent})}[\mathbf{f} \otimes \langle \mathbf{tag'}_{s(\texttt{newChild})}[\mathbf{f'}]_{\mathbf{fid'}} \rangle_{\mathrm{F}}]_{\mathbf{fid}})}{\texttt{appendChild}(\texttt{parent}, \texttt{newChild}), s, \mathbf{g} \rightsquigarrow s, \mathbf{g'}}$$

$$\frac{\mathbf{g} \equiv \mathrm{ap}(\mathbf{g'}, \langle \mathbf{tag'}_{s(\texttt{newChild})}[\mathbf{f'}]_{\mathbf{fid'}} \rangle_{\mathrm{F}}) \quad \mathrm{ap}(\mathbf{g'}, \varnothing_{\mathrm{F}}) \equiv \mathrm{ap}(\mathbf{g}, \mathbf{tag}_{s(\texttt{parent})}[\mathbf{f}]_{\mathbf{fid}}) \quad \mathbf{g'} \equiv \mathrm{ap}(\mathbf{g}, \mathbf{tag}_{s(\texttt{parent})}[\mathbf{f} \otimes \langle \mathbf{tag'}_{s(\texttt{newChild})}[\mathbf{f'}]_{\mathbf{fid'}} \rangle_{\mathrm{F}}]_{\mathbf{fid}})}{\texttt{appendChild}(\texttt{parent}, \texttt{newChild}), s, \mathbf{g} \rightsquigarrow s, \mathbf{g'}}$$

$$\frac{\mathbf{g} \equiv \mathrm{ap}(\mathbf{g}, \mathbf{tag}_{s(\texttt{parent})}[\mathbf{f}_1 \otimes \langle \mathbf{tag'}_{s(\texttt{oldChild})}[\mathbf{f}]_{\mathbf{fid}} \rangle_{\mathrm{F}} \otimes \mathbf{f}_2]_{\mathbf{fid'}}) \quad \mathbf{g'} \equiv \mathrm{ap}(\mathbf{g}, \mathbf{tag}_{s(\texttt{parent})}[\mathbf{f}_1 \otimes \mathbf{f}_2]_{\mathbf{fid'}}) \oplus \langle \mathbf{tag'}_{s(\texttt{oldChild})}[\mathbf{f}]_{\mathbf{fid}} \rangle_{\mathrm{G}}}{\texttt{removeChild}(\texttt{parent}, \texttt{oldChild}), s, \mathbf{g} \rightsquigarrow s, \mathbf{g'}}$$

$$\frac{\mathbf{g} \equiv \mathrm{ap}(\mathbf{g}, \mathbf{tag}_{s(\texttt{node})}[\mathbf{f}]_{\mathbf{fid}})}{\texttt{tag} := \texttt{getNodeName}(\texttt{node}), s, \mathbf{g} \rightsquigarrow [s|\texttt{tag} \leftarrow \mathbf{tag}], \mathbf{g}}$$

$$\frac{\mathbf{g} \equiv \mathrm{ap}(\mathbf{g}, \mathbf{tag}_{\mathbf{id}}[\mathbf{f}_1 \otimes \langle \mathbf{tag'}_{s(\texttt{node})}[\mathbf{f}]_{\mathbf{fid'}} \rangle_{\mathrm{F}} \otimes \mathbf{f}_2]_{\mathbf{fid}})}{\texttt{id} := \texttt{getParentNode}(\texttt{node}), s, \mathbf{g} \rightsquigarrow [s|\texttt{id} \leftarrow \mathbf{id}], \mathbf{g}} \quad \frac{\mathbf{g} \equiv \mathrm{ap}(\mathbf{g}, \langle \mathbf{tag'}_{s(\texttt{node})}[\mathbf{f}]_{\mathbf{fid'}} \rangle_{\mathrm{G}})}{\texttt{id} := \texttt{getParentNode}(\texttt{node}), s, \mathbf{g} \rightsquigarrow [s|\texttt{id} \leftarrow \mathbf{null}], \mathbf{g}}$$

$$\frac{\mathbf{g} \equiv \mathrm{ap}(\mathbf{g}, \mathbf{tag}_{s(\texttt{node})}[\mathbf{f}]_{\mathbf{fid}})}{\texttt{fid} := \texttt{getChildNodes}(\texttt{node}), s, \mathbf{g} \rightsquigarrow [s|\texttt{fid} \leftarrow \mathbf{fid}], \mathbf{g}} \quad \frac{[\![\texttt{Tag}]\!]s = \mathbf{tag} \quad \mathbf{g'} \equiv \mathbf{g} \oplus \langle \mathbf{tag}_{\mathbf{node}}[\varnothing_{\mathrm{F}}]_{\mathbf{fid}} \rangle_{\mathrm{G}} \quad \mathbf{node}, \mathbf{fid} \text{ fresh}}{\texttt{node} := \texttt{createNode}(\texttt{Tag}), s, \mathbf{g} \rightsquigarrow [s|\texttt{node} \leftarrow \mathbf{node}], \mathbf{g'}}$$

$$\frac{\mathbf{g} \equiv \mathrm{ap}(\mathbf{g}, \mathbf{tag}_{\mathbf{id}}[\mathbf{f}_1 \otimes \langle \mathbf{tag'}_{\mathbf{node}}[\mathbf{f}]_{\mathbf{fid}} \rangle_{\mathrm{F}} \otimes \mathbf{f}_2]_{s(\texttt{list})}) \quad \mathrm{len}(\mathbf{f}_1) = [\![\texttt{Int}]\!]s}{\texttt{node} := \texttt{item}(\texttt{list}, \texttt{Int}), s, \mathbf{g} \rightsquigarrow [s|\texttt{node} \leftarrow \mathbf{node}], \mathbf{g}} \quad \frac{\mathbf{g} \equiv \mathrm{ap}(\mathbf{g}, \mathbf{tag}_{\mathbf{id}}[\mathbf{f}]_{s(\texttt{list})}) \quad [\![\texttt{Int}]\!]s \geq \mathrm{len}(\mathbf{f}) \vee [\![\texttt{Int}]\!]s < 0}{\texttt{node} := \texttt{item}(\texttt{list}, \texttt{Int}), s, \mathbf{g} \rightsquigarrow [s|\texttt{node} \leftarrow \mathbf{null}], \mathbf{g}}$$

$$\frac{[\![\texttt{Id}]\!]s = \mathbf{id}}{\texttt{id}{:=}\texttt{Id}, s, \mathbf{g} \rightsquigarrow [s|\texttt{id} \leftarrow \mathbf{id}], \mathbf{g}} \quad \frac{[\![\texttt{Tag}]\!]s = \mathbf{tag}}{\texttt{tag}{:=}\texttt{Tag}, s, \mathbf{g} \rightsquigarrow [s|\texttt{tag} \leftarrow \mathbf{tag}], \mathbf{g}} \quad \frac{[\![\texttt{Int}]\!]s = \mathbf{n}}{\texttt{int}{:=}\texttt{Int}, s, \mathbf{g} \rightsquigarrow [s|\texttt{int} \leftarrow \mathbf{n}], \mathbf{g}} \quad \frac{[\![\texttt{Bool}]\!]s = \mathbf{b}}{\texttt{bool}{:=}\texttt{Bool}, s, \mathbf{g} \rightsquigarrow [s|\texttt{bool} \leftarrow \mathbf{b}], \mathbf{g}}$$

$$\frac{\mathtt{C}_1, s, \mathbf{g} \rightsquigarrow \mathtt{C}', s', \mathbf{g'}}{(\mathtt{C}_1 \,;\, \mathtt{C}_2), s, \mathbf{g} \rightsquigarrow (\mathtt{C}' \,;\, \mathtt{C}_2), s', \mathbf{g'}} \quad \frac{\mathtt{C}_1, s, \mathbf{g} \rightsquigarrow s', \mathbf{g'}}{(\mathtt{C}_1 \,;\, \mathtt{C}_2), s, \mathbf{g} \rightsquigarrow \mathtt{C}_2, s', \mathbf{g'}} \quad \frac{[\![\texttt{Bool}]\!]s = \mathbf{true}}{\texttt{if Bool then } \mathtt{C}_1 \texttt{ else } \mathtt{C}_2, s, \mathbf{g} \rightsquigarrow \mathtt{C}_1, s, \mathbf{g}}$$

$$\frac{[\![\texttt{Bool}]\!]s = \mathbf{false}}{\texttt{if Bool then } \mathtt{C}_1 \texttt{ else } \mathtt{C}_2, s, \mathbf{g} \rightsquigarrow \mathtt{C}_2, s, \mathbf{g}} \quad \frac{[\![\texttt{Bool}]\!]s = \mathbf{true}}{\texttt{while Bool do C}, s, \mathbf{g} \rightsquigarrow (\mathtt{C} \,;\, \texttt{while Bool do C}), s, \mathbf{g}}$$

$$\frac{[\![\texttt{Bool}]\!]s = \mathbf{false}}{\texttt{while Bool do C}, s, \mathbf{g} \rightsquigarrow s, \mathbf{g}} \quad \frac{}{\texttt{skip}, s, \mathbf{g} \rightsquigarrow s, \mathbf{g}} \quad \frac{\text{otherwise}}{\mathtt{C}, s, \mathbf{g} \rightsquigarrow \text{fault}}$$

Figure 1. Minimal DOM Operational Semantics

`node := item(list, Int)` sets the variable `node` to the `Int` + 1th node in the list pointed to by `list`, setting it to **null** if `Int` evaluates to an invalid index.

Notice that `removeChild` does not delete the tree identified by `oldChild`; instead it moves it to the root level of the grove. In fact, there is no way in Minimal DOM to delete data from the grove. This follows the example of DOM, which deliberately declines to specify any destructive memory management methods, so as to leave open the question of whether memory should be manually managed or garbage collected. It is natural therefore to think of programs written in 'pure' Minimal DOM (without destructive memory management extensions) as garbage collected programs.

DOM operations raise exceptions in 'exceptional circumstances, i.e., when an operation is impossible to perform' [DOM Specification, Section 1.2]. Examples include: trying to move a tree into its own subtree (e.g. using `appendChild`); and attempting to use `removeChild` to remove a non-existent subtree of a given tree. Where DOM calls for a DOMException, we raise a fault. This is compatible with the specification, which states that in languages that do not support exceptions, 'error conditions may be indicated using native error reporting mechanisms'.

We now give the formal operational semantics, as well as proof sketches for the minimality and sufficiency of Minimal DOM.

Definition 2.7 (Operational Semantics)**.** The operational semantics of Minimal DOM is given in Figure 1 by an evaluation relation $\rightsquigarrow$ relating configuration triples $\mathtt{C}, s, \mathbf{g}$, terminal states $s, \mathbf{g}$, and faults. $[s \mid x \leftarrow v]$ means the partial function s overwritten with $s(x) = v$.

Theorem 2.8 (Minimality of Minimal DOM)**.** *There is no redundancy in Minimal DOM – each command is necessary.*

Sketch proof. The 7 Minimal DOM commands may be divided into 2 groups: update commands (`appendChild`, `removeChild`, `createNode`) and lookup commands (`getParentNode`, `getChildNodes`, `item`, `getNodeName`). We justify each command in each group in turn.

Update Commands Only `appendChild`, `removeChild` and `createNode` change the state of the grove: `appendChild` and `removeChild` move a tree from one place in the grove to another; `createNode` introduces a new node into the grove. `createNode` is necessary since it is the only command to introduce fresh nodes. `removeChild` is necessary since it is the only command that can move a tree to the top level of the grove (`appendChild` requires that the programmer specify the parent node of the target location and the top of the grove has no such parent). `appendChild` is necessary since `removeChild` cannot move a tree to anywhere other than the top level of the grove.

Lookup Commands Only `getParentNode`, `getChildNodes`, `item` and `getNodeName` communicate information from the grove to the variable store. `getParentNode` is the only command that returns a node closer to the root of the tree. `getChildNodes` is the only command that returns a nodeList. `item` is the only command that returns a node further down the tree. `getNodeName` is the only command that returns a tag. □

Theorem 2.9 (Sufficiency of Minimal DOM)**.** *Minimal DOM is sufficient to describe the structural kernel of DOM Core Level 1.*

Sketch proof. The Node interface contains 9 relevant attributes and methods which are not implemented in Minimal DOM, while the NodeList interface is implemented completely. These missing attributes and methods correspond, in our imperative setting, to the following commands: `insertBefore`, `replaceChild`, `cloneNode`, `hasChildNodes`, `getLength`, `getFirstChild`, `getLastChild`, `getPreviousSibling` and `getNextSibling`. Implementations of each of these commands are given in the full version of this paper. the implementation of `getPreviousSibling` is discussed in Section 5.1. □

3. Context Logic

Context Logic was originally introduced to reason about tree update [CGZ05]. Here we apply Context Logic to to our DOM data structure, and in Section **??** give Local Hoare Reasoning about Minimal DOM based on this Context Logic reasoning.

The Minimal DOM language provides us with integer, reference, boolean and tag variables. In addition to these, we will require tree, forest, grove and context variables in our specifications. Our logic therefore uses a logical environment e as well as the variable store s of Minimal DOM.

Definition 3.1 (Logical environment)**.** A logical environment e is a total function sending data and context variables to their values. The environment contains the following types of environment variable: tree variables $\mathrm{Var_T}$, forest variables $\mathrm{Var_F}$, grove variables $\mathrm{Var_G}$, tree context variables $\mathrm{Var_{R\to T}}$ for $\mathrm{R} \in \{\mathrm{T},\mathrm{F}\}$, forest context variables $\mathrm{Var_{R\to F}}$ for the same R and grove context variables $\mathrm{Var_{D\to G}}$ for $\mathrm{D} \in \{\mathrm{T},\mathrm{F},\mathrm{G}\}$.

$$e : \begin{array}{c} (\mathrm{Var_T}\to\mathrm{T}) \times (\mathrm{Var_F}\to\mathrm{F}) \times (\mathrm{Var_G}\to\mathrm{G}) \times \\ (\mathrm{Var_{T\to T}}\to(\mathrm{T}\to\mathrm{T})) \times (\mathrm{Var_{T\to F}}\to(\mathrm{T}\to\mathrm{F})) \times (\mathrm{Var_{T\to G}}\to(\mathrm{T}\to\mathrm{G})) \times \\ (\mathrm{Var_{F\to T}}\to(\mathrm{F}\to\mathrm{T})) \times (\mathrm{Var_{F\to F}}\to(\mathrm{F}\to\mathrm{F})) \times (\mathrm{Var_{F\to G}}\to(\mathrm{F}\to\mathrm{G})) \times \\ \times\ (\mathrm{Var_{G\to G}}\to(\mathrm{G}\to\mathrm{G})) \end{array}$$

The notation $\mathrm{VAR_{ENV}}$ denotes the set of environment variables.

Context Logic consists of standard formulae constructed from the connectives of first-order logic, variables, expression tests, and quantification over variables. In addition, it has general *structural* formulae and *specific* formulae applicable to DOM. The structural formulae of Context Logic are constructed from an *application* connective for analysing context application, and its two corresponding *right adjoints*: for data types $\mathrm{D_1}, \mathrm{D_2} \in \{\mathrm{T},\mathrm{F},\mathrm{G}\}$,

- the application formula $P \circ_{\mathrm{D_1}} P_1$ describes data of e.g. type $\mathrm{D_2}$ that can be split into a context of type $\mathrm{D_1}{\to}\mathrm{D_2}$ satisfying P and disjoint subdata of type $\mathrm{D_1}$ satisfying P_1; the application connective is annotated with type information about the context hole, since this cannot be determined from the given data;
- one right adjoint $P \circ\!\!-_{\mathrm{D_2}} P_2$ describes data of e.g. type $\mathrm{D_1}$ which, *whenever* it is successfully placed in a context of type $\mathrm{D_1} \to \mathrm{D_2}$ satisfying P, results in data of type $\mathrm{D_2}$ satisfying P_2; the adjoint is annotated with type information about the resulting data, since this cannot be determined from the hole type;
- the right adjoint $P_1 \multimap P_2$ describes a context of e.g. type $\mathrm{D_1} \to \mathrm{D_2}$ which, *whenever* data of type $\mathrm{D_1}$ satisfying P_1 is successfully inserted into it, results in data of type $\mathrm{D_2}$ satisfying P_2; there is no type annotation as it can be inferred from the type of the given data.

Finally, we have model-specific formulae for analysing the tree, forest and grove structure. These correspond directly to the data structure definitions (Defns 2.1 and 2.2): for example, the tree definition $\mathbf{tag_{id}[f]_{fid}}$ corresponds to a tree formula $\mathtt{Tag}_{\mathtt{id}}[P]_{\mathtt{fid}}$, which uses the tag expression `Tag` and id variables `id` and `fid` to describe the node data, and a forest formula P to describe the subforest.

Definition 3.2 (Formulae)**.** Let A denote a data or context type of the form D or $\mathrm{D_1} \to \mathrm{D_2}$ for $\mathrm{D}, \mathrm{D_1}, \mathrm{D_2} \in \{\mathrm{T},\mathrm{F},\mathrm{G}\}$. The set of formulae for DOM are defined by:

$P ::=$

$P \Rightarrow P \mid \mathrm{false_A}$	Boolean formulae
$P \circ_{\mathrm{D_1}} P \mid P \circ\!\!-_{\mathrm{D_2}} P \mid P \multimap P$	structural formulae
$\ldots$ (see below) $\ldots$	DOM-specific formulae
$\mathtt{var_E} \mid \mathtt{Exp_V} = \mathtt{Exp_V}$	expression equality, $\mathtt{var_E} \in \mathrm{VAR_{ENV}}, \mathrm{V} \in \{\mathrm{ID},\mathrm{TAG},\mathbb{Z},\mathbb{B}\}$
$\mathtt{Int} = \mathrm{len}(\mathtt{f}) \mid \mathtt{Int} = \mathrm{len}(\mathtt{s})$	length equality
$\exists \mathtt{var}.\, P$	quantification, $\mathtt{var} \in \mathrm{VAR_{ENV}} \cup \mathrm{VAR_{STORE}}$

The *DOM-specific* formulae are given by:

$$\begin{array}{rl} P ::= \ldots & \mid -_{\mathrm{T}} \mid P_{\mathtt{id}}[P]_{\mathtt{fid}} \\ & \mid \varnothing_{\mathrm{F}} \mid -_{\mathrm{F}} \mid \langle P \rangle_{\mathrm{F}} \mid P \otimes P \\ & \mid \varnothing_{\mathrm{G}} \mid -_{\mathrm{G}} \mid \langle P \rangle_{\mathrm{G}} \mid P \oplus P \end{array}$$

The type annotations on the formulae enable us to define a simple typing relation $P{:}A$, where A is a data or context type, by induction on the structure of formula P. The Boolean formulae and quantified formulae inherit their types from the subformulae. The equalities satisfy arbitrary A, since they are really outside the typing system as they test the store rather than the data and context structures. We give the cases for the structural formulae and for the DOM-specific formulae for trees, and give one forest case; the cases for the other DOM-specific formulae are similar:

$$\begin{array}{rcl} (P_1 \circ_{\mathrm{D_1}} P_2){:}\mathrm{D_2} & \Leftrightarrow & P_1{:}\mathrm{D_1}{\to}\mathrm{D_2} \wedge P_2{:}\mathrm{D_1} \\ (P_1 \circ\!\!-_{\mathrm{D_2}} P_2){:}\mathrm{D_1} & \Leftrightarrow & P_1{:}\mathrm{D_1}{\to}\mathrm{D_2} \wedge P_2{:}\mathrm{D_2} \\ (P_1 \multimap P_2){:}\mathrm{D_1}{\to}\mathrm{D_2} & \Leftrightarrow & P_1{:}\mathrm{D_1} \wedge P_2{:}\mathrm{D_2} \\ \\ -_{\mathrm{T}}{:}\mathrm{T}{\to}\mathrm{T} & & \\ P_{\mathtt{id}}[P']_{\mathtt{fid}}{:}\mathrm{T} & \Leftrightarrow & P{:}\mathrm{S} \wedge P'{:}\mathrm{F} \\ P_{\mathtt{id}}[P']_{\mathtt{fid}}{:}\mathrm{R}{\to}\mathrm{T} & \Leftrightarrow & P{:}\mathrm{S} \wedge P'{:}\mathrm{R}{\to}\mathrm{F} \\ \\ (P_1 \otimes P_2){:}\mathrm{F} & \Leftrightarrow & P_1{:}\mathrm{F} \wedge P_2{:}\mathrm{F} \\ (P_1 \otimes P_2){:}\mathrm{R}{\to}\mathrm{F} & \Leftrightarrow & (P_1{:}\mathrm{R}{\to}\mathrm{F} \wedge P_2{:}\mathrm{F}) \vee (P_1{:}\mathrm{F} \wedge P_2{:}\mathrm{R}{\to}\mathrm{F}) \end{array}$$

where $\mathrm{R} \in \{\mathrm{T},\mathrm{F}\}$ denotes the possible tree or forest holes. The formula $P_{\mathtt{id}}[P']_{\mathtt{fid}}$ has two typings, depending on whether it describes a tree or tree context. The formula $P_1 \otimes P_2$ also has the two typings; since the forest context case has two options for typing the subformulae, depending on which one describes the forest context.

Definition 3.3 (Satisfaction Relation)**.** The satisfaction relation $e, s, \mathbf{a} \models_{\mathrm{A}} P$ is defined on environment e, variable store s, datum or context $\mathbf{a}$ of type A, and formula P of type A by induction on the structure of P:

$$\begin{array}{ll} e,s,\mathbf{a} \models_{\mathrm{A}} P \Rightarrow P' & \Leftrightarrow e,s,\mathbf{a} \models_{\mathrm{A}} P \Rightarrow e,s,\mathbf{a} \models_{\mathrm{A}} P' \\ e,s,\mathbf{a} \models_{\mathrm{A}} \mathrm{false_A} & never \\ e,s,\mathbf{a} \models_{\mathrm{A}} \mathtt{var_E} & \Leftrightarrow \mathbf{a} \equiv e(\mathtt{var_E}) \\ e,s,\mathbf{a} \models_{\mathrm{A}} \mathtt{Exp_V} = \mathtt{Exp'_V} & \Leftrightarrow [\![\mathtt{Exp_V}]\!]s = [\![\mathtt{Exp'_V}]\!]s \\ e,s,\mathbf{a} \models_{\mathrm{A}} \exists \mathtt{var_E}.\, P & \Leftrightarrow \exists \mathbf{b}.\, (e[\mathtt{var_E} \mapsto \mathbf{b}], s, \mathbf{a} \models_{\mathrm{A}} P) \\ e,s,\mathbf{a} \models_{\mathrm{A}} \exists \mathtt{var_V}.\, P & \Leftrightarrow \exists \mathbf{v}.\, (e, s[\mathtt{var_V} \mapsto \mathbf{v}], \mathbf{a} \models_{\mathrm{A}} P) \\ e,s,\mathbf{a} \models_{\mathrm{A}} \mathtt{Int} = \mathrm{len}(\mathtt{f}) & \Leftrightarrow [\![\mathtt{Int}]\!]s = |e(\mathtt{f})| \end{array}$$

for the structural formulae, we have

$$\begin{array}{ll} e,s,\mathbf{d}_2 \models_{\mathrm{D_2}} P_1 \circ_{\mathrm{D_1}} P_2 & \Leftrightarrow \exists \mathbf{cd}{:}(\mathrm{D_1}{\to}\mathrm{D_2}), \mathbf{d}_1{:}\mathrm{D_1}.\, \mathbf{d}_2 = \mathrm{ap}(\mathbf{cd}, \mathbf{d}_1) \\ & \quad \wedge e,s,\mathbf{cd} \models_{\mathrm{D_1}\to\mathrm{D_2}} P_1 \wedge e,s,\mathbf{d}_1 \models_{\mathrm{D_1}} P_2 \\ e,s,\mathbf{d}_1 \models_{\mathrm{D_1}} P_1 \circ\!\!-_{\mathrm{D_2}} P_2 & \Leftrightarrow \forall \mathbf{cd}{:}(\mathrm{D_1}{\to}\mathrm{D_2}).\, (e,s,\mathbf{cd} \models_{\mathrm{D_1}\to\mathrm{D_2}} P_1 \wedge \\ & \quad \mathrm{ap}(\mathbf{cd}, \mathbf{d}_1)\downarrow) \Rightarrow e,s,\mathrm{ap}(\mathbf{cd}, \mathbf{d}_1) \models_{\mathrm{D_2}} P_2 \\ e,s,\mathbf{cd}_2 \models_{\mathrm{D_1}\to\mathrm{D_2}} P_1 \multimap P_2 & \Leftrightarrow \forall \mathbf{d}_1{:}\mathrm{D_1}.\, e,s,\mathbf{d}_1 \models_{\mathrm{D_1}} P_1 \wedge \mathrm{ap}(\mathbf{cd}_2, \mathbf{d}_1)\downarrow \\ & \quad \Rightarrow e,s,\mathrm{ap}(\mathbf{cd}_2, \mathbf{d}_1) \models_{\mathrm{D_2}} P_2 \end{array}$$

and for the model-specific formulae, we have

$$
\begin{array}{ll}
e,s,\mathbf{t} \models_{\mathrm{T}} \mathtt{Tag}_{\mathtt{id}}[P']_{\mathtt{fid}} & \Leftrightarrow \exists \mathbf{f}{:}\mathrm{F}.\ (\mathbf{t} \equiv \mathtt{Tag}_{s(\mathtt{id})}[\mathbf{f}]_{s(\mathtt{fid})}) \wedge \\
& \quad e,s,\mathbf{f} \models_{\mathrm{F}} P' \\
e,s,\mathfrak{t} \models_{\mathrm{T}\to\mathrm{T}} -_{\mathrm{T}} & \Leftrightarrow \mathfrak{t} \equiv -_{\mathrm{T}} \\
e,s,\mathfrak{t} \models_{\mathrm{R}\to\mathrm{T}} \mathtt{tag}_{\mathtt{id}}[P']_{\mathtt{fid}} & \Leftrightarrow \exists \mathfrak{f}{:}(\mathrm{R}\to\mathrm{F}).\ (\mathfrak{t} \equiv \mathtt{Tag}_{s(\mathtt{id})}[\mathfrak{f}]_{s(\mathtt{fid})}) \wedge \\
& \quad e,s,\mathfrak{f} \models_{\mathrm{R}\to\mathrm{F}} P'
\end{array}
$$

$$
\begin{array}{ll}
e,s,\mathbf{f} \models_{\mathrm{F}} \varnothing_{\mathrm{F}} & \Leftrightarrow \mathbf{f} \equiv \varnothing_{\mathrm{F}} \\
e,s,\mathfrak{f} \models_{\mathrm{F}\to\mathrm{F}} -_{\mathrm{F}} & \Leftrightarrow \mathfrak{f} \equiv -_{\mathrm{F}} \\
e,s,\mathbf{f} \models_{\mathrm{F}} \langle P\rangle_{\mathrm{F}} & \Leftrightarrow \exists \mathbf{t}{:}\mathrm{T}.\ (\mathbf{f} \equiv \langle \mathbf{t}\rangle_{\mathrm{F}}) \wedge e,s,\mathbf{t} \models_{\mathrm{T}} P \\
e,s,\mathfrak{f} \models_{\mathrm{R}\to\mathrm{F}} \langle P\rangle_{\mathrm{F}} & \Leftrightarrow \exists \mathfrak{t}{:}(\mathrm{R}\to\mathrm{T}).\ (\mathfrak{f} \equiv \langle \mathfrak{t}\rangle_{\mathrm{F}}) \wedge e,s,\mathfrak{t} \models_{\mathrm{R}\to\mathrm{T}} P \\
e,s,\mathbf{f} \models_{\mathrm{F}} P_1 \otimes P_2 & \Leftrightarrow \exists \mathbf{f}_1{:}\mathrm{F}, \mathbf{f}_2{:}\mathrm{F}.\ (\mathbf{f} \equiv \mathbf{f}_1 \otimes \mathbf{f}_2) \wedge \\
& \quad e,s,\mathbf{f}_1 \models_{\mathrm{F}} P_1 \wedge e,s,\mathbf{f}_2 \models_{\mathrm{F}} P_2 \\
e,s,\mathfrak{f} \models_{\mathrm{R}\to\mathrm{F}} P_1 \otimes P_2 & \Leftrightarrow \exists \mathfrak{f}'{:}(\mathrm{R}\to\mathrm{F}), \mathbf{f}'{:}\mathrm{F}. \\
& \quad \begin{pmatrix} (\mathfrak{f} \equiv \mathfrak{f}' \otimes \mathbf{f}') \wedge \\ e,s,\mathfrak{f}' \models_{\mathrm{R}\to\mathrm{F}} P_1 \wedge e,s,\mathbf{f}' \models_{\mathrm{F}} P_2 \end{pmatrix} \vee \\
& \quad \begin{pmatrix} (\mathfrak{f} \equiv \mathbf{f}' \otimes \mathfrak{f}') \wedge \\ e,s,\mathbf{f}' \models_{\mathrm{F}} P_1 \wedge e,s,\mathfrak{f}' \models_{\mathrm{R}\to\mathrm{F}} P_2 \end{pmatrix}
\end{array}
$$

$$
\begin{array}{ll}
e,s,\mathbf{g} \models_{\mathrm{G}} \varnothing_{\mathrm{G}} & \Leftrightarrow \mathbf{g} \equiv \varnothing_{\mathrm{G}} \\
e,s,\mathfrak{g} \models_{\mathrm{G}\to\mathrm{G}} -_{\mathrm{G}} & \Leftrightarrow \mathfrak{g} \equiv -_{\mathrm{G}} \\
e,s,\mathbf{g} \models_{\mathrm{G}} \langle P\rangle_{\mathrm{G}} & \Leftrightarrow \exists \mathbf{t}{:}\mathrm{T}.\ (\mathbf{g} \equiv \langle \mathbf{t}\rangle_{\mathrm{G}}) \wedge e,s,\mathbf{t} \models_{\mathrm{T}} P \\
e,s,\mathfrak{g} \models_{\mathrm{R}\to\mathrm{G}} \langle P\rangle_{\mathrm{G}} & \Leftrightarrow \exists \mathfrak{t}{:}(\mathrm{R}\to\mathrm{T}).\ (\mathfrak{g} \equiv \langle \mathfrak{t}\rangle_{\mathrm{G}}) \wedge e,s,\mathfrak{t} \models_{\mathrm{R}\to\mathrm{T}} P_{\mathrm{CT}} \\
e,s,\mathbf{g} \models_{\mathrm{G}} P_1 \oplus P_2 & \Leftrightarrow \exists \mathbf{g}_1{:}\mathrm{G}, \mathbf{g}_2{:}\mathrm{G}.\ (\mathbf{g} \equiv \mathbf{g}_1 \oplus \mathbf{g}_2) \wedge \\
& \quad e,s,\mathbf{g}_1 \models_{\mathrm{G}} P_1 \wedge e,s,\mathbf{g}_2 \models_{\mathrm{G}} P_2 \\
e,s,\mathfrak{g} \models_{\mathrm{D}\to\mathrm{G}} P_1 \oplus P_2 & \Leftrightarrow \exists \mathfrak{g}'{:}(\mathrm{D}\to\mathrm{G}), \mathbf{g}{:}\mathrm{G}.\ (\mathfrak{g} \equiv \mathfrak{g}' \oplus \mathbf{g}) \wedge \\
& \quad e,s,\mathfrak{g}' \models_{\mathrm{CG}} P_1 \wedge e,s,\mathbf{g} \models_{\mathrm{G}} P_2
\end{array}
$$

The standard classical connectives are derivable: true, $\wedge$, $\vee$, $\neg$, $\forall$. We introduce notation for expressing 'somewhere (potentially deep down)' ($\Diamond_{\mathrm{D}_1\to\mathrm{D}_2} P$) and everywhere' ($\Box_{\mathrm{D}_1\to\mathrm{D}_2} P$), where $\mathrm{D}_1, \mathrm{D}_2 \in \{\mathrm{T}, \mathrm{F}, \mathrm{G}\}$. Similarly, we define the related concept of 'somewhere at this forest-level' ($\Diamond_{\otimes} P$) and 'everywhere at this forest-level' ($\Box_{\otimes} P$):

$$\Diamond_{\mathrm{D}_1\to\mathrm{D}_2} P \triangleq \mathrm{true}_{\mathrm{D}_1\to\mathrm{D}_2} \circ_{\mathrm{D}_1} P \quad \Diamond_{\otimes} P \triangleq (\mathrm{true}_{\mathrm{F}} \otimes -_{\mathrm{F}} \otimes \mathrm{true}_{\mathrm{F}}) \circ_{\mathrm{F}} P$$
$$\Box_{\mathrm{D}_1\to\mathrm{D}_2} P \triangleq \neg\Diamond_{\mathrm{D}_1\to\mathrm{D}_2} \neg P \quad \Box_{\otimes} P \triangleq \neg\Diamond_{\otimes} \neg P$$

We write $\mathtt{Bool}$ for $\mathtt{Bool} = \mathbf{true}$ and derive:

$$\mathtt{Tag}_{\mathtt{id}}[P] \triangleq \exists \mathtt{fid}.\, \mathtt{Tag}_{\mathtt{id}}[P]_{\mathtt{fid}} \quad \mathtt{Tag}[P] \triangleq \exists \mathtt{id}.\, \mathtt{Tag}_{\mathtt{id}}[P]$$

The order of binding precedence is: $\neg, \circ, \wedge, \vee, \{\circ\!\!-, -\!\!\circ\}$ and $\Rightarrow$, with no precedence between the elements in $\{\circ\!\!-, -\!\!\circ\}$.

Example 3.4 (Context Logic examples)**.** To demonstrate the expressive power of our logic we give some examples:

(a) Two equivalent ways of specifying a tree containing a node with name **a**, but otherwise unconstrained:

$$\exists \mathtt{id}, \mathtt{fid}.\, \mathrm{true}_{\mathrm{T}\to\mathrm{T}} \circ_{\mathrm{T}} (\mathbf{a}_{\mathtt{id}}[\mathrm{true}_{\mathrm{F}}]_{\mathtt{fid}}) \equiv \Diamond_{\mathrm{T}\to\mathrm{T}}(\mathbf{a}[\mathrm{true}_{\mathrm{F}}])$$

(b) A tree consisting of a **body** node with 0 or more **paragraph** nodes underneath:

$$\mathbf{body}[\Box_{\otimes}(\langle \mathrm{true}_{\mathrm{T}}\rangle_{\mathrm{F}} \Rightarrow \langle \mathbf{paragraph}[\mathrm{true}_{\mathrm{F}}]\rangle_{\mathrm{F}})]$$

The $\Box_{\otimes}$ constraint on the forest underneath **body** specifies that all its subforests that satisfy $\langle \mathrm{true}_{\mathrm{T}}\rangle_{\mathrm{F}}$ (in other words, all its subtrees) must also satisfy $\langle \mathbf{paragraph}[\mathrm{true}_{\mathrm{F}}]\rangle_{\mathrm{F}}$. This sort of formula turns out to be particularly useful when describing XML schema invariants in Section 5.3.

(c) A grove:

$$\mathfrak{g} \circ_{\mathrm{T}} (\mathtt{tag}_{\mathtt{id}}[\mathtt{f}]_{\mathtt{fid}})$$

containing a node $\mathtt{tag}_{\mathtt{id}}[\_]_{\mathtt{fid}}$ (described by the store variables `tag`, `id` and `fid`), inside a context $\mathfrak{g}$ and with a subforest `f` (described by the environment variables $\mathfrak{g}$ and `f`). We use this form of exact specification to specify that certain parts of the tree remain unchanged by a command.

(d) A grove:

$$\exists \mathfrak{g}, \mathtt{tag}, \mathtt{tag'}.\, (\varnothing_{\mathrm{X}} -\!\!\circ \mathfrak{g} \circ_{\mathrm{T}} (\mathtt{tag}_{\mathtt{node2}}[\mathrm{true}_{\mathrm{F}}])) \circ_{\mathrm{X}} \langle \mathtt{tag'}_{\mathtt{node1}}[\mathrm{true}_{\mathrm{F}}]\rangle_{\mathrm{X}}$$

containing the nodes `node1` and `node2`, where the node `node1` is not an ancestor of the node `node2`. This is parameterised by $\mathrm{X} \in \{\mathrm{F}, \mathrm{G}\}$ to cover both the case in which `node1` is a root level node ($\mathrm{X} = \mathrm{G}$) and that in which it is the child of some other node ($\mathrm{X} = \mathrm{F}$). This sort of formula occurs in the axiom of `appendChild`.

(e) The weakest precondition of the `appendChild` command:

$$\begin{array}{l}
\exists \mathtt{tag}, \mathtt{tag'}, \mathtt{fid}, \mathtt{fid'}, \mathtt{f}, \mathtt{f'}, \mathfrak{g}. \\
\big((\mathfrak{g} \circ_{\mathrm{T}} (\mathtt{tag}_{\mathtt{parent}}[\mathtt{f} \otimes \langle \mathtt{tag'}_{\mathtt{newChild}}[\mathtt{f'}]_{\mathtt{fid'}}\rangle_{\mathrm{F}}]_{\mathtt{fid}})) -\!\!\circ P\big) \circ_{\mathrm{G}} \\
\big((\varnothing_{\mathrm{X}} -\!\!\circ (\mathfrak{g} \circ_{\mathrm{T}} (\mathtt{tag}_{\mathtt{parent}}[\mathtt{f}]_{\mathtt{fid}}))) \langle \mathtt{tag'}_{\mathtt{newChild}}[\mathtt{f'}]_{\mathtt{fid'}}\rangle_{\mathrm{X}}\big)
\end{array}$$

This formula states that the node `newChild` is not an ancestor of the node `parent`, and that if the node `newChild` is moved to the end of `parent`'s list of children then some postcondition P will hold.

4. Local Hoare Reasoning

We use Context Logic applied to our DOM tree structure to provide local Hoare reasoning about Minimal DOM programs. This is possible because all Minimal DOM commands are local. A command is local if it satisfies two natural properties [IO01]: the *safety-monotonicity* property specifying that, if a command is safe in a given state (i.e., it does not fault), then it is safe in a larger state; and the *frame* property specifying that, if a command is safe in a given state, then any execution of the command on a larger state can be tracked to an execution on the smaller state.

With Minimal DOM, the formal operational semantics for the commands is defined on groves. The commands `appendChild`, `removeChild`, `createNode` do act at the grove level: `appendChild` potentially takes a subtree from one grove tree and appends it to a subtree from another grove tree; the other commands result in new grove trees. However, the commands `getNodeName`, `getChildNodes`, `item` essentially act on specific subtrees identified by the command, rather than at the grove level, and the command `getParentNode` is a hybrid, having different behaviour at the subtree level (where it returns the parent) and the grove level (where it returns **null**). We therefore provide two forms of Hoare triple, depending on whether we are reasoning about trees or groves. We use O'Hearn's fault-avoiding partial correctness interpretation of our local Hoare Triples on groves and trees, which says that if a state satisfies a precondition, then the command cannot fault and the resulting state must satisfy the postcondition.

Definition 4.1 (Local Hoare Triples)**.** Recall the evaluation relation $\rightsquigarrow$ relating configuration triples $\mathtt{C}, s, \mathbf{g}$, terminal states $s, \mathbf{g}$, and faults. The fault-avoiding partial correctness interpretation of local Hoare Triples is given by:

$$
\begin{array}{ll}
\{P\}\ \mathtt{C}\ \{Q\} \Leftrightarrow & (P{:}G \wedge Q{:}G \wedge \\
& \quad \forall e,s,\mathbf{g}.\ e,s,\mathbf{g} \models_{\mathrm{G}} P \Rightarrow \\
& \quad \mathtt{C},s,\mathbf{g} \not\rightsquigarrow \text{fault} \wedge \\
& \quad \forall s',\mathbf{g}'.\, \mathtt{C},s,\mathbf{g} \rightsquigarrow s',\mathbf{g}' \Rightarrow e,s',\mathbf{g}' \models_{\mathrm{G}} Q) \\
& \vee\ (P{:}T \wedge Q{:}T \wedge \\
& \quad \forall e,s,\mathbf{g}.\ e,s,\mathbf{g} \models_{\mathrm{G}} \langle P\rangle_{\mathrm{G}} \Rightarrow \\
& \quad \mathtt{C},s,\mathbf{g} \not\rightsquigarrow \text{fault} \wedge \\
& \quad \forall s',\mathbf{g}'.\, \mathtt{C},s,\mathbf{g} \rightsquigarrow s',\mathbf{g}' \Rightarrow e,s',\mathbf{g}' \models_{\mathrm{G}} \langle Q\rangle_{\mathrm{G}})
\end{array}
$$

Our interpretation of the local Hoare Triples on trees coerces those trees to groves using $\langle\rangle_{\mathrm{G}}$. This is necessary since $\rightsquigarrow$ is defined over configuration triples containing groves.

$$\begin{array}{rcl}
\{\,(\varnothing_{\mathrm{X}} \multimap (\mathtt{qg} \circ_{\mathrm{T}} (\mathtt{tag}_{\mathtt{parent}}[\mathtt{f}]_{\mathtt{fid}}))) \circ_{\mathrm{X}} \langle \mathtt{tag'}_{\mathtt{newChild}}[\mathtt{f'}]_{\mathtt{fid'}} \rangle_{\mathrm{X}} \,\} & \mathtt{appendChild(parent, newChild)} & \{\,\mathtt{qg} \circ_{\mathrm{T}} (\mathtt{tag}_{\mathtt{parent}}[\mathtt{f} \otimes \langle \mathtt{tag'}_{\mathtt{newChild}}[\mathtt{f'}]_{\mathtt{fid'}} \rangle_{\mathrm{F}}]_{\mathtt{fid}})\,\} \\
\{\,\langle \mathtt{ct} \circ_{\mathrm{T}} (\mathtt{tag}_{\mathtt{parent}}[\mathtt{f}_1 \otimes \langle \mathtt{tag'}_{\mathtt{oldChild}}[\mathtt{f}]_{\mathtt{fid'}} \rangle_{\mathrm{F}} \otimes \mathtt{f}_2]_{\mathtt{fid}}) \rangle_{\mathrm{G}} \,\} & \mathtt{removeChild(parent, oldChild)} & \{\,\langle \mathtt{ct} \circ_{\mathrm{T}} (\mathtt{tag}_{\mathtt{parent}}[\mathtt{f}_1 \otimes \mathtt{f}_2]_{\mathtt{fid}}) \rangle_{\mathrm{G}} \oplus \langle \mathtt{tag'}_{\mathtt{oldChild}}[\mathtt{f}]_{\mathtt{fid'}} \rangle_{\mathrm{G}} \,\} \\
\{\,\mathtt{tag'}_{\mathtt{node}}[\mathtt{f}]_{\mathtt{fid}} \,\} & \mathtt{tag := getNodeName(node)} & \{\,\mathtt{tag'}_{\mathtt{node}}[\mathtt{f}]_{\mathtt{fid}} \wedge (\mathtt{tag} = \mathtt{tag'}) \,\} \\
\{\,\mathtt{tag}_{\mathtt{node''}}[\mathtt{f}_1 \otimes \langle \mathtt{tag'}_{\mathtt{node}}[\mathtt{f}]_{\mathtt{fid'}} \rangle_{\mathrm{F}} \otimes \mathtt{f}_2]_{\mathtt{fid}} \,\} & \mathtt{id := getParentNode(node)} & \{\,\mathtt{tag}_{\mathtt{node''}}[\mathtt{f}_1 \otimes \langle \mathtt{tag'}_{\mathtt{node}}[\mathtt{f}]_{\mathtt{fid'}} \rangle_{\mathrm{F}} \otimes \mathtt{f}_2]_{\mathtt{fid}} \wedge (\mathtt{id{=}node''}) \,\} \\
\{\,\langle \mathtt{tag'}_{\mathtt{node}}[\mathtt{f}]_{\mathtt{fid'}} \rangle_{\mathrm{G}} \,\} & \mathtt{id := getParentNode(node)} & \{\,\langle \mathtt{tag'}_{\mathtt{node}}[\mathtt{f}]_{\mathtt{fid'}} \rangle_{\mathrm{G}} \wedge (\mathtt{id} = \mathbf{null}) \,\} \\
\{\,\mathtt{tag}_{\mathtt{node}}[\mathtt{f}]_{\mathtt{fid'}} \,\} & \mathtt{fid := getChildNodes(node)} & \{\,\mathtt{tag}_{\mathtt{node}}[\mathtt{f}]_{\mathtt{fid'}} \wedge (\mathtt{fid} = \mathtt{fid'}) \,\} \\
\{\,\varnothing_{\mathrm{G}} \,\} & \mathtt{node := createNode(Tag)} & \{\,\langle \mathtt{Tag}_{\mathtt{node}}[\varnothing_{\mathrm{F}}]_{\mathtt{fid}} \rangle_{\mathrm{G}} \,\} \\
\{\,\mathtt{tag}_{\mathtt{id}}[\mathtt{f}_1 \otimes \langle \mathtt{tag'}_{\mathtt{id'}}[\mathtt{f}]_{\mathtt{fid'}} \rangle_{\mathrm{F}} \otimes \mathtt{f}_2]_{\mathtt{list}} \wedge (\mathtt{Int} = \mathrm{len}(\mathtt{f}_1)) \,\} & \mathtt{node := item(list, Int)} & \{\,\mathtt{tag}_{\mathtt{id}}[\mathtt{f}_1 \otimes \langle \mathtt{tag'}_{\mathtt{id'}}[\mathtt{f}]_{\mathtt{fid'}} \rangle_{\mathrm{F}} \otimes \mathtt{f}_2]_{\mathtt{list}} \wedge (\mathtt{node} = \mathtt{id'}) \,\} \\
\{\,\mathtt{tag}_{\mathtt{id}}[\mathtt{f}]_{\mathtt{list}} \wedge (\mathtt{Int} < 0 \vee \mathtt{Int} \geq \mathrm{len}(\mathtt{f})) \,\} & \mathtt{node := item(list, Int)} & \{\,\mathtt{tag}_{\mathtt{id}}[\mathtt{f}]_{\mathtt{list}} \wedge (\mathtt{node{=}}\mathbf{null}) \,\}
\end{array}$$

Figure 2. Minimal DOM Axioms

$$\begin{array}{rcl}
\left\{ \begin{array}{l} \exists \mathtt{tag}, \mathtt{tag'}, \mathtt{fid}, \mathtt{fid'}, \mathtt{f}, \mathtt{f'}, \mathtt{qg}. \\ \end{array} \begin{array}{r} ((\mathtt{qg} \circ_{\mathrm{T}} (\mathtt{tag}_{\mathtt{parent}}[\mathtt{f} \otimes \langle \mathtt{tag'}_{\mathtt{newChild}}[\mathtt{f'}]_{\mathtt{fid'}} \rangle_{\mathrm{F}}]_{\mathtt{fid}})) \multimap P) \circ_{\mathrm{G}} \\ ((\varnothing_{\mathrm{X}} \multimap (\mathtt{qg} \circ_{\mathrm{T}} (\mathtt{tag}_{\mathtt{parent}}[\mathtt{f}]_{\mathtt{fid}}))) \circ_{\mathrm{X}} \langle \mathtt{tag'}_{\mathtt{newChild}}[\mathtt{f'}]_{\mathtt{fid'}} \rangle_{\mathrm{X}}) \end{array} \right\} & \mathtt{appendChild(parent, newChild)} & \{\,P\,\} \\
\left\{ \begin{array}{l} \exists \mathtt{tag}, \mathtt{tag'}, \mathtt{fid}, \mathtt{fid'}, \mathtt{f}, \mathtt{f}_1, \mathtt{f}_2, \mathtt{ct}. \\ \end{array} \begin{array}{r} ((\langle \mathtt{ct} \circ_{\mathrm{T}} (\mathtt{tag}_{\mathtt{parent}}[\mathtt{f}_1 \otimes \mathtt{f}_2]_{\mathtt{fid}}) \rangle_{\mathrm{G}} \oplus \langle \mathtt{tag'}_{\mathtt{oldChild}}[\mathtt{f}]_{\mathtt{fid'}} \rangle_{\mathrm{G}}) \multimap P) \circ_{\mathrm{G}} \\ (\langle \mathtt{ct} \circ_{\mathrm{T}} (\mathtt{tag}_{\mathtt{parent}}[\mathtt{f}_1 \otimes \langle \mathtt{tag'}_{\mathtt{oldChild}}[\mathtt{f}]_{\mathtt{fid'}} \rangle_{\mathrm{F}} \otimes \mathtt{f}_2]_{\mathtt{fid}}) \rangle_{\mathrm{G}}) \end{array} \right\} & \mathtt{removeChild(parent, oldChild)} & \{\,P\,\} \\
\{\, \exists \mathtt{tag'}. \Diamond_{\mathrm{T}\to\mathrm{G}} (\mathtt{tag'}_{\mathtt{node}}[\mathrm{true}_{\mathrm{F}}]) \wedge P[\mathtt{tag'}/\mathtt{tag}] \,\} & \mathtt{tag := getNodeName(node)} & \{\,P\,\} \\
\left\{ \begin{array}{l} \exists \mathtt{tag'}. \\ \end{array} \begin{array}{r} (\exists \mathtt{tag}, \mathtt{node''}. \Diamond_{\mathrm{T}\to\mathrm{G}} (\mathtt{tag}_{\mathtt{node''}}[\mathrm{true}_{\mathrm{F}} \otimes \langle \mathtt{tag'}_{\mathtt{node}}[\mathrm{true}_{\mathrm{F}}] \rangle_{\mathrm{F}} \otimes \mathrm{true}_{\mathrm{F}}]) \wedge P[\mathtt{node''}/\mathtt{id}]) \\ \vee (\Diamond_{\mathrm{G}\to\mathrm{G}} (\langle \mathtt{tag'}_{\mathtt{node}}[\mathrm{true}_{\mathrm{F}}] \rangle_{\mathrm{G}}) \wedge P[\mathbf{null}/\mathtt{id}]) \end{array} \right\} & \mathtt{id := getParentNode(node)} & \{\,P\,\} \\
\{\, \exists \mathtt{tag}, \mathtt{fid'}. \Diamond_{\mathrm{T}\to\mathrm{G}} (\mathtt{tag}_{\mathtt{node}}[\mathrm{true}_{\mathrm{F}}]_{\mathtt{fid'}}) \wedge P[\mathtt{fid'}/\mathtt{fid}] \,\} & \mathtt{fid := getChildNodes(node)} & \{\,P\,\} \\
\{\, (\forall \mathtt{node'}, \mathtt{fid}. \langle \mathtt{Tag}_{\mathtt{node}'}[\varnothing_{\mathrm{F}}]_{\mathtt{fid}} \rangle_{\mathrm{G}} \multimap P[\mathtt{node'}/\mathtt{node}]) \circ_{\mathrm{G}} (\varnothing_{\mathrm{G}}) \,\} & \mathtt{node := createNode(Tag)} & \{\,P\,\} \\
\left\{ \begin{array}{l} \exists \mathtt{tag}, \mathtt{id}. \\ \end{array} \begin{array}{r} \exists \mathtt{tag'}, \mathtt{id'}, \mathtt{f}_1. \Diamond_{\mathrm{T}\to\mathrm{G}} (\mathtt{tag}_{\mathtt{id}}[\mathtt{f}_1 \otimes \langle \mathtt{tag'}_{\mathtt{id'}}[\mathrm{true}_{\mathrm{F}}] \rangle_{\mathrm{F}} \otimes \mathrm{true}_{\mathrm{F}}]_{\mathtt{list}}) \wedge (\mathtt{Int} = \mathrm{len}(\mathtt{f}_1)) \wedge P[\mathtt{id'}/\mathtt{node}] \\ \vee \exists \mathtt{f}. \Diamond_{\mathrm{T}\to\mathrm{G}} (\mathtt{tag}_{\mathtt{id}}[\mathtt{f}]_{\mathtt{list}}) \wedge (\mathtt{Int} < 0 \vee \mathtt{Int} \geq \mathrm{len}(\mathtt{f})) \wedge P[\mathbf{null}/\mathtt{node}] \end{array} \right\} & \mathtt{node := item(list, Int)} & \{\,P\,\}
\end{array}$$

Figure 3. Minimal DOM Weakest Preconditions

Definition 4.2 (Command Axioms)**.** In Figure 2 we give the axioms for the Minimal DOM commands described in Section 2. In addition, we have the following axioms for assignment

$$\{\mathtt{d} \wedge (\mathtt{var'}_{\mathrm{V}} = \mathtt{Exp}_{\mathrm{V}})\}\ \mathtt{var}_{\mathrm{V}} := \mathtt{Exp}_{\mathrm{V}}\ \{\mathtt{d} \wedge (\mathtt{var}_{\mathrm{V}} = \mathtt{var'}_{\mathrm{V}})\}$$
$$\{\mathtt{d}\}\ \mathtt{skip}\ \{\mathtt{d}\}$$

where $\mathtt{d} \in \{\mathrm{Var}_{\mathrm{T}}, \mathrm{Var}_{\mathrm{G}}\}$.

The `appendChild` command has two axioms parameterised by $\mathrm{X} \in \{\mathrm{F}, \mathrm{G}\}$, corresponding to when `newChild` has a parent node and when it does not since it is at the top of the grove. `getParentNode` also has two axioms, returning the parent node when it exists and **null** when it does not. Similarly, the `item` command has two axioms, for the cases when the indices are within range or not. The axioms for assignment and skip are standard, and do not change the grove.

Definition 4.3 (Local Hoare Reasoning)**.** The local Hoare reasoning framework consists of the command axioms given in Definition 4.2 and seven general inference rules: the Rules of Sequencing, If-Then-Else, While, Consequence, Disjunction[1] and Auxiliary Variable Elimination, which are standard, and the Frame Rule, which permits local reasoning by allowing the inference of invariant properties implied by locality, and which is presented here in terms of context application:

$$\textsc{Sequencing:}\ \frac{\{P\}\,\mathtt{C}_1\,\{Q\} \quad \{Q\}\,\mathtt{C}_2\,\{R\}}{\{P\}\,\mathtt{C}_1\,;\,\mathtt{C}_2\,\{R\}}$$

$$\textsc{If-Then-Else:}\ \frac{\{\mathtt{Bool} \wedge P\}\,\mathtt{C}_1\,\{Q\} \quad \{\neg\mathtt{Bool} \wedge P\}\,\mathtt{C}_2\,\{Q\}}{\{P\}\,\mathtt{if\ Bool\ then\ C_1\ else\ C_2}\,\{Q\}}$$

$$\textsc{While:}\ \frac{\{\mathtt{Bool} \wedge P\}\,\mathtt{C}\,\{P\}}{\{P\}\,\mathtt{while\ Bool\ do\ C}\,\{\neg\mathtt{Bool} \wedge P\}}$$

$$\textsc{Consequence:}\ \frac{P' \Rightarrow P \quad \{P\}\,\mathtt{C}\,\{Q\} \quad Q \Rightarrow Q'}{\{P'\}\,\mathtt{C}\,\{Q'\}}$$

$$\textsc{Disjunction:}\ \frac{\{P\}\,\mathtt{C}\,\{Q\} \quad \{P'\}\,\mathtt{C}\,\{Q'\}}{\{P \vee P'\}\,\mathtt{C}\,\{Q \vee Q'\}}$$

$$\textsc{Aux Var Elim:}\ \frac{\{P\}\,\mathtt{C}\,\{Q\}}{\{\exists \mathtt{var}.\, P\}\,\mathtt{C}\,\{\exists \mathtt{var}.\, Q\}}\ \mathtt{var} \notin \mathrm{free}(\mathtt{C})$$

$$\textsc{Frame Rule:}\ \frac{\{P\}\,\mathtt{C}\,\{Q\}}{\{K \circ_{\mathrm{D}} P\}\,\mathtt{C}\,\{K \circ_{\mathrm{D}} Q\}}\ \begin{array}{l}\mathrm{mod}(\mathtt{C}) \cap \\ \mathrm{free}(K) = \emptyset\end{array}$$

where $P, Q\colon \mathrm{D}$, $K\colon \mathrm{D}{\to}\mathrm{D}'$ for $\mathrm{D}, \mathrm{D}' \in \{\mathrm{T}, \mathrm{G}\}$, and `var` is either an environment or a store variable. The set of free variables is standard and $\mathrm{mod}(\mathtt{C})$ is the set of all variables assigned to by `C`.

We conclude this section with a brief sanity check, showing that the weakest preconditions of the Minimal DOM commands are derivable in the logic. This means that our local Hoare reasoning is complete for straight line code.

Theorem 4.4 (Weakest Preconditions)**.** *The weakest preconditions of the Minimal DOM commands are derivable in the logic.*

Proof. The weakest preconditions for the commands are given in Figure 3. The derivations are provided in the full version of this paper. □

5. Examples

We present a number of examples of Minimal DOM reasoning. We illustrate the minimality of Minimal DOM by giving a representative example of a derivation of a DOM Core Level 1 command which is not included in Minimal DOM. We demonstrate the modular nature of Context Logic reasoning by giving a simple, concise derivation of a command which is not included in DOM. Finally, we demonstrate the potential applicability of the framework to real-world problems by proving that an example program will always maintain the properties specified by its accompanying XML schema.

[1] The Disjunction Rule is required for the commands with two axioms; the Conjunction Rule, meanwhile, is admissible.

getIndex derivation

$$\{\, \mathtt{tag}_{\mathtt{id}}[\mathtt{f} \otimes \langle \mathtt{tag'}_{\mathtt{node}}[\mathtt{f''}]_{\mathtt{fid'}}\rangle_{\mathrm{F}} \otimes \mathtt{f'}]_{\mathtt{nodeList}} \,\}$$
$$\mathtt{n} := 0 \ ; \ \mathtt{current} := \mathtt{item}(\mathtt{nodeList}, \mathtt{n}) \ ;$$
$$\left\{ \begin{array}{l} \exists \mathtt{tag''}, \mathtt{f}_1, \mathtt{f}_2 . (\mathtt{n} = \mathrm{len}(\mathtt{f}_1)) \\ \wedge\, \mathtt{tag}_{\mathtt{id}} \left[\left(\begin{array}{l} (\mathtt{f} \otimes \langle \mathtt{tag'}_{\mathtt{node}}[\mathtt{f''}]_{\mathtt{fid'}}\rangle_{\mathrm{F}}) \wedge \\ (\mathtt{f}_1 \otimes \langle \mathtt{tag''}_{\mathtt{current}}[\mathrm{true}_{\mathrm{F}}]\rangle_{\mathrm{F}} \otimes \mathtt{f}_2) \end{array} \right) \otimes \mathtt{f'} \right]_{\mathtt{nodeList}} \end{array} \right\}$$
$$\mathtt{while}\ (\mathtt{current} \neq \mathtt{node} \wedge \mathtt{current} \neq \mathbf{null})\ \mathtt{do}$$
$$\left\{ \begin{array}{l} \exists \mathtt{tag''}, \mathtt{tag'''}, \mathtt{id'''}, \mathtt{f}_1, \mathtt{f'}_2 . (\mathtt{n} = \mathrm{len}(\mathtt{f}_1)) \\ \wedge\, \mathtt{tag}_{\mathtt{id}} \left[\left(\begin{array}{l} (\mathtt{f} \otimes \langle \mathtt{tag'}_{\mathtt{node}}[\mathtt{f''}]_{\mathtt{fid'}}\rangle_{\mathrm{F}}) \wedge \\ \left(\begin{array}{l} \mathtt{f}_1 \otimes \langle \mathtt{tag''}_{\mathtt{current}}[\mathrm{true}_{\mathrm{F}}]\rangle_{\mathrm{F}} \otimes \\ \langle \mathtt{tag'''}_{\mathtt{id'''}}[\mathrm{true}_{\mathrm{F}}]\rangle_{\mathrm{F}} \otimes \mathtt{f'}_2 \end{array} \right) \end{array} \right) \otimes \mathtt{f'} \right]_{\mathtt{nodeList}} \end{array} \right\}$$
$$\mathtt{n} := \mathtt{n} + 1 \ ; \ \mathtt{current} := \mathtt{item}(\mathtt{nodelist}, \mathtt{n})$$
$$\left\{ \begin{array}{l} \exists \mathtt{tag'''}, \mathtt{f'}_1, \mathtt{f'}_2 . (\mathtt{n} = \mathrm{len}(\mathtt{f'}_1)) \\ \wedge\, \mathtt{tag}_{\mathtt{id}} \left[\left(\begin{array}{l} (\mathtt{f} \otimes \langle \mathtt{tag'}_{\mathtt{node}}[\mathtt{f''}]_{\mathtt{fid'}}\rangle_{\mathrm{F}}) \wedge \\ (\mathtt{f'}_1 \otimes \langle \mathtt{tag'''}_{\mathtt{current}}[\mathrm{true}_{\mathrm{F}}]\rangle_{\mathrm{F}} \otimes \mathtt{f'}_2) \end{array} \right) \otimes \mathtt{f'} \right]_{\mathtt{nodeList}} \end{array} \right\}$$
$$\left\{ \begin{array}{l} \exists \mathtt{tag''}, \mathtt{f}_1, \mathtt{f}_2 . (\mathtt{n} = \mathrm{len}(\mathtt{f}_1)) \wedge (\mathtt{current} = \mathtt{node}) \\ \wedge\, \mathtt{tag}_{\mathtt{id}} \left[\left(\begin{array}{l} (\mathtt{f} \otimes \langle \mathtt{tag'}_{\mathtt{node}}[\mathtt{f''}]_{\mathtt{fid'}}\rangle_{\mathrm{F}}) \wedge \\ (\mathtt{f}_1 \otimes \langle \mathtt{tag''}_{\mathtt{current}}[\mathrm{true}_{\mathrm{F}}]\rangle_{\mathrm{F}} \otimes \mathtt{f}_2) \end{array} \right) \otimes \mathtt{f'} \right]_{\mathtt{nodeList}} \end{array} \right\}$$
$$\{\, \mathtt{tag}_{\mathtt{id}}[\mathtt{f} \otimes \langle \mathtt{tag'}_{\mathtt{node}}[\mathtt{f''}]_{\mathtt{fid'}}\rangle_{\mathrm{F}} \otimes \mathtt{f'}]_{\mathtt{nodeList}} \wedge (\mathtt{n} = \mathrm{len}(\mathtt{f})) \,\}$$

getPreviousSibling derivation

$$\{\, \langle \mathtt{tag}_{\mathtt{node}}[\mathtt{f}]_{\mathtt{fid}}\rangle_{\mathrm{G}} \,\}$$
$$\mathtt{parent} := \mathtt{getParentNode}(\mathtt{node}) \ ;$$
$$\{\, \langle \mathtt{tag}_{\mathtt{node}}[\mathtt{f}]_{\mathtt{fid}}\rangle_{\mathrm{G}} \wedge (\mathtt{parent} = \mathbf{null}) \,\}$$
$$\mathtt{if}\ \mathtt{parent} := \mathbf{null}\ \mathtt{then}\ \mathtt{sibling} := \mathbf{null}\ \mathtt{else}\ \ldots$$
$$\{\, \langle \mathtt{tag}_{\mathtt{node}}[\mathtt{f}]_{\mathtt{fid}}\rangle_{\mathrm{G}} \wedge (\mathtt{sibling} = \mathbf{null}) \,\}$$

$$\{\, \mathtt{tag}_{\mathtt{id}}[\langle \mathtt{tag''}_{\mathtt{node}}[\mathtt{f''}]_{\mathtt{fid''}}\rangle_{\mathrm{F}} \otimes \mathtt{f}_2]_{\mathtt{fid}} \,\}$$
$$\mathtt{parent} := \mathtt{getParentNode}(\mathtt{node}) \ ; \ \mathtt{if}\ \mathtt{parent} := \mathbf{null}\ \mathtt{then}\ \ldots\ \mathtt{else}$$
$$\{\, \mathtt{tag}_{\mathtt{id}}[\langle \mathtt{tag''}_{\mathtt{node}}[\mathtt{f''}]_{\mathtt{fid''}}\rangle_{\mathrm{F}} \otimes \mathtt{f}_2]_{\mathtt{fid}} \wedge (\mathtt{parent} = \mathtt{id}) \,\}$$
$$\mathtt{children} := \mathtt{getChildNodes}(\mathtt{parent}) \ ; \ \mathtt{n} := \mathtt{getIndex}(\mathtt{children}, \mathtt{node}) \ ;$$
$$\{\, \mathtt{tag}_{\mathtt{id}}[\langle \mathtt{tag''}_{\mathtt{node}}[\mathtt{f''}]_{\mathtt{fid''}}\rangle_{\mathrm{F}} \otimes \mathtt{f}_2]_{\mathtt{fid}} \wedge (\mathtt{parent}{=}\mathtt{id}) \wedge (\mathtt{children}{=}\mathtt{fid}) \wedge (\mathtt{n}{=}0) \,\}$$
$$\mathtt{sibling} := \mathtt{item}(\mathtt{nodelist}, \mathtt{n} - 1)$$
$$\{\, \mathtt{tag}_{\mathtt{id}}[\langle \mathtt{tag''}_{\mathtt{node}}[\mathtt{f''}]_{\mathtt{fid''}}\rangle_{\mathrm{F}} \otimes \mathtt{f}_2]_{\mathtt{fid}} \wedge (\mathtt{sibling} = \mathbf{null}) \,\}$$

$$\{\, \mathtt{tag}_{\mathtt{id}}[\mathtt{f}_1 \otimes \langle \mathtt{tag'}_{\mathtt{id}}, [\mathtt{f'}]_{\mathtt{fid'}}\rangle_{\mathrm{F}} \otimes \langle \mathtt{tag''}_{\mathtt{node}}[\mathtt{f''}]_{\mathtt{fid''}}\rangle_{\mathrm{F}} \otimes \mathtt{f}_2]_{\mathtt{fid}} \,\}$$
$$\mathtt{parent} := \mathtt{getParentNode}(\mathtt{node}) \ ; \ \mathtt{if}\ \mathtt{parent} := \mathbf{null}\ \mathtt{then}\ \ldots\ \mathtt{else}$$
$$\{\, \mathtt{tag}_{\mathtt{id}}[\mathtt{f}_1 \otimes \langle \mathtt{tag'}_{\mathtt{id}}, [\mathtt{f'}]_{\mathtt{fid'}}\rangle_{\mathrm{F}} \otimes \langle \mathtt{tag''}_{\mathtt{node}}[\mathtt{f''}]_{\mathtt{fid''}}\rangle_{\mathrm{F}} \otimes \mathtt{f}_2]_{\mathtt{fid}} \wedge (\mathtt{parent}{=}\mathtt{id}) \,\}$$
$$\mathtt{children} := \mathtt{getChildNodes}(\mathtt{parent}) \ ; \ \mathtt{n} := \mathtt{getIndex}(\mathtt{children}, \mathtt{node}) \ ;$$
$$\left\{ \begin{array}{l} \mathtt{tag}_{\mathtt{id}}[\mathtt{f}_1 \otimes \langle \mathtt{tag'}_{\mathtt{id}}, [\mathtt{f'}]_{\mathtt{fid'}}\rangle_{\mathrm{F}} \otimes \langle \mathtt{tag''}_{\mathtt{node}}[\mathtt{f''}]_{\mathtt{fid''}}\rangle_{\mathrm{F}} \otimes \mathtt{f}_2]_{\mathtt{fid}} \\ \wedge\, (\mathtt{parent} = \mathtt{id}) \wedge (\mathtt{children} = \mathtt{fid}) \wedge (\mathtt{n} - 1 = \mathrm{len}(\mathtt{f}_1)) \end{array} \right\}$$
$$\mathtt{sibling} := \mathtt{item}(\mathtt{nodelist}, \mathtt{n} - 1)$$
$$\{\, \mathtt{tag}_{\mathtt{id}}[\mathtt{f}_1 \otimes \langle \mathtt{tag'}_{\mathtt{id}}, [\mathtt{f'}]_{\mathtt{fid'}}\rangle_{\mathrm{F}} \otimes \langle \mathtt{tag''}_{\mathtt{node}}[\mathtt{f''}]_{\mathtt{fid''}}\rangle_{\mathrm{F}} \otimes \mathtt{f}_2]_{\mathtt{fid}} \wedge (\mathtt{sibling}{=}\mathtt{id'}) \,\}$$

Figure 4. getIndex and getPreviousSibling Derivations

5.1 GetPreviousSibling

We define the DOM command `getPreviousSibling`. In doing so, we define the auxiliary command `getIndex`, which is not in DOM Core Level 1. The purpose of `getIndex` is to return the index of a given node in a given list. Here, we demonstrate the derivation of a specification for `getPreviousSibling`, and by necessity therefore also a derivation of `getIndex`. The implementations of `getPreviousSibling` and the auxiliary command `getIndex` are:

$$\begin{array}{l} \mathtt{n} := \mathtt{getIndex}(\mathtt{nodeList}, \mathtt{node}) \triangleq \\ \quad \mathtt{n} := 0 \ ; \ \mathtt{current} := \mathtt{item}(\mathtt{nodeList}, \mathtt{n}) \ ; \\ \quad \mathtt{while}\ (\mathtt{current} \neq \mathtt{node} \wedge \mathtt{current} \neq \mathbf{null})\ \mathtt{do} \\ \quad\quad \mathtt{n} := \mathtt{n} + 1 \ ; \ \mathtt{current} := \mathtt{item}(\mathtt{nodelist}, \mathtt{n}) \end{array}$$

$$\begin{array}{l} \mathtt{sibling} := \mathtt{getPreviousSibling}(\mathtt{node}) \triangleq \\ \quad \mathtt{parent} := \mathtt{getParentNode}(\mathtt{node}) \ ; \\ \quad \mathtt{if}\ \mathtt{parent} = \mathbf{null}\ \mathtt{then}\ \mathtt{sibling} := \mathbf{null}\ \mathtt{else} \\ \quad\quad \mathtt{children} := \mathtt{getChildNodes}(\mathtt{parent}) \ ; \\ \quad\quad \mathtt{n} := \mathtt{getIndex}(\mathtt{children}, \mathtt{node}) \ ; \\ \quad\quad \mathtt{sibling} := \mathtt{item}(\mathtt{nodelist}, \mathtt{n} - 1) \end{array}$$

The `getIndex` command uses a simple while loop to do a linear search of the nodes in the parameter `nodeList`, counting the elements in turn until the target `node` is found. It then returns the position of that node. The `getPreviousSibling` command uses `getParentNode` and `getChildNodes` to obtain the list of siblings of the parameter `node`. It then uses `getIndex` to find the position of `node` in that list, and `item` to return the previous one if it exists, or **null** otherwise. If `node` is a root level node and therefore has no siblings, `getPreviousSibling` returns **null**.

`getIndex` has the following specification when `node` is an element of `nodeList`:

$$\begin{array}{l} \{\, \mathtt{tag}_{\mathtt{id}}[\mathtt{f} \otimes \langle \mathtt{tag'}_{\mathtt{node}}[\mathtt{f''}]_{\mathtt{fid'}}\rangle_{\mathrm{F}} \otimes \mathtt{f'}]_{\mathtt{nodeList}} \,\} \\ \mathtt{n} := \mathtt{getIndex}(\mathtt{nodeList}, \mathtt{node}) \\ \{\, \mathtt{tag}_{\mathtt{id}}[\mathtt{f} \otimes \langle \mathtt{tag'}_{\mathtt{node}}[\mathtt{f''}]_{\mathtt{fid'}}\rangle_{\mathrm{F}} \otimes \mathtt{f'}]_{\mathtt{nodeList}} \wedge (\mathtt{n} = \mathrm{len}(\mathtt{f})) \,\} \end{array}$$

The precondition states that a tree identified by `node` is a child of a tree with a child list identified by `nodeList`. The postcondition states that the tree has remained the same, and that the store now records the position of the tree `node` in the variable `n`.

`getPreviousSibling`, meanwhile, can be best described using three complementary specifications, corresponding to when the node is at the grove level, the beginning of the nodeList, or elsewhere.

$$\begin{array}{l} \{\, \langle \mathtt{tag}_{\mathtt{node}}[\mathtt{f}]_{\mathtt{fid}}\rangle_{\mathrm{G}} \,\} \\ \mathtt{sibling} := \mathtt{getPreviousSibling}(\mathtt{node}) \\ \{\, \langle \mathtt{tag}_{\mathtt{node}}[\mathtt{f}]_{\mathtt{fid}}\rangle_{\mathrm{G}} \wedge (\mathtt{sibling} = \mathbf{null}) \,\} \end{array}$$

$$\begin{array}{l} \{\, \mathtt{tag}_{\mathtt{id}}[\langle \mathtt{tag''}_{\mathtt{node}}[\mathtt{f''}]_{\mathtt{fid''}}\rangle_{\mathrm{F}} \otimes \mathtt{f}_2]_{\mathtt{fid}} \,\} \\ \mathtt{sibling} := \mathtt{getPreviousSibling}(\mathtt{node}) \\ \{\, \mathtt{tag}_{\mathtt{id}}[\langle \mathtt{tag''}_{\mathtt{node}}[\mathtt{f''}]_{\mathtt{fid''}}\rangle_{\mathrm{F}} \otimes \mathtt{f}_2]_{\mathtt{fid}} \wedge (\mathtt{sibling} = \mathbf{null}) \,\} \end{array}$$

$$\begin{array}{l} \{\, \mathtt{tag}_{\mathtt{id}}[\mathtt{f}_1 \otimes \langle \mathtt{tag'}_{\mathtt{id}}, [\mathtt{f'}]_{\mathtt{fid'}}\rangle_{\mathrm{F}} \otimes \langle \mathtt{tag''}_{\mathtt{node}}[\mathtt{f''}]_{\mathtt{fid''}}\rangle_{\mathrm{F}} \otimes \mathtt{f}_2]_{\mathtt{fid}} \,\} \\ \mathtt{sibling} := \mathtt{getPreviousSibling}(\mathtt{node}) \\ \left\{ \begin{array}{l} \mathtt{tag}_{\mathtt{id}}[\mathtt{f}_1 \otimes \langle \mathtt{tag'}_{\mathtt{id}}, [\mathtt{f'}]_{\mathtt{fid'}}\rangle_{\mathrm{F}} \otimes \langle \mathtt{tag''}_{\mathtt{node}}[\mathtt{f''}]_{\mathtt{fid''}}\rangle_{\mathrm{F}} \otimes \mathtt{f}_2]_{\mathtt{fid}} \\ \quad \wedge\, (\mathtt{sibling} = \mathtt{id'}) \end{array} \right\} \end{array}$$

The derivations for these specifications are given in Figure 4.

5.2 InsertAfter

In a similar fashion to `getPreviousSibling`, we can use Minimal DOM to implement the DOM Core Level 1 command `insertBefore` which inserts a `newChild` into a `parent`'s list of children, immediately before some `refNode`:

$$\begin{array}{l} \mathtt{insertBefore}(\mathtt{parent}, \mathtt{newChild}, \mathtt{refNode}) \triangleq \\ \quad \mathtt{appendChild}(\mathtt{parent}, \mathtt{newChild}) \ ; \\ \quad \mathtt{if}\ \mathtt{refNode} = \mathbf{null}\ \mathtt{then\ skip\ else} \\ \quad\quad \mathtt{children} := \mathtt{getChildNodes}(\mathtt{parent}) \ ; \\ \quad\quad \mathtt{position} := \mathtt{getIndex}(\mathtt{children}, \mathtt{refNode}) \ ; \\ \quad\quad \mathtt{current} := \mathtt{item}(\mathtt{children}, \mathtt{position}) \ ; \\ \quad\quad \mathtt{while\ current} \neq \mathtt{newChild\ do} \\ \quad\quad\quad \mathtt{appendChild}(\mathtt{parent}, \mathtt{current}) \ ; \\ \quad\quad\quad \mathtt{current} := \mathtt{item}(\mathtt{children}, \mathtt{position}) \end{array}$$

The specification for this command has two cases: one in which the argument `refNode` is **null**; and another in which it is not. The case in which `refNode` is not **null** is as follows (where $\mathrm{X} \in \{\mathrm{F}, \mathrm{G}\}$ as

in Example 3.5d):

$$\left\{ \begin{array}{l} (\varnothing_{\mathrm{X}} \multimap \circ(\mathtt{g} \circ_{\mathrm{T}} (\mathtt{tag}_{\mathtt{parent}}[\mathtt{f}_1 \otimes \langle \mathtt{tag'}_{\mathtt{refNode}}[\mathtt{f}]_{\mathtt{fid'}} \rangle_{\mathrm{F}} \otimes \mathtt{f}_2]_{\mathtt{fid}}))) \\ \circ_{\mathrm{X}}(\langle \mathtt{tag''}_{\mathtt{newChild}}[\mathtt{f'}]_{\mathtt{fid''}} \rangle_{\mathrm{X}}) \end{array} \right\}$$

$$\mathtt{insertBefore}(\mathtt{parent}, \mathtt{newChild}, \mathtt{refNode})$$

$$\left\{ \mathtt{g} \circ_{\mathrm{T}} (\mathtt{tag}_{\mathtt{parent}} \begin{bmatrix} \mathtt{f}_1 \otimes \langle \mathtt{tag''}_{\mathtt{newChild}}[\mathtt{f'}]_{\mathtt{fid''}} \rangle_{\mathrm{F}} \\ \otimes \langle \mathtt{tag'}_{\mathtt{refNode}}[\mathtt{f}]_{\mathtt{fid'}} \rangle_{\mathrm{F}} \otimes \mathtt{f}_2 \end{bmatrix}_{\mathtt{fid}}) \right\}$$

Using `insertBefore`, one can implement another command, `insertAfter` (whose behaviour is as expected), which is *not* in DOM Core Level 1. In the case where `refNode` is not **null**, this simply corresponds to using two calls to `insertBefore`:

$$\begin{array}{l} \mathtt{insertAfter}(\mathtt{parent}, \mathtt{newChild}, \mathtt{refNode}) \triangleq \\ \quad \mathtt{insertBefore}(\mathtt{parent}, \mathtt{newChild}, \mathtt{refNode}) \ ; \\ \quad \mathtt{insertBefore}(\mathtt{parent}, \mathtt{refNode}, \mathtt{newChild}) \end{array}$$

In this case, `insertAfter` has the specification:

$$\left\{ \begin{array}{l} (\varnothing_{\mathrm{X}} \multimap \circ(\mathtt{g} \circ_{\mathrm{T}} (\mathtt{tag}_{\mathtt{parent}}[\mathtt{f}_1 \otimes \langle \mathtt{tag'}_{\mathtt{refNode}}[\mathtt{f}]_{\mathtt{fid'}} \rangle_{\mathrm{F}} \otimes \mathtt{f}_2]_{\mathtt{fid}}))) \\ \circ_{\mathrm{X}}(\langle \mathtt{tag''}_{\mathtt{newChild}}[\mathtt{f'}]_{\mathtt{fid''}} \rangle_{\mathrm{X}}) \end{array} \right\}$$

$$\mathtt{insertAfter}(\mathtt{parent}, \mathtt{newChild}, \mathtt{refNode});$$

$$\left\{ \mathtt{g} \circ_{\mathrm{T}} (\mathtt{tag}_{\mathtt{parent}} \begin{bmatrix} \mathtt{f}_1 \otimes \langle \mathtt{tag'}_{\mathtt{refNode}}[\mathtt{f}]_{\mathtt{fid'}} \rangle_{\mathrm{F}} \\ \otimes \langle \mathtt{tag''}_{\mathtt{newChild}}[\mathtt{f'}]_{\mathtt{fid''}} \rangle_{\mathrm{F}} \otimes \mathtt{f}_2 \end{bmatrix}_{\mathtt{fid}}) \right\}$$

which we can derive compositionally from the non-**null** case of the specification of `insertBefore`:

$$\left\{ \begin{array}{l} (\varnothing_{\mathrm{X}} \multimap \circ(\mathtt{g} \circ_{\mathrm{T}} (\mathtt{tag}_{\mathtt{parent}}[\mathtt{f}_1 \otimes \langle \mathtt{tag'}_{\mathtt{refNode}}[\mathtt{f}]_{\mathtt{fid'}} \rangle_{\mathrm{F}} \otimes \mathtt{f}_2]_{\mathtt{fid}}))) \\ \circ_{\mathrm{X}}(\langle \mathtt{tag''}_{\mathtt{newChild}}[\mathtt{f'}]_{\mathtt{fid''}} \rangle_{\mathrm{X}}) \end{array} \right\}$$

$$\mathtt{insertBefore}(\mathtt{parent}, \mathtt{newChild}, \mathtt{refNode});$$

$$\left\{ \mathtt{g} \circ_{\mathrm{T}} (\mathtt{tag}_{\mathtt{parent}} \begin{bmatrix} \mathtt{f}_1 \otimes \langle \mathtt{tag''}_{\mathtt{newChild}}[\mathtt{f'}]_{\mathtt{fid''}} \rangle_{\mathrm{F}} \\ \otimes \langle \mathtt{tag'}_{\mathtt{refNode}}[\mathtt{f}]_{\mathtt{fid'}} \rangle_{\mathrm{F}} \otimes \mathtt{f}_2 \end{bmatrix}_{\mathtt{fid}}) \right\}$$

$$\left\{ \begin{array}{l} (\varnothing_{\mathrm{F}} \multimap \circ(\mathtt{g} \circ_{\mathrm{T}} (\mathtt{tag}_{\mathtt{parent}}[\mathtt{f}_1 \otimes \langle \mathtt{tag''}_{\mathtt{newChild}}[\mathtt{f'}]_{\mathtt{fid''}} \rangle_{\mathrm{F}} \otimes \varnothing_{\mathrm{F}} \otimes \mathtt{f}_2]_{\mathtt{fid}}))) \\ \circ_{\mathrm{F}}(\langle \mathtt{tag'}_{\mathtt{refNode}}[\mathtt{f}]_{\mathtt{fid'}} \rangle_{\mathrm{F}}) \end{array} \right\}$$

$$\left\{ \begin{array}{l} (\varnothing_{\mathrm{F}} \multimap \circ(\mathtt{g} \circ_{\mathrm{T}} (\mathtt{tag}_{\mathtt{parent}}[\mathtt{f}_1 \otimes \langle \mathtt{tag''}_{\mathtt{newChild}}[\mathtt{f'}]_{\mathtt{fid''}} \rangle_{\mathrm{F}} \otimes \mathtt{f}_2]_{\mathtt{fid}}))) \\ \circ_{\mathrm{F}}(\langle \mathtt{tag'}_{\mathtt{refNode}}[\mathtt{f}]_{\mathtt{fid'}} \rangle_{\mathrm{F}}) \end{array} \right\}$$

$$\mathtt{insertBefore}(\mathtt{parent}, \mathtt{refNode}, \mathtt{newChild});$$

$$\left\{ \mathtt{g} \circ_{\mathrm{T}} (\mathtt{tag}_{\mathtt{parent}} \begin{bmatrix} \mathtt{f}_1 \otimes \langle \mathtt{tag'}_{\mathtt{refNode}}[\mathtt{f}]_{\mathtt{fid'}} \rangle_{\mathrm{F}} \\ \otimes \langle \mathtt{tag''}_{\mathtt{newChild}}[\mathtt{f'}]_{\mathtt{fid''}} \rangle_{\mathrm{F}} \otimes \mathtt{f}_2 \end{bmatrix}_{\mathtt{fid}}) \right\}$$

This example serves as a good illustration of the modularity of our reasoning. The specification of the composite command is of the same form as the specifications of each of the individual commands, and does not refer to any other specification. The nearest DOM equivalent would be an English language statement declaring that, where '$a \neq$ **null**', the command '$p.\mathtt{insertAfter}(a, b)$' is equivalent to the sequence of commands '$p.\mathtt{insertBefore}(a, b); p.\mathtt{insertBefore}(b, a)$'. This would require that the reader refer to the specification of `insertBefore` in order to understand that of `insertAfter`.

5.3 Proving Schema Invariants

When reasoning about programs, it is often desirable to prove a particular property about a program, rather than proving the whole (very complex) specification. One example of this involves proving XML schema invariants. For example, consider writing a program to update an XML document which complies with the following schema:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
<xs:element name="addressBook">
  <xs:element name="household" minOccurs="0"
              maxOccurs="unbounded">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="person" maxOccurs="unbounded"/>
        </xs:element>
        <xs:element name="address" type="string"/>
        <xs:element name="phone" type="string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:element>
```

This schema asserts that the root element of the document should be an '**addressBook**' node. That node may have zero or more children so long as they are '**household**' nodes. Those nodes must contain one or more '**person**' nodes, one '**address**' node and one '**phone**' node. Each of these third-level nodes have no children. In practice, these childless nodes should contain data of type 'string'. As stated in Section 2, Minimal DOM concentrates on the fundamental tree structure of XML, and not on the content of that structure, so we ignore this text data.

We specify that a tree consists of a valid addressBook document using a tree formula S, corresponding to the schema above:

$$S \triangleq \langle \mathbf{addressBook}[\text{households}] \rangle_{\mathrm{G}}$$

$$\text{households} \triangleq \Box_{\otimes}(\langle \text{true}_{\mathrm{T}} \rangle_{\mathrm{F}} \Rightarrow \langle \mathbf{household}[\langle \mathbf{person}[\varnothing_{\mathrm{F}}] \rangle_{\mathrm{F}} \otimes \text{people} \otimes \langle \mathbf{address}[\varnothing_{\mathrm{F}}] \rangle_{\mathrm{F}} \otimes \langle \mathbf{phone}[\varnothing_{\mathrm{F}}] \rangle_{\mathrm{F}}])$$

$$\text{people} \triangleq \Box_{\otimes}(\langle \text{true}_{\mathrm{T}} \rangle_{\mathrm{F}} \Rightarrow \langle \mathbf{person}[\varnothing_{\mathrm{F}}] \rangle_{\mathrm{F}})$$

Consider a program which updates the addressBook document when a specified person leaves of a given household. We give an implementation of this program in Minimal DOM which requires that the supplied parameter `house` refers to a '**household**' node in the addressBook, and that `leaver` refers to a '**person**' in that household. It moves `leaver` out of `house`, into a newly created house; checks if `house` is now empty; and, if it is, deletes it from the address book.

```
moveOut(house, leaver) ≜
    // Move leaver into a new house.
    book := getParentNode(house);
    newHouse := createNode('household');
    newAddr := createNode('address');
    newPhone := createNode('phone');
    appendChild(newHouse, leaver);
    appendChild(newHouse, newAddr);
    appendChild(newHouse, newPhone);
    appendChild(book, newHouse);
    // Check if old household is empty...
    kids := getChildNodes(house);
    firstChild := item(kids, 0);
    firstName := getNodeName(firstChild);
    if firstName = 'person' then skip else
        // ...and if so, remove it.
        removeChild(book, house);
```

Since Minimal DOM makes no attempt to store the data content of the XML structure, we do not require that the user of the program to specify address and phone data for the new house. In a language which did handle such data, to do so would be trivial.

The safety condition that `leaver` refers to a person within a household `house` can be expressed by the formula

$$P \triangleq \Diamond_{\mathrm{T} \to \mathrm{G}} \mathbf{household}_{\mathtt{house}}[\Diamond_{\otimes} \langle \mathbf{person}_{\mathtt{leaver}}[\varnothing_{\mathrm{F}}] \rangle_{\mathrm{F}}]$$

Given this precondition, we can show that `moveOut` maintains the schema predicate:

$$\left\{ (S \wedge P) \oplus \text{true}_{\mathrm{G}} \right\} \mathtt{moveOut}(\mathtt{house}, \mathtt{leaver}) \left\{ S \oplus \text{true}_{\mathrm{G}} \right\}$$

As explained in Section 2.2, we treat Minimal DOM as a garbage collected language. We therefore use true in the specification (in addition to, and disjoint from, our schema invariant) to refer to

$\{\ (S \wedge P) \oplus \text{true}_G\ \}$

$\{\ \langle \textbf{addressBook} \left[\text{households} \otimes \langle \textbf{household}_{\texttt{house}} \left[\text{people} \otimes \langle \textbf{person}_{\texttt{leaver}}[\varnothing_F]\rangle_F \otimes \text{people} \otimes \langle \textbf{address}[\varnothing_F]\rangle_F \otimes \langle \textbf{phone}[\varnothing_F]\rangle_F\right]\rangle_F \otimes \text{households}\right]\rangle_G \oplus \text{true}_G\ \}$

`moveOut(house, leaver)` $\triangleq$

// Move leaver into a new house.

`addr := getParentNode(house); newHouse := createNode(`**household**`); newAddr := createNode(`**address**`); newPhone := createNode(`**phone**`);`

$$\left\{ \begin{array}{l} \langle \textbf{addressBook}_{\texttt{addr}} \left[\text{households} \otimes \langle \textbf{household}_{\texttt{house}} \left[\text{people} \otimes \langle \textbf{person}_{\texttt{leaver}}[\varnothing_F]\rangle_F \otimes \text{people} \otimes \langle \textbf{address}[\varnothing_F]\rangle_F \otimes \langle \textbf{phone}[\varnothing_F]\rangle_F\right] \rangle_F \otimes \text{households}\right]\rangle_G \\ \quad \oplus \langle \textbf{household}_{\texttt{newHouse}}[\varnothing_F]\rangle_G \oplus \langle \textbf{address}_{\texttt{newAddr}}[\varnothing_F]\rangle_G \oplus \langle \textbf{phone}_{\texttt{newPhone}}[\varnothing_F]\rangle_G \oplus \text{true}_G \end{array} \right\}$$

`appendChild(newHouse, leaver); appendChild(newHouse, newAddr); appendChild(newHouse, newPhone); appendChild(addr, newHouse);`

$$\left\{ \left\langle \textbf{addressBook}_{\texttt{addr}} \begin{bmatrix} \text{households} \otimes \langle \textbf{household}_{\texttt{house}} \left[\text{people} \otimes \text{people} \otimes \langle \textbf{address}[\varnothing_F]\rangle_F \otimes \langle \textbf{phone}[\varnothing_F]\rangle_F\right]\rangle_F \otimes \\ \text{households} \otimes \langle \textbf{household}_{\texttt{newHouse}} \left[\langle \textbf{person}_{\texttt{leaver}}[\varnothing_F]\rangle_F \otimes \langle \textbf{address}[\varnothing_F]\rangle_F \otimes \langle \textbf{phone}[\varnothing_F]\rangle_F\right]\rangle_F \end{bmatrix} \right\rangle_G \oplus \text{true}_G \right\}$$

// Check if old household is empty...

`kids := getChildNodes(house); firstChild := item(kids, 0); firstName := getNodeName(firstChild);`

`if firstName =` **person** `then skip`

$$\left\{ \left\langle \textbf{addressBook}_{\texttt{addr}} \begin{bmatrix} \text{households} \otimes \langle \textbf{household}_{\texttt{house}} \left[\langle \textbf{person}[\varnothing_F]\rangle_F \otimes \text{people} \otimes \langle \textbf{address}[\varnothing_F]\rangle_F \otimes \langle \textbf{phone}[\varnothing_F]\rangle_F\right]\rangle_F \otimes \\ \text{households} \otimes \langle \textbf{household}_{\texttt{newHouse}} \left[\langle \textbf{person}[\varnothing_F]\rangle_F \otimes \langle \textbf{address}[\varnothing_F]\rangle_F \otimes \langle \textbf{phone}[\varnothing_F]\rangle_F\right]\rangle_F \end{bmatrix} \right\rangle_G \oplus \text{true}_G \right\} \Rightarrow \{\ S \oplus \text{true}_G\ \}$$

`else removeChild(addr, house);`

$$\{\ \langle \textbf{addressBook}_{\texttt{addr}} \left[\text{households} \otimes \text{households} \otimes \langle \textbf{household}_{\texttt{newHouse}} \left[\langle \textbf{person}[\varnothing_F]\rangle_F \otimes \langle \textbf{address}[\varnothing_F]\rangle_F \otimes \langle \textbf{phone}[\varnothing_F]\rangle_F\right]\rangle_F\right]\rangle_G \oplus \text{true}_G\ \} \Rightarrow \{\ S \oplus \text{true}_G\ \}$$

Figure 5. Schema Preservation Derivation

uncollected garbage which may safely be ignored. The proof for the specification is given in Figure 5.

6. Conclusion

Using Context Logic, we have developed local Hoare reasoning about Minimal DOM. Our reasoning is compositional and complete for straight-line code, which means that we can focus on a minimal set of DOM commands and prove invariant properties about simple programs.

We made the deliberate choice to work with the DOM tree structure (the trees, forests and groves), rather than the full DOM structure which also consists of text, attributes, etc. The tree structure is fundamental to DOM, since DOM views the other structures as nodes with simpler properties than tree nodes. We took the view that it was important to understand the reasoning of the fundamental tree structure first. We will extend our reasoning to full DOM in future, although we conjecture that there will be little additional conceptual reasoning in this extension.

We are at the beginning of our DOM project. We also aim to prove that an implementation of Minimal DOM is correct. A DOM implementation should have the same behaviour as other DOM implementations on different distributed sites. This only works if the implementation really does conform with DOM. We observed that, until recently [Ore07], Python mini-DOM was incorrect [Smi06][Whe07]. Since DOM is written in English, it is understandable that such errors occur. However, with our formal specification it is possible to prove that an implementation is correct. We are currently working on a DOM library for Smallfoot [BCO06], the verification tool for reasoning about C-programs using Separation Logic. In future, we aim to integrate our high-level reasoning about Minimal DOM with this low-level DOM library.

We will also explore a prototype verification tool for reasoning initially about Minimal DOM, and then about full DOM. The last example in Section 5.3, which verifies that an XML schema for describing an address book is an invariant of a simple Javascript program which moves a person to a new address, is particularly enticing. We would like to discover whether it is possible to provide a 'one-click' tool that checks if embedded Javascript in a web page can ever violate the schema assertions on that web page. We will first assess the expressivity of XML schema on the basic XML-tree structure with the Context-logic reasoning described here, and fully assess what sort of reasoning about Minimal DOM is possible by hand. We will then search for a decidable fragment of Context Logic, which hopefully captures enough of the schema reasoning, taking inspiration from the Smallfoot tool which verifies invariant properties of C-programs for manipulating lists.

References

[BCC+07] Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter O'Hearn, Thomas Wies, and Hongseok Yang. Shape analysis for composite data structures. In *CAV*, 2007.

[BCO06] J. Berdine, C. Calcagno, and P.W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *Proceedings of FMCO*, volume 4111 of *LNCS*, pages 115–137. Springer-Verlag, 2006.

[CGZ05] C. Calcagno, P. Gardner, and U. Zarfaty. Context logic & tree update. In *Proceedings of POPL*, pages 271–282. ACM Press, 2005.

[IO01] S. Isthiaq and P.W. O'Hearn. BI as an assertion language for mutable data structures. In *Proceedings of POPL*, pages 14–26. ACM Press, 2001.

[O'H05] Peter W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 2005.

[Ore07] Jason Orendorff. Compliance Patches for minidom. Included with Python, April 2007. Patches documented in the issue tracker at `http://bugs.python.org/issue1704134`.

[ORY01] P.W. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proceedings of CSL*, volume 2142 of *LNCS*, pages 1–19. Springer-Verlag, 2001.

[Smi06] Gareth Smith. A context logic approach to analysis and specification of xml update. PhD first year report, 2006.

[Var06] Various. Python: xml.dom.minidom. Included with Python, Documentation last updated September 2006. Documentation available at `http://docs.python.org/lib/module-xml.dom.minidom.html`.

[W3C00] W3C. Document Object Model (DOM) Level 1 Specification (2nd Edition). W3C working draft, September 2000. Available at `http://www.w3.org/TR/2000/WD-DOM-Level-1-20000929/`.

[W3C05] W3C. DOM: Document Object Model. W3C recommendation, Janurary 2005. Available at `http://www.w3.org/DOM/`.

[Whe07] Mark Wheelhouse. Dom: Towards a formal specification. Master's thesis, Imperial College, 2007.