Local Hoare Reasoning about DOM

Philippa A. Gardner, Gareth D. Smith, Mark J. Wheelhouse, Uri D. Zarfaty
Imperial College London, UK
{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 simple tree update to this real-world DOM application. Our reasoning not only formally specifies a significant subset of DOM Core Level 1, but can also be used to verify, for example, invariant properties of simple Javascript programs.

Categories and Subject Descriptors

F.3.1 [**Specifying and Verifying and Reasoning about Programs**]: Logics of programs

General Terms

Languages, Theory, Verification

1. INTRODUCTION

The Document Object Model (DOM) [13] 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 provides an interesting example of a library which is important enough to be given a comparatively formal specification. Over the years, it has evolved into a specification which, by consensus, seems to be correct. DOM is, however, written in English. This means that it is liable to misinterpretation; until recently, the Python mini-DOM implementation was incorrect. It also means that DOM is not compositional, in the sense that a specification of a composite command cannot be determined directly from the specification of its parts. DOM is thus larger than it need be, specifying for example composite commands such as `getPreviousSibling`. It also means that DOM is not complete; for example, it specifies the `insertBefore` command, but not its sister command `insertAfter`.

We provide a compositional specification of a significant fragment of DOM. Unlike DOM, we are able to work with a minimal set of commands and obtain complete reasoning for straight-line code. We do this by transferring techniques from program verification for reasoning about heaps [8] to XML, viewing XML as an in-place memory store as does DOM. In particular, O'Hearn, Reynolds and Yang have recently introduced local Hoare reasoning about shared data structures in memory using Separation Logic [8]. Calcagno, Gardner and Zarfaty adapted this work to provide local Hoare reasoning about simple tree update [4], introducing Context Logic for reasoning about both data and contexts. In this paper, we demonstrate that their reasoning scales to DOM's richer tree structure and update commands.

The Document Object Model

The documentation for DOM is substantial [14]. 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'. In this paper, we are 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.

PODS'08 June 9–12,2008, Vancouver, BC, Canada.

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 and simple text nodes, rather than dealing directly with more complex content of the structure such as Attributes, DocumentFragments, etc. The main conceptual difficulties lie with this tree and text structure; the other structures are presented in DOM as nodes with similar 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. For example, 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.

We observed that this safety error condition had been missed by Python mini-DOM [11, 15]; Orendorff has recently provided a patch which corrects this error [10]. The documentation for Python mini-DOM [12] 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 first 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. The operation therefore silently went ahead, creating a structure with a loop. If the loop was subsequently traversed by a program, then the program would diverge. With our style of local reasoning, this fundamental error in basic update behaviour would have been avoided.

Local Hoare Reasoning

We provide local Hoare reasoning about our fragment of DOM Core Level 1, inspired by 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 [8]. They summarise their key idea as follows:

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

Their work is having substantial impact, proving to be important for modular reasoning about large programs [1] and for concurrent, distributed programming [9]. It has led to the verification tool Smallfoot [2], a tool for automatically checking the absence of memory errors in C-programs based on a decidable fragment of Separation Logic. Ideas from Smallfoot are now being used in the program analysis tool SLAyer being developed at Microsoft Research Cambridge.

Following O'Hearn *et al.'s* work, Calcagno, Gardner and Zarfaty developed local Hoare reasoning about a simple tree-update language using Context Logic [4]. This work required a fundamental shift in how we reason about structured data. The key idea is that 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 disjointly about both data and this place of insertion (contexts).

In this paper, we provide local Hoare reasoning for our fragment of DOM Core Level 1, using Context Logic applied to DOM's comparatively rich tree structure. We illustrate our reasoning using 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 the formula

$$\exists \mathtt{name}, \mathtt{name}'.\, (\varnothing_{\mathrm{F}} \multimap (\mathtt{cg} \circ \mathtt{name}_{\mathtt{parent}}[\mathrm{true}_{\mathrm{F}}])) \circ \langle \mathtt{name}'_{\mathtt{newChild}} \rangle_{\mathrm{F}}$$

This formula states that the data structure can be split into two disjoint parts: a subforest containing a tree satisfying formula $\mathtt{name}'_{\mathtt{newChild}}$, which states that the top node is `newChild`, and a context satisfying formula $(\varnothing_{\mathrm{F}} \multimap (\mathtt{cg} \circ \mathtt{name}_{\mathtt{parent}}[\mathrm{true}_{\mathrm{F}}]))$ which states that, when the empty forest $\varnothing_{\mathrm{F}}$ is put in the hole, then the result contains 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 postcondition is

$$\exists \mathtt{name}, \mathtt{name}'.\, \mathtt{cg} \circ (\mathtt{name}_{\mathtt{parent}}[\mathrm{true}_{\mathrm{F}} \otimes \langle \mathtt{name}'_{\mathtt{newChild}} \rangle_{\mathrm{F}}])$$

This formula states that the resulting store has a changed subtree `parent`, which now has the subforest $\langle \mathtt{name}'_{\mathtt{newChild}} \rangle_{\mathrm{F}}$ at the end of its list of children.

Our specification is compositional in the sense that two Hoare triples compose if the postcondition of the first logically implies the precondition of the second. This compositionality enables us to focus on a minimal set of update commands, whereas DOM has to specify all the commands for which a specification is useful: e.g., we can derive the specification of the command `getPreviousSibling` with our reasoning, whereas DOM specifies it directly. Our specification is complete for straight-line code, which we prove using a standard technique of deriving the weakest preconditions of our commands: e.g., unlike DOM we can derive a specification of `insertAfter`. Finally, our reasoning can be used to verify invariant properties of simple programs: e.g., we show that a program for moving a person to a new address in an address book satisfies an XML schema invariant specifying that an XML document is indeed an address book.

2. MINIMAL DOM

We describe Minimal DOM, capturing the fragment of DOM Core Level 1 which deals with tree and text update. DOM is specified in an object-oriented manner, and hence encapsulates both data and behaviour into objects. We

separate the concerns, by first presenting an abstract data structure and then commands over that structure. This approach is consistent with DOM's 'simplified' view where all manipulation is done via the Node interface.

2.1 The Tree Structure

We focus on the basic XML tree structure with simple text content, presenting an abstract data structure consisting of trees, forests, groves and strings. Trees $\mathbf{t}$ correspond to (part of) the Node interface. Forests $\mathbf{f}$ are lists of trees and correspond to the sub-collections of the NodeList interface. Complete forests $[\mathbf{f}]_{\mathbf{fid}}$ with identifier $\mathbf{fid}$ correspond directly to the NodeList interface. Groves $\mathbf{g}$ are sets of completed trees and correspond to the object store in which Nodes exist. Strings $\mathbf{s}$ correspond to the type DOMString.

Like DOM, we update our data in place. This means that we must refer to subdata directly. Each node and list of children therefore has a unique identifier which can be directly referenced by a program using program variables.

Definition 1. (Trees, forests, groves and strings). Given an infinite set Id of node identifiers and a finite set Char of text characters, with a distinguished character #, we define trees $\mathbf{t} \in \mathrm{T}$, forests $\mathbf{f} \in \mathrm{F}$, groves $\mathbf{g} \in \mathrm{G}$ and strings $\mathbf{s} \in \mathrm{S}$ by: (with $\mathbf{id}, \mathbf{fid} \in \mathrm{Id}$ and $\mathbf{c} \in \mathrm{Char}$)

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

For well-formedness, the identifiers must be unique. We write $\mathbf{d}$ to denote arbitrary data, and $\mathbf{id} \notin \mathbf{d}$ for when the identifier $\mathbf{id}$ is not found in $\mathbf{d}$. There is a simple structural congruence, denoted $\equiv$, stating that $\otimes$ is associative with identity $\varnothing_{\mathrm{F}}$, that $\oplus$ is associative and commutative with identity $\varnothing_{\mathrm{G}}$, and that $\cdot$ is associative with identity $\varnothing_{\mathrm{S}}$. We use a notational shorthand for strings, writing abc for $\mathsf{a} \cdot \mathsf{b} \cdot \mathsf{c}$. Trees are built from *element* nodes $\mathbf{s}_{\mathbf{id}}$ and *text* nodes $\#\mathsf{text}_{\mathbf{id}}$, where $\#\mathsf{text}$ is a string with the distinguished character # and the string $\mathbf{s}$ is an element name which must not include the # character; we write EltNames $\subset$ S to denote this set of strings without #. We write $|\mathbf{f}|$, $|\mathbf{g}|$ and $|\mathbf{s}|$ for the length of a forest, size of a grove or length of a string respectively.

Notice that the list of children is given its own identifier $\mathbf{fid}$, following DOM which refers to such child lists directly (for example, see the DOM command `getChildNodes` in section 2.2). As an example, consider the XML structure:

```
⟨household⟩
  ⟨person⟩Tricia McMillan⟨/person⟩
  ⟨address⟩Islington⟨/address⟩
  ⟨phone⟩020 267 7091⟨/phone⟩
⟨/household⟩
```

which can be written as the tree:

$$\begin{array}{l} \mathsf{household}_{\mathbf{id}_1}[\\ \quad \langle \mathsf{person}_{\mathbf{id}_2}[\langle \#\mathsf{text}_{\mathbf{id}_6}[\mathsf{Tricia\ McMillan}] \rangle_{\mathrm{F}}]_{\mathbf{fid}_2} \rangle_{\mathrm{F}} \otimes \\ \quad \langle \mathsf{address}_{\mathbf{id}_3}[\langle \#\mathsf{text}_{\mathbf{id}_7}[\mathsf{Islington}] \rangle_{\mathrm{F}}]_{\mathbf{fid}_3} \rangle_{\mathrm{F}} \otimes \\ \quad \langle \mathsf{phone}_{\mathbf{id}_4}[\langle \#\mathsf{text}_{\mathbf{id}_8}[\mathsf{020\ 267\ 7091}] \rangle_{\mathrm{F}}]_{\mathbf{fid}_4} \rangle_{\mathrm{F}}]_{\mathbf{fid}_1} \end{array}$$

using an arbitrary choice of unique identifiers. We do not give identifiers to arbitrary forests $\mathbf{f}$ or strings, only to complete lists $[\mathbf{f}]_{\mathbf{fid}}$, as in DOM.

We also define natural contexts for each of our data structures. Contexts are not used in DOM. We shall see that they are essential for the context reasoning described in Section 3.

Definition 2. (Contexts). Given an infinite set Id of node identifiers and a finite set Char of single text characters, we define tree contexts $\mathbf{ct} \in \mathrm{CT}$, forest contexts $\mathbf{cf} \in \mathrm{CF}$, grove contexts $\mathbf{cg} \in \mathrm{CG}$ and string contexts $\mathbf{cs} \in \mathrm{CS}$ by

$$\begin{array}{rrl} \text{tree contexts} & \mathbf{ct} & ::= -_{\mathrm{T}} \mid \mathbf{s}_{\mathbf{id}}[\mathbf{cf}]_{\mathbf{fid}} \mid \#\mathsf{text}_{\mathbf{id}}[\mathbf{cs}] \\ \text{forest contexts} & \mathbf{cf} & ::= -_{\mathrm{F}} \mid \langle \mathbf{ct} \rangle_{\mathrm{F}} \mid \mathbf{cf} \otimes \mathbf{f} \mid \mathbf{f} \otimes \mathbf{cf} \\ \text{grove contexts} & \mathbf{cg} & ::= -_{\mathrm{G}} \mid \langle \mathbf{ct} \rangle_{\mathrm{G}} \mid \mathbf{cg} \oplus \mathbf{g} \\ \text{string contexts} & \mathbf{cs} & ::= -_{\mathrm{S}} \mid \mathbf{cs} \cdot \mathbf{s} \mid \mathbf{s} \cdot \mathbf{cs} \end{array}$$

As before, we have element names in EltNames, unique identifiers, and an analogous congruence on contexts.

Given data types $\mathrm{D}_1, \mathrm{D}_2 \in \{\mathrm{T}, \mathrm{F}, \mathrm{G}, \mathrm{S}\}$, we sometimes write $\mathbf{cd} \colon \mathrm{D}_1 \to \mathrm{D}_2$ to denote a context $\mathbf{cd} \in \mathrm{C}\mathbb{D}_2$ 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 [4]: string contexts have just string holes, but tree and forest contexts have tree, forest and string holes, and 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 3. (Context application). Given data types $\mathrm{D}_1, \mathrm{D}_2 \in \{\mathrm{T}, \mathrm{F}, \mathrm{G}, \mathrm{S}\}$, the partial application function $\mathrm{ap} : (\mathrm{D}_1 \to \mathrm{D}_2) \times \mathrm{D}_1 \rightharpoonup \mathrm{D}_2$ is defined by induction on the structure of the first argument (and undefined where not given):

$$\begin{array}{ll} \mathrm{ap}(-_{\mathrm{T}}, \mathbf{t}) & \triangleq \mathbf{t} \\ \mathrm{ap}(\mathbf{s}_{\mathbf{id}}[\mathbf{cf}]_{\mathbf{fid}}, \mathbf{d}_1) & \triangleq \mathbf{s}_{\mathbf{id}}[\mathrm{ap}(\mathbf{cf}, \mathbf{d}_1)]_{\mathbf{fid}} \text{ if } \mathbf{id}, \mathbf{fid} \notin \mathbf{d}_1 \\ & \quad \text{else undefined} \\ \mathrm{ap}(\#\mathsf{text}_{\mathbf{id}}[\mathbf{cs}], \mathbf{s}) & \triangleq \#\mathsf{text}_{\mathbf{id}}[\mathrm{ap}(\mathbf{cs}, \mathbf{s})] \\ \\ \mathrm{ap}(-_{\mathrm{F}}, \mathbf{f}) & \triangleq \mathbf{f} \\ \mathrm{ap}(\langle \mathbf{ct} \rangle_{\mathrm{F}}, \mathbf{d}_1) & \triangleq \langle \mathrm{ap}(\mathbf{ct}, \mathbf{d}_1) \rangle_{\mathrm{F}} \\ \mathrm{ap}(\mathbf{cf} \otimes \mathbf{f}, \mathbf{d}_1) & \triangleq \mathrm{ap}(\mathbf{cf}, \mathbf{d}_1) \otimes \mathbf{f} \\ \mathrm{ap}(\mathbf{f} \otimes \mathbf{cf}, \mathbf{d}_1) & \triangleq \mathbf{f} \otimes \mathrm{ap}(\mathbf{cf}, \mathbf{d}_1) \\ \\ \mathrm{ap}(-_{\mathrm{G}}, \mathbf{g}) & \triangleq \mathbf{g} \\ \mathrm{ap}(\langle \mathbf{ct} \rangle_{\mathrm{G}}, \mathbf{d}_1) & \triangleq \langle \mathrm{ap}(\mathbf{ct}, \mathbf{d}_1) \rangle_{\mathrm{G}} \\ \mathrm{ap}(\mathbf{cg} \oplus \mathbf{g}, \mathbf{d}_1) & \triangleq \mathrm{ap}(\mathbf{cg}, \mathbf{d}_1) \oplus \mathbf{g} \\ \\ \mathrm{ap}(-_{\mathrm{S}}, \mathbf{s}) & \triangleq \mathbf{s} \\ \mathrm{ap}(\mathbf{cs} \cdot \mathbf{s}', \mathbf{s}) & \triangleq \mathrm{ap}(\mathbf{cs}, \mathbf{s}) \cdot \mathbf{s}' \\ \mathrm{ap}(\mathbf{s}' \cdot \mathbf{cs}, \mathbf{s}) & \triangleq \mathbf{s}' \cdot \mathrm{ap}(\mathbf{cs}, \mathbf{s}) \end{array}$$

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, S, $\mathbb{Z}$ and $\mathbb{B}$. The expressions consist of variables and constants, concatenation on strings, arithmetic operations on integers, and logical operations on booleans. Variables of type Id also permit a **null** value, recording the absence of a node (e.g., 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 like standard imperative programs do not refer to literal heap addresses.

Definition 4. (VARIABLE STORE). The variable store s is a total function sending variables to values. The store contains four types of store variable: identifier variables $\mathrm{Var}_{\mathrm{ID}} = \{\mathtt{id}, \mathtt{fid}, \mathtt{list}, \mathtt{newChild}, \dots\}$, string variables $\mathrm{Var}_{\mathrm{S}} = \{\mathtt{str}, \mathtt{name}, \dots\}$, integer variables $\mathrm{Var}_{\mathbb{Z}} = \{\mathtt{int}, \mathtt{length}, \dots\}$ and boolean variables $\mathrm{Var}_{\mathbb{B}} = \{\mathtt{bool}, \dots\}$:

$$s : (\mathrm{Var}_{\mathrm{ID}} \to \mathrm{ID} \uplus \{\mathbf{null}\}) \times (\mathrm{Var}_{\mathrm{S}} \to \mathrm{S}) \times (\mathrm{Var}_{\mathbb{Z}} \to \mathbb{Z}) \times (\mathrm{Var}_{\mathbb{B}} \to \mathbb{B})$$

The notation $\mathrm{VAR}_{\mathrm{STORE}}$ denotes the set of store variables. By convention, all variable names are in lower camelCase. Note that, unlike element node names, string variable values may contain any character.

Definition 5. (EXPRESSIONS). Expressions build up values from constants and variables. There are four types: identifier expressions $\mathrm{Exp}_{\mathrm{ID}} = \{\mathtt{Id}, \dots\}$, string expressions $\mathrm{Exp}_{\mathrm{S}} = \{\mathtt{Str}, \mathtt{Arg}, \dots\}$, integer expressions $\mathrm{Exp}_{\mathbb{Z}} = \{\mathtt{Int}, \mathtt{Offset}, \dots\}$ and boolean expressions $\mathrm{Exp}_{\mathbb{B}} = \{\mathtt{Bool}, \dots\}$:

$$\begin{aligned} \mathtt{Id} &::= \mathbf{null} \mid \mathtt{id} \\ \mathtt{Str} &::= \varnothing_{\mathrm{S}} \mid \mathbf{c} \mid \mathtt{str} \mid \mathtt{Str} \cdot \mathtt{Str} \\ \mathtt{Int} &::= \mathbf{n} \mid \mathtt{int} \mid \mathtt{Int} + \mathtt{Int} \mid \mathtt{Int} - \mathtt{Int} \\ \mathtt{Bool} &::= \mathbf{false} \mid \mathtt{bool} \mid \mathtt{Bool} \Rightarrow \mathtt{Bool} \mid \mathtt{Id} = \mathtt{Id} \\ &\quad \mid \mathtt{Str} = \mathtt{Str} \mid \mathtt{Int} = \mathtt{Int} \mid \mathtt{Int} > \mathtt{Int} \end{aligned}$$

where $\mathtt{id} \in \mathrm{Var}_{\mathrm{ID}}$, $\mathbf{c} \in \mathrm{CHAR}$, $\mathtt{str} \in \mathrm{Var}_{\mathrm{S}}$, $\mathbf{n} \in \mathbb{Z}$, $\mathtt{int} \in \mathrm{Var}_{\mathbb{Z}}$ and $\mathtt{bool} \in \mathrm{Var}_{\mathbb{B}}$. Evaluation $[\![\mathtt{Exp}_V]\!]s$ of expression $\mathtt{Exp}_V$ on store s, for $\mathrm{V} \in \{\mathrm{ID}, \mathrm{S}, \mathbb{Z}, \mathbb{B}\}$, is defined as expected. By convention, expression names are in upper CamelCase.

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. We therefore specify the methods of the Node interface as imperative commands over a shared grove: e.g., the method call '`p.appendChild(c)`' becomes the command '`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: e.g., the CharacterData attribute '`n.data`' can be represented by the commands '`getData(n)`' and '`setData(n)`' while the '`n.parentNode`' attribute can be represented by the '`getParentNode(n)`' command alone. Some attributes and methods in the Node interface are omitted from Minimal DOM since they are not concerned with the tree structure or contents of text nodes. 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 commands `createNode` and `createTextNode`, which here perform the same functions as the DOM Document methods createElement and createTextNode.

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. For convenience, we implicitly assume procedural recursion.

Definition 6. (MINIMAL DOM). The Minimal DOM commands are: (with the arguments named as in DOM)

`C ::= appendChild(parent, newChild)`	append tree
`| removeChild(parent, oldChild)`	remove child
`| name := getNodeName(node)`	get node name
`| id := getParentNode(node)`	get parent node
`| fid := getChildNodes(node)`	get child nodes
`| node := createNode(Name)`	new elt. node
`| node := item(list, Int)`	get forest elt.
`| str:=substringData(node,Offset,Count)`	get substring
`| appendData(node, Arg)`	append string
`| deleteData(node, Offset, Count)`	erase substring
`| node := createTextNode(Str)`	new text node
`| id:=Id|str:=Str|int:=Int|bool:=Bool`	assignment
`| C ; C`	sequencing
`| if Bool then C else C`	if-then-else
`| while Bool do C`	while-do
`| skip`	skip

The DOM commands have the following behaviour:

`appendChild(parent, newChild)` moves the tree `newChild` to the end of `parent`'s child list. Requires that `parent` exists and is not a text node, and that `newChild` exists and is not an ancestor of `parent`, or it faults.

`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`, or it faults.

`name := getNodeName(node)` assigns to the variable `name` the nodeName value of `node` if it exists, or it faults. If `node` is a text node, then `name` = #text.

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

`fid := getChildNodes(node)` assigns to the variable `fid` the identifier of the child forest of the element `node`, which must exist or it faults.

`node := createNode(Name)` creates a new element node at the root of the grove, with fresh `id` and `fid` and a name equal to `Name`, and records its identifier in the variable `node`. It faults if $[\![\mathtt{Name}]\!]s \notin \mathrm{ELTNAMES}$.

`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. It faults if `list` does not exist.

`str := substringData(node,Offset,Count)` assigns to the variable `str` the substring of the string of the text node `node` starting at character `Offset` with length `Count`. If `Offset` + `Count` exceeds the string length, then all the characters to the string end are returned. Requires that `node` exists, `Offset` and `Count` be non-negative, and `Offset` be at most the string length, or it faults.

`appendData(node, Arg)` appends the string `Arg` to the end of the string contained in `node`. Requires that `node` exists and be a text node, or it faults.

`deleteData(node, Offset, Count)` deletes the substring of the string of `node` starting at the character `Offset` with length `Count`. If `Offset` + `Count` exceeds the string length, then all the characters to the string end are deleted. Requires that `node` exists, `Offset` and `Count` be non-negative, and `Offset` be at most the string length, or it faults.

`node := createTextNode(Str)` creates a new text node at the grove level, with fresh `id` and the string contained within the text node set to `Str`, and records the new node's identifier in the variable `node`.

The behaviour of these Minimal DOM commands can be described formally by an operational semantics (see [6]), using an evaluation relation $\rightsquigarrow$ relating configuration triples $\mathtt{C}, s, \mathbf{g}$ where $\mathtt{C}$ is a command, s is a store and $\mathbf{g}$ a grove, to either terminal states $s', \mathbf{g}'$, or fault. Notice that `removeChild` does not delete the tree identified by `oldChild`, but instead moves it to the grove. It is not possible to delete a tree from the grove in Minimal DOM. This follows 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) as garbage collected programs.

DOM operations raise exceptions in 'exceptional circumstances, i.e., when an operation is impossible to perform' [DOM, Section 1.2]. Examples include: trying to move a tree into its own subtree using `appendChild`; trying to create a node with `#text` as a name; and trying 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'.

THEOREM 1 (MINIMALITY/SUFFICIENCY). *Minimal DOM is indeed minimal, and sufficient to describe the tree structure and text nodes of DOM Core Level 1.*

3. CONTEXT LOGIC APPLIED TO DOM

We study Context Logic for analysing our DOM tree structure, following previous work on Context Logic for analysing simple trees (forests) [16]. Although the basic reasoning is the same, the transition was by no means easy due to the comparative complexity of the DOM structures.

With Context Logic, the fundamental idea is that, in order to provide local Hoare reasoning about tree update, we must reason about both trees and the interim contexts. Such local Hoare reasoning is not possible using, for example, Ambient Logic [5], despite its reasoning about static trees being analogous to the Separation Logic reasoning about heaps.

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 $\mathrm{D}, \mathrm{D}_1, \mathrm{D}_2 \in \{\mathrm{T}, \mathrm{F}, \mathrm{G}, \mathrm{S}\}$,

- the application formula $P \circ_{\mathrm{D}_1} P_1$ describes data of e.g. type D_2, which can be split into a context of type $\mathrm{D}_1 \to \mathrm{D}_2$ satisfying P, and disjoint subdata of type 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 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 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; and
- the right adjoint $P_1 \mathrel{-\!\!\circ} P_2$ describes a context of e.g. type $\mathrm{D}_1 \to \mathrm{D}_2$ which, *whenever* data of type D_1 satisfying P_1 is successfully inserted into it, results in data of type D_2 satisfying P_2; there is no type annotation as it can be inferred from the type of the given data.

The DOM-specific formulae analyse the data and context structure of DOM, and have a direct correspondence with the data structures given in definitions 1 and 2. For example, the specific forest formula $\langle \mathtt{p}_{\mathtt{id}}[\mathrm{true}_{\mathrm{F}}]_{\mathtt{fid}} \rangle_{\mathrm{F}}$ describes a forest containing one tree with top node labelled `p`, an arbitrary subforest, and identifiers determined by the values of the identifier variables given by the store. The formula

$$\exists \mathtt{id}, \mathtt{fid}.\, \mathrm{true}_{\mathrm{F}\to\mathrm{T}} \circ_{\mathrm{F}} \langle \mathtt{p}_{\mathtt{id}}[\mathrm{true}_{\mathrm{F}}]_{\mathtt{fid}} \rangle_{\mathrm{F}}$$

describes a tree which can be split into a context and a subforest containing one tree with top node labelled `p`. Formula

$$\exists \mathtt{id}, \mathtt{id}', \mathtt{fid}, \mathtt{fid}'.\, (\varnothing_{\mathrm{F}} \mathrel{-\!\!\circ} P) \circ_{\mathrm{F}} \langle \mathtt{p}_{\mathtt{id}}[\mathrm{true}_{\mathrm{F}}]_{\mathtt{fid}} \rangle_{\mathrm{F}}$$

describes a tree that can be split into a context and a subforest containing a tree with top node `p`. This time the context satisfies the property that, when the empty forest is put into the context hole, the resulting tree satisfies formula P.

Recall that Minimal DOM uses identifier, string, integer and boolean variables. For our Hoare reasoning, we also require tree, forest, grove and context variables. Our logic therefore uses a logical environment e as well as the store s. The logical environment assigns values for data variables in $\mathrm{Var}_{\mathrm{D}'}$, where $\mathrm{D}' \in \{\mathrm{T}, \mathrm{F}, \mathrm{G}\}$; string variables are excluded from the environment, as they are also program variables and are hence assigned values via the store. The environment also assigns values to context variables $\mathrm{Var}_{\mathrm{D}_1 \to \mathrm{D}_2}$. For simplicity, we let $\mathrm{D}_1, \mathrm{D}_2 \in \{\mathrm{T}, \mathrm{F}, \mathrm{G}, \mathrm{S}\}$. However, there are no contexts $\mathrm{R} \to \mathrm{S}$, $\mathrm{G} \to \mathrm{R}$ or $\mathrm{G} \to \mathrm{S}$ for $\mathrm{R} \in \{\mathrm{T}, \mathrm{F}\}$, so we assume the corresponding variable sets are empty.

Definition 7. (LOGICAL ENVIRONMENT). A logical environment e is a total function sending data variables $\mathrm{Var}_{\mathrm{D}'}$ and context variables $\mathrm{Var}_{\mathrm{D}_1 \to \mathrm{D}_2}$ to their values:

$$e : \prod_{\mathrm{D}' \in \{\mathrm{T},\mathrm{F},\mathrm{G}\}} \mathrm{Var}_{\mathrm{D}'} \to \mathrm{D}' \times \prod_{\mathrm{D}_1,\mathrm{D}_2 \in \{\mathrm{T},\mathrm{F},\mathrm{G},\mathrm{S}\}} (\mathrm{Var}_{\mathrm{D}_1 \to \mathrm{D}_2}) \to \mathrm{D}_1 \to \mathrm{D}_2$$

$\mathrm{VAR}_{\mathrm{ENV}}$ denotes the set of environment variables.

Definition 8. (FORMULAE FOR DOM). Let A denote a data or context type of the form D or $\mathrm{D}_1 \to \mathrm{D}_2$. The formulae for DOM are defined by:

$P ::= P \Rightarrow P \mid \mathrm{false}_{\mathrm{A}}$	Boolean formulae
$P \circ_{\mathrm{D}_1} P \mid P \circ\!\!-_{\mathrm{D}_2} P \mid P \mathrel{-\!\!\circ} P$	structural formulae
$\ldots$ (see below) $\ldots$	DOM-specific formulae
$\mathtt{var}_{\mathrm{E}} \mid \mathtt{Exp}_{\mathrm{V}} = \mathtt{Exp}_{\mathrm{V}}$	expression equality
$\mathtt{Int} = \mathrm{len}(\mathtt{f}) \mid \mathtt{Int} = \mathrm{len}(\mathtt{Str})$	length equality
$\mathrm{eltName}(\mathtt{Str})$	valid element name
$\exists \mathtt{var}.\, P$	quantification

where $\mathtt{var}_{\mathrm{E}}$ denotes a logical variable, $\mathrm{V} \in \{\mathrm{ID}, \mathrm{S}, \mathbb{Z}, \mathbb{B}\}$, $\mathtt{var} \in \mathrm{VAR}_{\mathrm{ENV}} \cup \mathrm{VAR}_{\mathrm{STORE}}$. The *DOM-specific* formulae are:

$$\begin{aligned} P ::= \ldots \quad & \mid -_{\mathrm{T}} \mid P_{\mathtt{id}}[P]_{\mathtt{fid}} \mid \#\mathsf{text}_{\mathtt{id}}[P] \\ & \mid \varnothing_{\mathrm{F}} \mid -_{\mathrm{F}} \mid \langle P \rangle_{\mathrm{F}} \mid P \otimes P \\ & \mid \varnothing_{\mathrm{G}} \mid -_{\mathrm{G}} \mid \langle P \rangle_{\mathrm{G}} \mid P \oplus P \\ & \mid -_{\mathrm{S}} \mid \mathtt{Str} \mid P \cdot P \end{aligned}$$

The type annotations on the formulae enable us to define a simple typing relation $P\colon A$, 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:

$$
\begin{array}{r}
(P_1 \circ_{\mathrm{D}_1} P_2)\colon \mathrm{D}_2 \Leftrightarrow P_1\colon \mathrm{D}_1{\to}\mathrm{D}_2 \wedge P_2\colon \mathrm{D}_1 \\
(P_1 \circ\!\!-_{\mathrm{D}_2} P_2)\colon \mathrm{D}_1 \Leftrightarrow P_1\colon \mathrm{D}_1{\to}\mathrm{D}_2 \wedge P_2\colon \mathrm{D}_2 \\
(P_1 -\!\!\circ P_2)\colon \mathrm{D}_1{\to}\mathrm{D}_2 \Leftrightarrow P_1\colon \mathrm{D}_1 \wedge P_2\colon \mathrm{D}_2 \\
-_{\mathrm{T}}\colon \mathrm{T}{\to}\mathrm{T} \\
P_{\mathtt{id}}[P']_{\mathtt{fid}}\colon \mathrm{T} \Leftrightarrow P\colon \mathrm{S} \wedge P'\colon \mathrm{F} \\
P_{\mathtt{id}}[P']_{\mathtt{fid}}\colon \mathrm{R}{\to}\mathrm{T} \Leftrightarrow P\colon \mathrm{S} \wedge P'\colon \mathrm{R}{\to}\mathrm{F} \\
\#\mathsf{text}_{\mathtt{id}}[P]\colon \mathrm{T} \Leftrightarrow P\colon \mathrm{S} \\
\#\mathsf{text}_{\mathtt{id}}[P]\colon \mathrm{S}{\to}\mathrm{T} \Leftrightarrow P\colon \mathrm{S}{\to}\mathrm{S} \\
(P_1 \otimes P_2)\colon \mathrm{F} \Leftrightarrow P_1\colon \mathrm{F} \wedge P_2\colon \mathrm{F} \\
(P_1 \otimes P_2)\colon \mathrm{R}{\to}\mathrm{F} \Leftrightarrow (P_1\colon \mathrm{R}{\to}\mathrm{F} \wedge P_2\colon \mathrm{F}) \vee (P_1\colon \mathrm{F} \wedge P_2\colon \mathrm{R}{\to}\mathrm{F})
\end{array}
$$

where $\mathrm{R} \in \{\mathrm{T}, \mathrm{F}, \mathrm{S}\}$ denotes the possible tree or forest context holes. The formulae $P_{\mathtt{id}}[P_1]_{\mathtt{fid}}$ and $\#\mathsf{text}_{\mathtt{id}}[P]$ have two typings, depending on whether they describe trees or tree contexts. $P_1 \otimes P_2$ also has the two typings; notice that the context case has two options for typing subformulae, depending on which one describes the forest context.

Definition 9. (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 P. A is a data or context type and $\mathrm{R} \in \{\mathrm{T}, \mathrm{F}, \mathrm{S}\}$ as above. For the Booleans, we have:

$$
\begin{array}{ll}
e, s, \mathbf{a} \models_{\mathrm{A}} P \Rightarrow P' \Leftrightarrow e, s, \mathbf{a} \models_{\mathrm{A}} P \Rightarrow e, s, \mathbf{a} \models_{\mathrm{A}} P' \\
e, s, \mathbf{a} \models_{\mathrm{A}} \mathrm{false}_{\mathrm{A}} & \textit{never}
\end{array}
$$

For the structural formulae, we have

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

For the DOM-specific formulae, we have:

$$
\begin{array}{ll}
e, s, \mathbf{ct} \models_{\mathrm{T}\to\mathrm{T}} -_{\mathrm{T}} & \Leftrightarrow \mathbf{ct} \equiv -_{\mathrm{T}} \\
e, s, \mathbf{t} \models_{\mathrm{T}} P_{\mathtt{id}}[P']_{\mathtt{fid}} & \Leftrightarrow \exists \mathbf{s}\colon\mathrm{S}, \mathbf{f}\colon\mathrm{F}.\, (\mathbf{t} \equiv \mathbf{s}_{s(\mathtt{id})}[\mathbf{f}]_{s(\mathtt{fid})}) \wedge \\
& \quad e, s, \mathbf{s} \models_{\mathrm{S}} P \wedge e, s, \mathbf{f} \models_{\mathrm{F}} P' \\
e, s, \mathbf{ct} \models_{\mathrm{R}\to\mathrm{T}} P_{\mathtt{id}}[P']_{\mathtt{fid}} & \Leftrightarrow \exists \mathbf{s}\colon\mathrm{S}, \mathbf{cf}\colon(\mathrm{R}{\to}\mathrm{T}).\, (\mathbf{ct} \equiv \mathbf{s}_{s(\mathtt{id})}[\mathbf{cf}]_{s(\mathtt{fid})}) \\
& \quad \wedge e, s, \mathbf{s} \models_{\mathrm{S}} P \wedge e, s, \mathbf{cf} \models_{\mathrm{R}\to\mathrm{F}} P' \\
e, s, \mathbf{t} \models_{\mathrm{T}} \#\mathsf{text}_{\mathtt{id}}[P] & \Leftrightarrow \exists \mathbf{s}\colon\mathrm{S}.\, (\mathbf{t} \equiv \#\mathsf{text}_{s(\mathtt{id})}[\mathbf{s}]) \wedge e, s, \mathbf{s} \models_{\mathrm{S}} P \\
e, s, \mathbf{ct} \models_{\mathrm{S}\to\mathrm{T}} \#\mathsf{text}_{\mathtt{id}}[P] & \Leftrightarrow \exists \mathbf{cs}\colon(\mathrm{S}{\to}\mathrm{S}).\, (\mathbf{ct} \equiv \#\mathsf{text}_{s(\mathtt{id})}[\mathbf{cs}]) \wedge \\
& \quad e, s, \mathbf{cs} \models_{\mathrm{S}\to\mathrm{S}} P
\end{array}
$$

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

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

$$
\begin{array}{ll}
e, s, \mathbf{cs} \models_{\mathrm{S}\to\mathrm{S}} -_{\mathrm{S}} & \Leftrightarrow \mathbf{cs} \equiv -_{\mathrm{S}} \\
e, s, \mathbf{s} \models_{\mathrm{S}} \mathtt{Str} & \Leftrightarrow \mathbf{s} \equiv [\![\mathtt{Str}]\!]s \\
e, s, \mathbf{s} \models_{\mathrm{S}} P_1 \cdot P_2 & \Leftrightarrow \exists \mathbf{s}_1\colon\mathrm{S}, \mathbf{s}_2\colon\mathrm{S}.\, (\mathbf{s} \equiv \mathbf{s}_1 \cdot \mathbf{s}_2) \wedge \\
& \quad e, s, \mathbf{s}_1 \models_{\mathrm{S}} P_1 \wedge e, s, \mathbf{s}_2 \models_{\mathrm{S}} P_2 \\
e, s, \mathbf{cs} \models_{\mathrm{S}\to\mathrm{S}} P_1 \cdot P_2 & \Leftrightarrow \exists \mathbf{cs}'\colon(\mathrm{S}{\to}\mathrm{S}), \mathbf{s}\colon(\mathrm{S}). \\
& \quad ((\mathbf{cs} \equiv \mathbf{cs}' \cdot \mathbf{s}') \wedge e, s, \mathbf{cs}' \models_{\mathrm{S}\to\mathrm{S}} P_1 \wedge e, s, \mathbf{s}' \models_{\mathrm{S}} P_2) \vee \\
& \quad ((\mathbf{cs} \equiv \mathbf{s}' \cdot \mathbf{cs}') \wedge e, s, \mathbf{s}' \models_{\mathrm{S}} P_1 \wedge e, s, \mathbf{cs}' \models_{\mathrm{S}\to\mathrm{S}} P_2)
\end{array}
$$

Finally, for the remaining formulae, we have:

$$
\begin{array}{ll}
e, s, \mathbf{a} \models_{\mathrm{A}} \mathtt{var}_{\mathrm{E}} & \Leftrightarrow \mathbf{a} \equiv e(\mathtt{var}_{\mathrm{E}}) \\
e, s, \mathbf{a} \models_{\mathrm{A}} \mathtt{Exp}_{\mathrm{V}} = \mathtt{Exp}'_{\mathrm{V}} & \Leftrightarrow [\![\mathtt{Exp}_{\mathrm{V}}]\!]s = [\![\mathtt{Exp}'_{\mathrm{V}}]\!]s \\
e, s, \mathbf{a} \models_{\mathrm{A}} \mathtt{Int} = \mathrm{len}(\mathtt{f}) & \Leftrightarrow [\![\mathtt{Int}]\!]s = |e(\mathtt{f})| \\
e, s, \mathbf{a} \models_{\mathrm{A}} \mathtt{Int} = \mathrm{len}(\mathtt{Str}) & \Leftrightarrow [\![\mathtt{Int}]\!]s = |[\![\mathtt{Str}]\!]s| \\
e, s, \mathbf{a} \models_{\mathrm{A}} \mathrm{eltName}(\mathtt{Str}) & \Leftrightarrow [\![\mathtt{Str}]\!]s \in \textsc{EltNames} \\
e, s, \mathbf{a} \models_{\mathrm{A}} \exists \mathtt{var}_{\mathrm{E}}.\, P & \Leftrightarrow \exists \mathbf{b}.\, e[\mathtt{var}_{\mathrm{E}} \mapsto \mathbf{b}], s, \mathbf{a} \models_{\mathrm{A}} P \\
e, s, \mathbf{a} \models_{\mathrm{A}} \exists \mathtt{var}_{\mathrm{V}}.\, P & \Leftrightarrow \exists \mathbf{v}.\, e, s[\mathtt{var}_{\mathrm{V}} \mapsto \mathbf{v}], \mathbf{a} \models_{\mathrm{A}} P
\end{array}
$$

The standard classical connectives 'true', $\wedge$, $\vee$, $\neg$, $\forall$ are all derivable. 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 $\mathrm{D}_1, \mathrm{D}_2 \in \{\mathrm{T}, \mathrm{F}, \mathrm{G}, \mathrm{S}\}$. Similarly, we define the related concept of 'somewhere at this forest-level' $\Diamond_{\otimes} P$ and 'everywhere at this forest-level' $\Box_{\otimes} P$:

$$
\begin{array}{ll}
\Diamond_{\mathrm{D}_1\to\mathrm{D}_2} P \triangleq \mathrm{true}_{\mathrm{D}_1\to\mathrm{D}_2} \circ_{\mathrm{D}_1} P & \Diamond_{\otimes} P \triangleq (\mathrm{true}_{\mathrm{F}} \otimes -_{\mathrm{F}} \otimes \mathrm{true}_{\mathrm{F}}) \circ_{\mathrm{F}} P \\
\Box_{\mathrm{D}_1\to\mathrm{D}_2} P \triangleq \neg\Diamond_{\mathrm{D}_1\to\mathrm{D}_2} \neg P & \Box_{\otimes} P \triangleq \neg\Diamond_{\otimes} \neg P
\end{array}
$$

As shorthand, we write $\mathtt{Bool}$ for $\mathtt{Bool} = \mathbf{true}$ and derive:

$$
P_{\mathtt{id}}[P'] \triangleq \exists \mathtt{fid}.\, P_{\mathtt{id}}[P']_{\mathtt{fid}} \qquad P[P'] \triangleq \exists \mathtt{id}.\, P_{\mathtt{id}}[P']
$$

$$
\#\mathsf{text} \triangleq \exists \mathtt{id}.\, \#\mathsf{text}_{\mathtt{id}}[\mathrm{true}_{\mathrm{S}}]
$$

$$
\mathtt{name}_{\mathtt{id}} \triangleq \mathtt{name}_{\mathtt{id}}[\mathrm{true}_{\mathrm{F}}] \vee (\mathtt{name} = \#\mathsf{text} \wedge \#\mathsf{text}_{\mathtt{id}}[\mathrm{true}_{\mathrm{S}}])
$$

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

Example 1. (Context Logic examples).

(a) Two equivalent ways of specifying a tree containing a node with name $\mathtt{p}$, but otherwise unconstrained:

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

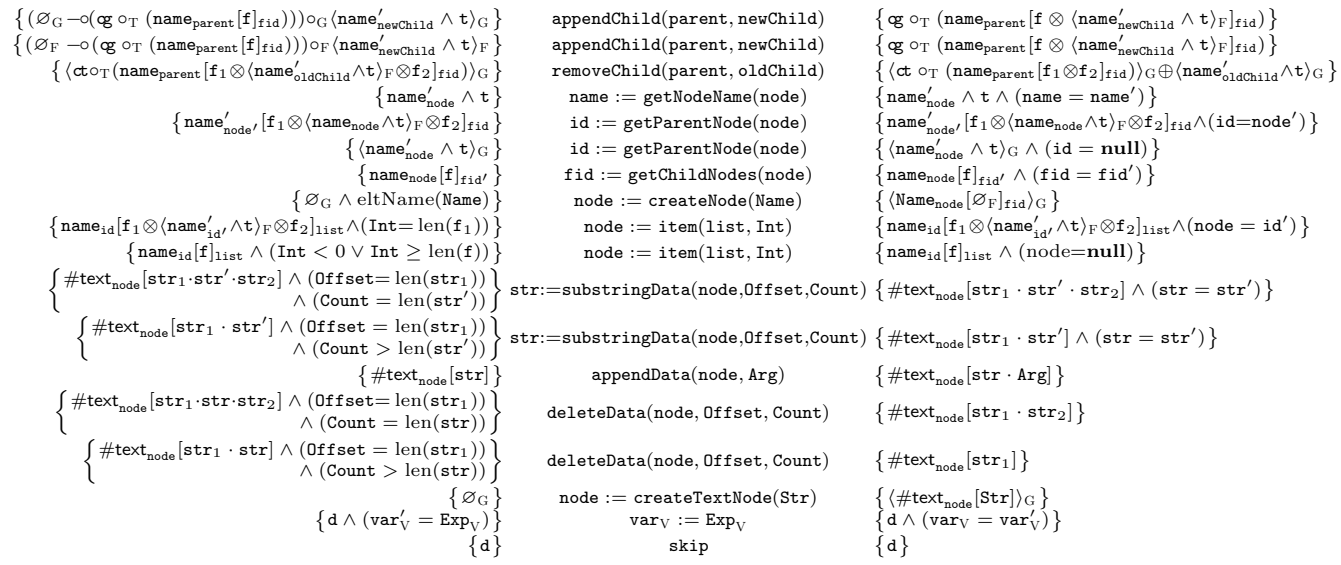
(b) A tree consisting of a body node with 0 or more paragraph nodes underneath:

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

The $\Box_{\otimes}$ constraint on the forest beneath body specifies that all the subforests that satisfy $\langle \mathrm{true}_{\mathrm{T}}\rangle_{\mathrm{F}}$—namely, the subtrees—also satisfy $\langle \mathsf{paragraph}[\mathrm{true}_{\mathrm{F}}]\rangle_{\mathrm{F}}$. This sort of formula turns out to be particularly useful when describing the XML schema invariant in Section 5.3.

(c)
$$
\langle \mathtt{name}_{\mathtt{node}} \wedge \mathtt{t}\rangle_{\mathrm{G}}
$$

A grove containing a node (which may be a text node or an element) described by the tree variable $\mathtt{t}$, with nodeName $\mathtt{name}$ and identifier $\mathtt{id}$. We use this form of exact specification to specify that certain parts of the tree remain unchanged by a command.



where $\mathtt{cg} \in \mathrm{Var}_{\mathrm{T}\to\mathrm{G}}$, $\mathtt{ct} \in \mathrm{Var}_{\mathrm{T}\to\mathrm{T}}$, $\mathtt{d} \in \mathrm{Var}_{\mathrm{D}}$ and $\mathrm{D} \in \{\mathrm{T}, \mathrm{G}\}$.

Figure 1: Minimal DOM Axioms

(d) $\exists \mathtt{cg}, \mathtt{name}, \mathtt{name}'.(\varnothing_{\mathrm{F}} \multimap (\mathtt{cg} \circ_{\mathrm{T}} (\mathtt{name}_{\mathtt{node2}}[\mathrm{true}_{\mathrm{F}}]))) \circ_{\mathrm{F}} \langle \mathtt{name}'_{\mathtt{node1}} \rangle_{\mathrm{F}}$

A grove containing the nodes `node1` and `node2`, where `node1` is not an ancestor of `node2`. This makes use of variable $\mathtt{cg} \in \mathrm{Var}_{\mathrm{T}\to\mathrm{G}}$. This sort of formula occurs in the axiom of `appendChild`.

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 [7]: 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`, `createTextNode` do act at the grove level: `appendChild` potentially takes a subtree of one grove tree and appends it to a subtree of another grove tree; the other commands result in new grove trees. However, the commands `getNodeName`, `getChildNodes`, `item`, `substringData`, `appendData`, `deleteData` 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 local Hoare triple, depending on whether we are reasoning about trees or groves. We use O'Hearn's fault-avoiding partial correctness interpretation of triples, which says that if a state satisfies a precondition, then the command cannot fault and the resulting state must satisfy the postcondition.

Definition 10. (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:

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

Notice that our interpretation of the Hoare triples on trees coerces those trees to groves using $\langle \_ \rangle_{\mathrm{G}}$. This is necessary as $\rightsquigarrow$ is defined for configuration triples containing groves.

Definition 11. (Command Axioms). The axioms for the basic Minimal DOM commands are given in Figure 1.

The `appendChild` command has two axioms, corresponding to when `newChild` has a parent node and when 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`, `substringData` and `deleteData` commands have 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 12. (Local Hoare Reasoning). The local Hoare reasoning framework consists of the command axioms given in Definition 11 and seven general inference rules: the Rules of Sequence, If Then Else, While, Consequence, Disjunction[1] and Auxiliary Variable Elimination, which are standard and so are not presented here, and the Frame Rule:

$$\text{Frame Rule: } \frac{\{P\}\ \mathtt{C}\ \{Q\}}{\{K \circ_{\mathrm{D}} P\}\ \mathtt{C}\ \{K \circ_{\mathrm{D}} Q\}}\ \mathrm{mod}(\mathtt{C}) \cap \mathrm{free}(K) = \emptyset$$

where $P, Q : D$ for $D \in \{\mathrm{T}, \mathrm{G}\}$, and K has type T→T, T→G or G→G as appropriate. The set of free variables $\mathrm{free}(K)$ is standard and $\mathrm{mod}(\mathtt{C})$ is the set of variables assigned to by $\mathtt{C}$.

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

The Frame Rule is key to local Hoare reasoning [8]. It allows properties of programs on a smaller state to be uniformly extended to properties about the larger state. Since our Hoare triples work on both trees and groves, a state may be made larger by extending a tree upwards (to obtain a larger tree or grove) or by extending a grove sideways (applying a context which contains a grove hole). The side-condition simply prevents clashes between the variables in context formula K and the store variables modified by the command.

As a sanity check, we note that the weakest preconditions of the Minimal DOM commands are derivable in the logic (see [6] for details). Our local Hoare reasoning is therefore complete for straight line code.

THEOREM 2 (WEAKEST PRECONDITIONS). *The weakest precondition triples of Minimal DOM are derivable.*

5. EXAMPLES

We present several examples of our Minimal DOM reasoning. We illustrate the minimal nature of Minimal DOM by specifying `getPreviousSibling`, a DOM Core Level 1 command that is not included in Minimal DOM (Section 5.1). We illustrate the completeness of our reasoning by specifying `insertAfter`, which is not in DOM (Section 5.2). Finally, we demonstrate the potential of our framework for reasoning about real-world programs by proving that an example program for manipulating address books maintains properties specified by an accompanying XML Schema (Section 5.3).

5.1 GetPreviousSibling

We define the DOM command `getPreviousSibling`, using an auxiliary command '`getIndex`' which is not in DOM Core Level 1 (and which is also useful in its own right). `getIndex` returns the index of a given node in a given list. The implementations of `getIndex` and `getPreviousSibling` are:

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

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

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` is described by two complementary specifications. When `node` is an element of `nodeList` (the only case used by `getPreviousSibling`), the specification is:

$$\begin{array}{l}\left\{\texttt{name}_{\texttt{id}}[\texttt{f} \otimes \langle \texttt{name}'_{\texttt{node}} \wedge \texttt{t}\rangle_{\mathrm{F}} \otimes \texttt{f}']_{\texttt{nodeList}}\right\} \\ \texttt{n} := \texttt{getIndex}(\texttt{nodeList},\texttt{node}) \\ \left\{\texttt{name}_{\texttt{id}}[\texttt{f} \otimes \langle \texttt{name}'_{\texttt{node}} \wedge \texttt{t}\rangle_{\mathrm{F}} \otimes \texttt{f}']_{\texttt{nodeList}} \wedge (\texttt{n} = \texttt{len}(\texttt{f}))\right\}\end{array}$$

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 is unaltered, and that the store now records the position of the tree `node` in the variable `n`. When `node` is not contained in `nodeList`, the store records the length of the list which is an invalid index.

`getPreviousSibling`, meanwhile, is described using three specifications, corresponding to when the node is at the grove level, the beginning of the nodeList, or elsewhere:

$$\begin{array}{l}\left\{\langle \texttt{name}_{\texttt{node}} \wedge \texttt{t}\rangle_{\mathrm{G}}\right\} \\ \texttt{sibling} := \texttt{getPreviousSibling}(\texttt{node}) \\ \left\{\langle \texttt{name}_{\texttt{node}} \wedge \texttt{t}\rangle_{\mathrm{G}} \wedge (\texttt{sibling} = \textbf{null})\right\}\end{array}$$

$$\begin{array}{l}\left\{\texttt{name}_{\texttt{id}}[\langle \texttt{name}''_{\texttt{node}} \wedge \texttt{t}'\rangle_{\mathrm{F}} \otimes \texttt{f}_2]_{\texttt{fid}}\right\} \\ \texttt{sibling} := \texttt{getPreviousSibling}(\texttt{node}) \\ \left\{\texttt{name}_{\texttt{id}}[\langle \texttt{name}''_{\texttt{node}} \wedge \texttt{t}'\rangle_{\mathrm{F}} \otimes \texttt{f}_2]_{\texttt{fid}} \wedge (\texttt{sibling} = \textbf{null})\right\}\end{array}$$

$$\begin{array}{l}\left\{\texttt{name}_{\texttt{id}}[\texttt{f}_1 \otimes \langle \texttt{name}'_{\texttt{id}'} \wedge \texttt{t}\rangle_{\mathrm{F}} \otimes \langle \texttt{name}''_{\texttt{node}} \wedge \texttt{t}'\rangle_{\mathrm{F}} \otimes \texttt{f}_2]_{\texttt{fid}}\right\} \\ \texttt{sibling} := \texttt{getPreviousSibling}(\texttt{node}) \\ \left\{\begin{array}{l}\texttt{name}_{\texttt{id}}[\texttt{f}_1 \otimes \langle \texttt{name}'_{\texttt{id}'} \wedge \texttt{t}\rangle_{\mathrm{F}} \otimes \langle \texttt{name}''_{\texttt{node}} \wedge \texttt{t}'\rangle_{\mathrm{F}} \otimes \texttt{f}_2]_{\texttt{fid}} \\ \quad \wedge (\texttt{sibling} = \texttt{id}')\end{array}\right\}\end{array}$$

The derivations for the specifications are given in Figure 2.

5.2 InsertAfter

In a similar fashion, we can implement the DOM Core Level 1 command `insertBefore`, which moves a subtree `newChild` into a `parent`'s list of children, immediately before some `refNode`. More interestingly, we can use `insertBefore` to implement `insertAfter`, which is not in DOM Core Level 1. The specifications for each of these commands have two cases: the case when `refNode` is **null** which we do not give here; and the case when `refNode` is a node in the tree or grove. In this later case, the specification of `insertBefore` is:

$$\begin{array}{l}\left\{\begin{array}{l}(\varnothing_{\mathrm{X}} \multimap_{\mathrm{o}} \texttt{cg} \circ_{\mathrm{T}} (\texttt{name}_{\texttt{parent}}[\texttt{f}_1 \otimes \langle \texttt{name}'_{\texttt{refNode}} \wedge \texttt{t}\rangle_{\mathrm{F}} \otimes \texttt{f}_2]_{\texttt{fid}})) \\ \quad \circ_{\mathrm{X}} \langle \texttt{name}''_{\texttt{newChild}} \wedge \texttt{t}'\rangle_{\mathrm{X}}\end{array}\right\} \\ \texttt{insertBefore}(\texttt{parent},\texttt{newChild},\texttt{refNode}) \\ \left\{\texttt{cg} \circ_{\mathrm{T}} (\texttt{name}_{\texttt{parent}}[\texttt{f}_1 \otimes \langle \texttt{name}''_{\texttt{newChild}} \wedge \texttt{t}'\rangle_{\mathrm{F}} \otimes \langle \texttt{name}'_{\texttt{refNode}} \wedge \texttt{t}\rangle_{\mathrm{F}} \otimes \texttt{f}_2]_{\texttt{fid}})\right\}\end{array}$$

where $\mathrm{X} \in \{\mathrm{F}, \mathrm{G}\}$ and $\texttt{cg} \in \mathrm{Var}_{\mathrm{T}\to\mathrm{G}}$. When $\texttt{refNode} \neq \textbf{null}$, `insertAfter` corresponds to two calls of `insertBefore`:

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

and has the specification:

$$\begin{array}{l}\left\{\begin{array}{l}(\varnothing_{\mathrm{X}} \multimap_{\mathrm{o}} \texttt{cg} \circ_{\mathrm{T}} (\texttt{name}_{\texttt{parent}}[\texttt{f}_1 \otimes \langle \texttt{name}'_{\texttt{refNode}} \wedge \texttt{t}\rangle_{\mathrm{F}} \otimes \texttt{f}_2]_{\texttt{fid}})) \\ \quad \circ_{\mathrm{X}} \langle \texttt{name}''_{\texttt{newChild}} \wedge \texttt{t}'\rangle_{\mathrm{X}}\end{array}\right\} \\ \texttt{insertAfter}(\texttt{parent},\texttt{newChild},\texttt{refNode}); \\ \left\{\texttt{cg} \circ_{\mathrm{T}} (\texttt{name}_{\texttt{parent}}[\texttt{f}_1 \otimes \langle \texttt{name}'_{\texttt{refNode}} \wedge \texttt{t}\rangle_{\mathrm{F}} \otimes \langle \texttt{name}''_{\texttt{newChild}} \wedge \texttt{t}'\rangle_{\mathrm{F}} \otimes \texttt{f}_2]_{\texttt{fid}})\right\}\end{array}$$

This specification may be derived compositionally from the non-**null** case of the specification of `insertBefore`:

$$\begin{array}{l}\left\{\begin{array}{l}(\varnothing_{\mathrm{X}} \multimap_{\mathrm{o}} \texttt{cg} \circ_{\mathrm{T}} (\texttt{name}_{\texttt{parent}}[\texttt{f}_1 \otimes \langle \texttt{name}'_{\texttt{refNode}} \wedge \texttt{t}\rangle_{\mathrm{F}} \otimes \texttt{f}_2]_{\texttt{fid}})) \\ \quad \circ_{\mathrm{X}} \langle \texttt{name}''_{\texttt{newChild}} \wedge \texttt{t}'\rangle_{\mathrm{X}}\end{array}\right\} \\ \texttt{insertBefore}(\texttt{parent},\texttt{newChild},\texttt{refNode}); \\ \left\{\texttt{cg} \circ_{\mathrm{T}} (\texttt{name}_{\texttt{parent}}[\texttt{f}_1 \otimes \langle \texttt{name}''_{\texttt{newChild}} \wedge \texttt{t}'\rangle_{\mathrm{F}} \otimes \langle \texttt{name}'_{\texttt{refNode}} \wedge \texttt{t}\rangle_{\mathrm{F}} \otimes \texttt{f}_2]_{\texttt{fid}})\right\} \\ \left\{\begin{array}{l}(\varnothing_{\mathrm{F}} \multimap_{\mathrm{o}} \texttt{cg} \circ_{\mathrm{T}} (\texttt{name}_{\texttt{parent}}[\texttt{f}_1 \otimes \langle \texttt{name}''_{\texttt{newChild}} \wedge \texttt{t}'\rangle_{\mathrm{F}} \otimes \varnothing_{\mathrm{F}} \otimes \texttt{f}_2]_{\texttt{fid}})) \\ \quad \circ_{\mathrm{F}} \langle \texttt{name}'_{\texttt{refNode}} \wedge \texttt{t}\rangle_{\mathrm{F}}\end{array}\right\} \\ \left\{\begin{array}{l}(\varnothing_{\mathrm{F}} \multimap_{\mathrm{o}} \texttt{cg} \circ_{\mathrm{T}} (\texttt{name}_{\texttt{parent}}[\texttt{f}_1 \otimes \langle \texttt{name}''_{\texttt{newChild}} \wedge \texttt{t}'\rangle_{\mathrm{F}} \otimes \texttt{f}_2]_{\texttt{fid}})) \\ \quad \circ_{\mathrm{F}} \langle \texttt{name}'_{\texttt{refNode}} \wedge \texttt{t}\rangle_{\mathrm{F}}\end{array}\right\} \\ \texttt{insertBefore}(\texttt{parent},\texttt{refNode},\texttt{newChild}); \\ \left\{\texttt{cg} \circ_{\mathrm{T}} (\texttt{name}_{\texttt{parent}}[\texttt{f}_1 \otimes \langle \texttt{name}'_{\texttt{refNode}} \wedge \texttt{t}\rangle_{\mathrm{F}} \otimes \langle \texttt{name}''_{\texttt{newChild}} \wedge \texttt{t}'\rangle_{\mathrm{F}} \otimes \texttt{f}_2]_{\texttt{fid}})\right\}\end{array}$$

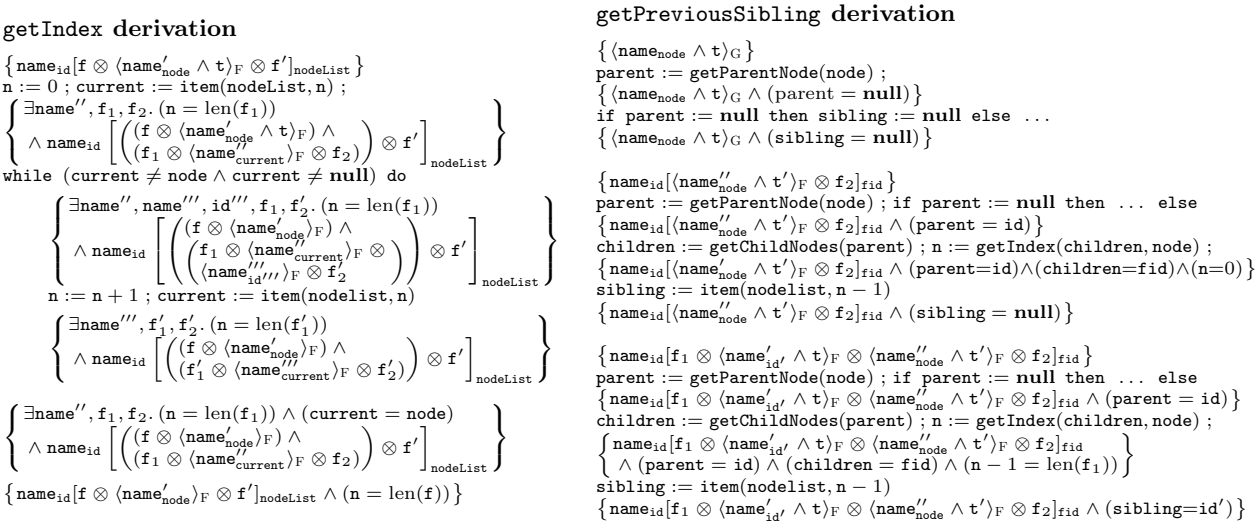


Figure 2: Derivations for the getIndex and getPreviousSibling Specifications

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 is an English language statement declaring that, assuming '$\mathtt{b} \neq \mathbf{null}$', the command '$\mathtt{p.insertAfter(a, b)}$' is equivalent to the sequence '$\mathtt{p.insertBefore(a, b)}$; $\mathtt{p.insertBefore(b, a)}$'. The reader must then refer to the specification of `insertBefore` in order to understand the specification 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 (often complex) specification. One example of this involves proving XML schema invariants. For example, consider writing a program to update an XML addressBook document which complies with the XML 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:complexType>
            <element name="name" type="string"/>
          </xs:complexType>
        </xs:element>
        <xs:element name="address" type="string"/>
        <xs:element name="phone" type="string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:element>
```

The schema asserts that the root element of the document should be an addressBook node, whose children should be zero or more household nodes. These household nodes should contain one or more person nodes, one address node and one phone node. Each of these third-level nodes should contain data of type 'string'.

We can specify this XML schema using the grove formula S:

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

$$\mathrm{households} \triangleq \Box_{\otimes}(\langle \mathrm{true}_{\mathrm{T}}\rangle_{\mathrm{F}} \Rightarrow \left\langle \mathsf{household}\left[\begin{array}{l}\langle \mathsf{person}[\mathrm{txts}]\rangle_{\mathrm{F}} \otimes \mathrm{people} \otimes \\ \langle \mathsf{address}[\mathrm{txts}]\rangle_{\mathrm{F}} \otimes \langle \mathsf{phone}[\mathrm{txts}]\rangle_{\mathrm{F}}\end{array}\right]\right\rangle_{\mathrm{F}})$$

$$\mathrm{txts} \triangleq \langle \#\mathsf{text}\rangle_{\mathrm{F}} \otimes \Box_{\otimes}(\langle \mathrm{true}_{\mathrm{T}}\rangle_{\mathrm{F}} \Rightarrow \langle \#\mathsf{text}\rangle_{\mathrm{F}})$$

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

Now consider a Minimal DOM program which updates the addressBook document when a specified person `leaver` moves house. The program moves `leaver` out of its current household, into a newly created house with a user-supplied address and phone number. The program checks if the original household is now empty and, if it is, deletes it.

```
moveHouse(leaver, newAddress, newPhone) ≜
    // Move leaver into a new house.
    house := getParentNode(leaver);
    book := getParentNode(house);
    newHouse := createNode('household');
    appendChild(newHouse, leaver);
    addr := createNode('address');
    adtxt := createTextNode(newAddress);
    appendChild(addr, adtxt);
    appendChild(newHouse, addr);
    phn := createNode('phone');
    phntxt := createTextNode(newPhone);
    appendChild(phn, phntxt);
    appendChild(newHouse, phn);
    appendChild(book, newHouse);
    // Remove old house if empty.
    kids := getChildNodes(house);
    firstChild := item(kids, 0);
    firstName := getNodeName(firstChild);
    if firstName = 'person' then skip
     else removeChild(book, house);
```

The safety condition that `leaver` refers to a person node can be simply expressed by the formula

$$P \triangleq \Diamond_{\mathrm{T}\to\mathrm{G}} \mathsf{person}_{\mathtt{leaver}}[\mathrm{true}_{\mathrm{F}}]$$

$\{S \wedge P\}$

$\{\langle \mathsf{addressBook}\,[\mathrm{households} \otimes \langle \mathsf{household}\,[\mathrm{people} \otimes \langle \mathsf{person}_{\mathsf{leaver}}[\mathrm{txts}]\rangle_{\mathrm{F}} \otimes \mathrm{people} \otimes \langle \mathsf{address}[\mathrm{txts}]\rangle_{\mathrm{F}} \otimes \langle \mathsf{phone}[\mathrm{txts}]\rangle_{\mathrm{F}}]\rangle_{\mathrm{F}} \otimes \mathrm{households}]\rangle_{\mathrm{G}}\}$

`moveHouse(leaver, newAddress, newPhone)` $\triangleq$

```
// Move leaver into a new house.
house := getParentNode(leaver); book := getParentNode(house); newHouse := createNode('household'); appendChild(newHouse, leaver);
addr := createNode('address'); adtxt := createTextNode(newAddress); appendChild(addr, adtxt); appendChild(newHouse, addr);
phn := createNode('phone'); phntxt := createTextNode(newPhone); appendChild(phn, phntxt); appendChild(newHouse, phn);
```

$\left\{\begin{array}{l}\langle \mathsf{addressBook}_{\mathsf{book}}\,[\mathrm{households} \otimes \langle \mathsf{household}_{\mathsf{house}}\,[\mathrm{people} \otimes \langle \mathsf{address}[\mathrm{txts}]\rangle_{\mathrm{F}} \otimes \langle \mathsf{phone}[\mathrm{txts}]\rangle_{\mathrm{F}}]\rangle_{\mathrm{F}} \otimes \mathrm{households}]\rangle_{\mathrm{G}} \\ \oplus\ \langle \mathsf{household}_{\mathsf{newHouse}}[\langle \mathsf{person}_{\mathsf{leaver}}[\mathrm{txts}]\rangle_{\mathrm{F}} \otimes \langle \mathsf{address}[\mathsf{newAddress}]\rangle_{\mathrm{F}} \otimes \langle \mathsf{phone}[\mathsf{newPhone}]\rangle_{\mathrm{F}}]\rangle_{\mathrm{G}}\end{array}\right\}$

```
appendChild(book, newHouse);
```

$\{\langle \mathsf{addressBook}_{\mathsf{book}}\,[\mathrm{households} \otimes \langle \mathsf{household}_{\mathsf{house}}\,[\mathrm{people} \otimes \langle \mathsf{address}[\mathrm{txts}]\rangle_{\mathrm{F}} \otimes \langle \mathsf{phone}[\mathrm{txts}]\rangle_{\mathrm{F}}]\rangle_{\mathrm{F}} \otimes \mathrm{households}]\rangle_{\mathrm{G}}\}$

```
// Remove old house if empty.
kids := getChildNodes(house); firstChild := item(kids, 0); firstName := getNodeName(firstChild);
if firstName = 'person' then skip
```

$\{\langle \mathsf{addressBook}\,[\mathrm{households} \otimes \langle \mathsf{household}\,[\langle \mathsf{person}[\mathrm{txts}]\rangle_{\mathrm{F}} \otimes \mathrm{people} \otimes \langle \mathsf{address}[\mathrm{txts}]\rangle_{\mathrm{F}} \otimes \langle \mathsf{phone}[\mathrm{txts}]\rangle_{\mathrm{F}}]\rangle_{\mathrm{F}} \otimes \mathrm{households}]\rangle_{\mathrm{G}}\}$

```
else removeChild(book, house);
```

$\{\langle \mathsf{addressBook}[\mathrm{households}]\rangle_{\mathrm{G}} \oplus \mathrm{true}_{\mathrm{G}}\}$

$\{S \oplus \mathrm{true}_{\mathrm{G}}\}$

Figure 3: Schema Preservation Derivation

We can prove that `moveHouse` maintains the schema formula S provided that this safety formula P also holds:

$$\{S \wedge P\}\ \texttt{moveHouse(leaver, newAddress, newPhone)}\ \{S \oplus \mathrm{true}_{\mathrm{G}}\}$$

As discussed in Section 2.2, we treat Minimal DOM as a garbage collected language. We thus have $\mathrm{true}_{\mathrm{G}}$ in the post-condition to refer to uncollected garbage generated by the program, which is safely ignored. The proof is in Figure 3.

6. CONCLUSION

Using Context Logic, we have developed local Hoare reasoning about Minimal DOM. Our reasoning is compositional and complete for straight-line code. This 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 and text nodes, rather than the full DOM structure which also consists of Attributes, DocumentFragments, etc. We took the view that it was important to understand the reasoning about the fundamental tree structure first, especially since DOM treats these other structures as nodes with similar properties. We will extend our reasoning to DOM Core Level 1 in future. We conjecture that there will be little additional conceptual reasoning to this extension. We have however already found holes in the DOM specification: for example, DOM does not specify how the normalize method of the Element class interacts with pointers in the store.

A future goal is to develop a prototype verification tool for reasoning first about Minimal DOM, and then about DOM Core Level 1. Calcagno, Dinsdale-Young and Gardner have recently shown that model-checking and validity are decidable for quantifier-free Context Logic applied to simple trees. We believe this result will extend to Context Logic for the DOM tree structure presented here. With quantifiers however, Context Logic applied both to simple trees and to the DOM tree structure becomes undecidable, just as for Separation Logic and Ambient Logic. We will investigate tractable fragments of Context Logic, inspired by the verification tool Smallfoot based on a fragment of Separation Logic. In particular, the last example in Section 5.3, which verifies that an XML address book schema is an invariant of a simple program which moves a person to a new address, is particularly enticing. An ambitious challenge is to provide a 'one-click' tool that checks if, for example, embedded Javascript in a web page can ever violate the schema assertions on that web page.

Acknowledgements We thank Brotherstone, Calcagno, Dinsdale-Young, Kahramanogullari, O'Hearn and Raza for insightful discussions, and acknowledge financial support from EPSRC, Microsoft and the Royal Academy of Engineering.

7. REFERENCES

[1] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *CAV*, 2007.

[2] J. Berdine, C. Calcagno, and P. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, 2006.

[3] C. Calcagno, T. Dinsdale-Young, and P. Gardner. Context logic and regular languages: Expressivity results. 2008.

[4] C. Calcagno, P. Gardner, and U. Zarfaty. Context logic & tree update. In *POPL*, 2005.

[5] C. Calcagno, P. Gardner, and U. Zarfaty. Context logic as modal logic: Completeness and parametric inexpressivity. In *POPL*, 2007.

[6] P. Gardner, G. Smith, M. Wheelhouse, and U. Zarfaty. Local hoare reasoning about DOM. Extended version, available online at `http://www.doc.ic.ac.uk/~pg/publications.html`.

[7] S. Isthiaq and P. O'Hearn. BI as an assertion language for mutable data structures. In *POPL*, 2001.

[8] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL*, 2001.

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

[10] J. Orendorff. Compliance Patches for minidom. Included with Python, 2007. See `http://bugs.python.org/issue1704134`.

[11] G. Smith. A context logic approach to analysis and specification of xml update. PhD 1st year report, 2006.

[12] Various. Python: xml.dom.minidom.

[13] W3C. Document Object Model (DOM) Level 1 Specification (2nd Ed.). Working draft, 2000. `http://www.w3.org/TR/2000/WD-DOM-Level-1-20000929/`.

[14] W3C. DOM: Document Object Model. W3C recommendation, 2005. See `http://www.w3.org/DOM/`.

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

[16] U. Zarfaty and P. Gardner. Local reasoning about tree update. In *MFPS*, 2006.