# Small Specifications for Tree Update

Philippa Gardner and Mark Wheelhouse

Imperial College London, {pg, mjw03}@doc.ic.ac.uk

**Abstract.** O'Hearn, Reynolds and Yang introduced Separation Logic to provide modular reasoning about simple, mutable data structures in memory. They were able to construct small specifications of programs, by reasoning about the local parts of memory accessed by programs. Gardner, Calcagno and Zarfaty generalised this work, introducing Context Logic to reason about more complex data structures. In particular, they developed a formal, compositional specification of the Document Object Model, a W3C XML update library. Whilst keeping to the spirit of local reasoning, they were not able to retain small specifications. We introduce Segment Logic, which provides a more fine-grained analysis of the tree structure and yields small specifications. As well as being aesthetically pleasing, small specifications are important for reasoning about concurrent tree update.

## 1 Introduction

Separation Logic [13], introduced by O'Hearn, Reynolds and Yang, provides modular reasoning about mutable data structures in memory. The idea is to reason about the small, local parts of memory (the footprint) that are accessed by a program. The resulting modular reasoning has been used to notable success for verifying memory safety properties of large C-programs [1], and for reasoning about concurrent imperative programs [14]. Calcagno, Gardner and Zarfaty generalised this work to more complex data structures, such as those found on the Web, by introducing Context Logic for reasoning about arbitrary structured data update [3]. Their original work applied Context Logic reasoning to a simple tree update language. With Smith and Zarfaty, Gardner and Wheelhouse have since applied Context Logic reasoning to the W3C Document Object Model (DOM) [6], a library for in-place XML update [18].

Our goal is to design and formally specify a concurrent XML update language. Such a language will enable web applications to make the most of the dynamic nature of XML. For example, with Wikipedia, users currently copy articles on to their browsers, before updating and returning them to Wikipedia to be integrated with the main site. They cannot view Wikipedia (or a scientific database or information in the Cloud) as a shared XML memory store that can be concurrently updated by many clients, because methods for safely performing such operations are poorly understood. We almost have the technology to develop a safe, formally specified language for concurrent XML update, drawing on our experience experience with sequential DOM [6] and O'Hearn's Concurrent Separation Logic [14]. However, we are missing one ingredient.

In our DOM work, we were not able to provide small specifications for all our DOM programs. In particular, our reasoning for the basic move commands,

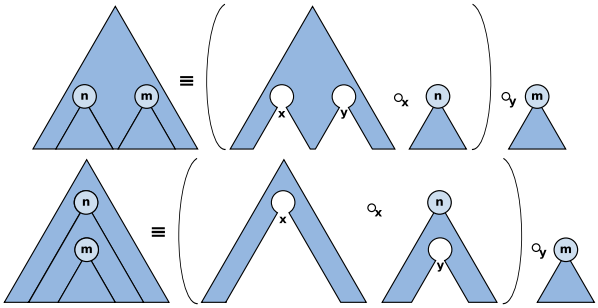



**Fig. 1.** Splitting up the Working Tree using Multi-holed Contexts.

such as DOM's `appendChild`, used axioms which required a substantial over-approximation of the footprint. Whilst this over-approximation was acceptable for reasoning about sequential programs, it is a serious limitation when reasoning about concurrent programs. In this paper, we solve this limitation, by introducing Segment Logic to provide a more fine-grained analysis of structured data update in general, and tree update in particular. We provide small axioms for all the basic commands of a simple tree update language; it is straightforward to extend our ideas to DOM [6]. Although this paper focuses on a sequential tree update language, we believe it provides the technology necessary for our future work on reasoning about concurrent tree update.

To motivate Segment Logic, consider the DOM command $\texttt{appendChild}(n, m)$ which moves the tree with top node identified by $m$ to be the last child of the tree identified by $n$. **Fig. 1** indicates how the working tree splits in the two cases where $\texttt{appendChild}(n, m)$ does not fault: it succeeds when $n$ and $m$ are in different parts of the tree and when $m$ is under $n$; it faults when $m$ is above $n$. The axiom for $\texttt{appendChild}(n, m)$ using multi-holed Context Logic [2] is [1]:

$$\begin{array}{c}\{(C \circ_\alpha n[c_1]) \circ_\beta m[\mathrm{tree}(c_2)]\}\\ \texttt{appendChild}(n, m)\\ \{(C \circ_\alpha n[c_1 \otimes m[\mathrm{tree}(c_2)]]) \circ_\beta \varnothing\}\end{array}$$

The precondition specifies that the working tree can be split into a subtree with top node $m$, and a tree context with hole variable $\beta$ ($y$ in **Fig. 1**) satisfying the separating application formula $C \circ_\alpha n[c_1]$. This formula states that the context can be further split into a subcontext with top node $n$ and an unspecified context with hole $\alpha$ ($x$ in **Fig. 1**) given by context variable $C$. The postcondition states that the tree at $m$ is moved to be the last child of $n$ and is replaced by the empty tree. The surrounding context, denoted by variable $C$, remains the same.

The problem with this `appendChild` axiom is that it is not small. The precondition is not the intuitive footprint. The only part of the tree that $\texttt{appendChild}(n, m)$ requires is the tree at $m$ which is being moved, and the tree or context with top node $n$ (actually node $n$ is enough) whose children are being extended by $m$. However, our precondition does not just use $m$ and $n$. It also requires the surrounding context denoted by $C$. It is possible to put additional constraints on $C$

[1] In [6], the axiom for appendChild is given using single-holed Context Logic. The multi-holed Context Logic axiom is simpler, but still not suitable for concurrent reasoning.

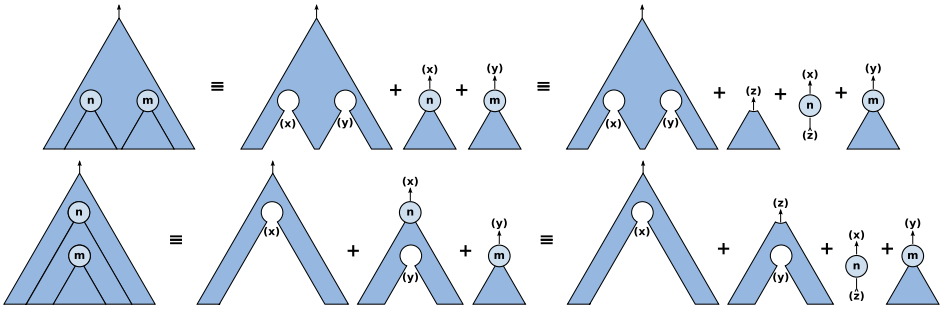



**Fig. 2.** Splitting up the Working Tree using Tree Segments.

to insist that the context is minimal. But this is not the point. We need a finer way of analysing the tree in order to capture the intuitive footprint of the command.

Instead of basing our reasoning on multi-holed tree contexts and application, we base our reasoning on *tree segments*. With multi-holed contexts, the working tree is split into a context and subtrees which have lost the information about where they originated from; the application function determines which holes get filled. With segments, the working tree is split into tree segments which still 'know' how to join back together again. As well as unique hole labels, tree segments have unique hole addresses which determine which holes the segments fill. For example, consider **Fig. 2**. In both cases, the working tree is split into a bunch of tree segments. The hole labels (in the holes) and the hole addresses (on the arrows) determine how the tree segments join back up to form the original tree. Notice that hole labels and addresses have brackets around them, denoting that they are bound. In the syntax, we will use a hiding operator $(x)$, analogous to the restriction operator of Milner's $\pi$-calculus [12].

Moreover, consider the right-hand equalities of **Fig. 2**. In both cases, the tree segment with top node identified by $n$ has been split into just the node $n$ at the same address and fresh hole label $z$, plus another tree segment at address $z$ which contains the children of node $n$. We shall see that the node $n$ and the tree with top node $m$ are all that is required to provide the small axiom for `appendChild`. **Fig. 2** thus indicates how we can uniformly separate the minimal data required in order to reason about appendChild. It is possible to take this separation to the extreme, by cutting up the tree structure into a collection of nodes, with the hole labels and addresses showing how the nodes are joined together (a spaghetti of wires analogous to a heap representation). However, this is not how we use the hole information. We only cut up the tree in a minimal way in order to provide the right segment about which to reason.

We introduce Segment Logic for reasoning about our tree segments. It is like Context Logic in that it reasons directly about high-level trees. It is like Separation Logic in that it uses a commutative separating conjunction $*$, rather than the non-commutative separating application of Context Logic. Using Segment Logic, the small axiom for $\texttt{appendChild}(n, m)$ is:

$$\{\alpha{\leftarrow}n[\gamma] * \beta{\leftarrow}m[\mathrm{tree}(c)]\}$$
$$\texttt{appendChild}(n, m)$$
$$\{\alpha{\leftarrow}n[\gamma \otimes m[\mathrm{tree}(c)]] * \beta{\leftarrow}\varnothing_{\mathrm{T}}\}$$

The precondition specifies two tree segments: a node $n$ at address variable $\alpha$ ($x$ in **Fig. 2**) and a complete tree whose top node is $m$ at address $\beta$ ($y$ in **Fig 2**). The postcondition states that the tree at $m$ moves to be the last child of $n$ and is replaced by the empty tree. The axiom is small, with the precondition capturing the intuitive footprint of `appendChild`$(n, m)$. We can extend the axiom to larger tree segments using the normal separation frame rule, a rule for the hiding quantification, and the rule for logical consequence. In fact, instead of using the hiding quantifier as primitive, we use the basic revelation connective (and a revelation frame rule), the revelation magic wand (the revelation right adjoint), and the fresh label quantification (and a fresh variable elimination rule), inspired by the work of Gabbay and Pitts [5], and Cardelli and Gordon [4]. Interestingly, we shall see that these more primitive constructs are important for describing the weakest preconditions.

## 2 Tree Update Language

We study a simple, but expressive, high-level tree update language for manipulating finite, ordered, unranked trees, with unique node identifiers for specifying the locations of updates as in DOM. Our tree structures are left intentionally simple. It is straightforward to incorporate (and reason about) additional data such as text data and attributes (see [6]). To simplify our exposition, we work with multi-holed tree contexts [2]. Throughout this paper we use countably infinite and disjoint sets $\mathrm{I} = \{m, n, ...\}$ for location names and $\mathrm{X} = \{x, y, z, ...\}$ for hole identifiers.

**Definition 1 (Tree Contexts).** *Multi-holed tree contexts $c \in \mathrm{C}_{\mathrm{I},\mathrm{X}}$ are defined by:*

$$\begin{array}{rrll} \text{tree context} & c ::= & \varnothing_{\mathrm{C}} & \textit{empty tree context} \\ & & x & \textit{hole identifier } x \textit{ used as a hole label} \\ & & n[c] & \textit{tree context with top node } n \\ & & c \otimes c & \textit{composition of tree contexts} \end{array}$$

*with the restriction that each hole identifier, $x \in \mathrm{X}$, and location name, $n \in \mathrm{I}$, occur at most once in a tree context $c$, and subject to an equivalence $c_1 \equiv c_2$ stating that $\otimes$ is associative with identity $\varnothing_{\mathrm{C}}$. The set of hole identifiers that occur in tree context $c$ is denoted $free(c)$. A tree context with no context holes (a complete tree) is denoted $t$.*

**Definition 2 (Context Application).** *Context Application is defined as a set of partial functions $ap_x : \mathrm{C}_{\mathrm{I},\mathrm{X}} \times \mathrm{C}_{\mathrm{I},\mathrm{X}} \rightharpoonup \mathrm{C}_{\mathrm{I},\mathrm{X}}$ indexed by hole labels $x$:*

$$ap_x(c_1, c_2) = \begin{cases} c_1[c_2/x] & \textit{if } x \in free(c_1) \textit{ and } free(c_1) \cap free(c_2) = \{\} \\ \textit{undefined otherwise} \end{cases}$$

We abbreviate $ap_x(c_1, c_2)$ by $c_1 \circ_x c_2$. We often omit the $\varnothing_{\mathrm{C}}$ leaves from a tree context to make it more readable, writing $n[m \otimes p]$ instead of $n[m[\varnothing_{\mathrm{C}}] \otimes p[\varnothing_{\mathrm{C}}]]$.

Our update language is a high-level, stateful, sequential, imperative language, based on variable assignment and update commands as in DOM. The program state is made up of two components: the working tree which contains all of the nodes we will be manipulating with our programs; and a high-level variable store containing variables for node identifiers.

**Definition 3 (Variable Store).** *The variable store $\sigma \in \Sigma$ is a finite partial function*

$$\sigma : Var_{\mathrm{I}} \rightharpoonup_{fin} \mathrm{I} \cup \{\mathit{null}\}$$

*mapping location name variables $Var_{\mathrm{I}} = \{m, n, ...\}$ to location names or **null**. We write $\sigma[n \mapsto n]$ for the variable store $\sigma$ overwritten with $\sigma(n) = n$.*

To specify location name values, our language uses simple expressions. Location names are specified either with location name variables or the constant ***null***; we forbid direct reference to constant location names other than ***null***. We also require simple Boolean expressions for conditional tests in our language.

**Definition 4 (Expressions).** *Location name expressions $N \in Exp_{\mathrm{I}}$ and Boolean expressions $B \in Exp_{\mathbb{B}}$ are defined by:*

$$N ::= n \mid \mathit{null} \qquad\qquad n \in Var_{\mathrm{I}}$$
$$B ::= N = N \mid \mathbf{false} \mid B \Rightarrow B$$

The valuation of an expression $E$ in a store $\sigma$ is written $[\![E]\!]\sigma$ and has the obvious semantics. The classical Boolean connectives **true**, $\neg$, $\wedge$ and $\vee$ are derivable.

DOM commands tend to update whole trees although, for example, the DOM commands `getNodeName` and `createNode` manipulate single nodes. Since our reasoning analyses tree segments, we explore a language for manipulating tree segments with primitive commands that update either a single node $n$ or the whole subtree beneath some node $n$; the commands for updating whole trees are derivable (Example 1).

**Definition 5 (Tree Update Language).** *The commands of the tree update language consist of the node update commands $\mathbb{C}_{nodeUp}$, the tree update commands $\mathbb{C}_{treeUp}$ and the standard skip, assignment, local, sequencing, if-then-else and while-do commands:*

| | | |
|---|---|---|
| $\mathbb{C}_{nodeUp} ::=$ | $n' :=$ `getUp`$(n)$ | *get parent of node n and record it in $n'$* |
| | $n' :=$ `getLeft`$(n)$ | *get previous sibling of node n* |
| | $n' :=$ `getRight`$(n)$ | *get next sibling of node n* |
| | $n' :=$ `getFirst`$(n)$ | *get first child of node n* |
| | $n' :=$ `getLast`$(n)$ | *get last child of node n* |
| | `insertNodeAbove`$(n)$ | *insert a new node above node n* |
| | `deleteNode`$(n)$ | *delete node n* |
| | `moveNodeAbove`$(n, m)$ | *move node m above node n* |
| | `moveNodeLeft`$(n, m)$ | *move node m to the left of node n* |
| | `moveNodeRight`$(n, m)$ | *move node m to the right of node n* |
| | `prependNode`$(n, m)$ | *prepend node m to children of node n* |
| | `appendNode`$(n, m)$ | *append node m to children of node n* |

| | | |
|---|---|---|
| $\mathbb{C}_{treeUp} ::=$ | `deleteSubtree`$(n)$ | *delete subtree (subforest) beneath node n* |
| | `moveSubLeft`$(n, m)$ | *move children of node m to the left of node n* |
| | `moveSubRight`$(n, m)$ | *move children of node m to the right of node n* |
| | `prependSub`$(n, m)$ | *prepend children of node m to children of node n* |
| | `appendSub`$(n, m)$ | *append children of node m to children of node n* |

*The set of free variables of a command $\mathbb{C}$ is denoted free($\mathbb{C}$), and the set of variables modified by $\mathbb{C}$ is denoted mod($\mathbb{C}$).*

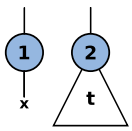
The behavior of these commands should be self-explanatory. The node update commands consist of get commands that return a neighboring node in the tree, a node insertion command that puts a fresh node into the tree above node $n$ (the other node insertion commands are derivable), a delete command that removes a node from the tree, and node move commands that take a node out of the tree and put it in a new position. These node move commands leave the children of the moved node $m$ as children of $m$'s old parent. The tree update commands work on subtrees of an identified node. They consist of a delete command that removes an entire subtree from the tree, and subtree move commands that take a subtree out of the tree and put it in a new position.

These commands are sufficient to express a wide range of tree manipulation, as illustrated by the examples below. Our command set is not minimal: for example, we could derive the `deleteSubtree` command using a combination of get commands, node deletion and recursion. However, we believe the commands chosen provide a natural and expressive tree update language. We give the operational semantics in Section 3 using tree segments, rather than trees or tree contexts, as this simplifies the description of our reasoning in Section 5.

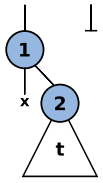
In [8], there are also commands for inserting whole trees. This is achieved by including tree shapes (trees without identifiers) in the variable store. Here we avoid such complications by omitting commands for tree insertion and copying. The reasoning presented here extends simply to these extra commands.

We now give example programs which will be used to illustrate our reasoning in Section 5. In all of these examples, $\sigma(n) = 1$ and $\sigma(m) = 2$.

*Example 1 (Move).* DOM has the command `appendChild`, whereas our language has `appendNode` and `appendSub`. We implement the standard `appendChild` as:



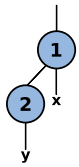

```
appendChild(n, m) ≜ local temp := null in {
                      insertNodeAbove(m) ;
                      temp := getUp(m) ;
                      appendNode(n, m) ;
                      appendSub(m, temp) ;
                      deleteNode(temp)
                    }
```

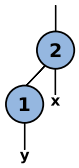



The diagrams illustrate the intuitive effect of the program on the working tree. For the program above not to fault, it requires the node $n = 1$ and the complete tree at node $m = 2$ (the left-hand diagram and the intuitive footprint). The complete tree at $m$ ensures that $m$ is not an ancestor of $n$. The result of the program is to move the tree at $m = 2$ to be under the node $n = 1$ (the right-hand diagram). Our reasoning captures the intuition illustrated by these diagrams.

*Example 2 (Simple Swap).* Our node update commands enable us to define programs that act on arbitrary segments of the tree. For example, consider the program `simple`($n$) which swaps a node $n$ with its first child:

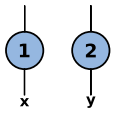
```
simple(n) ≜ local temp := null, first := null in {
              insertNodeAbove(n) ;
              temp := getUp(n) ;
              first := getFirst(n) ;
              moveNodeAbove(first, n) ;
              moveNodeAbove(temp, first) ;
              deleteNode(temp)
            }
```

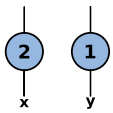



*Example 3 (General Swap).* The program nodeSwap(*n*, *m*) swaps the positions of arbitrary nodes *n* and *m* of a tree leaving their subtrees stationary:

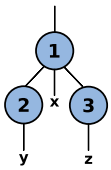
```
nodeSwap(n, m) ≜ local temp := null in {
                   insertNodeAbove(n) ;
                   temp := getUp(n) ;
                   moveNodeAbove(m, n) ;
                   moveNodeAbove(temp, m) ;
                   deleteNode(temp)
                 }
```

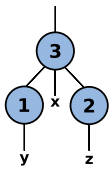



*Example 4 (Node Rotate).* The program nodeCycle(*n*) takes the node *n*, its first and last child, and rotates these nodes with *n* taking the place of first child, first child taking the place of last child, and last child taking the place of *n*:

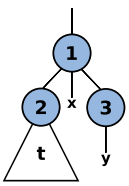
```
nodeCycle(n) ≜ local first := null, last := null in {
                 first := getFirst(n) ;
                 last := getLast(n) ;
                 nodeSwap(n, last) ;
                 nodeSwap(n, first)
               }
```

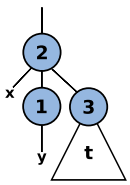



*Example 5 (Combining Move and Node Swap).* Consider a simple cyclic list of pictures used, for example, to view properties on an estate agent's web page. It can be implemented as a tree structure, with the root node of the tree containing the ID of the picture currently being displayed, the picture itself being stored beneath the last of its children (under node 3 below), and the other pictures in the list being stored beneath their ID nodes as the rest of the root node's children. The program queuePop(*n*) cycles the pictures so that the current picture moves to the back of the list and the next picture is displayed. Notice the use of appendChild which includes the complete tree *t* in the footprint.

```
queuePop(n) ≜ local next := null, info := null in {
                next := getFirst(n) ;
                info := getLast(n) ;
                nodeSwap(n, info) ;
                nodeSwap(next, info) ;
                appendChild(next, info)
              }
```



## 3 Tree Segments

We are not able to provide a small specification of the `appendChild` command (and the `appendSub` command) using tree contexts. We are able to provide small specifications for all our tree update commands using tree segments.

**Definition 6 (Tree Segments).** *Tree segments $s \in \mathrm{S}_{\mathrm{I},X}$ are defined by the grammar:*

| | | | |
|---|---|---|---|
| tree segment | $s ::=$ | $\varnothing_{\mathrm{S}}$ | *empty tree segment* |
| | | $x{\leftarrow}c$ | *tree context $c$ addressed by hole identifier $x$* |
| | | $s + s$ | *disjoint union* |
| | | $(x)(s)$ | *hiding, hole identifier $x$ bound in tree segment $s$* |

*with the restriction that each hole identifier $x \in \mathrm{X}$ occurs free at most once as a hole label and at most once as a hole address in tree segment $s$, and each location name, $n \in \mathrm{I}$, occurs at most once in $s$. The set $free(s)$ denotes the set of free hole identifiers in $s$.*

With tree contexts, we have the application $(1[x \otimes 3]) \circ_x 2 = 1[2 \otimes 3]$. The application $\circ_x$ binds $x$, and declares that hole $x$ is filled by the argument 2. With tree segments, we have the equivalence $(x)(z{\leftarrow}1[x \otimes 3] + x{\leftarrow}2) \equiv z{\leftarrow}1[2 \otimes 3]$. In this case, it is the segment $x{\leftarrow}2$ with address $x$ that declares that 2 should go into the hole $x$, and the hiding operator $(x)$ which binds $x$ in the segment.

**Definition 7 (Tree Segment Equivalence).** *An equivalence relation $\equiv$ over tree segments is defined by the following axioms and the natural structural rules:*

$$\begin{array}{rcl@{\qquad}rcl} s + \varnothing_{\mathrm{S}} & \equiv & s & s_1 + s_2 & \equiv & s_2 + s_1 \\ (x)(\varnothing_{\mathrm{S}}) & \equiv & \varnothing_{\mathrm{S}} & s_1 + (s_2 + s_3) & \equiv & (s_1 + s_2) + s_3 \end{array}$$

$$\begin{array}{rcll} (x)(y)(s) & \equiv & (y)(x)(s) & \\ (x)(s) & \equiv & (y)(s[y/x]) & \text{if} \quad y \notin free(s) \\ (x)(y{\leftarrow}c + s) & \equiv & y{\leftarrow}c + (x)(s) & \text{if} \quad x \neq y \text{ and } x \notin free(c) \\ (x)(y{\leftarrow}c_1 + x{\leftarrow}c_2) & \equiv & y{\leftarrow}(c_1 \circ_x c_2) & \text{if} \quad x \in free(c_1) \end{array}$$

Most of the axioms involving hiding follow from analogous axioms for the restriction operator of the $\pi$-calculus [12]. The last hiding axiom is specific to tree segments. It enables us to pull apart and compress segments, as illustrated in **Fig. 3**. A tree segment is in its compressed form if it cannot be further compressed using this last axiom. A tree segment is well-formed if and only if its compressed form is cycle free; that is, the hole labels and hole addresses are disjoint in its compressed form. We only work with well-formed tree segments in this paper.

**Fig. 3** demonstrates a graphical interpretation of segments. The left-hand side of **Fig. 3** will come as no surprise to those familiar with graphical process models: for example, Milner's work on process graphs [11]. Here, the hole identifiers describe the edges of the graph (wires). However, this is not the only use of hole identifiers. Consider the right-hand side of **Fig. 3**. Here, the hole identifier $y$ is used to address multiple edges of a graph. More than this, consider the `appendChild` command in Example 1. The tree segment $z{\leftarrow}1[x] + y{\leftarrow}2[t]$ updates to $z{\leftarrow}1[x \otimes 2[t]] + y{\leftarrow}\varnothing_{\mathrm{C}}$: before update the segment $y{\leftarrow}2[t]$ states that a tree is at address $y$; after update $y{\leftarrow}\varnothing_{\mathrm{C}}$ states that the empty tree is at address $y$.

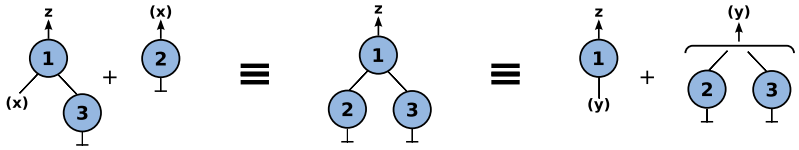



**Fig. 3.** Equivalent Tree Segments: $(x)$ and $(y)$ denote hidden hole labels and addresses.

This example illustrates that edge arity is not necessarily preserved by update. Our hole identifiers should therefore not be regarded as describing graph edges. Instead, they describe tree fragments.

Notice that our language manipulates nodes and complete trees. It does not refer to hole identifiers in any way. However, the operational semantics is greatly simplified by using either tree contexts or tree segments. We choose tree segments, as this leads to a simpler interpretation of Hoare triples in Section 5.

**Definition 8 (Operational Semantics).** *We give the operational semantics for the basic commands of the tree update language in* **Fig. 4** *using an evaluation relation* $\rightsquigarrow$ *relating configuration triples* $\mathbb{C}, \sigma, \mathbf{s}$*, terminal states* $\sigma, \mathbf{s}$*, and faults, where* $\mathbb{C}$ *is a command,* $\sigma$ *is a variable store and* $\mathbf{s}$ *is a tree segment. The set of variables of a command* $\mathbb{C}$ *is denoted free*($\mathbb{C}$) *and is contained within the domain of* $\sigma$*, denoted dom*($\sigma$)*. We omit the standard cases for skip, assignment, local, sequencing, if-then-else and while-do.*

Our style of local Hoare reasoning about programs requires that the commands of our language be local. A command is local if it satisfies two properties, initially introduced in [10], known as the *safety-monotonicity* property and the *frame* property. The *safety-monotonicity* property specifies that, if a command is safe (does not fault) in a given state, then it is safe in a larger state. The *frame* property specifies that, if a command is safe in a given state, then any execution on a larger state can be tracked to an execution on the smaller state. The commands of our language presented here (and the DOM commands [6]) are local. For example, consider the behavior of $n' := \texttt{getRight}(n)$. If the right sibling of $n$ exists, then its identifier is stored at $n'$. If $n$ is the last child of some parent node (meaning $n$ can never obtain a right sibling via segment composition), then $n'$ stores the value ***null***. However, if the node $n$ is not present in the tree, or $n$ has no right sibling or parent, then the command must fault in order to be local.

## 4 Segment Logic

We introduce Segment Logic. First, we present the *logical environment* which contains logical variables for tree contexts, tree segments and hole identifiers. Location name variables have the standard dual role as both program variables and logical variables. They are declared in the variable store, but can be quantified like logical variables.

**Definition 9 (Logical Environment).** *An environment* $e \in \mathrm{E}$ *is a set of functions*

$$e : (LVar_{\mathrm{C}} \to \mathrm{C_{I\ ,X}}) \times (LVar_{\mathrm{S}} \to \mathrm{S_{I\ ,X}}) \times (LVar_{\mathrm{X}} \to \mathrm{X})$$

*mapping tree context variables* $LVar_{\mathrm{C}} = \{\mathtt{c}, ...\}$ *to tree contexts, tree segment variables* $LVar_{\mathrm{S}} = \{\mathtt{s}, ...\}$ *to tree segments, and hole identifier variables* $LVar_{\mathrm{X}} = \{\alpha, \beta, \gamma, \delta ...\}$ *to hole identifiers. We write* $e[\mathtt{lvar} \mapsto \boldsymbol{val}]$ *for* $e$ *overwritten with* $e(\mathtt{lvar}) = \boldsymbol{val}$*.*

$$\frac{\sigma(n) = n \quad s \equiv (w,x,y,z)(s' + x{\leftarrow}m[y \otimes n[w] \otimes z])}{n' := \texttt{getUp}(n), \sigma, s \rightsquigarrow \sigma[n' \mapsto m], s}$$

$$\frac{\sigma(n) = n \quad s \equiv (x,y,z)(s' + x{\leftarrow}n[y] \otimes m[z])}{n' := \texttt{getRight}(n), \sigma, s \rightsquigarrow \sigma[n' \mapsto m], s} \qquad \frac{\sigma(n) = n \quad s \equiv (x,y,z)(s' + x{\leftarrow}m[z \otimes n[y]])}{n' := \texttt{getRight}(n), \sigma, s \rightsquigarrow \sigma[n' \mapsto \mathit{null}], s}$$

$$\frac{\sigma(n) = n \quad s \equiv (x,y,z)(s' + x{\leftarrow}n[y \otimes m[z]])}{n' := \texttt{getLast}(n), \sigma, s \rightsquigarrow \sigma[n' \mapsto m], s} \qquad \frac{\sigma(n) = n \quad s \equiv (x)(s' + x{\leftarrow}n[\varnothing_C])}{n' := \texttt{getLast}(n), \sigma, s \rightsquigarrow \sigma[n' \mapsto \mathit{null}], s}$$

$$\frac{\sigma(n) = n \quad s \equiv (x,y)(s'' + x{\leftarrow}n[y]) \quad m \text{ fresh id} \quad s' \equiv (x,y)(s'' + x{\leftarrow}m[n[y]])}{\texttt{insertNodeAbove}(n), \sigma, s \rightsquigarrow \sigma, s'} \qquad \frac{\sigma(n) = n \quad s \equiv (x,y)(s'' + x{\leftarrow}n[y]) \quad s' \equiv (x,y)(s'' + x{\leftarrow}y)}{\texttt{deleteNode}(n), \sigma, s \rightsquigarrow \sigma, s'} \qquad \frac{\sigma(n) = n \quad s \equiv (x)(s'' + x{\leftarrow}n[t]) \quad s' \equiv (x)(s'' + x{\leftarrow}n[\varnothing_C])}{\texttt{deleteSubtree}(n), \sigma, s \rightsquigarrow \sigma, s'}$$

$$\frac{\sigma(n) = n \quad \sigma(m) = m \quad s \equiv (w,x,y,z)(s'' + x{\leftarrow}n[z] + y{\leftarrow}m[w]) \quad s' \equiv (w,x,y,z)(s'' + x{\leftarrow}n[z \otimes m] + y{\leftarrow}w)}{\texttt{appendNode}(n,m), \sigma, s \rightsquigarrow \sigma, s'} \qquad \frac{\sigma(n) = n \quad \sigma(m) = m \quad s \equiv (x,y,z)(s'' + x{\leftarrow}n[z] + y{\leftarrow}m[t]) \quad s' \equiv (x,y,z)(s'' + x{\leftarrow}n[z \otimes t] + y{\leftarrow}m)}{\texttt{appendSub}(n,m), \sigma, s \rightsquigarrow \sigma, s'}$$

$$\frac{\sigma(n) = n \quad \sigma(m) = m \quad s \equiv (w,x,y,z)(s'' + x{\leftarrow}n[z] + y{\leftarrow}m[w]) \quad s' \equiv (w,x,y,z)(s'' + x{\leftarrow}n[m[z]] + y{\leftarrow}w)}{\texttt{moveNodeAbove}(n,m), \sigma, s \rightsquigarrow \sigma, s'}$$

For get and move, only some of the cases are given; the other cases are analogous. Our commands fault when the program state does not satisfy any of the preconditions for that command.

**Fig. 4.** Operational Semantics for the Basic Tree Update Commands.

Segment Logic for trees consists of segment formulae and tree formulae. Just as in Separation Logic and Context Logic, segment formulae consist of classical formulae, structural formulae and specific formulae for describing the structure of data (in this case trees). For this paper, we have chosen to use tree formulae in the style of Ambient Logic [4], although we do not see a reason why $P_T$ could not be first-order logic formulae for describing trees or even XDuce types [9]. Note that adapting this work to other data structures, such as sequences, just involves changing the tree formulae (or types) to sequence formulae (or types).

**Definition 10 (Formulae).** *The formulae of Segment Logic for trees consist of the segment formulae $P_S$ and tree formulae $P_T$ given by:*

| | | |
|---|---|---|
| $P_S, Q_S ::= P_S \Rightarrow P_S \mid \mathbf{false}_S$ | $P_T, Q_T ::= P_T \Rightarrow P_T \mid \mathbf{false}_T$ | *Classical* |
| $\mid \varnothing_S \mid P_S * P_S \mid P_S {-\!\!*} P_S \mid \alpha ® P_S \mid \alpha \neg ® P_S$ | | *Structural* |
| $\mid \exists \mathtt{var}.\, P_S \mid \exists \mathtt{1var}.\, P_S \mid \forall \alpha.\, P_S$ | $\mid \exists \mathtt{var}.\, P_T \mid \exists \mathtt{1var}.\, P_T$ | *Quantifiers* |
| $\mid \alpha{\leftarrow}P_T$ | $\mid \varnothing_T \mid \alpha \mid \mathtt{n}[P_T] \mid P_T \otimes P_T$ | *Specific* |
| $\mid \mathtt{s} \mid \mathtt{B}$ | $\mid \mathtt{c} \mid \mathtt{B} \mid @_T \alpha$ | *Expression* |

*Let $free(P_S)$ and $free(P_T)$ denote the appropriate sets of free variables: $\alpha$ is free in $\alpha ® P_S$ $\alpha \neg ® P_S$, $\alpha{\leftarrow}P_T$ and $@_T\alpha$, bound in $\forall \alpha.\, P_S$. $\mathtt{var}$ is a location name variable and $\mathtt{1var}$ is a logical variable. The binding precedence, strongest first, is: $\otimes$, $\leftarrow$, $*$, $®$, $-\!\!*$, $\neg ®$, $\Rightarrow$.*

The separating connective $*$, its unit $\varnothing_S$ and its right adjoint (the separating magic wand) $-\!\!*$, are structural formulae which are known from the Separation Logic literature: formula $P_S * Q_S$ describes a segment that can be separated into a segment satisfying $P_S$ and a disjoint segment satisfying $Q_S$; formula $\varnothing_S$ describes the empty segment; and formula $P_S -\!\!* Q_S$ describes a segment that, whenever it is joined to a segment satisfying $P_S$, results in a segment satisfying $Q_S$.

Before explaining the other structural formulae, we explain the specific segment formulae: the formula $\alpha{\leftarrow}P_T$ describes a tree segment with hole address given by the value of variable $\alpha$ and tree context satisfying tree formula $P_T$. The tree formulae follow the style of reasoning given by Ambient Logic. For

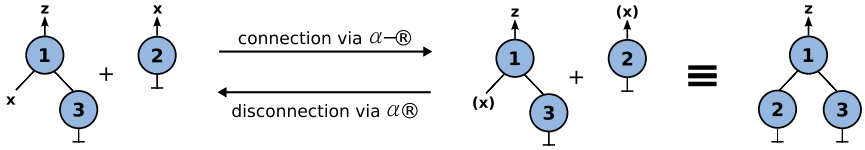



**Fig. 5.** Connection and Disconnection of Tree Segments.

the specific tree formulae, we have $\varnothing_T$ specifying the empty tree context, $\alpha$ specifying a hole label given by the value of variable $\alpha$, $n[P_T]$ specifying a tree context with top node denoted by node variable $n$ and subtree satisfying $P_T$, and the composition formula $P_T \otimes Q_T$ describing a tree context which can be split into one context satisfying $P_T$ and the other disjoint context satisfying $Q_T$. The tree expression formulae include the formula $@_T\alpha$ which describes a tree that contains $\alpha$ free; the analogous formula for tree segments is derivable.

The other structural connectives are the revelation connective, ®, and its right adjoint, the revelation magic wand –®. As far as we are aware, these connectives have not been used in the local reasoning setting before. Together with the freshness quantifier $И\alpha$, they have been used in the Ambient Logic [4], following the work of Pitts and Gabbay [5]. The freshness quantifier enables us to pick a completely new hole identifier. The formula $\alpha®P_S$ describes a segment with a top-level hiding binder given by the value of $\alpha$ such that, after the hiding is removed, the remaining segment satisfies $P_S$. Consider the tree segment $z\leftarrow1[2\otimes 3] \equiv (x)(z\leftarrow1[x \otimes 3] + x\leftarrow2)$. It satisfies the formula $\alpha®(\beta\leftarrow1[\alpha \otimes \mathbf{true}_T] + \alpha\leftarrow2)$, when $\alpha = x, \beta = z$. The revelation connective $\alpha®$ strips off the hiding binder, disconnecting the tree into the fragments $z\leftarrow1[x\otimes3]+x\leftarrow2$ as illustrated in **Fig. 5**. By contrast, the formula $\alpha$ –® $P_S$ describes a segment which satisfies $P_S$ if it is extended with a hiding binder over the hole identifier stored in variable $\alpha$. For example, the tree segment $z\leftarrow1[x\otimes3]+x\leftarrow2$ satisfies the formula $\alpha–®(\beta\leftarrow1[2 \otimes 3])$ when $\alpha = x$. The revelation magic wand $\alpha$ –® adds the binder $(x)$ to the segment to obtain the tree segment $(x)(z\leftarrow1[x \otimes 3] + x\leftarrow2) \equiv z\leftarrow1[2 \otimes 3]$, thus connecting the fragmented tree into the whole tree as illustrated in **Fig. 5**. Analogous to the separating magic wand, we shall see that the revelation magic wand is important for giving the weakest preconditions of commands.

**Definition 11 (Satisfaction Relation).** *Given a logical environment $e$ and a variable store $\sigma$, the semantics of Segment Logic is given in* **Fig. 6** *by two satisfaction relations $e, \sigma, s \models_S P_S$ and $e, \sigma, c \models_T P_T$ defined on tree segments and tree contexts respectively.*

**Definition 12 (Derived Formulae).** *The standard classical logic connectives are derived from* **false** *and $\Rightarrow$ as usual, and the following useful formulae are defined:*

$$\begin{array}{rlrl} tree(P_T) & \triangleq P_T \wedge \neg\exists\alpha.\, @_T\alpha & @_S\alpha & \triangleq H\beta.\, (\mathbf{true}_S * (\alpha\leftarrow\beta \vee \beta\leftarrow\alpha)) \\ n & \triangleq n[\varnothing_T] & \Diamond P_S & \triangleq \mathbf{true}_S * P_S \\ \circ[P_T] & \triangleq \exists m.\, m[P_T] & H\alpha.\, P_S & \triangleq И\alpha.\, \alpha®P_S \end{array}$$

Formula $tree(P_T)$ describes a complete tree satisfying $P_T$. Formula $n$ describes a leaf node identified by $n$. Formula $\circ[P_T]$ allows us to drop the identifier of a node. Formula $@_S\alpha$ describes a tree segment that contains $\alpha$ free. Formula $\Diamond P_S$ allows us to express that somewhere in the tree segment there is a segment satisfying $P_S$. Finally, formula $H\alpha.\, P_S$ provides the standard hiding quantification [4] allowing us to quantify over hidden (restricted) labels.

$$
\begin{array}{ll}
e,\sigma,\boldsymbol{s} \models_{\mathrm{S}} \varnothing_{\mathrm{S}} & \Leftrightarrow \boldsymbol{s} \equiv \varnothing_{\mathrm{S}} \\
e,\sigma,\boldsymbol{s} \models_{\mathrm{S}} P_{\mathrm{S}} * Q_{\mathrm{S}} & \Leftrightarrow \exists \boldsymbol{s}_1, \boldsymbol{s}_2.\, \boldsymbol{s} \equiv \boldsymbol{s}_1 + \boldsymbol{s}_2 \wedge e,\sigma,\boldsymbol{s}_1 \models_{\mathrm{S}} P_{\mathrm{S}} \wedge e,\sigma,\boldsymbol{s}_2 \models_{\mathrm{S}} Q_{\mathrm{S}} \\
e,\sigma,\boldsymbol{s} \models_{\mathrm{S}} P_{\mathrm{S}} \mathrel{-\!\!*} Q_{\mathrm{S}} & \Leftrightarrow \forall \boldsymbol{s}'.\, e,\sigma,\boldsymbol{s}' \models_{\mathrm{S}} P_{\mathrm{S}} \wedge (\boldsymbol{s}+\boldsymbol{s}')\!\downarrow \Rightarrow e,\sigma,\boldsymbol{s}+\boldsymbol{s}' \models_{\mathrm{S}} Q_{\mathrm{S}} \\
e,\sigma,\boldsymbol{s} \models_{\mathrm{S}} \alpha \circledR P_{\mathrm{S}} & \Leftrightarrow \exists \boldsymbol{x}, \boldsymbol{s}'.\, e(\alpha) = \boldsymbol{x} \wedge \boldsymbol{s} \equiv (\boldsymbol{x})(\boldsymbol{s}') \wedge e,\sigma,\boldsymbol{s}' \models_{\mathrm{S}} P_{\mathrm{S}} \\
e,\sigma,\boldsymbol{s} \models_{\mathrm{S}} \alpha \mathrel{-\!\!\circledR} P_{\mathrm{S}} & \Leftrightarrow \exists \boldsymbol{x}, \boldsymbol{s}'.\, e(\alpha) = \boldsymbol{x} \wedge \boldsymbol{s}' \equiv (\boldsymbol{x})(\boldsymbol{s}) \wedge e,\sigma,\boldsymbol{s}' \models_{\mathrm{S}} P_{\mathrm{S}} \\
e,\sigma,\boldsymbol{s} \models_{\mathrm{S}} \mathit{И}\alpha.\, P_{\mathrm{S}} & \Leftrightarrow \exists \boldsymbol{x}.\, \boldsymbol{x}\#e,\boldsymbol{s} \wedge e[\alpha \mapsto \boldsymbol{x}],\sigma,\boldsymbol{s} \models_{\mathrm{S}} P_{\mathrm{S}} \\
e,\sigma,\boldsymbol{s} \models_{\mathrm{S}} \boldsymbol{\alpha}\!\leftarrow\! P_{\mathrm{T}} & \Leftrightarrow \exists \boldsymbol{c}, \boldsymbol{x}.\, e(\alpha) = \boldsymbol{x} \wedge \boldsymbol{s} \equiv \boldsymbol{x}\!\leftarrow\!\boldsymbol{c} \wedge e,\sigma,\boldsymbol{c} \models_{\mathrm{T}} P_{\mathrm{T}} \\
e,\sigma,\boldsymbol{s} \models_{\mathrm{S}} \boldsymbol{s} & \Leftrightarrow \boldsymbol{s} \equiv e(\boldsymbol{s}) \\
e,\sigma,\boldsymbol{s} \models_{\mathrm{S}} B & \Leftrightarrow [\![B]\!]\sigma = \mathbf{true}
\end{array}
\qquad
\begin{array}{ll}
e,\sigma,\boldsymbol{c} \models_{\mathrm{T}} \varnothing_{\mathrm{T}} & \Leftrightarrow \boldsymbol{c} \equiv \varnothing_{\mathrm{C}} \\
e,\sigma,\boldsymbol{c} \models_{\mathrm{T}} \alpha & \Leftrightarrow \boldsymbol{c} \equiv e(\alpha) \\
e,\sigma,\boldsymbol{c} \models_{\mathrm{T}} \boldsymbol{n}[P_{\mathrm{T}}] & \Leftrightarrow \exists \boldsymbol{c}_1.\, \boldsymbol{c} \equiv \sigma(\boldsymbol{n})[\boldsymbol{c}_1] \\
 & \quad \wedge e,\sigma,\boldsymbol{c}_1 \models_{\mathrm{T}} P_{\mathrm{T}} \\
e,\sigma,\boldsymbol{c} \models_{\mathrm{T}} P_{\mathrm{T}} \otimes Q_{\mathrm{T}} & \Leftrightarrow \exists \boldsymbol{c}_1, \boldsymbol{c}_2.\, \boldsymbol{c} \equiv \boldsymbol{c}_1 \otimes \boldsymbol{c}_2 \\
 & \quad \wedge e,\sigma,\boldsymbol{c}_1 \models_{\mathrm{T}} P_{\mathrm{T}} \\
 & \quad \wedge e,\sigma,\boldsymbol{c}_2 \models_{\mathrm{T}} Q_{\mathrm{T}} \\
e,\sigma,\boldsymbol{c} \models_{\mathrm{T}} \boldsymbol{c} & \Leftrightarrow \boldsymbol{c} \equiv e(\boldsymbol{c}) \\
e,\sigma,\boldsymbol{c} \models_{\mathrm{T}} B & \Leftrightarrow [\![B]\!]\sigma = \mathbf{true} \\
e,\sigma,\boldsymbol{c} \models_{\mathrm{T}} @_{\mathrm{T}}\alpha & \Leftrightarrow e(\alpha) \in \mathit{free}(\boldsymbol{c})
\end{array}
$$

$(\boldsymbol{s})\!\downarrow$ denotes that $\boldsymbol{s}$ is well formed. $\boldsymbol{x}\#\boldsymbol{s}$ denotes that $\boldsymbol{x}$ is fresh with respect to $\boldsymbol{s}$.
We omit the standard semantics for $P \Rightarrow Q$, **false** and $\exists \boldsymbol{v}.\, P$.

**Fig. 6.** Satisfaction Relations of Segment Logic for Trees.

*Example 6 (Segment Logic Examples).*

(a) The segment formula $\alpha\!\leftarrow\!\boldsymbol{n}[\gamma] * \beta\!\leftarrow\!\boldsymbol{m}[\delta]$ describes a tree segment consisting of a node $\boldsymbol{n}$ with address $\alpha$ and context hole $\gamma$, and node $\boldsymbol{m}$ with address $\beta$ and context hole $\delta$. The variables $\boldsymbol{n}$ and $\boldsymbol{m}$ cannot denote the same node identifier: similarly, $\alpha$, $\beta$ cannot denote the same hole identifier; neither can $\gamma$, $\delta$.

(b) The segment formula $\alpha\!\leftarrow\!\boldsymbol{n}[\gamma] * \beta\!\leftarrow\!\boldsymbol{m}[\mathrm{tree}(\boldsymbol{c})]$ describes a tree segment consisting of a single node $\boldsymbol{n}$ at address $\alpha$ and a complete tree (a tree with no holes) with top node $\boldsymbol{m}$ at address $\beta$. This formula is the safety precondition for the small axiom of the `appendSub`$(\boldsymbol{n},\boldsymbol{m})$ command. In particular, the formula states that $\boldsymbol{m}$ cannot be an ancestor of $\boldsymbol{n}$, as $\boldsymbol{n}$ is disjoint from the tree $\boldsymbol{c}$.

(c) The segment formula $\mathit{H}\alpha,\beta.\, (\delta\!\leftarrow\!\boldsymbol{r}[\alpha \otimes \beta] * \alpha\!\leftarrow\!\boldsymbol{n}[\gamma] * \beta\!\leftarrow\!\boldsymbol{m}[\mathrm{tree}(\boldsymbol{c})])$ describes a tree segment consisting of a node $\boldsymbol{r}$ at address $\delta$ whose children are given by the holes $\alpha$ and $\beta$, and a tree segment satisfying the formula in Example (b). The labels $\alpha$,$\beta$ are under the hiding quantification, and hence denote fresh, unequal, identifiers. This formula is equivalent to $\delta\!\leftarrow\!\boldsymbol{r}[\boldsymbol{n}[\gamma] \otimes \boldsymbol{m}[\mathrm{tree}(\boldsymbol{c})]]$, which states that there is a node $\boldsymbol{r}$ at address $\delta$ whose children are $\boldsymbol{n}$ and $\boldsymbol{m}$.

(d) To specify our language, it is enough to work with the hiding quantification. However, to describe the weakest preconditions, we must use revelation. For example, the weakest precondition of the `deleteSubtree`$(\boldsymbol{n})$ command is $\exists \boldsymbol{c}.\, \mathit{И}\alpha.\, \alpha\circledR( (\alpha\!\leftarrow\!\boldsymbol{n}[\varnothing_{\mathrm{T}}] \mathrel{-\!\!*} (\alpha \mathrel{-\!\!\circledR} P_{\mathrm{S}})) * \alpha\!\leftarrow\!\boldsymbol{n}[\mathrm{tree}(\boldsymbol{c})] )$. This formula describes a tree segment which can be separated into a complete tree, with top node $\boldsymbol{n}$ at a fresh address $\boldsymbol{x}$ denoted by $\alpha$, and a segment $\boldsymbol{s}$ satisfying $(\alpha\!\leftarrow\!\boldsymbol{n}[\varnothing_{\mathrm{T}}]) \mathrel{-\!\!*} (\alpha \mathrel{-\!\!\circledR} P_{\mathrm{S}})$. The segment $\boldsymbol{s}$, when extended to $(\boldsymbol{x})(\boldsymbol{x}\!\leftarrow\!\boldsymbol{n}[\varnothing_{\mathrm{T}}] + \boldsymbol{s})$, satisfies $P_{\mathrm{S}}$.

## 5 Local Hoare Reasoning

We use Segment Logic to provide local Hoare reasoning about programs written in the language given in Definition 5. First, we give a fault avoiding, partial correctness interpretation of local Hoare triples following [19]. Informally, $\{P_{\mathrm{S}}\}\ \mathbb{C}\ \{Q_{\mathrm{S}}\}$ means that, when $P_{\mathrm{S}}$ holds for a tree segment $\boldsymbol{s}$, then command $\mathbb{C}$ does not fault when run on $\boldsymbol{s}$ and the result, if $\mathbb{C}$ terminates, satisfies $Q_{\mathrm{S}}$.

**Definition 13 (Local Hoare Triples).** *Recall the evaluation relation $\rightsquigarrow$ relating configuration triples $\mathbb{C},\sigma,\boldsymbol{s}$, terminal states $\sigma,\boldsymbol{s}$ and faults in* **Fig. 4**. *The fault-avoiding partial correctness interpretation of local Hoare Triples is given below:*

$$
\begin{array}{rl}
\{P_{\mathrm{S}}\}\ \mathbb{C}\ \{Q_{\mathrm{S}}\} & \Leftrightarrow\ \forall e,\sigma,\boldsymbol{s}.\ \mathit{free}(\mathbb{C}) \subseteq \mathit{dom}(\sigma) \wedge \mathit{free}(P_{\mathrm{S}}) \cup \mathit{free}(Q_{\mathrm{S}}) \subseteq \mathit{dom}(\sigma) \cup \mathit{dom}(e) \\
 & \quad \wedge\ e,\sigma,\boldsymbol{s} \models_{\mathrm{S}} P_{\mathrm{S}} \Rightarrow \mathbb{C},\sigma,\boldsymbol{s} \not\rightsquigarrow \mathit{fault} \wedge \forall \sigma', \boldsymbol{s}'.\, \mathbb{C},\sigma,\boldsymbol{s} \rightsquigarrow \sigma',\boldsymbol{s}' \Rightarrow e,\sigma',\boldsymbol{s}' \models_{\mathrm{S}} Q_{\mathrm{S}}
\end{array}
$$

$$
\begin{array}{rcl}
\{\alpha{\leftarrow}m[\beta \otimes n[\delta] \otimes \gamma] \wedge (n' = n_0)\} & n' := \texttt{getUp}(n) & \{(\alpha{\leftarrow}m[\beta \otimes n[\delta] \otimes \gamma])_{[n_0/n']} \wedge (n' = m)\} \\
\{\alpha{\leftarrow}n[\delta] \otimes m[\beta] \wedge (n' = n_0)\} & n' := \texttt{getRight}(n) & \{(\alpha{\leftarrow}n[\delta] \otimes m[\beta])_{[n_0/n']} \wedge (n' = m)\} \\
\{\alpha{\leftarrow}m[\beta \otimes n[\delta]] \wedge (n' = n_0)\} & n' := \texttt{getRight}(n) & \{(\alpha{\leftarrow}m[\beta \otimes n[\delta]])_{[n_0/n']} \wedge (n' = \mathit{null})\} \\
\{\alpha{\leftarrow}n[\delta \otimes m[\beta]] \wedge (n' = n_0)\} & n' := \texttt{getLast}(n) & \{(\alpha{\leftarrow}n[\delta \otimes m[\beta]])_{[n_0/n']} \wedge (n' = m)\} \\
\{\alpha{\leftarrow}n[\varnothing_{\mathrm{T}}] \wedge (n' = n_0)\} & n' := \texttt{getLast}(n) & \{(\alpha{\leftarrow}n[\varnothing_{\mathrm{T}}])_{[n_0/n']} \wedge (n' = \mathit{null})\} \\
\{\alpha{\leftarrow}n[\beta]\} & \texttt{insertNodeAbove}(n) & \{\alpha{\leftarrow}\circ[n[\beta]]\} \\
\{\alpha{\leftarrow}n[\beta]\} & \texttt{deleteNode}(n) & \{\alpha{\leftarrow}\beta\} \\
\{\alpha{\leftarrow}n[\mathrm{tree}(c)]\} & \texttt{deleteSubtree}(n) & \{\alpha{\leftarrow}n[\varnothing_{\mathrm{T}}]\} \\
\{\alpha{\leftarrow}n[\gamma] * \beta{\leftarrow}m[\delta]\} & \texttt{moveNodeAbove}(n, m) & \{\alpha{\leftarrow}m[n[\gamma]] * \beta{\leftarrow}\delta\} \\
\{\alpha{\leftarrow}n[\gamma] * \beta{\leftarrow}m[\delta]\} & \texttt{appendNode}(n, m) & \{\alpha{\leftarrow}n[\gamma \otimes m[\varnothing_{\mathrm{T}}]] * \beta{\leftarrow}\delta\} \\
\{\alpha{\leftarrow}n[\gamma] * \beta{\leftarrow}m[\mathrm{tree}(c)]\} & \texttt{appendSub}(n, m) & \{\alpha{\leftarrow}n[\gamma \otimes \mathrm{tree}(c)] * \beta{\leftarrow}m[\varnothing_{\mathrm{T}}]\}
\end{array}
$$

$n'$ and $n_0$ are distinct. The omitted commands have analogous or standard axioms.

**Fig. 7.** A Selection of the Small Axioms for the Basic Tree Update Commands.

**Definition 14 (Small Axioms).** *The Small Axioms for the basic tree update commands from* **Fig. 4** *are given in* **Fig. 7**.

With Raza, Gardner has developed the formal definitions of footprints and small specifications for abstract local functions using Abstract Separation Logic [15]. It would be interesting to extend this abstract theory to the tree segments and reasoning studied here, and prove that the axioms really are small.

**Definition 15 (Inference Rules).** *The local reasoning inference rules include the standard Hoare Logic Rules for Sequencing, Consequence, Disjunction, Local Variable, If-Then-Else, While-Do, and the rules for Separation Frame, Revelation Frame, Auxiliary Variable Elimination and Fresh Label Elimination given by:*

$$\mathrm{SF}: \quad \frac{\{P_{\mathrm{S}}\}\ \mathbb{C}\ \{Q_{\mathrm{S}}\}}{\{P_{\mathrm{S}} * R_{\mathrm{S}}\}\ \mathbb{C}\ \{Q_{\mathrm{S}} * R_{\mathrm{S}}\}} \quad \mathit{mod}(\mathbb{C}) \cap \mathit{free}(R_{\mathrm{S}}) = \{\} \qquad \mathrm{RF}: \quad \frac{\{P_{\mathrm{S}}\}\ \mathbb{C}\ \{Q_{\mathrm{S}}\}}{\{\alpha \circledR P_{\mathrm{S}}\}\ \mathbb{C}\ \{\alpha \circledR Q_{\mathrm{S}}\}}$$

$$\mathrm{AVE}: \quad \frac{\{P_{\mathrm{S}}\}\ \mathbb{C}\ \{Q_{\mathrm{S}}\}}{\{\exists n.\, P_{\mathrm{S}}\}\ \mathbb{C}\ \{\exists n.\, Q_{\mathrm{S}}\}} \quad n \notin \mathit{free}(\mathbb{C}) \qquad \mathrm{FVE}: \quad \frac{\{P_{\mathrm{S}}\}\ \mathbb{C}\ \{Q_{\mathrm{S}}\}}{\{\mathcal{U}\alpha.\, P_{\mathrm{S}}\}\ \mathbb{C}\ \{\mathcal{U}\alpha.\, Q_{\mathrm{S}}\}}$$

*Recall that the set of variables modified by a command $\mathbb{C}$ is denoted $\mathit{mod}(\mathbb{C})$.*

The Separation Frame rule is standard from Separation Logic and allows us to extend the working tree with tree segments that are not used by the command. The Revelation Frame rule is similar. It allows us to add hiding binders to the working tree, since hole identifiers are not used by any of our commands. The Fresh Variable Elimination rule is analogous to the standard Auxiliary Variable Elimination rule. To see how we use these rules, consider again the small axiom of appendSub. We can extend this axiom using our inference rules to obtain:

$$\{\mathsf{H}\alpha, \beta.\, (\epsilon{\leftarrow}r[\alpha \otimes \beta] * \alpha{\leftarrow}n[\gamma] * \beta{\leftarrow}m[\mathrm{tree}(c)])\}$$
$$\texttt{appendSub}(n, m)$$
$$\{\mathsf{H}\alpha, \beta.\, (\epsilon{\leftarrow}r[\alpha \otimes \beta] * \alpha{\leftarrow}n[\gamma \otimes \mathrm{tree}(c)] * \beta{\leftarrow}m[\varnothing_{\mathrm{T}}])\}$$

This triple can be simplified using the consequence rule following the discussion in Example 6(c). In this example, it is natural to use the derived hiding quantification. Following the discussion in Example 6(d) we can see that the primitive revelation connectives are, however, necessary to obtain the weakest preconditions, a selection of which are shown in **Fig. 8**.

$$\begin{array}{rcl}
\{\exists m, n_0.\, \mathsf{H}\alpha,\beta,\gamma,\delta.\, \Diamond\alpha{\leftarrow}m[\beta \otimes n[\delta] \otimes \gamma] \wedge (n' = n_0) \wedge (\alpha,\beta,\gamma,\delta \mathrel{-\!\!\circledR} P_{S[m/n']})\} & n' := \mathtt{getUp}(n) & \{P_S\} \\
\left\{\begin{array}{r}\exists m, n_0.\, \mathsf{H}\alpha,\beta,\delta.\\ \end{array}\begin{array}{r}\Diamond\alpha{\leftarrow}n[\delta] \otimes m[\beta] \wedge (n' = n_0) \wedge (\alpha,\beta,\delta \mathrel{-\!\!\circledR} P_{S[m/n']})\\ \vee\, \Diamond\alpha{\leftarrow}m[\beta \otimes n[\delta]] \wedge (n' = n_0) \wedge (\alpha,\beta,\delta \mathrel{-\!\!\circledR} P_{S[null/n']})\end{array}\right\} & n' := \mathtt{getRight}(n) & \{P_S\} \\
\left\{\begin{array}{r}\exists m, n_0.\, \mathsf{H}\alpha,\beta,\delta.\\ \end{array}\begin{array}{r}\Diamond\alpha{\leftarrow}n[\delta \otimes m[\beta]] \wedge (n' = n_0) \wedge (\alpha,\beta,\delta \mathrel{-\!\!\circledR} P_{S[m/n']})\\ \vee\, \Diamond\alpha{\leftarrow}n[\varnothing_T] \wedge (n' = n_0) \wedge (\alpha \mathrel{-\!\!\circledR} P_{S[null/n']})\end{array}\right\} & n' := \mathtt{getLast}(n) & \{P_S\} \\
\{\mathsf{H}\alpha,\beta.\, ((\alpha{\leftarrow}\circ[n[\beta]] \mathrel{-\!\!*} (\alpha,\beta \mathrel{-\!\!\circledR} P_S)) * \alpha{\leftarrow}n[\beta])\} & \mathtt{insertNodeAbove}(n) & \{P_S\} \\
\{\mathsf{H}\alpha,\beta.\, ((\alpha{\leftarrow}\beta \mathrel{-\!\!*} (\alpha,\beta \mathrel{-\!\!\circledR} P_S)) * \alpha{\leftarrow}n[\beta])\} & \mathtt{deleteNode}(n) & \{P_S\} \\
\{\exists c.\, \mathsf{H}\alpha.\, ((\alpha{\leftarrow}n[\varnothing_T] \mathrel{-\!\!*} (\alpha \mathrel{-\!\!\circledR} P_S)) * \alpha{\leftarrow}n[\mathsf{tree}(c)])\} & \mathtt{deleteSubtree}(n) & \{P_S\} \\
\{\mathsf{H}\alpha,\beta,\gamma,\delta.\, (((\alpha{\leftarrow}m[n[\gamma]] * \beta{\leftarrow}\delta) \mathrel{-\!\!*} (\alpha,\beta,\gamma,\delta \mathrel{-\!\!\circledR} P_S)) * (\alpha{\leftarrow}n[\gamma] * \beta{\leftarrow}m[\delta]))\} & \mathtt{moveNodeAbove}(n,m) & \{P_S\} \\
\{\mathsf{H}\alpha,\beta,\gamma,\delta.\, (((\alpha{\leftarrow}n[\gamma \otimes m[\varnothing_T]] * \beta{\leftarrow}\delta) \mathrel{-\!\!*} (\alpha,\beta,\gamma,\delta \mathrel{-\!\!\circledR} P_S)) * (\alpha{\leftarrow}n[\gamma] * \beta{\leftarrow}m[\delta]))\} & \mathtt{appendNode}(n,m) & \{P_S\} \\
\{\exists c.\, \mathsf{H}\alpha,\beta,\gamma.\, (((\alpha{\leftarrow}n[\gamma \otimes \mathsf{tree}(c)] * \beta{\leftarrow}m[\varnothing_T]) \mathrel{-\!\!*} (\alpha,\beta,\gamma \mathrel{-\!\!\circledR} P_S)) * (\alpha{\leftarrow}n[\gamma] * \beta{\leftarrow}m[\mathsf{tree}(c)]))\} & \mathtt{appendSub}(n,m) & \{P_S\}
\end{array}$$

**Fig. 8.** A Selection of the Weakest Preconditions for our Basic Tree Update Commands.

**Theorem 1 (Soundness and Completeness).** *The small axioms and inference rules are sound. For straight line code, they are also complete.*

*Proof Sketch.* Soundness is straightforward to prove. Completeness for straight line code follows from the derivability of the the weakest preconditions (**Fig. 8**) from the small axioms (**Fig. 7**). See the full paper [7] for further details.

*Example 7 (Specifying appendChild).* In Example 1 we gave the `appendChild` program. The command's specification and its derivation are:

$$\begin{array}{c}\{\alpha{\leftarrow}n[\gamma] * \beta{\leftarrow}m[\mathsf{tree}(c)]\}\\ \mathtt{appendChild}(n,m)\\ \{\alpha{\leftarrow}n[\gamma \otimes m[\mathsf{tree}(c)]] * \beta{\leftarrow}\varnothing_T\}\end{array}$$

$$\begin{array}{l}
\{\alpha{\leftarrow}n[\gamma] * \beta{\leftarrow}m[\mathsf{tree}(c)]\}\\
\quad \mathtt{local}\ \mathit{temp} := \mathbf{\mathit{null}}\ \mathtt{in}\ \{\\
\{\alpha{\leftarrow}n[\gamma] * \beta{\leftarrow}m[\mathsf{tree}(c)] \wedge (\mathit{temp} = \mathbf{\mathit{null}})\}\\
\{\mathsf{H}\delta.\, \alpha{\leftarrow}n[\gamma] * \beta{\leftarrow}m[\delta] * \delta{\leftarrow}\mathsf{tree}(c) \wedge (\mathit{temp} = \mathbf{\mathit{null}})\}\\
\quad \mathtt{insertNodeAbove}(m)\ ;\\
\{\mathsf{H}\delta.\, \alpha{\leftarrow}n[\gamma] * \beta{\leftarrow}\circ[m[\delta]] * \delta{\leftarrow}\mathsf{tree}(c) \wedge (\mathit{temp} = \mathbf{\mathit{null}})\}\\
\quad \mathit{temp} := \mathtt{getUp}(m)\ ;\\
\{\mathsf{H}\delta.\, \alpha{\leftarrow}n[\gamma] * \beta{\leftarrow}\mathit{temp}[m[\delta]] * \delta{\leftarrow}\mathsf{tree}(c)\}\\
\{\mathsf{H}\epsilon,\delta.\, \alpha{\leftarrow}n[\gamma] * \beta{\leftarrow}\mathit{temp}[\epsilon] * \epsilon{\leftarrow}m[\delta] * \delta{\leftarrow}\mathsf{tree}(c)\}\\
\quad \mathtt{appendNode}(n,m)\ ;\\
\{\mathsf{H}\epsilon,\delta.\, \alpha{\leftarrow}n[\gamma \otimes m[\varnothing_T]] * \beta{\leftarrow}\mathit{temp}[\epsilon] * \epsilon{\leftarrow}\delta * \delta{\leftarrow}\mathsf{tree}(c)\}\\
\{\alpha{\leftarrow}n[\gamma \otimes m[\varnothing_T]] * \beta{\leftarrow}\mathit{temp}[\mathsf{tree}(c)]\}\\
\{\mathsf{H}\epsilon,\delta.\, \alpha{\leftarrow}n[\gamma \otimes \epsilon] * \epsilon{\leftarrow}m[\delta] * \delta{\leftarrow}\varnothing_T * \beta{\leftarrow}\mathit{temp}[\mathsf{tree}(c)]\}\\
\quad \mathtt{appendSub}(m, \mathit{temp})\ ;\\
\{\mathsf{H}\epsilon,\delta.\, \alpha{\leftarrow}n[\gamma \otimes \epsilon] * \epsilon{\leftarrow}m[\delta \otimes \mathsf{tree}(c)] * \delta{\leftarrow}\varnothing_T * \beta{\leftarrow}\mathit{temp}[\varnothing_T]\}\\
\{\alpha{\leftarrow}n[\gamma \otimes m[\mathsf{tree}(c)]] * \beta{\leftarrow}\mathit{temp}[\varnothing_T]\}\\
\quad \mathtt{deleteNode}(\mathit{temp})\quad \}\\
\{\alpha{\leftarrow}n[\gamma \otimes m[\mathsf{tree}(c)]] * \beta{\leftarrow}\varnothing_T\}
\end{array}$$

Throughout the proof, we use the rules to separate out the footprint of a command, apply the appropriate small axiom, then use the rules to compress the result back into the original tree.

*Example 8 (Specifying Node Manipulation).* In Examples 2, 3 and 4 we gave three node manipulation programs: `simple`$(n)$, `nodeSwap`$(n,m)$ and `nodeCycle`$(n)$. The specifications for each of these programs are:

$$\begin{array}{ccc}
\{\alpha{\leftarrow}n[m[\beta] \otimes \gamma]\} & \{\alpha{\leftarrow}n[\gamma] * \beta{\leftarrow}m[\delta]\} & \{\alpha{\leftarrow}n[m[\beta] \otimes \gamma \otimes 1[\delta]]\}\\
\mathtt{simple}(n) & \mathtt{nodeSwap}(n,m) & \mathtt{nodeCycle}(n)\\
\{\alpha{\leftarrow}m[n[\beta] \otimes \gamma]\} & \{\alpha{\leftarrow}m[\gamma] * \beta{\leftarrow}n[\delta]\} & \{\alpha{\leftarrow}1[n[\beta] \otimes \gamma \otimes m[\delta]]\}
\end{array}$$

The derivations of these specifications are shown in **Fig. 9**.

*Example 9 (Specifying queuePop).* In a similar way, we can derive the following specification for the `queuePop` program from Example 5:

$$\begin{array}{c}\{\alpha{\leftarrow}n[m[\mathsf{tree}(c)] \otimes \gamma \otimes i[\beta]]\}\\ \mathtt{queuePop}(n)\\ \{\alpha{\leftarrow}m[\gamma \otimes n[\beta] \otimes i[\mathsf{tree}(c)]]\}\end{array}$$

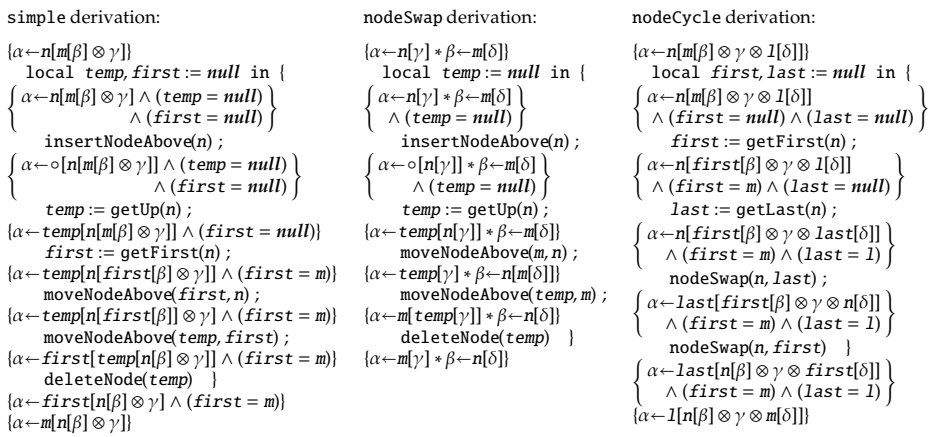



**Fig. 9.** Derivations of the Specifications for `simple`, `nodeSwap` and `nodeCycle`.

## 6 Conclusion

We have introduced Segment Logic for reasoning about structured data update in general, and tree update in particular. Using Segment Logic, we have demonstrated that it is possible to give small axioms for tree update commands such as DOM's `appendChild` command. In this paper, we have concentrated on a simple, lightweight tree update language. It is straightforward to transfer the techniques developed here to Featherweight DOM [6]. We do not envisage difficulties with extending the approach to the full DOM specification [16].

A typical Segment Logic proof separates the working tree into segments, identifying the tree segment which corresponds to the footprint of a command. It applies a small axiom to this segment, and compresses the updated fragment back into the original tree. This separation and compression is key to our Segment Logic reasoning. It is not unlike the unfolding and folding of abstract predicates, due to Parkinson and Vafeiadis [17]. The difference is that reasoning using Separation Logic with abstract predicates is implementation dependent: the formula $slist(l, i)$ describes a list $l$ implemented as a singly-linked list with heap address $i$; the formula $dlist(l, i, j)$ describes a list $l$ implemented as a doubly-linked list with heap addresses $i$ and $j$. By contrast, reasoning using Segment Logic is implementation independent: the formula $\alpha{\leftarrow}P$ describes e.g. a list or tree identified and satisfying formula or data type $P$ at abstract address $\alpha$.

Our next step is to design and formally specify a concurrent XML update language, combining ideas from Featherweight DOM, Concurrent Separation Logic [14] and Segment Logic. We believe that Segment Logic provides us with crucial technology for achieving this goal. For example, consider the program $\texttt{deleteTree}(n) \parallel \texttt{deleteTree}(m)$, which should succeed if the two trees being called are disjoint. A Segment Logic specification of this program is:

$$\{\alpha{\leftarrow}n[\mathrm{tree}(c_1)] * \beta{\leftarrow}m[\mathrm{tree}(c_2)]\}$$
$$\texttt{deleteTree}(n) \parallel \texttt{deleteTree}(m)$$
$$\{\alpha{\leftarrow}\varnothing_{\mathrm{T}} * \beta{\leftarrow}\varnothing_{\mathrm{T}}\}$$

Segment Logic allows us to establish such natural disjointness properties, since it combines reasoning directly about the abstract tree structure with using the

separating conjunction $*$. Our goal is to extend the update language presented here with parallel composition and critical regions, and adapt the Concurrent Separation Logic reasoning to provide a formal, compositional specification of a concurrent XML update language.

**Acknowledgments:** We thank Thomas Dinsdale-Young for many interesting discussions regarding this work and Viktor Vafeiadis for the name Segment Logic. Gardner acknowledges support of a Microsoft/RAEng Senior Research Fellowship. Wheelhouse acknowledges support of an EPSRC DTA award.

## References

1. J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. *FMCO, LNCS*, 4111, 2005.
2. C. Calcagno, T. Dinsdale-Young, and P. Gardner. Adjunct elimination in context logic for trees. *APLAS, LNCS*, 4807, 2007.
3. C. Calcagno, P. Gardner, and U. Zarfaty. Context logic and tree update. *POPL, ACM*, 40, 2005.
4. L. Cardelli and A. D. Gordon. Ambient logic. *Mathematical Structures in Computer Science, in press*, 2006.
5. M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13, 2002.
6. P. Gardner, G. Smith, M. Wheelhouse, and U. Zarfaty. Local Hoare reasoning about DOM. *PODS, ACM*, 27, 2008.
7. P. Gardner and M. Wheelhouse. Small specifications for tree update, extended version. http://www.doc.ic.ac.uk/~mjw03/PersonalWebpage/pdfs/moveFull.pdf, 2009.
8. P. Gardner and U. Zarfaty. Reasoning about high-level tree update and its low-level implementation. Technical Report DTR09-9, Imperial College, 2009.
9. H. Hosoya and B. C. Pierce. Xduce: A statically typed XML processing language. *TOIT, ACM*, 3, 2003.
10. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. *POPL, ACM*, 36, 2001.
11. R. Milner. Pi-nets: A graphical form of $\pi$-calculus. *ESOP, LNCS*, 788, 1994.
12. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes I & II. *Information and Computation*, 100, 1992.
13. P. W. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. *CSL, LNCS*, 15, 2001.
14. P. W. OHearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375, 2007.
15. M. Raza and P. Gardner. Footprints in local reasoning. *FoSSaCS, LNCS*, 4962, 2008.
16. G. Smith. Providing a formal specification for DOM core level 1. PhD Thesis, to be submitted December 2009.
17. V. Vafeiadis. Modular fine-grained concurrency verification. Technical Report UCAM-CL-TR-726, Cambridge, 2008.
18. W3C. Dom: Document object model. W3C recommendation. http://www.w3.org/DOM/, 2005.
19. H. Yang and P. W. O'Hearn. A semantic basis for local reasoning. *FoSSaCS, LNCS*, 2303, 2002.