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 local Hoare reasoning about mutable data structures using Separation Logic. They reason about the local parts of the memory accessed by programs, and thus construct their smallest complete specifications. Gardner *et al.* generalised their work, using Context Logic to reason about structured data at the same level of abstraction as the data itself. In particular, they developed a formal specification of the Document Object Model, a W3C XML update library. Whilst they kept to the spirit of local reasoning, they were not able to retain small specifications: for example, the specification of appendChild was not small. We show how to obtain small specifications by working with a more fine-grained context structure, allowing us to work with arbitrary tree fragments.

Key words: Specification, logical reasoning, program verification, locality

1 Introduction

Separation Logic [12], 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 the memory (the footprint) that are accessed by a program. In particular, they introduced *small axioms* for specifying the atomic commands, using the smallest heaps possible to obtain complete specifications of programs, and the *frame rule* for extending the reasoning to larger heaps. The resulting modular reasoning has been used to notable success for verifying memory safety properties of large C-programs.

Calcagno, Gardner and Zarfaty generalised Separation Logic to reason about more complex data structures, such as those found on the web, by providing a fundamental shift in the reasoning. Structured data update typically identifies the portion of data to be replaced, removes it, and inserts the new data in the same place. Gardner *et al.* introduced Context Logic to reason about both data and this place of insertion (contexts). Their original work applied Context Logic to reason about a simple tree update language, with analogous small axioms for the basic tree update commands and a generalised frame rule.

With Smith and Zarfaty, Gardner and Wheelhouse have applied Context Logic to provide a concise, compositional specification of the W3C Document Object Model (DOM) [17],[7], a library for XML update. In our initial paper[6], we introduced and reasoned about Featherweight DOM (called Minimal DOM in the paper), a fragment of DOM which concentrates on the DOM tree structure rather than the full DOM structure. The compositionality of our reasoning means that, as well as specifying the basic DOM commands, we can also reason about simple JavaScript programs which call DOM. Gardner and Smith have extended

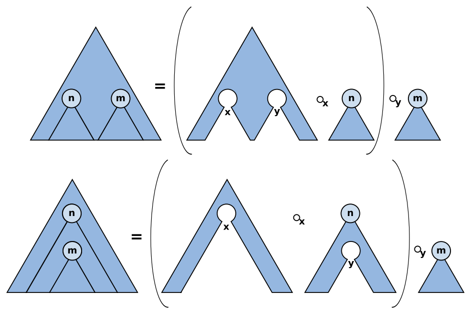


Fig. 1. Splitting up the working tree using multi-holed contexts.

the reasoning to the full DOM Core Level 1 specification[15]. This extension was a substantial piece of work, not because of the reasoning, but because the full DOM specification is large, underspecified and difficult to interpret.

Context Logic reasoning can be adapted to many familiar context styles associated with structured data. In our initial papers and the DOM work, we chose to use single-holed contexts because it was enough to introduce our ideas of reasoning about tree update. However, although our DOM specification keeps to the spirit of local reasoning, it does not have small axioms for all the atomic commands. This can be illustrated by the command `appendChild`(n, m) which moves the tree with top node identified by DOM identifier m to be the last child of the tree identified by n. Since n and m may be in distinct parts of the tree, it certainly seems natural to move to multi-holed Context Logic[2]. However, we shall see that multi-holed contexts are not enough.

Consider Figure 1 which indicates how the working tree splits in the two cases where `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 `appendChild`(n, m) in multi-holed Context Logic is:

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

Figure 1 shows, in each successful case, how the tree satisfies the precondition. The precondition specifies that the working tree can be split into a subtree with top node identified by m, and a context with hole variable β (equals y in the figure) satisfying context formula $C \circ_\alpha n[c_1]$. This formula $C \circ_\alpha n[c_1]$ states that the context can be split into a subcontext with top node n and an unspecified context with hole α (equals x in figure) given by context variable C. The postcondition states that the tree at m moves to be the last child of n, the empty tree replaces the tree at m, and the surrounding context denoted by variable C remains the same.

The problem with this `appendChild`(n, m) axiom is that it is not small, since it uses variable C to stand for a surrounding context which contains both n and m. We could put additional constraints on C to insist that the context is minimal, but this is not the point. Intuitively, the only part of the tree that `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. We need a finer way of splitting the tree to be able to capture this footprint.

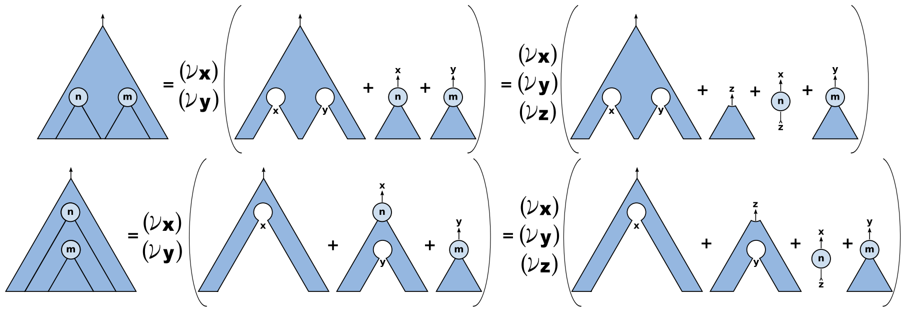


Fig. 2. Splitting up the working tree using tree fragments.

We use *tree fragments*. The idea is that the working tree can be split into tree fragments that can be reasoned about separately, but which still 'know' how they join back together. They are similar to multi-holed tree contexts in that they have unique hole labels; they are different in the way they join together. Tree fragments have unique hole addresses, which determine which holes the fragments can fill. With multi-holed contexts, it is the application function that determines which hole gets filled. Consider figure 2. In both example cases, the working tree is split into a bunch of tree fragments; the hole labels and addresses determine how the tree fragments join back up to form the original tree.

There are several further features about our tree fragments to observe. Consider the right-hand equalities of figure 2. In both cases, the tree fragment with top node n has been split into just the top node n with the same address and fresh hole label z, and another tree fragment with address z. 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`. If we take this to the extreme, we can cut up the tree structure into a collection of nodes with hole spaghetti (similar to heap cells), where the hole labels and addresses show how the nodes are joined together. Although this is possible, this is not how we use the hole labels and addresses. We only cut up the tree in a minimal way in order to provide the right fragment about which to reason. Also notice that the hole labels and addresses have been hidden by a freshness operator; the ν in the figure. With the restriction, the fragments can be compressed into the larger fragments indicated by the figure. Without the restriction, the fragments cannot be compressed and the hole labels and addresses would behave rather like pointers and addresses in a heap.

We introduce Context Logic for analysing tree fragments. It is analogous to our previous work on single-holed and multi-holed Context Logic in the sense that we analyse fragments of high-level trees. It is different from the previous work in that we use the commutative separating conjunction $*$ of Separation Logic rather than a non-commutative separating application. We also use the revelation connectives and freshness quantification of Gabbay and Pitts [5] and Cardelli and Gordon [4]. Interestingly, we shall see that these constructs are important for the weakest preconditions. Using this Context Logic for analysing tree fragments, we are able to give a small axiom for `appendChild`(n, m):

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

The precondition specifies two tree fragments: a tree fragment at variable address α with node $\mathbf{n}$ and a tree fragment at address β with a complete tree whose top node is $\mathbf{m}$. The postcondition states that the tree at $\mathbf{m}$ moves to be the last child of $\mathbf{n}$ being replaced by the empty tree. The axiom is small, in the sense that it captures the intuitive footprint of `appendChild`$(\mathbf{n}, \mathbf{m})$. We can extend the axiom to larger tree fragments using the normal frame rule for separation conjunction, a non-standard frame rule for revelation, and a rule for freshness quantification.

We must especially point out the difference in spirit between this work on reasoning about high-level tree update, and the work of O'Hearn, Parkinson and colleagues on reasoning about mutable data structures represented in heaps. The reasoning is at a different level of abstraction. O'Hearn and Parkinson are working with C-programs and object-oriented programs, where it is natural to work with the basic heap model and build up layers of abstraction. Our focus is on high-level tree update languages such as that specified by the DOM library, where we must work with the tree structures directly.

2 Tree Update Language

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

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

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

with the restriction that each hole label, $\mathbf{x} \in \mathrm{X}$, and location name, $\mathbf{n} \in \mathrm{I}$, occur at most once in a tree context $\mathbf{c}$, and subject to an equivalence $\mathbf{c}_1 \equiv \mathbf{c}_2$ stating that the $\otimes$ operator is associative with identity $\varnothing_{\mathrm{C}}$. The set of hole labels that occur in tree context $\mathbf{c}$ is denoted by $fn(\mathbf{c})$. We use $\mathbf{t}$, $\mathbf{t}_1$, $\mathbf{t}_2$ to denote tree contexts with no context holes.

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

$$ap_{\mathbf{x}}(\mathbf{c}_1, \mathbf{c}_2) = \begin{cases} \mathbf{c}_1[\mathbf{c}_2/\mathbf{x}] & \textit{if } \mathbf{x} \in fn(\mathbf{c}_1) \textit{ and } fn(\mathbf{c}_1) \cap fn(\mathbf{c}_2) \subseteq \{\mathbf{x}\} \\ \textit{undefined otherwise} \end{cases}$$

We abbreviate $\mathrm{ap}_{\mathbf{x}}(\mathbf{c}_1, \mathbf{c}_2)$ by $\mathbf{c}_1 \textcircled{x} \mathbf{c}_2$. We often omit the $\varnothing_{\mathrm{C}}$ leaves from a tree context to make it more readable, writing $\mathbf{n}[\mathbf{m} \otimes \mathbf{p}]$ instead of $\mathbf{n}[\mathbf{m}[\varnothing_{\mathrm{C}}] \otimes \mathbf{p}[\varnothing_{\mathrm{C}}]]$.

Our update language is a high-level, stateful, imperative language, based on variable assignment, and update commands. The program state is made up of two components. The first component is the working tree which contains all of the nodes we will be manipulating with our programs. The second component is a high-level variable store containing variables for both node identifiers and tree-shapes (trees modulo renaming of identifiers, allowing high-level manipulation of tree structures). The choice of having tree shapes, as opposed to trees, in the store illustrates a seemingly paradoxical property of high-level, imperative update: while some way of identifying nodes is required to specify the location of in-place updates, these identifiers are typically not considered an important part of the high-level structure itself.

Definition 3 (Tree-shapes). *Tree-shapes $\boldsymbol{t}_\circ \in \mathrm{T}_\circ$ are defined by the grammar:*

$$\begin{array}{lrrl} \text{tree-shape} & \boldsymbol{t}_\circ ::= & \varnothing_\mathrm{T} & \textit{empty tree} \\ & & \circ[\boldsymbol{t}_\circ] & \textit{tree node} \\ & & \boldsymbol{t}_\circ \otimes \boldsymbol{t}_\circ & \textit{tree composition} \end{array}$$

We write $\langle \boldsymbol{t} \rangle$ for the shape of a tree $\boldsymbol{t}$, where $\langle \varnothing_\mathrm{C} \rangle = \varnothing_\mathrm{T}$, $\langle \boldsymbol{n}[\boldsymbol{t}] \rangle = \circ[\langle \boldsymbol{t} \rangle]$ and $\langle \boldsymbol{t} \otimes \boldsymbol{t}' \rangle = \langle \boldsymbol{t} \rangle \otimes \langle \boldsymbol{t}' \rangle$. We only store complete trees. We write $\boldsymbol{t} \simeq \boldsymbol{t}'$ when $\langle \boldsymbol{t} \rangle = \langle \boldsymbol{t}' \rangle$.

Definition 4 (Variable Store). *The variable store $s \in \mathrm{S}$ consists of a pair of finite partial functions*

$$s : (Var_\mathrm{I} \rightharpoonup_{fin} \mathrm{I} \cup \{\boldsymbol{null}\}) \times (Var_{\mathrm{T}_\circ} \rightharpoonup_{fin} \mathrm{T}_\circ)$$

mapping location name variables $Var_\mathrm{I} = \{\mathtt{m}, \mathtt{n}, ...\}$ to location names or $\boldsymbol{null}$, and tree shape variables $Var_{\mathrm{T}_\circ} = \{\mathtt{t}, ...\}$ to tree shapes. We write $s[\mathtt{n} \mapsto \boldsymbol{n}]$ for the variable store s overwritten with $s(\mathtt{n}) = \boldsymbol{n}$, and similarly for $s[\mathtt{t} \mapsto \boldsymbol{t}_\circ]$.

To specify location name and tree-shape 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***. Tree-shapes are specified using a combination of tree-shape variables and constant tree-shape structures. We also require simple Boolean expressions.

Definition 5 (Expressions). *Location name expressions $\mathtt{N} \in Exp_\mathrm{I}$, tree-shape expressions $\mathtt{T} \in Exp_{\mathrm{T}_\circ}$ and Boolean expressions $\mathtt{B} \in Exp_\mathbb{B}$ are defined by the grammars:*

$$\begin{array}{ll} \mathtt{N} ::= \mathtt{n} \mid \boldsymbol{null} & \mathtt{n} \in Var_\mathrm{I} \\ \mathtt{T} ::= \varnothing_T \mid \mathtt{t} \mid \circ[\mathtt{T}] \mid \mathtt{T} \otimes \mathtt{T} & \mathtt{t} \in Var_{\mathrm{T}_\circ} \\ \mathtt{B} ::= \mathtt{N} = \mathtt{N} \mid \mathtt{T} = \mathtt{T} \mid \mathbf{false} \mid \mathtt{B} \Rightarrow \mathtt{B} & \end{array}$$

The valuation of an expression $\mathtt{E}$ in a store s is written $[\![\mathtt{E}]\!]s$ and has the obvious semantics. The standard classical Boolean connectives $\neg$, $\wedge$ and $\vee$ are derivable.

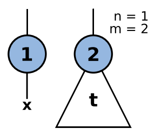
In previous work [9], we concentrated mainly on tree update commands for changing a tree with some top node $\boldsymbol{n}$. Here, we give node commands and subtree commands for changing the subtree under node $\boldsymbol{n}$. The language here is more flexible, allowing us to manipulate pieces of trees; the language in [3] is implementable. The node update commands consist of look-up commands that return a neighboring node in the tree, a delete command that removes a node from the tree, one node insertion command that puts a fresh node into the tree (the others are derivable), and node move commands that take a node out of the tree and replace it in a new position. The node movement 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 and consist of a copy command that stores the shape of a subtree, a delete command that removes an entire subtree from the tree, insertion commands that add new nodes to the tree and subtree move commands that take a subtree out of the tree and replace it in a new position.

Definition 6 (Tree Update Language). *The commands of the tree update language are defined by the node update commands $\mathbb{C}_{nodeUp}$, the tree update commands $\mathbb{C}_{treeUp}$, and the standard skip, variable assignment, sequencing, if-then-else and while commands:*

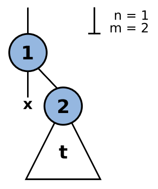
$\mathbb{C}_{nodeUp} ::=$	$n' := \texttt{getUp}(n)$	*get parent of node n*
	$n' := \texttt{getLeft}(n)$	*get previous sibling of node n*
	$n' := \texttt{getRight}(n)$	*get next sibling of node n*
	$n' := \texttt{getFirst}(n)$	*get first child of node n*
	$n' := \texttt{getLast}(n)$	*get last child of node n*
	$\texttt{deleteNode}(n)$	*delete node n*
	$\texttt{insertNodeAbove}(n)$	*insert a new node above node n*
	$\texttt{moveNodeAbove}(n, m)$	*move node m above node n*
	$\texttt{moveNodeLeft}(n, m)$	*move node m to the left of node n*
	$\texttt{moveNodeRight}(n, m)$	*move node m to the right of node n*
	$\texttt{prependNode}(n, m)$	*prepend node m to children of node n*
	$\texttt{appendNode}(n, m)$	*append node m to children of node n*
$\mathbb{C}_{treeUp} ::=$	$x := \texttt{copy}(n)$	*copy shape of subtree starting at node n*
	$\texttt{deleteSubtree}(n)$	*delete subtree beneath node n*
	$\texttt{insertLeft}(n, T)$	*insert tree shape T to the left of node n*
	$\texttt{insertRight}(n, T)$	*insert tree shape T to the right of node n*
	$\texttt{insertFirst}(n, T)$	*insert tree shape T as first child of n*
	$\texttt{insertLast}(n, T)$	*insert tree shape T as last child of n*
	$\texttt{moveSubLeft}(n, m)$	*move children of node m to the left of node n*
	$\texttt{moveSubRight}(n, m)$	*move children of node m to the right of node n*
	$\texttt{prependSub}(n, m)$	*prepend children of node m to children of node n*
	$\texttt{appendSub}(n, m)$	*append children of node m to children of node n*

The intuitive behavior of these commands should be self-explanatory. These commands are sufficient to express a wide range of tree manipulation. For example, allocation of a new tree or node can by expressed by the insertion of a literal tree-shape. In particular, inserting the tree-shape expression $\circ[\varnothing_T]$ creates a single new node, with fresh identifier, at a location given by the insert command. The only node insertion command that we need to give explicitly is for inserting a fresh node above an existing node. Our command set is not minimal, for example we could derive the copy command using a combination of lookup, insertion and recursion. We believe the commands chosen lead to a natural and expressive tree update language. We give the operational semantics in section 3, using tree fragments rather than trees, as it simplifies the interpretation of the Hoare triples in section 5.

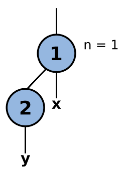
Example 1 (Move). DOM uses the command `appendChild`, whereas here we have `appendNode` and `appendSub`.. We use our basic commands to provide the standard `appendChild`(n, m) command. The diagrams illustrate the effect of the program on the part of a tree necessary for the program to run without faulting, with $n = 1$ and $m = 2$. The complete subtree beneath node m is needed as the whole tree is cut out of its original place in the tree and appended to node n.



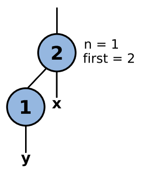
```
appendChild(n, m) ≜ insertRight(m, ∘[∅_T]);
                    temp := getRight(m);
                    appendSub(temp, m);
                    appendNode(n, m);
                    appendSub(m, temp);
                    deleteNode(temp)
```



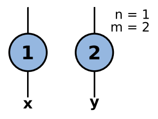
Example 2 (Simple Swap). Our node update commands enable us to define programs that act on arbitrary fragments of the tree. For example, consider the program `simple`(n) which swaps a node n with its first child:



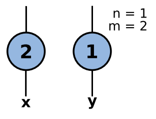
```
simple(n) ≜ insertNodeAbove(n);
            temp := getUp(n);
            first := getFirst(n);
            moveNodeAbove(first, n);
            moveNodeAbove(temp, first);
            deleteNode(temp)
```



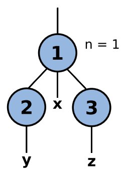
Example 3 (General Swap). The program `nodeSwap`(n, m) swaps the positions of arbitrary nodes n and m of a tree leaving their subtrees stationary:



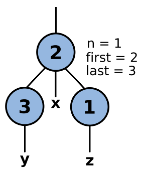
```
nodeSwap(n, m) ≜ insertNodeAbove(n);
                 temp := getUp(n);
                 moveNodeAbove(m, n);
                 moveNodeAbove(temp, m);
                 deleteNode(temp)
```



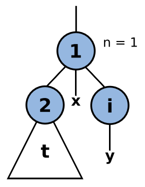
Example 4 (Node Rotate). Consider the program `nodeCycle`(n); which takes n, its first and last child and rotates these nodes with n taking the place of last child, last child taking the place of first child, and first child taking the place of n:



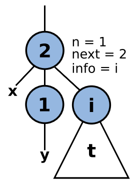
```
nodeCycle(n) ≜ first := getFirst(n);
               last := getLast(n);
               nodeSwap(n, last);
               nodeSwap(n, first)
```



Example 5 (Move and Node Swap). Consider a simple hierarchical queuing system given by the program `queuePop`(n) which puts the top element of the hierarchy to the back of the queue and promotes the next element to the top of the queue, carefully maintaining the data related to these elements:



```
queuePop(n) ≜ next := getFirst(n);
              info := getLast(n);
              nodeSwap(n, info);
              nodeSwap(next, info);
              appendChild(next, info)
```



In Section 6 we look at the reasoning of these programs and show how we can specify their behavior from the specifications of their component commands.

3 Tree Fragments

We now give our definition of tree fragments.

Definition 7 (Tree Fragments). *Tree fragments $f \in \mathrm{F}_{\mathrm{I}\ ,\mathrm{X}}$ are defined by the grammar:*

tree fragment	$f ::=$	$\varnothing_{\mathrm{F}}$	*empty tree fragment*
		$x{\leftarrow}c$	*tree context c with hole address x*
		$f + f$	*disjoint union*
		$(\nu x)(f)$	*label restriction*

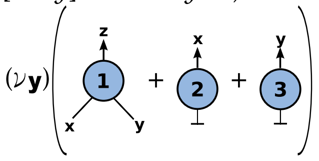
with the restriction that each label, $x \in \mathrm{X}$, occurs free at most once as a hole address and at most once as a hole label in a tree fragment f, and each location name, $n \in \mathrm{I}$, occurs at most once in the a fragment f. Tree fragments are also required to be cycle free. The set of hole labels and hole addresses that occur free in tree fragment f is denoted $fn(f)$.

Definition 8 (Tree Fragment Equivalence). *An equivalence relation $\equiv$ over tree fragments is defined by the following axioms:*

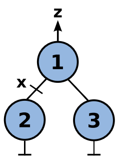
$$\begin{array}{rcll}
f + \varnothing_{\mathrm{F}} & \equiv & f & \\
f_1 + f_2 & \equiv & f_2 + f_1 & \\
f_1 + (f_2 + f_3) & \equiv & (f_1 + f_2) + f_3 & \\
(\nu x)(\varnothing_{\mathrm{F}}) & \equiv & \varnothing_{\mathrm{F}} & \\
(\nu x)(\nu y)(f) & \equiv & (\nu y)(\nu x)(f) & \\
(\nu x)(f) & \equiv & (\nu y)(f[y/x]) & \textit{if} \quad y \notin fn(f) \\
(\nu x)(y{\leftarrow}c + f) & \equiv & y{\leftarrow}c + (\nu x)(f) & \textit{if} \quad x \neq y \textit{ and } x \notin fn(c) \\
(\nu x)(y{\leftarrow}c_1 + x{\leftarrow}c_2) & \equiv & y{\leftarrow}c_1 \circledx c_2 & \textit{if} \quad x \in fn(c_1)
\end{array}$$

Most of these axioms involving restriction are unsurprising and follow from the π calculus [11]. The last restriction axiom is crucial and enables us to split and join tree fragments at will, as illustrated in Figure 2 of the introduction.

Restriction is well known as a mechanism for hiding names (wires) in Milner's process graphs in particular, and in arbitrary graphs in general. Our tree fragments have similarities and differences with this approach. The tree fragment $(\nu y)(z{\leftarrow}1[x \otimes y] + x{\leftarrow}2 + y{\leftarrow}3)$ can be illustrated as:



or even



$$\frac{s(n) = \mathbf{n} \quad f \equiv (\nu x)(f' + x{\leftarrow}\mathbf{n}[t])}{\mathtt{t} := \mathtt{copy}(n), s, f \rightsquigarrow s[\mathtt{t} \mapsto \langle \mathbf{n}[t] \rangle], f} \qquad \frac{s(n) = \mathbf{n} \quad f \equiv (\nu w,x,y,z)(f' + x{\leftarrow}\mathbf{m}[y \otimes \mathbf{n}[w] \otimes z])}{n' := \mathtt{getUp}(n), s, f \rightsquigarrow s[n' \mapsto \mathbf{m}], f}$$

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

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

$$\frac{\begin{array}{r} s(n) = \mathbf{n} \quad f \equiv (\nu x,y)(f'' + x{\leftarrow}\mathbf{n}[y]) \\ f' \equiv (\nu x,y)(f'' + x{\leftarrow}y) \end{array}}{\mathtt{deleteNode}(n), s, f \rightsquigarrow s, f'} \qquad \frac{\begin{array}{r} s(n) = \mathbf{n} \quad f \equiv (\nu x)(f'' + x{\leftarrow}\mathbf{n}[t]) \\ f' \equiv (\nu x)(f'' + x{\leftarrow}\mathbf{n}[\varnothing_C]) \end{array}}{\mathtt{deleteSubtree}(n), s, f \rightsquigarrow s, f'} \qquad \frac{\begin{array}{c} s(n) = \mathbf{n} \quad \langle t \rangle = [\![T]\!]s \quad t \text{ has fresh ids} \\ f \equiv (\nu x,y)(f'' + x{\leftarrow}\mathbf{n}[y]) \\ f' \equiv (\nu x,y)(f'' + x{\leftarrow}\mathbf{n}[y] \otimes t) \end{array}}{\mathtt{insertRight}(n, T), s, f \rightsquigarrow s, f'}$$

$$\frac{\begin{array}{r} s(n) = \mathbf{n} \quad f \equiv (\nu x,y)(f'' + x{\leftarrow}\mathbf{n}[y]) \\ \mathbf{n}' \text{ fresh id} \quad f' \equiv (\nu x,y)(f'' + x{\leftarrow}\mathbf{n}'[\mathbf{n}[y]]) \end{array}}{\mathtt{insertNodeAbove}(n), s, f \rightsquigarrow s, f'} \qquad \frac{\begin{array}{l} s(n) = \mathbf{n} \quad f \equiv (\nu w,x,y,z)(f'' + x{\leftarrow}\mathbf{n}[z] + y{\leftarrow}\mathbf{m}[w]) \\ s(m) = \mathbf{m} \quad f' \equiv (\nu w,x,y,z)(f'' + x{\leftarrow}\mathbf{m}[\mathbf{n}[z]] + y{\leftarrow}w) \end{array}}{\mathtt{moveNodeAbove}(n, m), s, f \rightsquigarrow s, f'}$$

$$\frac{\begin{array}{l} s(n) = \mathbf{n} \quad f \equiv (\nu w,x,y,z)(f'' + x{\leftarrow}\mathbf{n}[z] + y{\leftarrow}\mathbf{m}[w]) \\ s(m) = \mathbf{m} \quad f' \equiv (\nu w,x,y,z)(f'' + x{\leftarrow}\mathbf{n}[z \otimes \mathbf{m}] + y{\leftarrow}w) \end{array}}{\mathtt{appendNode}(n, m), s, f \rightsquigarrow s, f'} \qquad \frac{\begin{array}{l} s(n) = \mathbf{n} \quad f \equiv (\nu x,y,z)(f'' + x{\leftarrow}\mathbf{n}[z] + y{\leftarrow}\mathbf{m}[t]) \\ s(m) = \mathbf{m} \quad f' \equiv (\nu x,y,z)(f'' + x{\leftarrow}\mathbf{n}[z \otimes t] + y{\leftarrow}\mathbf{m}) \end{array}}{\mathtt{appendSub}(n, m), s, f \rightsquigarrow s, f'}$$

The cases for skip, assignment, sequencing, if-then-else and while-do are omitted as they are standard.
For get, insert, and move only some of the cases are given; the other cases are analogous.
Our commands fault when the datastructure does not satisfy any of the preconditions for that command.

Fig. 3. Operational Semantics of the Tree Update Language

and is analogous to the graph approach. However, we are not only using hole labels for wires. Consider the `appendChild` command in example 1. The tree fragment $(z{\leftarrow}1[x] + y{\leftarrow}2[t])$ updates to $(z{\leftarrow}1[x \otimes 2[t]] + y{\leftarrow}\varnothing_C)$: before update the fragment $y{\leftarrow}2[t]$ states that a tree can be put in hole z; after update $y{\leftarrow}\varnothing_C$ states that the empty tree can be put on hole z. In general, unlike heaps, we do not have a sense of arity being preserved by update: before update a node can have a certain number of children; after update it can have a different number of children. The closest work to tree fragments that we have come across is work by Back, which does not have restriction, but otherwise is analogous.

Our programming language manipulates nodes and complete trees. It does not refer to hole labels or hole addresses in any way. However, the operational semantics are greatly simplified by using either tree contexts or tree fragments. We use tree fragments, as this leads to a simpler interpretation of Hoare triples in section 5. We give the operational semantics of the tree update language in Figure 3. We use an evaluation relation $\rightsquigarrow$ relating configuration triples $\mathbb{C}, s, f$, terminal states s, f, and faults, where f refers to a tree fragment and the free program variables of a command $\mathbb{C}$ are free($\mathbb{C}$).

Our style of reasoning 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. A state can be made larger via disjoint tree fragment union or via label restriction. Separate from the reasoning, we also believe that the property of locality leads to good language design. Low-level imperative commands are typically local [12]. In our work on specifying

DOM [7] we demonstrated that the DOM commands are also local. Here, we insist on locality. Consider for example 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 context 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 if it is to be local. The behavior of the other update cases are similar.

4 Context Logic

First, we present the *logical environment* which is a set of functions mapping logical tree context variables to tree contexts, tree fragment variables to tree fragments, and context label variables to context labels. These variables allow us to refer to unchanged data in our pre- and post-conditions (see Definition 14), and make use of quantification in our weakest preconditions (see Figure 6). Tree-shape program variables refer to specific store values and are hence not quantified. We permit location name variables to have the standard dual role as both program variables and logical variables, hence they can be quantified.

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

$$e : (LVar_{\mathrm{C}} \to \mathrm{C}_{\mathrm{I}\ ,\mathrm{X}}) \times (LVar_{\mathrm{F}} \to \mathrm{F}_{\mathrm{I}\ ,\mathrm{X}}) \times (LVar_{\mathrm{X}} \to \mathrm{X})$$

mapping tree context variables $LVar_{\mathrm{C}} = \{\mathtt{c}, ...\}$ to tree contexts, tree fragment variables $LVar_{\mathrm{F}} = \{\mathtt{f}, ...\}$ to tree fragments and label variables $LVar_{\mathrm{X}} = \{\alpha, \beta, \gamma, \delta ...\}$ to labels.

We write $e[x \mapsto v]$ for the environment e overwritten with $e(x) = v$.

We are going to work with Context Logic for tree fragments. In fact, our logic has much in common with Separation Logic [12]. In particular, we have the standard classical formulae (additive connectives) and structural formulae (multiplicative connectives) from Separation Logic. The most important of these are the separation connective $*$ and its right adjoint $-\!\!*$. Given tree fragment formulae P_F and P'_F: the formula $P_F * P'_F$ describes a tree fragment that can be split into a tree fragment satisfying P_F and a separate tree fragment satisfying P'_F; and the formula $P_F -\!\!* P'_F$ describes a tree fragment which, when joined to a tree fragment satisfying P_F, results in a tree fragment satisfying P'_F.

We include label restriction in our tree fragments, which means it is natural to have freshness quantification $\reflectbox{N}\alpha$ [1] and revelation connectives ® and $-$® from Ambient Logic [5],[4]. These constructs are essential for our weakest preconditions. Given tree fragment formula P_F and hole label α: the formula $\reflectbox{N}\alpha.\, P_F$ describes a tree fragment that with a fresh label stored in variable α satisfies P_F; the formula α®P_F describes a tree fragment with a top level restriction of the value of α and, after removing that restriction, the remaining tree fragment satisfies P_F; and the formula α $-$® P_F describes a tree fragment which satisfies P_F once it has been extended with a restriction over label stored in variable α.

[1] For our model, it would be possible to use the existential quantification for hole labels instead of the freshness quantification. We choose freshness since it is the natural quantification to accompany revelation.

$$
\begin{array}{lll}
e,s,\boldsymbol{c} \models_C P_C \Rightarrow P'_C & \Leftrightarrow & e,s,\boldsymbol{c} \models_C P_C \\
 & & \quad \Rightarrow e,s,\boldsymbol{c} \models_C P'_C \\
e,s,\boldsymbol{c} \models_C \mathbf{false}_C & \Leftrightarrow & \textit{never} \\
e,s,\boldsymbol{c} \models_C \varnothing_C & \Leftrightarrow & \boldsymbol{c} \equiv \varnothing_C \\
e,s,\boldsymbol{c} \models_C \alpha & \Leftrightarrow & \boldsymbol{c} \equiv e(\alpha) \\
e,s,\boldsymbol{c} \models_C \boldsymbol{n}[P_C] & \Leftrightarrow & \exists \boldsymbol{c}_1.\, \boldsymbol{c} \equiv s(\boldsymbol{n})[\boldsymbol{c}_1] \\
 & & \quad \wedge e,s,\boldsymbol{c}_1 \models_C P_C \\
e,s,\boldsymbol{c} \models_C P_C \otimes P'_C & \Leftrightarrow & \exists \boldsymbol{c}_1, \boldsymbol{c}_2.\, \boldsymbol{c} \equiv \boldsymbol{c}_1 \otimes \boldsymbol{c}_2 \\
 & & \quad \wedge e,s,\boldsymbol{c}_1 \models_C P_C \\
 & & \quad \wedge e,s,\boldsymbol{c}_2 \models_C P'_C \\
e,s,\boldsymbol{c} \models_C T & \Leftrightarrow & \langle \boldsymbol{c} \rangle \equiv [\![T]\!]s \\
e,s,\boldsymbol{c} \models_C \mathtt{c} & \Leftrightarrow & \boldsymbol{c} \equiv e(\mathtt{c}) \\
e,s,\boldsymbol{c} \models_C \langle \mathtt{c} \rangle & \Leftrightarrow & \boldsymbol{c} \simeq e(\mathtt{c}) \\
e,s,\boldsymbol{c} \models_C B & \Leftrightarrow & [\![B]\!]s = \mathbf{true} \\
e,s,\boldsymbol{c} \models_C @\alpha & \Leftrightarrow & e(\alpha) \in fn(\boldsymbol{c})
\end{array}
$$

$$
\begin{array}{lll}
e,s,\boldsymbol{f} \models_F P_F \Rightarrow P'_F & \Leftrightarrow & e,s,\boldsymbol{f} \models_F P_F \Rightarrow e,s,\boldsymbol{f} \models_F P'_F \\
e,s,\boldsymbol{f} \models_F \mathbf{false}_F & \Leftrightarrow & \textit{never} \\
e,s,\boldsymbol{f} \models_F \varnothing_F & \Leftrightarrow & \boldsymbol{f} \equiv \varnothing_F \\
e,s,\boldsymbol{f} \models_F \boldsymbol{\alpha} \leftarrow P_C & \Leftrightarrow & \exists \boldsymbol{c}, \boldsymbol{x}.\, e(\alpha) = \boldsymbol{x} \wedge \boldsymbol{f} \equiv \boldsymbol{x} \leftarrow \boldsymbol{c} \wedge e,s,\boldsymbol{c} \models_C P_C \\
e,s,\boldsymbol{f} \models_F P_F * P'_F & \Leftrightarrow & \exists \boldsymbol{f}_1, \boldsymbol{f}_2.\, \boldsymbol{f} \equiv \boldsymbol{f}_1 + \boldsymbol{f}_2 \wedge e,s,\boldsymbol{f}_1 \models_F P_F \wedge e,s,\boldsymbol{f}_2 \models_F P'_F \\
e,s,\boldsymbol{f} \models_F \alpha \circledR P_F & \Leftrightarrow & \exists \boldsymbol{x}, \boldsymbol{f}'.\, e(\alpha) = \boldsymbol{x} \wedge \boldsymbol{f} \equiv (\boldsymbol{\nu x})(\boldsymbol{f}') \wedge e,s,\boldsymbol{f}' \models_F P_F \\
e,s,\boldsymbol{f} \models_F P_F \mathrel{-\!\!*} P'_F & \Leftrightarrow & \forall \boldsymbol{f}'.\, e,s,\boldsymbol{f}' \models_F P_F \wedge (\boldsymbol{f} + \boldsymbol{f}')\!\downarrow \Rightarrow e,s,\boldsymbol{f} + \boldsymbol{f}' \models_F P'_F \\
e,s,\boldsymbol{f} \models_F \alpha \mathrel{-\!\!\circledR} P_F & \Leftrightarrow & \exists \boldsymbol{x}, \boldsymbol{f}'.\, e(\alpha) = \boldsymbol{x} \wedge \boldsymbol{f}' \equiv (\boldsymbol{\nu x})(\boldsymbol{f}) \wedge e,s,\boldsymbol{f}' \models P_F \\
e,s,\boldsymbol{f} \models_F \mathtt{f} & \Leftrightarrow & \boldsymbol{f} \equiv e(\mathtt{f}) \\
e,s,\boldsymbol{f} \models_F B & \Leftrightarrow & [\![B]\!]s = \mathbf{true} \\
e,s,\boldsymbol{f} \models_F \exists \mathtt{var}.\, P_F & \Leftrightarrow & \exists v.\, e,s[\mathtt{var} \mapsto v],\boldsymbol{f} \models_F P_F \\
e,s,\boldsymbol{f} \models_F \exists \mathtt{lvar}.\, P_F & \Leftrightarrow & \exists v.\, e[\mathtt{lvar} \mapsto v],s,\boldsymbol{f} \models_F P_F \\
e,s,\boldsymbol{f} \models_F \text{И}\alpha.\, P_F & \Leftrightarrow & \exists \boldsymbol{x}.\, \boldsymbol{x} \# e,\boldsymbol{f} \wedge e[\alpha \mapsto \boldsymbol{x}],s,\boldsymbol{f} \models_F P_F
\end{array}
$$

Fig. 4. Satisfaction Relations of Context Logic for Tree Fragments.

We also use specific formulae for our tree fragment model. The tree context specific formulae are standard and include the variable α which expresses that a tree context is a context hole labeled whose label is the value of variable α. The specific connectives for tree fragments consist of $\varnothing_F$, describing an empty tree fragment, and $\alpha \leftarrow P_C$, describing a tree context satisfying P_C with hole address given by the value of variable α. This specific formula $\alpha \leftarrow P_C$ is analogous to the atomic formula $\boldsymbol{n} \mapsto \boldsymbol{n}_1, \ldots, \boldsymbol{n}_n$ except that we work with hole variables not node identifier variables. We also have existential quantification over location name, tree context and tree fragment variables. Finally, we add the tree context expression formula $@\alpha$ which describes a tree context that contains α free; the analogous formula for tree fragments is derivable.

Definition 10 (Formulae). *The formulae of Context Logic for tree fragments include tree context formulae P_C and tree fragment formulae P_F given by:*

$$
\begin{array}{lll}
P_C ::= & P_F ::= & \\
\quad P_C \Rightarrow P_C \mid \mathbf{false}_C & \quad P_F \Rightarrow P_F \mid \mathbf{false}_F & \textit{Classical formulae} \\
 & \mid P_F * P_F \mid P_F \mathrel{-\!\!*} P_F \mid \alpha \circledR P_F \mid \alpha \mathrel{-\!\!\circledR} P_F & \textit{Structural formulae} \\
\mid \varnothing_C \mid \alpha \mid \boldsymbol{n}[P_C] \mid P_C \otimes P_C & \mid \varnothing_F \mid \alpha \leftarrow P_C & \textit{Specific formulae} \\
\mid T \mid \mathtt{c} \mid \langle \mathtt{c} \rangle \mid B \mid @\alpha & \mid \mathtt{f} \mid B & \textit{Expression formulae} \\
 & \mid \exists \mathit{var}.\, P_F \mid \exists \mathit{lvar}.\, P_F \mid \text{И}\alpha.\, P_F & \textit{Quantification}
\end{array}
$$

Notice that the structure of the tree fragment formulae are orthogonal to the structure of the tree context formulae. It is easy to adapt this approach to other data structures such as sequences and terms.

Definition 11 (Satisfaction Relation). *Given a logical environment e and a variable store s, the semantics of Context Logic for tree fragments (see Figure 4) is given by two satisfaction relations $e,s,\boldsymbol{c} \models_C P_C$ and $e,s,\boldsymbol{f} \models_F P_F$ defined on tree contexts and tree fragments.*

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

$$
\begin{array}{rclcrcl}
\mathit{tree}(P_C) & \triangleq & P_C \wedge \neg \exists \alpha.\, @\alpha & \qquad & \Diamond P_F & \triangleq & \mathbf{true}_F * P_F \\
\boldsymbol{n} & \triangleq & \boldsymbol{n}[\varnothing_C] & & \mathsf{H}\alpha.\, P_F & \triangleq & \text{И}\alpha.\, \alpha \circledR P_F \\
\circ[P_C] & \triangleq & \exists \boldsymbol{m}.\, \boldsymbol{m}[P_C] & & & &
\end{array}
$$

Formula tree(P_C) describes a complete tree. Formula $\mathbf{n}$ allows us to drop the need to mention when a subtree is empty. Formula $\circ[P_C]$ allows us to drop the identifier of a node. Formula $\Diamond P_F$ allows us to express that, somewhere in the tree fragment, P_F holds. Finally, the hiding quantification, $\mathsf{H}\alpha$, is shorthand for revelation over a fresh label.

Example 6 (Context Logic Examples).

(a) The tree fragment formula $\alpha{\leftarrow}\mathbf{n}[\gamma] * \beta{\leftarrow}\mathbf{m}[\delta]$ describes a tree fragment consisting of a node $\mathbf{n}$ with address α and context hole γ and node $\mathbf{m}$ with address β and context hole δ; the $\mathbf{n}$ and $\mathbf{m}$ are non-equal. Now consider the fragment formula:

$$\omega{\leftarrow}\mathbf{n}[\gamma] \otimes \mathbf{m}[\delta] \quad \Leftrightarrow \quad \alpha, \beta \circledR (\omega{\leftarrow}\alpha \otimes \beta * \alpha{\leftarrow}\mathbf{n}[\gamma] * \beta{\leftarrow}\mathbf{m}[\delta])$$

The first formula states that the nodes $\mathbf{n}$ and $\mathbf{m}$ are siblings with address ω. The second formula states that the node $\mathbf{n}$ has address α, the node $\mathbf{m}$ has address β and the holes α and β are siblings with address ω. The labels α and β are revealed in the fragment and thus these two formulae are equivalent. The separation connective $*$ allows us to add more pieces to the tree fragment and the revelation connective $\circledR$ allows us to link up, and break apart, these pieces.

(b) The tree fragment formula $\alpha{\leftarrow}\mathbf{n}[\gamma] * \beta{\leftarrow}\mathbf{m}[\mathrm{tree}(\mathsf{c})]$ describes a tree fragment consisting of a single node $\mathbf{n}$ at address α and a complete tree with top node $\mathbf{m}$ at address β. This formula is the precondition of the small axiom of `appendChild`. In particular, due to the satisfaction relation for $*$, we know that node $\mathbf{n}$ cannot appear in the tree with top node $\mathbf{m}$ without violating the unique location name requirement of the tree fragment. This style of formula allows us to capture the tree fragments required for the successful running of the move subtree commands of our update language, elegantly expressing both the case where the trees at $\mathbf{n}$ and $\mathbf{m}$ are disjoint and the case where $\mathbf{n}$ is an ancestor of $\mathbf{m}$.

(c) The tree fragment formula $\exists \mathsf{c}.\, \mathsf{H}\alpha.\, ((\alpha{\leftarrow}\mathbf{n}[\varnothing_C] -\!\!* (\alpha -\!\!\circledR P_F)) * \alpha{\leftarrow}\mathbf{n}[\mathrm{tree}(\mathsf{c})])$ describes a tree fragment which can be split into a complete tree with top node $\mathbf{n}$ at address α and a tree fragment, that when extended by node $\mathbf{n}$ with the empty tree beneath it satisfies some property P_F. The use of $\circledR$ hides the label α allowing us to pull the tree with top node $\mathbf{n}$ out of the larger fragment. The use of $-\!\!\circledR$ describes putting the extracted tree fragment back into the larger fragment. If this tree fragment has had the subtree at $\mathbf{n}$ removed then the property P_F must now hold. This formula is the weakest precondition of the `deleteSubtree`($\mathbf{n}$) command.

5 Local Hoare Reasoning

We use the logic defined in Section 4 to provide local Hoare reasoning about programs written in the language defined in Definition 6. First we give a fault avoiding partial correctness interpretation of local Hoare triples following [18].

$$\begin{array}{rcl}
\{\emptyset_F\} & \texttt{skip} & \{\emptyset_F\} \\
\{\emptyset_F \wedge (n = n_0)\} & n := N & \{\emptyset_F \wedge (n = N[n_0/n])\} \\
\{\emptyset_F \wedge (t = t_0)\} & t := T & \{\emptyset_F \wedge (t = T[t_0/t])\} \\
\{\alpha\leftarrow n[\mathrm{tree}(c)]\} & t := \texttt{copy}(n) & \{\alpha\leftarrow n[\mathrm{tree}(c)] \wedge (t = \langle n[\mathrm{tree}(c)]\rangle)\} \\
\{\alpha\leftarrow m[\beta \otimes n[\delta] \otimes \gamma] \wedge (n' = n_0)\} & n' := \texttt{getUp}(n) & \{\alpha\leftarrow m[\beta \otimes n_{[n_0/n']}[\delta] \otimes \gamma] \wedge (n' = m)\} \\
\{\alpha\leftarrow n[\delta] \otimes m[\beta] \wedge (n' = n_0)\} & n' := \texttt{getRight}(n) & \{\alpha\leftarrow n_{[n_0/n']}[\delta] \otimes m[\beta] \wedge (n' = m)\} \\
\{\alpha\leftarrow m[\beta \otimes n[\delta]] \wedge (n' = n_0)\} & n' := \texttt{getRight}(n) & \{\alpha\leftarrow m[\beta \otimes n_{[n_0/n']}[\delta]] \wedge (n' = \boldsymbol{null})\} \\
\{\alpha\leftarrow n[\delta \otimes m[\beta]] \wedge (n' = n_0)\} & n' := \texttt{getLast}(n) & \{\alpha\leftarrow n_{[n_0/n']}[\delta \otimes m[\beta]] \wedge (n' = m)\} \\
\{\alpha\leftarrow n[\emptyset_C] \wedge (n' = n_0)\} & n' := \texttt{getLast}(n) & \{\alpha\leftarrow n_{[n_0/n']}[\emptyset_C] \wedge (n' = \boldsymbol{null})\} \\
\{\alpha\leftarrow n[\beta]\} & \texttt{deleteNode}(n) & \{\alpha\leftarrow \beta\} \\
\{\alpha\leftarrow n[\mathrm{tree}(c)]\} & \texttt{deleteSubtree}(n) & \{\alpha\leftarrow n[\emptyset_C]\} \\
\{\alpha\leftarrow n[\beta]\} & \texttt{insertNodeAbove}(n) & \{\alpha\leftarrow \circ[n[\beta]]\} \\
\{\alpha\leftarrow n[\beta]\} & \texttt{insertRight}(n, T) & \{\alpha\leftarrow n[\beta] \otimes 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[\emptyset_C]] * \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[\emptyset_C]\}
\end{array}$$

Fig. 5. Small Axioms for the Tree Update Language.

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

$$\begin{array}{lcl}
\{P_F\}\ \mathbb{C}\ \{Q_F\} & \Leftrightarrow & \forall e, s, f.\ free(\mathbb{C}) \cup free(P) \cup free(Q) \subseteq dom(s) \wedge e, s, f \models_F P_F \\
& & \Rightarrow \mathbb{C}, s, f \not\rightsquigarrow fault \wedge \forall s', f'.\ \mathbb{C}, s, f \rightsquigarrow s', f' \Rightarrow e, s', f' \models_F Q_F
\end{array}$$

Definition 14 (Small Axioms). *The Small Axioms are given in Figure 5.*

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

F L E :

$$\frac{\{P_F\}\ \mathbb{C}\ \{Q_F\}}{\{\mathsf{И}\alpha.\, P_F\}\ \mathbb{C}\ \{\mathsf{И}\alpha.\, Q_F\}}$$

A V E :

$$\frac{\{P_F\}\ \mathbb{C}\ \{Q_F\}}{\{\exists n.\, P_F\}\ \mathbb{C}\ \{\exists n.\, Q_F\}} \quad n \notin Free(\mathbb{C})$$

R F :

$$\frac{\{P_F\}\ \mathbb{C}\ \{Q_F\}}{\{\alpha \circledR P_F\}\ \mathbb{C}\ \{\alpha \circledR Q_F\}}$$

S F :

$$\frac{\{P_F\}\ \mathbb{C}\ \{Q_F\}}{\{P_F * R_F\}\ \mathbb{C}\ \{Q_F * R_F\}} \quad Mod(\mathbb{C}) \cap Free(R_F) = \{\}$$

The Auxiliary Variable Elimination and Separation Frame rules are standard from Separation Logic. The Revelation Frame rule is the natural consequence of having restriction in the model. The Fresh Label Elimination rule is analogous to the Auxiliary Variable Elimination rule.
Our reasoning system is sound. The weakest preconditions of our tree update commands (given in Figure 6) are derivable; proof in full paper [8]. This means that our local Hoare reasoning is complete for straight line code.

6 Examples

We provide specifications for each of the example programs given in section 2. We make the assumption that all locally defined variables, such as `temp` in the appendChild program, are disjoint from all other program variables.

$$\begin{array}{rcl}
\{P_F\} & \mathtt{skip} & \{P_F\} \\
\{\exists n_0.\,(n = n_0) \wedge P_F[N/n]\} & n := N & \{P_F\} \\
\{\exists t_0.\,(t = t_0) \wedge P_F[T/t]\} & t := T & \{P_F\} \\
\{\exists c.\,\mathsf{H}\alpha.\,\Diamond\alpha{\leftarrow}n[\mathrm{tree}(c)] \wedge (\alpha \mathbin{-\circledR} P_F[\langle n[\mathrm{tree}(c)]\rangle/t])\} & t := \mathtt{copy}(n) & \{P_F\} \\
\{\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 \mathbin{-\circledR} P_F[m/n'])\} & n' := \mathtt{getUp}(n) & \{P_F\} \\
\{\exists m, n_0.\,\mathsf{H}\alpha,\beta,\delta.\;\begin{array}{r}\Diamond\alpha{\leftarrow}n[\delta] \otimes m[\beta] \wedge (n' = n_0) \wedge (\alpha,\beta,\delta \mathbin{-\circledR} P_F[m/n']) \\ \vee\, \Diamond\alpha{\leftarrow}m[\beta \otimes n[\delta]] \wedge (n' = n_0) \wedge (\alpha,\beta,\delta \mathbin{-\circledR} P_F[\mathbf{null}/n'])\end{array}\} & n' := \mathtt{getRight}(n) & \{P_F\} \\
\{\exists m, n_0.\,\mathsf{H}\alpha,\beta,\delta.\;\begin{array}{r}\Diamond\alpha{\leftarrow}n[\delta \otimes m[\beta]] \wedge (n' = n_0) \wedge (\alpha,\beta,\delta \mathbin{-\circledR} P_F[m/n']) \\ \vee\, \Diamond\alpha{\leftarrow}n[\varnothing_C] \wedge (n' = n_0) \wedge (\alpha \mathbin{-\circledR} P_F[\mathbf{null}/n'])\end{array}\} & n' := \mathtt{getLast}(n) & \{P_F\} \\
\{\mathsf{H}\alpha,\beta.\,((\alpha{\leftarrow}\beta \multimap (\alpha,\beta \mathbin{-\circledR} P_F)) * \alpha{\leftarrow}n[\beta])\} & \mathtt{deleteNode}(n) & \{P_F\} \\
\{\exists c.\,\mathsf{H}\alpha.\,((\alpha{\leftarrow}n[\varnothing_C] \multimap (\alpha \mathbin{-\circledR} P_F)) * \alpha{\leftarrow}n[\mathrm{tree}(c)])\} & \mathtt{deleteSubtree}(n) & \{P_F\} \\
\{\mathsf{H}\alpha,\beta.\,((\alpha{\leftarrow}\circ[n[\beta]] \multimap (\alpha,\beta \mathbin{-\circledR} P_F)) * \alpha{\leftarrow}n[\beta])\} & \mathtt{insertNodeAbove}(n) & \{P_F\} \\
\{\mathsf{H}\alpha,\beta.\,((\alpha{\leftarrow}n[\beta] \otimes T \multimap (\alpha,\beta \mathbin{-\circledR} P_F)) * \alpha{\leftarrow}n[\beta])\} & \mathtt{insertRight}(n, T) & \{P_F\} \\
\{\mathsf{H}\alpha,\beta,\gamma,\delta.\,(((\alpha{\leftarrow}m[n[\gamma]] * \beta{\leftarrow}\delta) \multimap (\alpha,\beta,\gamma,\delta \mathbin{-\circledR} P_F)) * (\alpha{\leftarrow}n[\gamma] * \beta{\leftarrow}m[\delta]))\} & \mathtt{moveNodeAbove}(n, m) & \{P_F\} \\
\{\mathsf{H}\alpha,\beta,\gamma,\delta.\,(((\alpha{\leftarrow}n[\gamma \otimes m[\varnothing_C]] * \beta{\leftarrow}\delta) \multimap (\alpha,\beta,\gamma,\delta \mathbin{-\circledR} P_F)) * (\alpha{\leftarrow}n[\gamma] * \beta{\leftarrow}m[\delta]))\} & \mathtt{appendNode}(n, m) & \{P_F\} \\
\{\exists c.\,\mathsf{H}\alpha,\beta,\gamma.\,(((\alpha{\leftarrow}n[\gamma \otimes \mathrm{tree}(c)] * \beta{\leftarrow}m[\varnothing_C]) \multimap (\alpha,\beta,\gamma \mathbin{-\circledR} P_F)) * (\alpha{\leftarrow}n[\gamma] * \beta{\leftarrow}m[\mathrm{tree}(c)]))\} & \mathtt{appendSub}(n, m) & \{P_F\}
\end{array}$$

Fig. 6. Weakest Preconditions of the atomic commands given in Figure 3.

appendChild: In example 1 of section 2 we gave the program `appendChild`. Its specification and derivation are:

$$\begin{array}{c}
\{\alpha{\leftarrow}n[\gamma] * \beta{\leftarrow}m[\mathrm{tree}(c)]\} \\
\mathtt{appendChild}(n, m) \\
\{\alpha{\leftarrow}n[\gamma \otimes m[\mathrm{tree}(c)]] * \beta{\leftarrow}\varnothing_C\}
\end{array}
\qquad
\begin{array}{l}
\{\alpha{\leftarrow}n[\gamma] * \beta{\leftarrow}m[\mathrm{tree}(c)]\} \\
\{\mathsf{H}\delta.\,\alpha{\leftarrow}n[\gamma] * \beta{\leftarrow}m[\delta] * \delta{\leftarrow}\mathrm{tree}(c)\} \\
\mathtt{insertRight}(m, \circ[\varnothing_T]); \\
\{\mathsf{H}\delta.\,\alpha{\leftarrow}n[\gamma] * \beta{\leftarrow}m[\delta] \otimes \circ[\varnothing_T] * \delta{\leftarrow}\mathrm{tree}(c)\} \\
temp := \mathtt{getLeft}(m); \\
\{\mathsf{H}\delta.\,\alpha{\leftarrow}n[\gamma] * \beta{\leftarrow}m[\delta] \otimes temp[\varnothing_C] * \delta{\leftarrow}\mathrm{tree}(c)\} \\
\{\alpha{\leftarrow}n[\gamma] * \beta{\leftarrow}m[\mathrm{tree}(c)] \otimes temp[\varnothing_C]\} \\
\{\mathsf{H}\delta,\epsilon.\,\alpha{\leftarrow}n[\gamma] * \beta{\leftarrow}\delta \otimes \epsilon * \delta{\leftarrow}m[\mathrm{tree}(c)] * \epsilon{\leftarrow}temp[\varnothing_C]\} \\
\mathtt{appendSub}(temp, m); \\
\{\mathsf{H}\delta,\epsilon.\,\alpha{\leftarrow}n[\gamma] * \beta{\leftarrow}\delta \otimes \epsilon * \delta{\leftarrow}m[\varnothing_C] * \epsilon{\leftarrow}temp[\mathrm{tree}(c)]\} \\
\mathtt{appendNode}(n, m); \\
\{\mathsf{H}\delta,\epsilon.\,\alpha{\leftarrow}n[\gamma \otimes m[\varnothing_C]] * \beta{\leftarrow}\delta \otimes \epsilon * \delta{\leftarrow}\varnothing_C * \epsilon{\leftarrow}temp[\mathrm{tree}(c)]\} \\
\mathtt{appendSub}(m, temp); \\
\{\mathsf{H}\delta,\epsilon.\,\alpha{\leftarrow}n[\gamma \otimes m[\mathrm{tree}(c)]] * \beta{\leftarrow}\delta \otimes \epsilon * \delta{\leftarrow}\varnothing_C * \epsilon{\leftarrow}temp[\varnothing_C]\} \\
\mathtt{deleteNode}(temp) \\
\{\mathsf{H}\delta,\epsilon.\,\alpha{\leftarrow}n[\gamma \otimes m[\mathrm{tree}(c)]] * \beta{\leftarrow}\delta \otimes \epsilon * \delta{\leftarrow}\varnothing_C * \epsilon{\leftarrow}\varnothing_C\} \\
\{\alpha{\leftarrow}n[\gamma \otimes m[\mathrm{tree}(c)]] * \beta{\leftarrow}\varnothing_C\}
\end{array}$$

The `appC`(n, m) program below has equivalent behavior to `appendChild`(n, m) modulo renaming of the tree at m, and an analogous specification:

$$\begin{array}{ll}
\mathtt{appC}(n, m) \triangleq t := \mathtt{copy}(m); & \{\alpha{\leftarrow}n[\gamma] * \beta{\leftarrow}m[\mathrm{tree}(c)]\} \\
\qquad\qquad\quad\;\; \mathtt{deleteTree}(m); & \qquad\quad \mathtt{appC}(n, m) \\
\qquad\qquad\quad\;\; \mathtt{insertLast}(n, t) & \{\alpha{\leftarrow}n[\gamma \otimes \langle m[\mathrm{tree}(c)]\rangle] * \beta{\leftarrow}\varnothing_C\}
\end{array}$$

The derivation of this specification is similar to the derivation of `appendChild` given above. In multi-holed Context logic we could give small specifications for each of the atomic commands used to construct the `appC` program. However, we cannot provide a small specification for `appC` directly from the small axioms of these atomic commands. As we discussed in the introduction, we instead must use a specification with a context variable C to describe the linking context between nodes n and m.

Node Manipulation: In examples 2, 3 and 4 of section 2 we gave three node manipulation programs; `simple`(n), `nodeSwap`(n, m) and `nodeCycle`(n). The specifications for each of these programs are:

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

The derivations of these specifications are shown in Figure 7.

`simple` derivation:

```
$\{\alpha\leftarrow n[m[\beta]\otimes\gamma]\}$
insertNodeAbove($n$);
$\{\alpha\leftarrow \circ\,[n[m[\beta]\otimes\gamma]]\}$
$\mathit{temp}$ := getUp($n$);
$\{\alpha\leftarrow \mathit{temp}[n[m[\beta]\otimes\gamma]]\}$
$\mathit{first}$ := getFirst($n$);
$\{\alpha\leftarrow \mathit{temp}[n[\mathit{first}[\beta]\otimes\gamma]]\wedge(m=\mathit{first})\}$
moveNodeAbove($\mathit{first}, n$);
$\{\alpha\leftarrow \mathit{temp}[n[\mathit{first}[\beta]]\otimes\gamma]\wedge(m=\mathit{first})\}$
moveNodeAbove($\mathit{temp}, \mathit{first}$);
$\{\alpha\leftarrow \mathit{first}[\mathit{temp}[n[\beta]\otimes\gamma]]\wedge(m=\mathit{first})\}$
deleteNode($\mathit{temp}$)
$\{\alpha\leftarrow \mathit{first}[n[\beta]\otimes\gamma]\wedge(m=\mathit{first})\}$
$\{\alpha\leftarrow m[n[\beta]\otimes\gamma]\}$
```

`nodeSwap` derivation:

```
$\{\alpha\leftarrow n[\gamma]*\beta\leftarrow m[\delta]\}$
insertNodeAbove($n$);
$\{\alpha\leftarrow \circ\,[n[\gamma]]*\beta\leftarrow m[\delta]\}$
$\mathit{temp}$ := getUp($n$);
$\{\alpha\leftarrow \mathit{temp}[n[\gamma]]*\beta\leftarrow m[\delta]\}$
moveNodeAbove($m, n$);
$\{\alpha\leftarrow \mathit{temp}[\gamma]*\beta\leftarrow n[m[\delta]]\}$
moveNodeAbove($\mathit{temp}, m$);
$\{\alpha\leftarrow m[\mathit{temp}[\gamma]]*\beta\leftarrow n[\delta]\}$
deleteNode($\mathit{temp}$)
$\{\alpha\leftarrow m[\gamma]*\beta\leftarrow n[\delta]\}$
```

`nodeCycle` derivation:

```
$\{\alpha\leftarrow n[m[\beta]\otimes\gamma\,\mathtt{l}[\delta]]\}$
$\mathit{first}$ := getFirst($n$);
$\left\{\begin{array}{r}\alpha\leftarrow n[\mathit{first}[\beta]\otimes\gamma\,\mathtt{l}[\delta]]\\ \wedge(\mathit{first}=m)\end{array}\right\}$
$\mathit{last}$ := getLast($n$);
$\left\{\begin{array}{c}\alpha\leftarrow n[\mathit{first}[\beta]\otimes\gamma\,\mathit{last}[\delta]]\\ \wedge(\mathit{first}=m)\wedge(\mathit{last}=\mathtt{l})\end{array}\right\}$
nodeSwap($n, \mathit{last}$);
$\left\{\begin{array}{c}\alpha\leftarrow \mathit{last}[\mathit{first}[\beta]\otimes\gamma\, n[\delta]]\\ \wedge(\mathit{first}=m)\wedge(\mathit{last}=\mathtt{l})\end{array}\right\}$
nodeSwap($n, \mathit{first}$)
$\left\{\begin{array}{c}\alpha\leftarrow \mathit{last}[n[\beta]\otimes\gamma\,\mathit{first}[\delta]]\\ \wedge(\mathit{first}=m)\wedge(\mathit{last}=\mathtt{l})\end{array}\right\}$
$\{\alpha\leftarrow \mathtt{l}[n[\beta]\otimes\gamma\, m[\delta]]\}$
```

Fig. 7. Derivations of the specifications for `simple`, `nodeSwap` and `nodeCycle`.

Hierarchical Queue: The specification and derivation of the queuePop program from example 5 of section 2 are:

$$\{\alpha\leftarrow n[m[\mathrm{tree}(c)]\otimes\gamma\otimes i[\beta]]\}$$
$$\text{queuePop}(n)$$
$$\{\alpha\leftarrow m[\gamma\otimes n[\beta]\otimes i[\mathrm{tree}(c)]]\}$$

```
$\{\alpha\leftarrow n[m[\mathrm{tree}(c)]\otimes\gamma\otimes i[\beta]]\}$
$\mathit{next}$ := getFirst($n$);
$\{\alpha\leftarrow n[\mathit{next}[\mathrm{tree}(c)]\otimes\gamma\otimes i[\beta]]\wedge(\mathit{next}=m)\}$
$\mathit{info}$ := getLast($n$);
$\{\alpha\leftarrow n[\mathit{next}[\mathrm{tree}(c)]\otimes\gamma\otimes \mathit{info}[\beta]]\wedge(\mathit{next}=m)\wedge(\mathit{info}=i)\}$
nodeSwap($n, \mathit{info}$);
$\{\alpha\leftarrow \mathit{info}[\mathit{next}[\mathrm{tree}(c)]\otimes\gamma\otimes n[\beta]]\wedge(\mathit{next}=m)\wedge(\mathit{info}=i)\}$
nodeSwap($\mathit{next}, \mathit{info}$);
$\{\alpha\leftarrow \mathit{next}[\mathit{info}[\mathrm{tree}(c)]\otimes\gamma\otimes n[\beta]]\wedge(\mathit{next}=m)\wedge(\mathit{info}=i)\}$
appendChild($\mathit{next}, \mathit{info}$)
$\{\alpha\leftarrow \mathit{next}[\gamma\otimes n[\beta]\otimes \mathit{info}[\mathrm{tree}(c)]]\wedge(\mathit{next}=m)\wedge(\mathit{info}=i)\}$
$\{\alpha\leftarrow m[\gamma\otimes n[\beta]\otimes i[\mathrm{tree}(c)]]\}$
```

7 Conclusion

We have shown how to give small axioms for commands such as the `appendChild` command, by developing Context Logic reasoning for tree fragments. It is straightforward to transfer the techniques developed here to Featherweight DOM [7]. For this paper, we have worked with the intuitive understanding of what it means for command axioms to be small. With Raza, Gardner has developed the formal definitions of footprints and small specifications for abstract local functions using Abstract Separation Logic [14]. It would be interesting to extend this abstract theory to the tree fragments and reasoning studied here, and prove that the axioms really are small.

We believe the results presented here form a pivotal step in the development of Context Logic reasoning. The key point about Context Logic is that it reasons about structured data at the same level of abstraction as the data itself. Our previous work used various forms of separating application, which were appropriate for the applications we had in mind. Here, we move nearer to Separation Logic reasoning. We use the separating conjunction for reasoning about disjoint tree fragments, and the revelation connectives and freshness quantification for reasoning about restriction. This reasoning style means that we can pull out different tree fragments from the working tree, update them, and put them back again in any undetermined order. The ideas in this paper provides us with the technology to extend our reasoning to concurrent tree update following O'Hearn's work on concurrent Separation Logic [1],[13],[16].

References

1. S. Brookes. A semantics for concurrent separation logic. *Theor. Comput. Sci.*, 375, 2007.
2. C. Calcagno, T. Dinsdale-Young, and P. Gardner. Adjuct elimination in context logic for trees. In *APLAS*, 2007.
3. C. Calcagno, P. Gardner, and U. Zarfaty. Context logic and tree update. In *POPL*, 2005.
4. L. Cardelli and A. D. Gordon. Ambient logic. 2006.
5. M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding.
6. P. Gardner, G. Smith, M. Wheelhouse, and U. Zarfaty. Dom: Towards a formal specification. In *Plan-X: Programming Language Techniques for XML*, 2008.
7. P. Gardner, G. Smith, M. Wheelhouse, and U. Zarfaty. Local hoare reasoning about dom. In *PODS: Symposium on Principles of Database Systems*, 2008.
8. P. Gardner and M. Wheelhouse. Small specifications for tree update, 2009. http://www.doc.ic.ac.uk/~mjw03/PersonalWebpage/papers.html.
9. P. Gardner and U. Zarfaty. Integrated reasoning about high-level tree update and a low-level implementation. submitted to publication.
10. S. Ishtiaq and P. W. O'Hearn. Bi as an assertion language for mutable data structures. In *POPL*, pages 14–26, 2001.
11. R. Milner. A calculus of mobile processes, parts. *I and II. Information and Computation*, 100:1–77, 1992.
12. P. O'Hearn, J. Reynolds, and H. Yang. *Local Reasoning about Programs that Alter Data Structures*, volume 2142. January 2001.
13. P. W. Ohearn. Resources, concurrency and local reasoning. In *Theoretical Computer Science*, pages 49–67. Springer, 2004.
14. M. Raza and P. Gardner. Footprints in local reasoning. In *FoSSaCS '08: Proceedings of the 11th International Conference on Foundations of Software Science and Computation Structures*, volume 4962, pages 201–215, London, UK, 2008. Springer.
15. G. Smith. Providing a formal specification for dom core level 1, 2009. PhD Thesis. Ongoing work.
16. V. Vafeiadis and M. Parkinson. A marriage of rely/guarantee and separation logic. In *In 18th CONCUR*, pages 256–271. Springer, 2007.
17. W3C. Dom: Document object model. W3C recommendation, 2005. http://www.w3.org/DOM/.
18. H. Yang and P. W. O'Hearn. A semantic basis for local reasoning. In *FoSSaCS '02: Proceedings of the 5th International Conference on Foundations of Software Science and Computation Structures*, pages 402–416, London, UK, 2002. Springer-Verlag.