

Small Specifications for Tree Update (Full Version)

Philippa Gardner and Mark Wheelhouse

Imperial College London, Department of Computing
pg@doc.ic.ac.uk 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 update. 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. In this paper 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, such small axioms are essential for reasoning about concurrent tree update.

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

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 [12]. 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 [17].

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

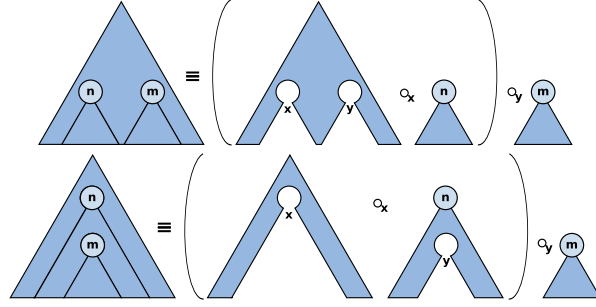


Fig. 1. Splitting up the Working Tree using Multi-holed contexts.

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 with sequential DOM [6] and O’Hearn’s Concurrent Separation Logic [12]. 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, 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 `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 `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)` using multi-holed Context Logic [2] is ¹:

$$\begin{aligned} & \{(C \circ_{\alpha} n[c_1]) \circ_{\beta} m[\text{tree}(c_2)]\} \\ & \text{appendChild}(n, m) \\ & \{(C \circ_{\alpha} n[c_1 \otimes m[\text{tree}(c_2)]]\} \circ_{\beta} \emptyset \end{aligned}$$

The precondition specifies that the working tree can be split into a subtree with top node m , and a tree context with hole variable β (y in Fig. 1) satisfying the

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.

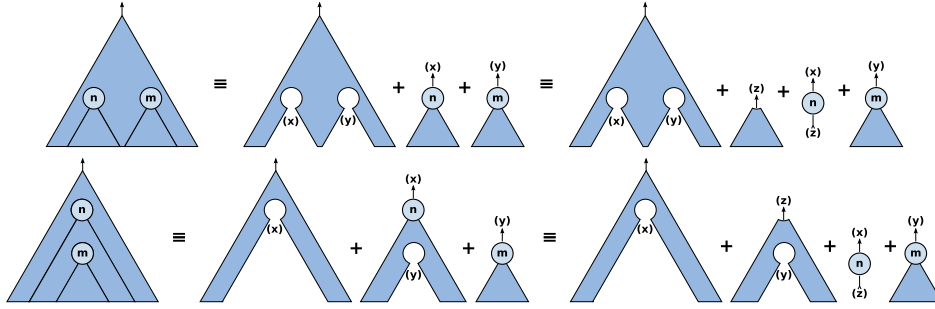


Fig. 2. Splitting up the Working Tree using Tree Segments.

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 α (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 `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 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 π -calculus [11].

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 `appendChild`(n, m) is:

$$\frac{\{\alpha \leftarrow n[\gamma] * \beta \leftarrow m[\text{tree}(c)]\}}{\text{appendChild}(n, m)} \{\alpha \leftarrow n[\gamma \otimes m[\text{tree}(c)]] * \beta \leftarrow \emptyset_T\}$$

The precondition specifies two tree segments: a node n at address variable α (x in **Fig. 2**) and a complete tree whose top node is m at address β (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 $\text{ID} = \{m, n, \dots\}$ for location names and $X = \{x, y, z, \dots\}$ for hole identifiers.

Definition 1 (Tree Contexts). *Multi-holed tree contexts $c \in C_{\text{ID}, X}$ are defined by:*

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

with the restriction that each hole identifier, $x \in X$, and location name, $n \in \text{Id}$, 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 \emptyset_C . The set of hole identifiers that occur in tree context c is denoted $\text{free}(c)$. 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 $ap_x: C_{\text{Id},X} \times C_{\text{Id},X} \rightarrow C_{\text{Id},X}$ indexed by hole labels x :

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

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

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 $\sigma \in \Sigma$ consists of a finite partial function

$$\sigma: \text{Var}_{\text{Id}} \rightarrow_{\text{fin}} \text{Id} \cup \{\text{null}\}$$

mapping location name variables $\text{Var}_{\text{Id}} = \{m, n, \dots\}$ to location names or **null**. We write $\sigma[n \mapsto n]$ for the variable store σ overwritten with $\sigma(n) = n$. We write $\sigma \setminus n$ for the variable store σ with the variable n removed from its domain.

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 $N \in \text{Exp}_{\text{Id}}$ and Boolean expressions $B \in \text{Exp}_{\text{B}}$ are defined by:

$$\begin{aligned} N &::= n \mid \text{null} & n &\in \text{Var}_{\text{Id}} \\ B &::= N = N \mid \text{false} \mid B \Rightarrow B \end{aligned}$$

The valuation of an expression E in a store σ is written $\llbracket E \rrbracket \sigma$ and has the obvious semantics. The standard 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 are defined by the node update commands \mathbb{C}_{nodeUp} , the tree update commands \mathbb{C}_{treeUp} , and the standard skip, variable assignment, local variable, sequencing, if-then-else and while-do commands:*

$\mathbb{C} ::= \text{skip}$	<i>no operation</i>
$n := N$	<i>variable assignment</i>
$\text{local } v := E \text{ in } \mathbb{C}$	<i>local variable</i>
\mathbb{C}_{nodeUp}	<i>node update commands</i>
\mathbb{C}_{treeUp}	<i>tree update commands</i>
$\mathbb{C}; \mathbb{C}$	<i>sequencing</i>
$\text{if } B \text{ then } \mathbb{C} \text{ else } \mathbb{C}$	<i>if-then-else branching</i>
$\text{while } B \text{ do } \mathbb{C}$	<i>while-do loop</i>

where,

$\mathbb{C}_{nodeUp} ::= n' := \text{getUp}(n)$	<i>get parent of node n and record it in n'</i>
$n' := \text{getLeft}(n)$	<i>get previous sibling of node n</i>
$n' := \text{getRight}(n)$	<i>get next sibling of node n</i>
$n' := \text{getFirst}(n)$	<i>get first child of node n</i>
$n' := \text{getLast}(n)$	<i>get last child of node n</i>
$\text{deleteNode}(n)$	<i>delete node n</i>
$\text{insertNodeAbove}(n)$	<i>insert a new node above node n</i>
$\text{insertNodeLeft}(n)$	<i>insert a new node to the left of node n</i>
$\text{insertNodeRight}(n)$	<i>insert a new node to the right of node n</i>
$\text{insertNodeFirst}(n)$	<i>insert a new node as the first child of node n</i>
$\text{insertNodeLast}(n)$	<i>insert a new node as the last child of node n</i>
$\text{moveNodeAbove}(n, m)$	<i>move node m above node n</i>
$\text{moveNodeLeft}(n, m)$	<i>move node m to the left of node n</i>
$\text{moveNodeRight}(n, m)$	<i>move node m to the right of node n</i>
$\text{prependNode}(n, m)$	<i>prepend node m to children of node n</i>
$\text{appendNode}(n, m)$	<i>append node m to children of node n</i>
$\mathbb{C}_{treeUp} ::= \text{deleteSubtree}(n)$	<i>delete subtree (subforest) beneath node n</i>
$\text{moveSubLeft}(n, m)$	<i>move children of node m to the left of node n</i>
$\text{moveSubRight}(n, m)$	<i>move children of node m to the right of node n</i>
$\text{prependSub}(n, m)$	<i>prepend children of node m to children of node n</i>
$\text{appendSub}(n, m)$	<i>append children of node m to children of node n</i>

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

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, node insertion commands that put a fresh node into the tree in relation to node n , 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

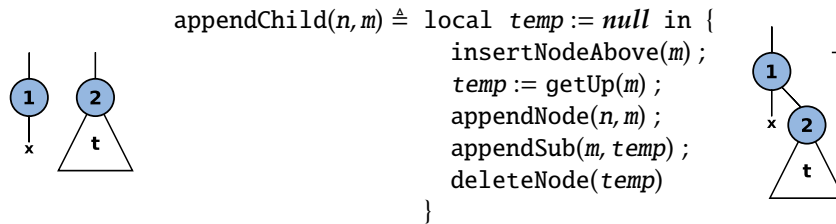
old parent. The tree update commands work on subtrees of an identified node. They consist of a delete command that removes an entire subtree from the tree, and subtree move commands that take a subtree out of the tree and put it in a new position.

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 [7], 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.

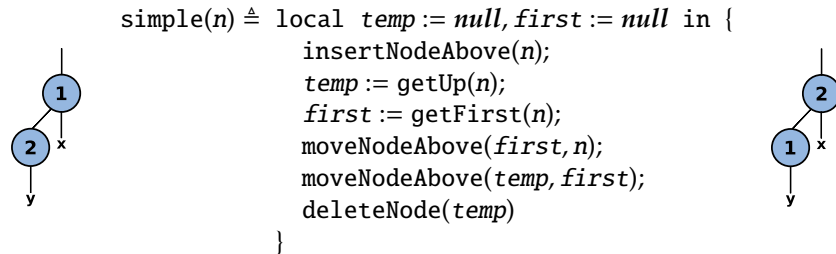
We now give example programs which will be used to illustrate our reasoning in Section ?? . In all of these examples, $\sigma(n) = 1$ and $\sigma(m) = 2$.

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

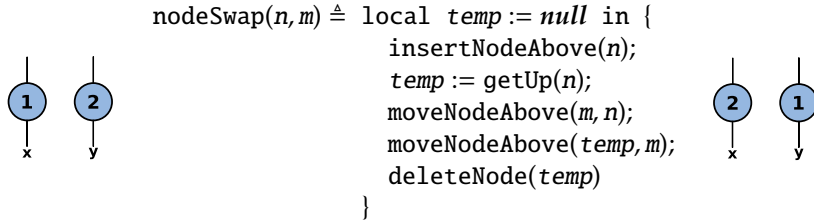


The diagrams illustrate the intuitive effect of the program on the working tree. For the program above not to fault, it requires the node $n = 1$ and the complete tree at node $m = 2$ (the left-hand diagram and the intuitive footprint). The complete tree at m ensures that m is not an ancestor of n . 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.

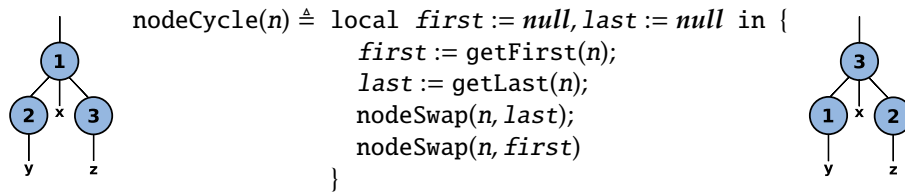
Example 2 (Simple Swap). Our node update commands enable us to define programs that act on arbitrary segments of the tree. For example, consider the program `simple(n)` which swaps a node n with its first child:



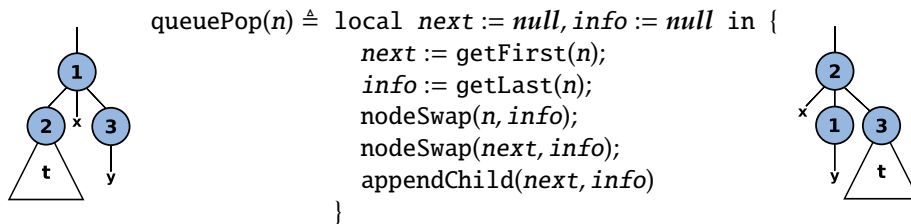
Example 3 (General Swap). The program `nodeSwap(n, m)` swaps the positions of arbitrary nodes n and m of a tree leaving their subtrees stationary:



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 :



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.



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 $s \in \mathbb{S}_{\text{Id}, X}$ are defined by:

$$\begin{aligned} \text{tree segment } s ::= & \quad \emptyset_{\mathbb{S}} \quad \text{empty tree segment} \\ & \quad x \leftarrow c \quad \text{tree context } c \text{ addressed by hole identifier } x \\ & \quad s + s \quad \text{disjoint union} \\ & \quad (x)(s) \quad \text{hiding, hole identifier } x \text{ bound in tree segment } s \end{aligned}$$

with the restriction that each hole identifier $x \in X$ 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, $n \in \text{Id}$, occurs at most once in s . The set $\text{free}(s)$ 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:

$$\begin{aligned} s + \emptyset_{\mathbb{S}} & \equiv s \\ s_1 + s_2 & \equiv s_2 + s_1 \\ s_1 + (s_2 + s_3) & \equiv (s_1 + s_2) + s_3 \\ (x)(\emptyset_{\mathbb{S}}) & \equiv \emptyset_{\mathbb{S}} \\ (x)(y)(s) & \equiv (y)(x)(s) \\ (x)(s) & \equiv (y)(s[y/x]) \quad \text{if } y \notin \text{fn}(s) \\ (x)(y \leftarrow c + s) & \equiv y \leftarrow c + (x)(s) \quad \text{if } x \neq y \text{ and } x \notin \text{fn}(c) \\ (x)(y \leftarrow c_1 + x \leftarrow c_2) & \equiv y \leftarrow c_1 \circ_x c_2 \quad \text{if } x \in \text{fn}(c_1) \end{aligned}$$

Most of the axioms involving hiding follow from analogous axioms for the restriction operator of the π -calculus [11]. 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 [10]. 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 $z \leftarrow 1[x \otimes 2[t]] + y \leftarrow \emptyset_C$: before update the segment $y \leftarrow 2[t]$ states that a tree is at address y ; after update $y \leftarrow \emptyset_C$ states that the empty tree is at address y . 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.

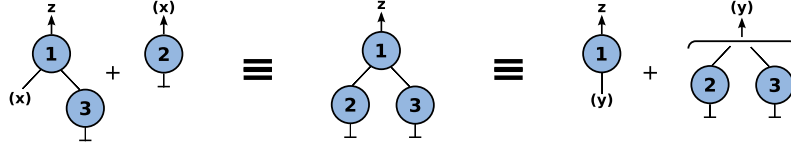


Fig. 3. Equivalent Tree Segments: (x) and (y) denote hidden hole labels and addresses.

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 \mathbf{C}, σ, s , terminal states σ, s , and faults, where \mathbf{C} is a command, σ is a variable store and s is a tree segment. The set of variables of a command \mathbf{C} is denoted $\text{free}(\mathbf{C})$ and is contained within the domain of σ , denoted $\text{dom}(\sigma)$. 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 [9], 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 $n' := \text{getRight}(n)$. 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 $e \in E$ is a set of functions

$$e : (LVar_C \rightarrow C_{\text{Id}, X}) \times (LVar_S \rightarrow S_{\text{Id}, X}) \times (LVar_X \rightarrow X)$$

$$\begin{array}{c}
\frac{}{\text{skip}, \sigma, s \rightsquigarrow \sigma, s} \quad \frac{\llbracket M \rrbracket \sigma = n}{n := N, \sigma, s \rightsquigarrow \sigma[n \mapsto n], s} \\
\frac{v \in \text{dom}(\sigma) \quad \sigma(v) = n_0 \quad \mathbb{C}, \sigma[v \mapsto n], s \rightsquigarrow^* \sigma', s'}{\text{local } v := n \text{ in } \mathbb{C}, \sigma, s \rightsquigarrow \sigma'[v \mapsto n_0], s'} \quad \frac{v \notin \text{dom}(\sigma) \quad \mathbb{C}, \sigma[v \mapsto n], s \rightsquigarrow^* \sigma', s'}{\text{local } v := n \text{ in } \mathbb{C}, \sigma, s \rightsquigarrow \sigma' \setminus v, s'} \\
\frac{\mathbb{C}_1, \sigma, s \rightsquigarrow \mathbb{C}'_1, \sigma', s'}{\mathbb{C}_1; \mathbb{C}_2, \sigma, s \rightsquigarrow \mathbb{C}'_1; \mathbb{C}_2, \sigma', s'} \quad \frac{\mathbb{C}_1, \sigma, s \rightsquigarrow \sigma', s'}{\mathbb{C}_1; \mathbb{C}_2, \sigma, s \rightsquigarrow \mathbb{C}_2, \sigma', s'} \\
\frac{\llbracket B \rrbracket \sigma = \text{true}}{\text{if } B \text{ then } \mathbb{C}_1 \text{ else } \mathbb{C}_2, \sigma, s \rightsquigarrow \mathbb{C}_1, \sigma, s} \quad \frac{\llbracket B \rrbracket \sigma = \text{false}}{\text{if } B \text{ then } \mathbb{C}_1 \text{ else } \mathbb{C}_2, \sigma, s \rightsquigarrow \mathbb{C}_2, \sigma, s} \\
\frac{\llbracket B \rrbracket \sigma = \text{true}}{\text{while } B \text{ do } \mathbb{C}, \sigma, s \rightsquigarrow \mathbb{C}; \text{while } B \text{ do } \mathbb{C}, \sigma, s} \quad \frac{\llbracket B \rrbracket \sigma = \text{false}}{\text{while } B \text{ do } \mathbb{C}, \sigma, s \rightsquigarrow \sigma, s} \\
\frac{\sigma(n) = n \quad s \equiv (w, x, y, z)(s' + x \leftarrow m[y \otimes n[w] \otimes z])}{n' := \text{getUp}(n), \sigma, s \rightsquigarrow \sigma[n' \mapsto m], s} \\
\frac{\sigma(n) = n \quad s \equiv (x, y, z)(s' + x \leftarrow m[y] \otimes n[z])}{n' := \text{getLeft}(n), \sigma, s \rightsquigarrow \sigma[n' \mapsto m], s} \quad \frac{\sigma(n) = n \quad s \equiv (x, y, z)(s' + x \leftarrow m[n[z] \otimes y])}{n' := \text{getLeft}(n), \sigma, s \rightsquigarrow \sigma[n' \mapsto \text{null}], s} \\
\frac{\sigma(n) = n \quad s \equiv (x, y, z)(s' + x \leftarrow n[y] \otimes m[z])}{n' := \text{getRight}(n), \sigma, s \rightsquigarrow \sigma[n' \mapsto m], s} \quad \frac{\sigma(n) = n \quad s \equiv (x, y, z)(s' + x \leftarrow m[z \otimes n[y]])}{n' := \text{getRight}(n), \sigma, s \rightsquigarrow \sigma[n' \mapsto \text{null}], s} \\
\frac{\sigma(n) = n \quad s \equiv (x, y, z)(s' + x \leftarrow n[m[y] \otimes z])}{n' := \text{getFirst}(n), \sigma, s \rightsquigarrow \sigma[n' \mapsto m], s} \quad \frac{\sigma(n) = n \quad s \equiv (x, y, z)(s' + x \leftarrow n[\otimes_C])}{n' := \text{getFirst}(n), \sigma, s \rightsquigarrow \sigma[n' \mapsto \text{null}], s} \\
\frac{\sigma(n) = n \quad s \equiv (x, y, z)(s' + x \leftarrow n[y \otimes m[z]])}{n' := \text{getLast}(n), \sigma, s \rightsquigarrow \sigma[n' \mapsto m], s} \quad \frac{\sigma(n) = n \quad s \equiv (x, y, z)(s' + x \leftarrow n[\otimes_C])}{n' := \text{getLast}(n), \sigma, s \rightsquigarrow \sigma[n' \mapsto \text{null}], s} \\
\frac{\sigma(n) = n \quad s \equiv (x, y)(s'' + x \leftarrow n[y]) \quad s' \equiv (x, y)(s'' + x \leftarrow y)}{\text{deleteNode}(n), \sigma, s \rightsquigarrow \sigma, s'} \quad \frac{\sigma(n) = n \quad s \equiv (x)(s'' + x \leftarrow n[t]) \quad s' \equiv (x)(s'' + x \leftarrow n[\otimes_C])}{\text{deleteSubtree}(n), \sigma, s \rightsquigarrow \sigma, s'} \quad \frac{\sigma(n) = n \quad s \equiv (x, y)(s'' + x \leftarrow n[y]) \quad n' \text{ fresh id } \quad s' \equiv (x, y)(s'' + x \leftarrow n'[n[y]])}{\text{insertNodeAbove}(n), \sigma, s \rightsquigarrow \sigma, s'} \\
\frac{\sigma(n) = n \quad s \equiv (x, y)(s'' + x \leftarrow n[y]) \quad n' \text{ fresh id } \quad s' \equiv (x, y)(s'' + x \leftarrow n[y] \otimes n'[\otimes_C])}{\text{insertNodeRight}(n), \sigma, s \rightsquigarrow \sigma, s'} \quad \frac{\sigma(n) = n \quad s \equiv (x, y)(s'' + x \leftarrow n[y]) \quad n' \text{ fresh id } \quad s' \equiv (x, y)(s'' + x \leftarrow n'[\otimes_C] \otimes n[y])}{\text{insertNodeLeft}(n), \sigma, s \rightsquigarrow \sigma, s'} \\
\frac{\sigma(n) = n \quad s \equiv (x, y)(s'' + x \leftarrow n[y]) \quad n' \text{ fresh id } \quad s' \equiv (x, y)(s'' + x \leftarrow n[n'[\otimes_C] \otimes y])}{\text{insertNodeFirst}(n), \sigma, s \rightsquigarrow \sigma, s'} \quad \frac{\sigma(n) = n \quad s \equiv (x, y)(s'' + x \leftarrow n[y]) \quad n' \text{ fresh id } \quad s' \equiv (x, y)(s'' + x \leftarrow n[y \otimes n'[\otimes_C]])}{\text{insertNodeLast}(n), \sigma, s \rightsquigarrow \sigma, s'} \\
\frac{\sigma(n) = n \quad s \equiv (w, x, y, z)(s'' + x \leftarrow n[z] + y \leftarrow m[w]) \quad \sigma(m) = m \quad s' \equiv (w, x, y, z)(s'' + x \leftarrow m[n[z]] + y \leftarrow w)}{\text{moveNodeAbove}(n, m), \sigma, s \rightsquigarrow \sigma, s'} \\
\frac{\sigma(n) = n \quad s \equiv (w, x, y, z)(s'' + x \leftarrow n[y] + z \leftarrow m[w]) \quad \sigma(m) = m \quad s' \equiv (w, x, y, z)(s'' + x \leftarrow m \otimes n[y] + z \leftarrow w)}{\text{moveNodeLeft}(n, m), \sigma, s \rightsquigarrow \sigma, s'} \quad \frac{\sigma(n) = n \quad s \equiv (x, y, z)(s'' + x \leftarrow n[y] + z \leftarrow m[t]) \quad \sigma(m) = m \quad s' \equiv (x, y, z)(s'' + x \leftarrow t \otimes n[y] + z \leftarrow m)}{\text{moveSubLeft}(n, m), \sigma, s \rightsquigarrow \sigma, s'} \\
\frac{\sigma(n) = n \quad s \equiv (w, x, y, z)(s'' + x \leftarrow n[y] + z \leftarrow m[w]) \quad \sigma(m) = m \quad s' \equiv (w, x, y, z)(s'' + x \leftarrow n[y] \otimes m + z \leftarrow w)}{\text{moveNodeRight}(n, m), \sigma, s \rightsquigarrow \sigma, s'} \quad \frac{\sigma(n) = n \quad s \equiv (x, y, z)(s'' + x \leftarrow n[y] + z \leftarrow m[t]) \quad \sigma(m) = m \quad s' \equiv (x, y, z)(s'' + x \leftarrow n[y] \otimes t + z \leftarrow m)}{\text{moveSubRight}(n, m), \sigma, s \rightsquigarrow \sigma, s'} \\
\frac{\sigma(n) = n \quad s \equiv (w, x, y, z)(s'' + x \leftarrow n[y] + z \leftarrow m[w]) \quad \sigma(m) = m \quad s' \equiv (w, x, y, z)(s'' + x \leftarrow n[m \otimes y] + z \leftarrow w)}{\text{prependNode}(n, m), \sigma, s \rightsquigarrow \sigma, s'} \quad \frac{\sigma(n) = n \quad s \equiv (x, y, z)(s'' + x \leftarrow n[y] + z \leftarrow m[t]) \quad \sigma(m) = m \quad s' \equiv (x, y, z)(s'' + x \leftarrow n[t \otimes y] + z \leftarrow m)}{\text{prependSub}(n, m), \sigma, s \rightsquigarrow \sigma, s'} \\
\frac{\sigma(n) = n \quad s \equiv (w, x, y, z)(s'' + x \leftarrow n[z] + y \leftarrow m[w]) \quad \sigma(m) = m \quad s' \equiv (w, x, y, z)(s'' + x \leftarrow n[z \otimes m] + y \leftarrow w)}{\text{appendNode}(n, m), \sigma, s \rightsquigarrow \sigma, s'} \quad \frac{\sigma(n) = n \quad s \equiv (x, y, z)(s'' + x \leftarrow n[z] + y \leftarrow m[t]) \quad \sigma(m) = m \quad s' \equiv (x, y, z)(s'' + x \leftarrow n[z \otimes t] + y \leftarrow m)}{\text{appendSub}(n, m), \sigma, s \rightsquigarrow \sigma, s'} \\
\frac{}{\text{otherwise}} \\
\frac{}{\mathbb{C}, \sigma, s \rightsquigarrow \text{fault}}
\end{array}$$

Fig. 4. Operational Semantics for the Tree Update Language

mapping tree context variables $LVar_C = \{c, \dots\}$ to tree contexts, tree segment variables $LVar_S = \{s, \dots\}$ to tree segments, and hole identifier variables $LVar_X = \{\alpha, \beta, \gamma, \delta, \dots\}$ to hole identifiers. We write $e[\mathbf{lvar} \mapsto \mathbf{val}]$ for e overwritten with $e(\mathbf{lvar}) = \mathbf{val}$.

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 [8]. 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:*

$P_S, Q_S ::= P_S \Rightarrow P_S \mid \mathbf{false}_S$	$P_T, Q_T ::= P_T \Rightarrow P_T \mid \mathbf{false}_T$	<i>Classical</i>
$\mid \emptyset_S \mid P_S * P_S \mid P_S \multimap P_S \mid \alpha \textcircled{R} P_S \mid \alpha \textcircled{-R} P_S$		<i>Structural</i>
$\mid \exists \mathbf{var}. P_S \mid \exists \mathbf{lvar}. P_S \mid \forall \alpha. P_S$	$\mid \exists \mathbf{var}. P_T \mid \exists \mathbf{lvar}. P_T$	<i>Quantifiers</i>
$\mid \alpha \leftarrow P_T$	$\mid \emptyset_T \mid \alpha \mid n[P_T] \mid P_T \otimes P_T$	<i>Specific</i>
$\mid s \mid B$	$\mid c \mid B \mid @_T \alpha$	<i>Expression</i>

Let $free(P_S)$ and $free(P_T)$ denote the appropriate sets of free variables: α is free in $\alpha \textcircled{R} P_S$, $\alpha \textcircled{-R} P_S$, $\alpha \leftarrow P_T$ and $@_T \alpha$, bound in $\forall \alpha. P_S$. \mathbf{var} is a location name variable and \mathbf{lvar} is a logical variable. The binding precedence, strongest first, is: \otimes , \leftarrow , $*$, \textcircled{R} , \multimap , $\textcircled{-R}$, \Rightarrow .

The separating connective $*$, its unit \emptyset_S and its right adjoint (the separating magic wand) \multimap , 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 \emptyset_S describes the empty segment; and formula $P_S \multimap Q_S$ 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 α and tree context satisfying tree formula P_T . The tree formulae follow the style of reasoning given by Ambient Logic. For the specific tree formulae, we have \emptyset_T specifying the empty tree context, α specifying a hole label given by the value of variable α , $n[P_T]$ 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 α free; the analogous formula for tree segments is derivable.

The other structural connectives are the revelation connective, \textcircled{R} , and its right adjoint, the revelation magic wand $\textcircled{-R}$. As far as we are aware, these connectives have not been used in the local reasoning setting before. Together with the freshness quantifier $\forall \alpha$, they have been used in the Ambient Logic [4],

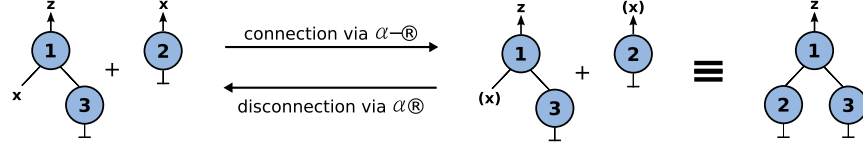


Fig. 5. Connection and Disconnection of Tree Segments.

following the work of Pitts and Gabbay [5]. The freshness quantifier enables us to pick a completely new hole identifier. The formula $\alpha\text{-}\mathbb{R}P_S$ describes a segment with a top-level hiding binder given by the value of α such that, after the hiding is removed, the remaining segment satisfies P_S . Consider the tree segment $z\leftarrow 1[2\otimes 3] \equiv (x)(z\leftarrow 1[x\otimes 3] + x\leftarrow 2)$. It satisfies the formula $\alpha\text{-}\mathbb{R}(\beta\leftarrow 1[\alpha\otimes \text{true}_T] + \alpha\leftarrow 2)$, when $\alpha = x, \beta = z$. The revelation connective $\alpha\text{-}\mathbb{R}$ strips off the hiding binder, disconnecting the tree into the fragments $z\leftarrow 1[x\otimes 3] + x\leftarrow 2$ as illustrated in Fig. 5. By contrast, the formula $\alpha\text{-}\mathbb{R}P_S$ describes a segment which satisfies P_S if it is extended with a hiding binder over the hole identifier stored in variable α . For example, the tree segment $z\leftarrow 1[x\otimes 3] + x\leftarrow 2$ satisfies the formula $\alpha\text{-}\mathbb{R}(\beta\leftarrow 1[2\otimes 3])$ when $\alpha = x$. The revelation magic wand $\alpha\text{-}\mathbb{R}$ adds the binder (x) to the segment to obtain the tree segment $(x)(z\leftarrow 1[x\otimes 3] + x\leftarrow 2) \equiv z\leftarrow 1[2\otimes 3]$, 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 σ , 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 where $(s)\downarrow$ denotes that s is well formed and $x\#\mathbf{s}$ denotes that x is fresh with respect to \mathbf{s} .

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

$$\begin{array}{ll}
\text{tree}(P_T) \triangleq P_T \wedge \neg \exists \alpha. @_T \alpha & @_S \alpha \triangleq \mathbf{H}\beta. (\text{true}_S * (\alpha\leftarrow \beta \vee \beta\leftarrow \alpha)) \\
n \triangleq n[\emptyset_T] & \mathbf{H}\alpha. P_S \triangleq \mathbf{V}\alpha. \alpha\text{-}\mathbb{R}P_S \\
\circ[P_T] \triangleq \exists m. m[P_T] & \mathbf{V}\alpha, \beta. P_S \triangleq \mathbf{V}\alpha. \mathbf{V}\beta. P_S \\
\Diamond P_S \triangleq \text{true}_S * P_S & \alpha, \beta\text{-}\mathbb{R}P_S \triangleq \alpha\text{-}\mathbb{R}\beta\text{-}\mathbb{R}P_S \\
\Diamond P_T \triangleq \mathbf{H}\alpha. (\text{true}_S * \alpha\leftarrow P_T) & \alpha, \beta\text{-}\mathbb{R}P_S \triangleq \alpha\text{-}\mathbb{R}\beta\text{-}\mathbb{R}P_S
\end{array}$$

Formula $\text{tree}(P_T)$ 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. Formulae $\Diamond P_S$ and $\Diamond P_T$ allow us to express that, somewhere in the tree segment there is a segment satisfying P_S or P_T respectively. Formula $@_S \alpha$ describes a tree segment that contains α free. Formula $\mathbf{H}\alpha. P_S$ provides the standard hiding quantification [4] allowing us to quantify over hidden (restricted) labels. Finally we give a number of shorthands for simplifying our notation for freshness, revelation and the revelation right adjoint.

$e, \sigma, s \models_S P_S \Rightarrow Q_S \Leftrightarrow e, \sigma, s \models_S P_S$ $\Leftrightarrow e, \sigma, s \models_S Q_S$	$e, \sigma, c \models_T P_T \Rightarrow Q_T \Leftrightarrow e, \sigma, c \models_T P_T$ $\Leftrightarrow e, \sigma, c \models_T Q_T$
$e, \sigma, s \models_S \text{false}_S \quad \text{never}$	$e, \sigma, c \models_T \text{false}_T \quad \text{never}$
$e, \sigma, s \models_S \emptyset_S \Leftrightarrow s \equiv \emptyset_S$	$e, \sigma, c \models_T \emptyset_T \Leftrightarrow c \equiv \emptyset_C$
$e, \sigma, s \models_S \alpha \leftarrow P_T \Leftrightarrow \exists c, x. e(\alpha) = x$ $\wedge s \equiv x \leftarrow c$	$e, \sigma, c \models_T \alpha \Leftrightarrow c \equiv e(\alpha)$
$e, \sigma, s \models_S P_S * Q_S \Leftrightarrow \exists s_1, s_2. s \equiv s_1 + s_2$ $\wedge e, \sigma, s_1 \models_S P_S$ $\wedge e, \sigma, s_2 \models_S Q_S$	$e, \sigma, c \models_T n[P_T] \Leftrightarrow \exists c_1. c \equiv \sigma(n)[c_1]$ $\wedge e, \sigma, c_1 \models_T P_T$
$e, \sigma, s \models_S \alpha @ P_S \Leftrightarrow \exists x, s'. e(\alpha) = x$ $\wedge s \equiv (x)(s')$ $\wedge e, \sigma, s' \models_S P_S$	$e, \sigma, c \models_T P_T \otimes Q_T \Leftrightarrow \exists c_1, c_2. c \equiv c_1 \otimes c_2$ $\wedge e, \sigma, c_1 \models_T P_T$ $\wedge e, \sigma, c_2 \models_T Q_T$
$e, \sigma, s \models_S P_S \multimap Q_S \Leftrightarrow \forall s'. e, \sigma, s' \models_S P_S$ $\wedge (s + s') \downarrow$ $\Rightarrow e, \sigma, s + s' \models_S Q_S$	$e, \sigma, c \models_T c \Leftrightarrow c \equiv e(c)$
$e, \sigma, s \models_S \alpha \text{-} @ P_S \Leftrightarrow \forall x, s'. e(\alpha) = x$ $\wedge s' \equiv (x)(s)$ $\wedge e, \sigma, s' \models_S P_S$	$e, \sigma, c \models_T B \Leftrightarrow \llbracket B \rrbracket \sigma = \text{true}$
$e, \sigma, s \models_S s \Leftrightarrow s \equiv e(s)$	$e, \sigma, c \models_T @ \alpha \Leftrightarrow e(\alpha) \in \text{free}(c)$
$e, \sigma, s \models_S B \Leftrightarrow \llbracket B \rrbracket \sigma = \text{true}$	$e, \sigma, c \models_T \exists \text{var}. P_T \Leftrightarrow \exists v. e, \sigma[\text{var} \mapsto v], c \models_T P_T$
$e, \sigma, s \models_S \exists \text{var}. P_S \Leftrightarrow \exists v. e, \sigma[\text{var} \mapsto v], s \models_S P_S$	$e, \sigma, c \models_T \exists ! \text{var}. P_T \Leftrightarrow \exists v. e[! \text{var} \mapsto v], \sigma, c \models_T P_T$
$e, \sigma, s \models_S \exists ! \text{var}. P_S \Leftrightarrow \exists v. e[! \text{var} \mapsto v], \sigma, s \models_S P_S$	
$e, \sigma, s \models_S \mathcal{I} \alpha. P_S \Leftrightarrow \exists x. x \# e, s \wedge e[\alpha \mapsto x], \sigma, s \models_S P_S$	

Fig. 6. Satisfaction Relations of Segment Logic.

Example 6 (Segment Logic Examples).

- (a) The segment formula $\alpha \leftarrow n[\gamma] * \beta \leftarrow m[\delta]$ describes a tree segment consisting of a node n with address α and context hole γ , and node m with address β and context hole δ . The variables n and m cannot denote the same node identifier; similarly, α, β cannot denote the same hole identifier; neither can γ, δ .
- (b) The segment formula $\alpha \leftarrow n[\gamma] * \beta \leftarrow m[\text{tree}(c)]$ describes a tree segment consisting of a single node n at address α and a complete tree (a tree with no holes) with top node m at address β . This formula is the safety precondition for the small axiom of the `appendSub(n, m)` command. In particular, the formula states that m cannot be an ancestor of n , as n is disjoint from the tree c .
- (c) The segment formula $\mathcal{H} \alpha, \beta. (\delta \leftarrow r[\alpha \otimes \beta] * \alpha \leftarrow n[\gamma] * \beta \leftarrow m[\text{tree}(c)])$ describes a tree segment consisting of a node r at address δ whose children are given by the holes α and β , and a tree segment satisfying the formula in Example (b). The labels α, β are under the hiding quantification, and hence denote fresh, unequal, identifiers. This formula is equivalent to $\delta \leftarrow r[n[\gamma] \otimes m[\text{tree}(c)]]$, which states that there is a node r at address δ whose children are n and 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(n)` command is $\exists c. \mathcal{I} \alpha. \alpha @ ((\alpha \leftarrow n[\emptyset_T] \multimap (\alpha \text{-} @ P_S)) * \alpha \leftarrow n[\text{tree}(c)])$. This formula describes a tree segment which can be separated into a complete tree, with top node n at a fresh address x denoted by α , and a segment s satisfying $(\alpha \leftarrow n[\emptyset_T]) \multimap (\alpha \text{-} @ P_S)$. The segment s , when extended to $(x)(x \leftarrow n[\emptyset_T] + s)$, satisfies P_S .

$\{\emptyset_S\}$	skip	$\{\emptyset_S\}$
$\{\emptyset_S \wedge (n = n_0)\}$	$n := N$	$\{\emptyset_S \wedge (n = N_{[n_0/n]})\}$
$\{\alpha \leftarrow m[\beta \otimes n[\delta] \otimes \gamma] \wedge (n' = n_0)\}$	$n' := \text{getUp}(n)$	$\{(\alpha \leftarrow m[\beta \otimes n[\delta] \otimes \gamma])_{[n_0/n']} \wedge (n' = m)\}$
$\{\alpha \leftarrow m[\beta] \otimes n[\delta] \wedge (n' = n_0)\}$	$n' := \text{getLeft}(n)$	$\{(\alpha \leftarrow m[\beta] \otimes n[\delta])_{[n_0/n']} \wedge (n' = m)\}$
$\{\alpha \leftarrow m[n[\delta] \otimes \beta] \wedge (n' = n_0)\}$	$n' := \text{getLeft}(n)$	$\{(\alpha \leftarrow m[n[\delta] \otimes \beta])_{[n_0/n']} \wedge (n' = \text{null})\}$
$\{\alpha \leftarrow n[\delta] \otimes m[\beta] \wedge (n' = n_0)\}$	$n' := \text{getRight}(n)$	$\{(\alpha \leftarrow n[\delta] \otimes m[\beta])_{[n_0/n']} \wedge (n' = m)\}$
$\{\alpha \leftarrow m[\beta \otimes n[\delta]] \wedge (n' = n_0)\}$	$n' := \text{getRight}(n)$	$\{(\alpha \leftarrow m[\beta \otimes n[\delta]])_{[n_0/n']} \wedge (n' = \text{null})\}$
$\{\alpha \leftarrow n[m[\beta] \otimes \delta] \wedge (n' = n_0)\}$	$n' := \text{getFirst}(n)$	$\{(\alpha \leftarrow n[m[\beta] \otimes \delta])_{[n_0/n']} \wedge (n' = m)\}$
$\{\alpha \leftarrow n[\emptyset_T] \wedge (n' = n_0)\}$	$n' := \text{getFirst}(n)$	$\{(\alpha \leftarrow n[\emptyset_T])_{[n_0/n']} \wedge (n' = \text{null})\}$
$\{\alpha \leftarrow n[\delta \otimes m[\beta]] \wedge (n' = n_0)\}$	$n' := \text{getLast}(n)$	$\{(\alpha \leftarrow n[\delta \otimes m[\beta]])_{[n_0/n']} \wedge (n' = m)\}$
$\{\alpha \leftarrow n[\emptyset_T] \wedge (n' = n_0)\}$	$n' := \text{getLast}(n)$	$\{(\alpha \leftarrow n[\emptyset_T])_{[n_0/n']} \wedge (n' = \text{null})\}$
$\{\alpha \leftarrow n[\beta]\}$	deleteNode(n)	$\{\alpha \leftarrow \beta\}$
$\{\alpha \leftarrow n[\text{tree}(c)]\}$	deleteSubtree(n)	$\{\alpha \leftarrow n[\emptyset_T]\}$
$\{\alpha \leftarrow n[\beta]\}$	insertNodeAbove(n)	$\{\alpha \leftarrow \circ[n[\beta]]\}$
$\{\alpha \leftarrow n[\beta]\}$	insertNodeLeft(n)	$\{\alpha \leftarrow \circ[\emptyset_T] \otimes n[\beta]\}$
$\{\alpha \leftarrow n[\beta]\}$	insertNodeRight(n)	$\{\alpha \leftarrow n[\beta] \otimes \circ[\emptyset_T]\}$
$\{\alpha \leftarrow n[\beta]\}$	insertNodeFirst(n)	$\{\alpha \leftarrow n[\circ[\emptyset_T] \otimes \beta]\}$
$\{\alpha \leftarrow n[\beta]\}$	insertNodeLast(n)	$\{\alpha \leftarrow n[\beta \otimes \circ[\emptyset_T]]\}$
$\{\alpha \leftarrow n[\gamma] * \beta \leftarrow m[\delta]\}$	moveNodeAbove(n, m)	$\{\alpha \leftarrow m[n[\gamma]] * \beta \leftarrow \delta\}$
$\{\alpha \leftarrow n[\gamma] * \beta \leftarrow m[\delta]\}$	moveNodeLeft(n, m)	$\{\alpha \leftarrow m[\emptyset_T] \otimes n[\gamma] * \beta \leftarrow \delta\}$
$\{\alpha \leftarrow n[\gamma] * \beta \leftarrow m[\delta]\}$	moveNodeRight(n, m)	$\{\alpha \leftarrow n[\gamma] \otimes m[\emptyset_T] * \beta \leftarrow \delta\}$
$\{\alpha \leftarrow n[\gamma] * \beta \leftarrow m[\delta]\}$	prependNode(n, m)	$\{\alpha \leftarrow n[m[\emptyset_T] \otimes \gamma] * \beta \leftarrow \delta\}$
$\{\alpha \leftarrow n[\gamma] * \beta \leftarrow m[\delta]\}$	appendNode(n, m)	$\{\alpha \leftarrow n[\gamma \otimes m[\emptyset_T]] * \beta \leftarrow \delta\}$
$\{\alpha \leftarrow n[\gamma] * \beta \leftarrow m[\text{tree}(c)]\}$	moveSubLeft(n, m)	$\{\alpha \leftarrow \text{tree}(c) \otimes n[\gamma] * \beta \leftarrow m[\emptyset_T]\}$
$\{\alpha \leftarrow n[\gamma] * \beta \leftarrow m[\text{tree}(c)]\}$	moveSubRight(n, m)	$\{\alpha \leftarrow n[\gamma] \otimes \text{tree}(c) * \beta \leftarrow m[\emptyset_T]\}$
$\{\alpha \leftarrow n[\gamma] * \beta \leftarrow m[\text{tree}(c)]\}$	prependSub(n, m)	$\{\alpha \leftarrow n[\text{tree}(c) \otimes \gamma] * \beta \leftarrow m[\emptyset_T]\}$
$\{\alpha \leftarrow n[\gamma] * \beta \leftarrow m[\text{tree}(c)]\}$	appendSub(n, m)	$\{\alpha \leftarrow n[\gamma \otimes \text{tree}(c)] * \beta \leftarrow m[\emptyset_T]\}$

n' and n_0 are distinct.

Fig. 7. Small Axioms for the Tree Basic Update Commands.

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 [18]. Informally, $\{P_S\} \mathbb{C} \{Q_S\}$ means that, when P_S holds for a tree segment s , then command \mathbb{C} does not fault when run on s and the result, if \mathbb{C} terminates, satisfies Q_S .

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

$$\{P_S\} \mathbb{C} \{Q_S\} \Leftrightarrow \forall e, \sigma, s. \text{free}(\mathbb{C}) \subseteq \text{dom}(\sigma) \wedge \text{free}(P_S) \cup \text{free}(Q_S) \subseteq \text{dom}(\sigma) \cup \text{dom}(e) \\ \wedge e, \sigma, s \models_S P_S \Rightarrow \mathbb{C}, \sigma, s \not\rightsquigarrow \text{fault} \wedge \forall \sigma', s'. \mathbb{C}, \sigma, s \rightsquigarrow \sigma', s' \Rightarrow e, \sigma', s' \models_S Q_S$$

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 [14]. 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:*

$$\begin{array}{l}
\text{CONSEQUENCE:} \\
\frac{P_S \Rightarrow P'_S \quad \{P'_S\} \mathbf{C} \{Q'_S\} \quad Q'_S \Rightarrow Q_S}{\{P_S\} \mathbf{C} \{Q_S\}} \\
\\
\text{DISJUNCTION:} \\
\frac{\{P_S\} \mathbf{C} \{Q_S\} \quad \{P'_S\} \mathbf{C} \{Q'_S\}}{\{P_S \vee P'_S\} \mathbf{C} \{Q_S \vee Q'_S\}} \\
\\
\text{SEPARATION FRAME:} \\
\frac{\{P_S\} \mathbf{C} \{Q_S\}}{\{P_S * R_S\} \mathbf{C} \{Q_S * R_S\}} \quad \text{mod}(\mathbf{C}) \cap \text{free}(R_S) = \{\} \\
\\
\text{REVELATION FRAME:} \\
\frac{\{P_S\} \mathbf{C} \{Q_S\}}{\{\alpha @ P_S\} \mathbf{C} \{\alpha @ Q_S\}} \\
\\
\text{AUXILIARY VARIABLE ELIMINATION:} \\
\frac{\{P_S\} \mathbf{C} \{Q_S\}}{\{\exists n. P_S\} \mathbf{C} \{\exists n. Q_S\}} \quad n \notin \text{free}(\mathbf{C}) \\
\\
\text{FRESH LABEL ELIMINATION:} \\
\frac{\{P_S\} \mathbf{C} \{Q_S\}}{\{\forall \alpha. P_S\} \mathbf{C} \{\forall \alpha. Q_S\}} \\
\\
\text{SEQUENCING:} \\
\frac{\{P_S\} \mathbf{C} \{R_S\} \quad \{R_S\} \mathbf{C}' \{Q_S\}}{\{P_S\} \mathbf{C}; \mathbf{C}' \{Q_S\}} \\
\\
\text{IF-THEN-ELSE:} \\
\frac{\{B \wedge P_S\} \mathbf{C} \{Q_S\} \quad \{\neg B \wedge P_S\} \mathbf{C}' \{Q_S\}}{\{P_S\} \text{ if } B \text{ then } \mathbf{C} \text{ else } \mathbf{C}' \{Q_S\}} \\
\\
\text{WHILE-DO:} \\
\frac{\{B \wedge P_S\} \mathbf{C} \{P_S\}}{\{P_S\} \text{ while } B \text{ do } \mathbf{C} \{-BP_S\}} \\
\\
\text{LOCAL VARIABLE}_1: \\
\frac{\{P_S \wedge (v = n_0)\} \mathbf{C} \{Q_S\}}{\{P_S\} \text{ local } v := n_0 \text{ in } \mathbf{C} \{Q_S\}} \quad v \notin \text{free}(n_0, P_S, Q_S) \\
\\
\text{LOCAL VARIABLE}_2: \\
\frac{\{P_S\} \text{ local } v' := n_0 \text{ in } \mathbf{C}[v'/v] \{Q_S\}}{\{P_S\} \text{ local } v := n_0 \text{ in } \mathbf{C} \{Q_S\}} \quad v' \notin \text{free}(\mathbf{C})
\end{array}$$

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

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

	$\{P_S\}$	skip	$\{P_S\}$
	$\{\exists n_0. (n = n_0) \wedge P_S[n/n]\}$	$n := N$	$\{P_S\}$
$\{\exists m, n_0. H\alpha, \beta, \gamma, \delta. \diamond \alpha \leftarrow m[\beta \otimes n[\delta] \otimes \gamma] \wedge (n' = n_0) \wedge (\alpha, \beta, \gamma, \delta \text{ -- } \textcircled{R} P_S[m/n'])\}$		$n' := \text{getUp}(n)$	$\{P_S\}$
$\left\{ \begin{array}{l} \exists m, n_0. H\alpha, \beta, \delta. \diamond \alpha \leftarrow m[\beta] \otimes n[\delta] \wedge (n' = n_0) \wedge (\alpha, \beta, \delta \text{ -- } \textcircled{R} P_S[m/n']) \\ \vee \diamond \alpha \leftarrow m[n[\delta] \otimes \beta] \wedge (n' = n_0) \wedge (\alpha, \beta, \delta \text{ -- } \textcircled{R} P_S[mult/n']) \end{array} \right\}$		$n' := \text{getLeft}(n)$	$\{P_S\}$
$\left\{ \begin{array}{l} \exists m, n_0. H\alpha, \beta, \delta. \diamond \alpha \leftarrow n[\delta] \otimes m[\beta] \wedge (n' = n_0) \wedge (\alpha, \beta, \delta \text{ -- } \textcircled{R} P_S[m/n']) \\ \vee \diamond \alpha \leftarrow m[\beta \otimes n[\delta]] \wedge (n' = n_0) \wedge (\alpha, \beta, \delta \text{ -- } \textcircled{R} P_S[mult/n']) \end{array} \right\}$		$n' := \text{getRight}(n)$	$\{P_S\}$
$\left\{ \begin{array}{l} \exists m, n_0. H\alpha, \beta, \delta. \diamond \alpha \leftarrow n[m[\beta] \otimes \delta] \wedge (n' = n_0) \wedge (\alpha, \beta, \delta \text{ -- } \textcircled{R} P_S[m/n']) \\ \vee \diamond \alpha \leftarrow n[\emptyset_T] \wedge (n' = n_0) \wedge (\alpha \text{ -- } \textcircled{R} P_S[mult/n']) \end{array} \right\}$		$n' := \text{getFirst}(n)$	$\{P_S\}$
$\left\{ \begin{array}{l} \exists m, n_0. H\alpha, \beta, \delta. \diamond \alpha \leftarrow n[\delta \otimes m[\beta]] \wedge (n' = n_0) \wedge (\alpha, \beta, \delta \text{ -- } \textcircled{R} P_S[m/n']) \\ \vee \diamond \alpha \leftarrow n[\emptyset_T] \wedge (n' = n_0) \wedge (\alpha \text{ -- } \textcircled{R} P_S[mult/n']) \end{array} \right\}$		$n' := \text{getLast}(n)$	$\{P_S\}$
$\{H\alpha, \beta. ((\alpha \leftarrow \beta \text{ -- } (\alpha, \beta \text{ -- } \textcircled{R} P_S)) * \alpha \leftarrow n[\beta])\}$		deleteNode(n)	$\{P_S\}$
$\{\exists c. H\alpha. ((\alpha \leftarrow n[\emptyset_T] \text{ -- } (\alpha \text{ -- } \textcircled{R} P_S)) * \alpha \leftarrow n[\text{tree}(c)])\}$		deleteSubtree(n)	$\{P_S\}$
$\{H\alpha, \beta. ((\alpha \leftarrow n[\beta]) \text{ -- } (\alpha, \beta \text{ -- } \textcircled{R} P_S)) * \alpha \leftarrow n[\beta]\}$		insertNodeAbove(n)	$\{P_S\}$
$\{H\alpha, \beta. ((\alpha \leftarrow n[\emptyset_T] \otimes n[\beta]) \text{ -- } (\alpha, \beta \text{ -- } \textcircled{R} P_S)) * \alpha \leftarrow n[\beta]\}$		insertNodeLeft(n)	$\{P_S\}$
$\{H\alpha, \beta. ((\alpha \leftarrow n[\beta] \otimes n[\emptyset_T]) \text{ -- } (\alpha, \beta \text{ -- } \textcircled{R} P_S)) * \alpha \leftarrow n[\beta]\}$		insertNodeRight(n)	$\{P_S\}$
$\{H\alpha, \beta. ((\alpha \leftarrow n[\emptyset_T] \otimes \beta) \text{ -- } (\alpha, \beta \text{ -- } \textcircled{R} P_S)) * \alpha \leftarrow n[\beta]\}$		insertNodeFirst(n)	$\{P_S\}$
$\{H\alpha, \beta. ((\alpha \leftarrow n[\beta \otimes n[\emptyset_T]]) \text{ -- } (\alpha, \beta \text{ -- } \textcircled{R} P_S)) * \alpha \leftarrow n[\beta]\}$		insertNodeLast(n)	$\{P_S\}$
$\{H\alpha, \beta, \gamma, \delta. (((\alpha \leftarrow m[n[\gamma]]) * \beta \leftarrow \delta) \text{ -- } (\alpha, \beta, \gamma, \delta \text{ -- } \textcircled{R} P_S)) * (\alpha \leftarrow n[\gamma] * \beta \leftarrow m[\delta])\}$		moveNodeAbove(n, m)	$\{P_S\}$
$\{H\alpha, \beta, \gamma, \delta. (((\alpha \leftarrow m[\emptyset_T] \otimes n[\gamma]) * \beta \leftarrow \delta) \text{ -- } (\alpha, \beta, \gamma, \delta \text{ -- } \textcircled{R} P_S)) * (\alpha \leftarrow n[\gamma] * \beta \leftarrow m[\delta])\}$		moveNodeLeft(n, m)	$\{P_S\}$
$\{H\alpha, \beta, \gamma, \delta. (((\alpha \leftarrow n[\gamma] \otimes m[\emptyset_T]) * \beta \leftarrow \delta) \text{ -- } (\alpha, \beta, \gamma, \delta \text{ -- } \textcircled{R} P_S)) * (\alpha \leftarrow n[\gamma] * \beta \leftarrow m[\delta])\}$		moveNodeRight(n, m)	$\{P_S\}$
$\{H\alpha, \beta, \gamma, \delta. (((\alpha \leftarrow n[m[\emptyset_T] \otimes \gamma]) * \beta \leftarrow \delta) \text{ -- } (\alpha, \beta, \gamma, \delta \text{ -- } \textcircled{R} P_S)) * (\alpha \leftarrow n[\gamma] * \beta \leftarrow m[\delta])\}$		prependNode(n, m)	$\{P_S\}$
$\{H\alpha, \beta, \gamma, \delta. (((\alpha \leftarrow n[\gamma \otimes m[\emptyset_T]]) * \beta \leftarrow \delta) \text{ -- } (\alpha, \beta, \gamma, \delta \text{ -- } \textcircled{R} P_S)) * (\alpha \leftarrow n[\gamma] * \beta \leftarrow m[\delta])\}$		appendNode(n, m)	$\{P_S\}$
$\{\exists c. H\alpha, \beta, \gamma. (((\alpha \leftarrow \text{tree}(c) \otimes n[\gamma]) * \beta \leftarrow m[\emptyset_T]) \text{ -- } (\alpha, \beta, \gamma \text{ -- } \textcircled{R} P_S)) * (\alpha \leftarrow n[\gamma] * \beta \leftarrow m[\text{tree}(c)])\}$		moveSubLeft(n, m)	$\{P_S\}$
$\{\exists c. H\alpha, \beta, \gamma. (((\alpha \leftarrow n[\gamma] \otimes \text{tree}(c)) * \beta \leftarrow m[\emptyset_T]) \text{ -- } (\alpha, \beta, \gamma \text{ -- } \textcircled{R} P_S)) * (\alpha \leftarrow n[\gamma] * \beta \leftarrow m[\text{tree}(c)])\}$		moveSubRight(n, m)	$\{P_S\}$
$\{\exists c. H\alpha, \beta, \gamma. (((\alpha \leftarrow n[\text{tree}(c) \otimes \gamma]) * \beta \leftarrow m[\emptyset_T]) \text{ -- } (\alpha, \beta, \gamma \text{ -- } \textcircled{R} P_S)) * (\alpha \leftarrow n[\gamma] * \beta \leftarrow m[\text{tree}(c)])\}$		prependSub(n, m)	$\{P_S\}$
$\{\exists c. H\alpha, \beta, \gamma. (((\alpha \leftarrow n[\gamma \otimes \text{tree}(c)]) * \beta \leftarrow m[\emptyset_T]) \text{ -- } (\alpha, \beta, \gamma \text{ -- } \textcircled{R} P_S)) * (\alpha \leftarrow n[\gamma] * \beta \leftarrow m[\text{tree}(c)])\}$		appendSub(n, m)	$\{P_S\}$

Fig. 8. Weakest Preconditions for the Basic Tree Update Commands.

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:

$$\begin{aligned} & \{H\alpha, \beta. (\epsilon \leftarrow r[\alpha \otimes \beta] * \alpha \leftarrow n[\gamma] * \beta \leftarrow m[\text{tree}(c)])\} \\ & \quad \text{appendSub}(n, m) \\ & \{H\alpha, \beta. (\epsilon \leftarrow r[\alpha \otimes \beta] * \alpha \leftarrow n[\gamma \otimes \text{tree}(c)] * \beta \leftarrow m[\emptyset_T])\} \end{aligned}$$

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.

Theorem 1 (Soundness). *The small axioms and inference rules are sound.*

Proof. Soundness follows from existing soundness results of Context Logic and Separation Logic. The only inference rules new to this logic are The rules of Fresh Label Elimination and Revelation Frame. The soundness for Fresh Label elimination is sound by the same argument as for Auxillary Variable Elimination, with freshness in place of existence. The Revelation Frame rule is sound due to the fact that revealing a label in a formula does not affect the truth of that formula, thus validity is trivially preserved.

Theorem 2 (Completeness). *For straight line code the small axioms and inference rules are sound.*

Proof. The weakest preconditions of our tree update commands (given in Fig. 8.) are derivable from the small axioms (given in Fig. 7). The full derivations are given in Appendix C. It is simple to show that they are indeed the weakest preconditions.

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

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

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

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:

$$\begin{array}{l}
\{\alpha \leftarrow n[m[\text{tree}(c)] \otimes \gamma \otimes i[\beta]]\} \\
\text{queuePop}(n) \\
\{\alpha \leftarrow m[\gamma \otimes n[\beta] \otimes i[\text{tree}(c)]]\}
\end{array}
\quad
\begin{array}{l}
\{\alpha \leftarrow n[m[\text{tree}(c)] \otimes \gamma \otimes i[\beta]]\} \\
\text{local } \text{next} := \text{null}, \text{info} := \text{null} \text{ in } \{ \\
\{\alpha \leftarrow n[m[\text{tree}(c)] \otimes \gamma \otimes i[\beta]] \wedge (\text{next} := \text{null}) \wedge (\text{info} = \text{null})\} \\
\text{next} := \text{getFirst}(n); \\
\{\alpha \leftarrow n[\text{next}[\text{tree}(c)] \otimes \gamma \otimes i[\beta]] \wedge (\text{next} = m) \wedge (\text{info} = \text{null})\} \\
\text{info} := \text{getLast}(n); \\
\{\alpha \leftarrow n[\text{next}[\text{tree}(c)] \otimes \gamma \otimes \text{info}[\beta]] \wedge (\text{next} = m) \wedge (\text{info} = i)\} \\
\text{nodeSwap}(n, \text{info}); \\
\{\alpha \leftarrow \text{info}[\text{next}[\text{tree}(c)] \otimes \gamma \otimes n[\beta]] \wedge (\text{next} = m) \wedge (\text{info} = i)\} \\
\text{nodeSwap}(\text{next}, \text{info}); \\
\{\alpha \leftarrow \text{next}[\text{info}[\text{tree}(c)] \otimes \gamma \otimes n[\beta]] \wedge (\text{next} = m) \wedge (\text{info} = i)\} \\
\text{appendChild}(\text{next}, \text{info}) \} \\
\{\alpha \leftarrow \text{next}[\gamma \otimes n[\beta] \otimes \text{info}[\text{tree}(c)]] \wedge (\text{next} = m) \wedge (\text{info} = i)\} \\
\{\alpha \leftarrow m[\gamma \otimes n[\beta] \otimes i[\text{tree}(c)]]\}
\end{array}$$

simple derivation:	nodeSwap derivation:	nodeCycle derivation:
$\{\alpha \leftarrow n[m[\beta] \otimes \gamma]\}$ $\text{local } temp, first := null \text{ in } \{$ $\{\alpha \leftarrow n[m[\beta] \otimes \gamma] \wedge (temp = null)\}$ $\wedge (first = null)\}$ $\text{insertNodeAbove}(n);$ $\{\alpha \leftarrow \circ [n[m[\beta] \otimes \gamma]] \wedge (temp = null)\}$ $\wedge (first = null)\}$ $temp := getUp(n);$ $\{\alpha \leftarrow temp[n[m[\beta] \otimes \gamma]] \wedge (first = null)\}$ $first := getFirst(n);$ $\{\alpha \leftarrow temp[n[first[\beta] \otimes \gamma]] \wedge (first = m)\}$ $\text{moveNodeAbove}(first, n);$ $\{\alpha \leftarrow temp[n[first[\beta]] \otimes \gamma] \wedge (first = m)\}$ $\text{moveNodeAbove}(temp, first);$ $\{\alpha \leftarrow first[temp[n[\beta] \otimes \gamma]] \wedge (first = m)\}$ $\text{deleteNode}(temp)\}$ $\{\alpha \leftarrow first[n[\beta] \otimes \gamma] \wedge (first = m)\}$ $\{\alpha \leftarrow m[n[\beta] \otimes \gamma]\}$	$\{\alpha \leftarrow n[\gamma] * \beta \leftarrow m[\delta]\}$ $\text{local } temp := null \text{ in } \{$ $\{\alpha \leftarrow n[\gamma] * \beta \leftarrow m[\delta]\}$ $\wedge (temp = null)\}$ $\text{insertNodeAbove}(n);$ $\{\alpha \leftarrow \circ [n[\gamma]] * \beta \leftarrow m[\delta]\}$ $\wedge (temp = null)\}$ $temp := getUp(n);$ $\{\alpha \leftarrow temp[n[\gamma]] * \beta \leftarrow m[\delta]\}$ $\text{moveNodeAbove}(m, n);$ $\{\alpha \leftarrow temp[\gamma] * \beta \leftarrow n[m[\delta]]\}$ $\text{moveNodeAbove}(temp, m);$ $\{\alpha \leftarrow m[temp[\gamma]] * \beta \leftarrow n[\delta]\}$ $\text{deleteNode}(temp)\}$ $\{\alpha \leftarrow m[\gamma] * \beta \leftarrow n[\delta]\}$	$\{\alpha \leftarrow n[m[\beta] \otimes \gamma \otimes I[\delta]]\}$ $\text{local } first, last := null \text{ in } \{$ $\{\alpha \leftarrow n[m[\beta] \otimes \gamma \otimes I[\delta]]\}$ $\wedge (first = null) \wedge (last = null)\}$ $first := getFirst(n);$ $\{\alpha \leftarrow n[first[\beta] \otimes \gamma \otimes I[\delta]]\}$ $\wedge (first = m) \wedge (last = null)\}$ $last := getLast(n);$ $\{\alpha \leftarrow n[first[\beta] \otimes \gamma \otimes last[\delta]]\}$ $\wedge (first = m) \wedge (last = l)\}$ $\text{nodeSwap}(n, last);$ $\{\alpha \leftarrow last[first[\beta] \otimes \gamma \otimes n[\delta]]\}$ $\wedge (first = m) \wedge (last = l)\}$ $\text{nodeSwap}(n, first)\}$ $\{\alpha \leftarrow last[n[\beta] \otimes \gamma \otimes first[\delta]]\}$ $\wedge (first = m) \wedge (last = l)\}$ $\{\alpha \leftarrow l[n[\beta] \otimes \gamma \otimes m[\delta]]\}$

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 [15].

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 [16]. 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 α .

Our next step is to design and formally specify a concurrent XML update language, combining ideas from Featherweight DOM, Concurrent Separation Logic [12] and Segment Logic. We believe that Segment Logic provides us with crucial technology for achieving this goal. For example, consider the program `deleteTree(n) || deleteTree(m)`, which should succeed if the two trees being

called are disjoint. A Segment Logic specification of this program is:

$$\begin{aligned} & \{\alpha \leftarrow n[\text{tree}(c_1)] * \beta \leftarrow m[\text{tree}(c_2)]\} \\ & \text{deleteTree}(n) \parallel \text{deleteTree}(m) \\ & \{\alpha \leftarrow \emptyset_T * \beta \leftarrow \emptyset_T\} \end{aligned}$$

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.

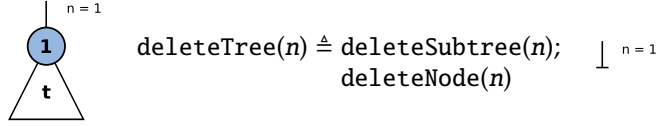
References

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 U. Zarfaty. Reasoning about high-level tree update and its low-level implementation. Technical Report DTR09-9, Imperial College, 2009.
8. H. Hosoya and B. C. Pierce. Xduce: A statically typed XML processing language. *TOIT, ACM*, 3, 2003.
9. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. *POPL, ACM*, 36, 2001.
10. R. Milner. Pi-nets: A graphical form of π -calculus. *ESOP, LNCS*, 788, 1994.
11. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes I & II. *Information and Computation*, 100, 1992.
12. P. W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375, 2007.
13. P. W. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. *CSL, LNCS*, 15, 2001.
14. M. Raza and P. Gardner. Footprints in local reasoning. *FoSSaCS, LNCS*, 4962, 2008.
15. G. Smith. Providing a formal specification for DOM core level 1. PhD Thesis, to be submitted December 2009.

16. V. Vafeiadis. Modular fine-grained concurrency verification. Technical Report UCAM-CL-TR-726, Cambridge, 2008.
17. W3C. Dom: Document object model. W3C recommendation. <http://www.w3.org/DOM/>, 2005.
18. H. Yang and P. W. O'Hearn. A semantic basis for local reasoning. *FoSSaCS, LNCS*, 2303, 2002.

A Specifying deleteTree

A common command in many tree update languages is `deleteTree(n)` which deletes an entire tree starting at node n . Here we show how to implement this command as a simple program using the basic commands of our tree update language.



The specification, and derivation, for the above program are:

$$\begin{array}{l} \{ \alpha \leftarrow n[\text{tree}(c)] \} \\ \text{deleteTree}(n) \\ \{ \alpha \leftarrow \emptyset_T \} \end{array} \quad \begin{array}{l} \{ \alpha \leftarrow n[\text{tree}(c)] \} \\ \text{deleteSubtree}(n); \\ \{ \alpha \leftarrow n[\emptyset_T] \} \\ \{ \text{H}\beta. \alpha \leftarrow n[\beta] * \beta \leftarrow \emptyset_T \} \\ \text{deleteNode}(n) \\ \{ \text{H}\beta. \alpha \leftarrow \beta * \beta \leftarrow \emptyset_T \} \\ \{ \alpha \leftarrow \emptyset_T \} \end{array}$$

B Useful Lemmas

The following are a series of Lemmas that are useful when applying the rule of consequence:

$$\begin{array}{l} (P_S \multimap Q_S) * P_S \Leftrightarrow \diamond P_S \wedge Q_S \quad \text{for a precise data structure} \\ \alpha \text{-}\mathbb{R}(\alpha \mathbb{R} P_S) \Rightarrow P_S \\ \alpha \mathbb{R}(\alpha \text{-}\mathbb{R} P_S) \Rightarrow P_S \\ P_S * \emptyset_S \Leftrightarrow P_S \\ P_S * Q_S \Leftrightarrow Q_S * P_S \\ P_S * (Q_S * R_S) \Leftrightarrow (P_S * Q_S) * R_S \\ \alpha \mathbb{R} \emptyset_S \Leftrightarrow \emptyset_S \\ \alpha \mathbb{R}(\beta \leftarrow (P_T \wedge \text{-}@ \alpha) * P_S) \wedge \alpha \neq \beta \Leftrightarrow \beta \leftarrow (P_T \wedge \text{-}@ \alpha) * \alpha \mathbb{R}(P_S) \wedge \alpha \neq \beta \\ \alpha \mathbb{R} P_S \wedge \text{-}@ \beta \Leftrightarrow \beta \mathbb{R} P_S[\beta/\alpha] \\ \alpha \mathbb{R}(\beta \mathbb{R} P_S) \Leftrightarrow \beta \mathbb{R}(\alpha \mathbb{R} P_S) \\ \text{H}\alpha(\beta \leftarrow P_T \wedge @ \alpha * \alpha \leftarrow Q_T) \Leftrightarrow \beta \leftarrow P_T[Q_T/\alpha] \end{array}$$

The proofs are simple and follow directly from the satisfaction relations of our logic and the equivalence relation for tree fragments.

C Weakest Precondition Derivations

We annotate each proof step with the rule(s) used to make that logical inference. The annotations used are given below showing which inference rule from Definition 15 each annotation refers to:

—DISJ	DISJUNCTION
—CONS	CONSEQUENCE
—AUX	AUXILIARY VARIABLE ELIMINATION
—FRESH	FRESH LABEL ELIMINATION
—SEP	SEPARATION FRAME
—REV	REVELATION FRAME

skip

$$\begin{array}{c}
 \{\emptyset_S\} \\
 \text{skip} \\
 \{\emptyset_S\} \\
 \hline
 \{\emptyset_S * (\emptyset_S \multimap P_S)\} \\
 \text{skip} \\
 \{\emptyset_S * (\emptyset_S \multimap P_S)\} \\
 \hline
 \{P_S\} \\
 \text{skip} \\
 \{P_S\}
 \end{array}$$

SEP

CONS

node assignment

$$\begin{array}{c}
 \{\emptyset_S \wedge (n = n_0)\} \\
 n := N \\
 \{\emptyset_S \wedge (n = N_{[n_0/n]})\} \\
 \hline
 \{(\emptyset_S \wedge (n = n_0)) * (\emptyset_S \multimap P_{S[N_{[n_0/n]}/n]})\} \\
 n := N \\
 \{(\emptyset_S \wedge (n = N_{[n_0/n]}) * (\emptyset_S \multimap P_{S[N_{[n_0/n]}/n]})\} \\
 \hline
 \{\emptyset_S * (\emptyset_S \multimap P_{S[N_{[n_0/n]}/n]}) \wedge (n = n_0)\} \\
 n := N \\
 \{\emptyset_S * (\emptyset_S \multimap P_{S[N_{[n_0/n]}/n]}) \wedge (n = N_{[n_0/n]})\} \\
 \hline
 \{P_{S[N_{[n_0/n]}/n]} \wedge (n = n_0)\} \\
 n := N \\
 \{P_{S[N_{[n_0/n]}/n]} \wedge (n = N_{[n_0/n]})\} \\
 \hline
 \{\exists n_0. (n = n_0) \wedge P_{S[N/n]}\} \\
 n := N \\
 \{P_S\}
 \end{array}$$

SEP

CONS

CONS

CONS

CONS/AUX

getUp

$$\begin{array}{c}
\{\alpha \leftarrow m[\beta \otimes n[\delta] \otimes \gamma] \wedge (n' = n_0)\} \\
n' := \text{getUp}(n) \\
\{(\alpha \leftarrow m[\beta \otimes n[\delta] \otimes \gamma])_{[n_0/n']}\} \wedge (n' = m)\} \\
\hline
\text{SEP} \\
\{(\alpha \leftarrow m[\beta \otimes n[\delta] \otimes \gamma] \wedge (n' = n_0)) * ((\alpha \leftarrow m[\beta \otimes n[\delta] \otimes \gamma])_{[n_0/n']}) \multimap \alpha, \beta, \gamma, \delta \text{-}\textcircled{R} P_{S[m/n']})\} \\
n' := \text{getUp}(n) \\
\{((\alpha \leftarrow m[\beta \otimes n[\delta] \otimes \gamma])_{[n_0/n']}) \wedge (n' = m) * ((\alpha \leftarrow m[\beta \otimes n[\delta] \otimes \gamma])_{[n_0/n']}) \multimap \alpha, \beta, \gamma, \delta \text{-}\textcircled{R} P_{S[m/n']})\} \\
\hline
\text{CONS/REV} \\
\{\alpha, \beta, \gamma, \delta \textcircled{R} (\alpha \leftarrow m[\beta \otimes n[\delta] \otimes \gamma] * ((\alpha \leftarrow m[\beta \otimes n[\delta] \otimes \gamma])_{[n_0/n']}) \multimap \alpha, \beta, \gamma, \delta \text{-}\textcircled{R} P_{S[m/n']}) \wedge (n' = n_0)\} \\
n' := \text{getUp}(n) \\
\{\alpha, \beta, \gamma, \delta \textcircled{R} ((\alpha \leftarrow m[\beta \otimes n[\delta] \otimes \gamma])_{[n_0/n']}) * ((\alpha \leftarrow m[\beta \otimes n[\delta] \otimes \gamma])_{[n_0/n']}) \multimap \alpha, \beta, \gamma, \delta \text{-}\textcircled{R} P_{S[m/n']}) \wedge (n' = m)\} \\
\hline
\text{CONS} \\
\{\alpha, \beta, \gamma, \delta \textcircled{R} (\diamond(\alpha \leftarrow m[\beta \otimes n[\delta] \otimes \gamma])_{[n_0/n']}) \wedge (\alpha, \beta, \gamma, \delta \text{-}\textcircled{R} P_{S[m/n']}) \wedge (n' = n_0)\} \\
n' := \text{getUp}(n) \\
\{\alpha, \beta, \gamma, \delta \textcircled{R} (\alpha, \beta, \gamma, \delta \text{-}\textcircled{R} P_{S[m/n']}) \wedge (n' = m)\} \\
\hline
\text{CONS/FRESH} \\
\{\forall \alpha, \beta, \gamma, \delta. \alpha, \beta, \gamma, \delta \textcircled{R} (\diamond(\alpha \leftarrow m[\beta \otimes n[\delta] \otimes \gamma])_{[n_0/n']}) \wedge (n' = n_0) \wedge (\alpha, \beta, \gamma, \delta \text{-}\textcircled{R} P_{S[m/n']})\} \\
n' := \text{getUp}(n) \\
\{P_{S[m/n']}\} \wedge (n' = m)\} \\
\hline
\text{CONS/AUX} \\
\{\exists m, n_0. \forall \alpha, \beta, \gamma, \delta. \diamond(\alpha \leftarrow m[\beta \otimes n[\delta] \otimes \gamma] \wedge (n' = n_0) \wedge (\alpha, \beta, \gamma, \delta \text{-}\textcircled{R} P_{S[m/n']})\} \\
n' := \text{getUp}(n) \\
\{P_S\}
\end{array}$$

getLeft

$$\begin{array}{c}
\{\alpha \leftarrow m[\beta] \otimes n[\delta] \wedge (n' = n_0)\} \\
n' := \text{getLeft}(n) \\
\{(\alpha \leftarrow m[\beta] \otimes n[\delta])_{[n_0/n']}\} \wedge (n' = m)\} \\
\hline
\text{SEP} \\
\{(\alpha \leftarrow m[\beta] \otimes n[\delta] \wedge (n' = n_0)) * ((\alpha \leftarrow m[\beta] \otimes n[\delta])_{[n_0/n']}) \multimap (\alpha, \beta, \delta \text{-}\textcircled{R} P_{S[m/n']})\} \\
n' := \text{getLeft}(n) \\
\{((\alpha \leftarrow m[\beta] \otimes n[\delta])_{[n_0/n']}) \wedge (n' = m) * ((\alpha \leftarrow m[\beta] \otimes n[\delta])_{[n_0/n']}) \multimap (\alpha, \beta, \delta \text{-}\textcircled{R} P_{S[m/n']})\} \\
\hline
\text{CONS/REV} \\
\{\alpha, \beta, \delta \textcircled{R} (\alpha \leftarrow m[\beta] \otimes n[\delta] * ((\alpha \leftarrow m[\beta] \otimes n[\delta])_{[n_0/n']}) \multimap (\alpha, \beta, \delta \text{-}\textcircled{R} P_{S[m/n']})) \wedge (n' = n_0)\} \\
n' := \text{getLeft}(n) \\
\{\alpha, \beta, \delta \textcircled{R} ((\alpha \leftarrow m[\beta] \otimes n[\delta])_{[n_0/n']}) * ((\alpha \leftarrow m[\beta] \otimes n[\delta])_{[n_0/n']}) \multimap (\alpha, \beta, \delta \text{-}\textcircled{R} P_{S[m/n']}) \wedge (n' = m)\} \\
\hline
\text{CONS} \\
\{\alpha, \beta, \delta \textcircled{R} (\diamond(\alpha \leftarrow m[\beta] \otimes n[\delta])_{[n_0/n']}) \wedge (\alpha, \beta, \delta \text{-}\textcircled{R} P_{S[m/n']}) \wedge (n' = n_0)\} \\
n' := \text{getLeft}(n) \\
\{\alpha, \beta, \delta \textcircled{R} (\alpha, \beta, \delta \text{-}\textcircled{R} P_{S[m/n']}) \wedge (n' = m)\} \\
\hline
\text{CONS/FRESH} \\
\{\forall \alpha, \beta, \delta. \alpha, \beta, \delta \textcircled{R} (\diamond(\alpha \leftarrow m[\beta] \otimes n[\delta])_{[n_0/n']}) \wedge (n' = n_0) \wedge (\alpha, \beta, \delta \text{-}\textcircled{R} P_{S[m/n']})\} \\
n' := \text{getLeft}(n) \\
\{P_{S[m/n']}\} \wedge (n' = m)\} \\
\hline
\text{CONS/AUX} \\
\{\exists m, n_0. \forall \alpha, \beta, \delta. \diamond(\alpha \leftarrow m[\beta] \otimes n[\delta] \wedge (n' = n_0) \wedge (\alpha, \beta, \delta \text{-}\textcircled{R} P_{S[m/n']})\} \\
n' := \text{getLeft}(n) \\
\{P_S\}
\end{array}$$

$$\begin{array}{c}
\{\alpha \leftarrow m[n[\delta] \otimes \beta] \wedge (n' = n_0)\} \\
n' := \text{getLeft}(n) \\
\{(\alpha \leftarrow m[n[\delta] \otimes \beta])_{[n_0/n']}\} \wedge (n' = \text{null})\} \\
\hline
\text{SEP} \\
\{(\alpha \leftarrow m[n[\delta] \otimes \beta] \wedge (n' = n_0)) * ((\alpha \leftarrow m[n[\delta] \otimes \beta])_{[n_0/n']}) \rightarrow (\alpha, \beta, \delta \text{ -}\textcircled{\text{R}} P_{S[mult/n']})\} \\
n' := \text{getLeft}(n) \\
\{((\alpha \leftarrow m[n[\delta] \otimes \beta])_{[n_0/n']}) \wedge (n' = \text{null}) * ((\alpha \leftarrow m[n[\delta] \otimes \beta])_{[n_0/n']}) \rightarrow (\alpha, \beta, \delta \text{ -}\textcircled{\text{R}} P_{S[mult/n']})\} \\
\hline
\text{CONS/REV} \\
\{\alpha, \beta, \delta \textcircled{\text{R}} (\alpha \leftarrow m[n[\delta] \otimes \beta]) * ((\alpha \leftarrow m[n[\delta] \otimes \beta])_{[n_0/n']}) \rightarrow (\alpha, \beta, \delta \text{ -}\textcircled{\text{R}} P_{S[mult/n']}) \wedge (n' = n_0)\} \\
n' := \text{getLeft}(n) \\
\{\alpha, \beta, \delta \textcircled{\text{R}} ((\alpha \leftarrow m[n[\delta] \otimes \beta])_{[n_0/n']}) * ((\alpha \leftarrow m[n[\delta] \otimes \beta])_{[n_0/n']}) \rightarrow (\alpha, \beta, \delta \text{ -}\textcircled{\text{R}} P_{S[mult/n']}) \wedge (n' = \text{null})\} \\
\hline
\text{CONS} \\
\{\alpha, \beta, \delta \textcircled{\text{R}} (\diamond (\alpha \leftarrow m[n[\delta] \otimes \beta])_{[n_0/n']}) \wedge (\alpha, \beta, \delta \text{ -}\textcircled{\text{R}} P_{S[mult/n']}) \wedge (n' = n_0)\} \\
n' := \text{getLeft}(n) \\
\{\alpha, \beta, \delta \textcircled{\text{R}} (\alpha, \beta, \delta \text{ -}\textcircled{\text{R}} P_{S[mult/n']}) \wedge (n' = \text{null})\} \\
\hline
\text{CONS/FRESH} \\
\{\forall \alpha, \beta, \delta, \alpha, \beta, \delta \textcircled{\text{R}} (\diamond (\alpha \leftarrow m[n[\delta] \otimes \beta])_{[n_0/n']}) \wedge (n' = n_0) \wedge (\alpha, \beta, \delta \text{ -}\textcircled{\text{R}} P_{S[mult/n']})\} \\
n' := \text{getLeft}(n) \\
\{P_{S[mult/n']}\} \wedge (n' = \text{null})\} \\
\hline
\text{CONS/AUX} \\
\{\exists m, n_0, \text{H}\alpha, \beta, \delta, \diamond \alpha \leftarrow m[n[\delta] \otimes \beta] \wedge (n' = n_0) \wedge (\alpha, \beta, \delta \text{ -}\textcircled{\text{R}} P_{S[mult/n']})\} \\
n' := \text{getLeft}(n) \\
\{P_S\} \\
\hline
\text{DISJ} \\
\left\{ \begin{array}{l} \exists m, n_0, \text{H}\alpha, \beta, \delta, \diamond \alpha \leftarrow m[\beta] \otimes n[\delta] \wedge (n' = n_0) \wedge (\alpha, \beta, \delta \text{ -}\textcircled{\text{R}} P_{S[m/n']}) \\ \vee \diamond \alpha \leftarrow m[n[\delta] \otimes \beta] \wedge (n' = n_0) \wedge (\alpha, \beta, \delta \text{ -}\textcircled{\text{R}} P_{S[mult/n']}) \end{array} \right\} \\
n' := \text{getLeft}(n) \\
\{P_S\}
\end{array}$$

getRight

$$\begin{array}{c}
\{\alpha \leftarrow n[\delta] \otimes m[\beta] \wedge (n' = n_0)\} \\
n' := \text{getRight}(n) \\
\{(\alpha \leftarrow n[\delta] \otimes m[\beta])_{[n_0/n']}\} \wedge (n' = m)\} \\
\hline
\text{SEP} \\
\{(\alpha \leftarrow n[\delta] \otimes m[\beta] \wedge (n' = n_0)) * ((\alpha \leftarrow n[\delta] \otimes m[\beta])_{[n_0/n']}) \rightarrow (\alpha, \beta, \delta \text{ -}\textcircled{\text{R}} P_{S[m/n']})\} \\
n' := \text{getRight}(n) \\
\{((\alpha \leftarrow n[\delta] \otimes m[\beta])_{[n_0/n']}) \wedge (n' = m) * ((\alpha \leftarrow n[\delta] \otimes m[\beta])_{[n_0/n']}) \rightarrow (\alpha, \beta, \delta \text{ -}\textcircled{\text{R}} P_{S[m/n']})\} \\
\hline
\text{CONS/REV} \\
\{\alpha, \beta, \delta \textcircled{\text{R}} (\alpha \leftarrow n[\delta] \otimes m[\beta]) * ((\alpha \leftarrow n[\delta] \otimes m[\beta])_{[n_0/n']}) \rightarrow (\alpha, \beta, \delta \text{ -}\textcircled{\text{R}} P_{S[m/n']}) \wedge (n' = n_0)\} \\
n' := \text{getRight}(n) \\
\{\alpha, \beta, \delta \textcircled{\text{R}} ((\alpha \leftarrow n[\delta] \otimes m[\beta])_{[n_0/n']}) * ((\alpha \leftarrow n[\delta] \otimes m[\beta])_{[n_0/n']}) \rightarrow (\alpha, \beta, \delta \text{ -}\textcircled{\text{R}} P_{S[m/n']}) \wedge (n' = m)\} \\
\hline
\text{CONS} \\
\{\alpha, \beta, \delta \textcircled{\text{R}} (\diamond (\alpha \leftarrow n[\delta] \otimes m[\beta])_{[n_0/n']}) \wedge (\alpha, \beta, \delta \text{ -}\textcircled{\text{R}} P_{S[m/n']}) \wedge (n' = n_0)\} \\
n' := \text{getRight}(n) \\
\{\alpha, \beta, \delta \textcircled{\text{R}} (\alpha, \beta, \delta \text{ -}\textcircled{\text{R}} P_{S[m/n']}) \wedge (n' = m)\} \\
\hline
\text{CONS/FRESH} \\
\{\forall \alpha, \beta, \delta, \alpha, \beta, \delta \textcircled{\text{R}} (\diamond (\alpha \leftarrow n[\delta] \otimes m[\beta])_{[n_0/n']}) \wedge (n' = n_0) \wedge (\alpha, \beta, \delta \text{ -}\textcircled{\text{R}} P_{S[m/n']})\} \\
n' := \text{getRight}(n) \\
\{P_{S[m/n']}\} \wedge (n' = m)\} \\
\hline
\text{CONS/AUX} \\
\{\exists m, n_0, \text{H}\alpha, \beta, \delta, \diamond \alpha \leftarrow n[\delta] \otimes m[\beta] \wedge (n' = n_0) \wedge (\alpha, \beta, \delta \text{ -}\textcircled{\text{R}} P_{S[m/n']})\} \\
n' := \text{getRight}(n) \\
\{P_S\}
\end{array}$$

$$\begin{array}{c}
\{\alpha \leftarrow m[\beta \otimes n[\delta]] \wedge (n' = n_0)\} \\
n' := \text{getRight}(n) \\
\{(\alpha \leftarrow m[\beta \otimes n[\delta]])_{[n_0/n']}\} \wedge (n' = \text{null})\} \\
\hline
\text{SEP} \\
\{(\alpha \leftarrow m[\beta \otimes n[\delta]] \wedge (n' = n_0)) * ((\alpha \leftarrow m[\beta \otimes n[\delta]])_{[n_0/n']}) \multimap (\alpha, \beta, \delta \text{ -}\mathbb{R} P_{S\{null/n'\}})\} \\
n' := \text{getRight}(n) \\
\{((\alpha \leftarrow m[\beta \otimes n[\delta]])_{[n_0/n']}) \wedge (n' = \text{null}) * ((\alpha \leftarrow m[\beta \otimes n[\delta]])_{[n_0/n']}) \multimap (\alpha, \beta, \delta \text{ -}\mathbb{R} P_{S\{null/n'\}})\} \\
\hline
\text{CONS/REV} \\
\{\alpha, \beta, \delta \mathbb{R} (\alpha \leftarrow m[\beta \otimes n[\delta]] * ((\alpha \leftarrow m[\beta \otimes n[\delta]])_{[n_0/n']}) \multimap (\alpha, \beta, \delta \text{ -}\mathbb{R} P_{S\{null/n'\}})) \wedge (n' = n_0)\} \\
n' := \text{getRight}(n) \\
\{\alpha, \beta, \delta \mathbb{R} ((\alpha \leftarrow m[\beta \otimes n[\delta]])_{[n_0/n']}) * ((\alpha \leftarrow m[\beta \otimes n[\delta]])_{[n_0/n']}) \multimap (\alpha, \beta, \delta \text{ -}\mathbb{R} P_{S\{null/n'\}}) \wedge (n' = \text{null})\} \\
\hline
\text{CONS} \\
\{\alpha, \beta, \delta \mathbb{R} (\diamond (\alpha \leftarrow m[\beta \otimes n[\delta]])_{[n_0/n']} \wedge (\alpha, \beta, \delta \text{ -}\mathbb{R} P_{S\{null/n'\}}) \wedge (n' = n_0))\} \\
n' := \text{getRight}(n) \\
\{\alpha, \beta, \delta \mathbb{R} (\alpha, \beta, \delta \text{ -}\mathbb{R} P_{S\{null/n'\}}) \wedge (n' = \text{null})\} \\
\hline
\text{CONS/FRESH} \\
\{\forall \alpha, \beta, \delta. \alpha, \beta, \delta \mathbb{R} (\diamond (\alpha \leftarrow m[\beta \otimes n[\delta]])_{[n_0/n']} \wedge (n' = n_0) \wedge (\alpha, \beta, \delta \text{ -}\mathbb{R} P_{S\{null/n'\}}))\} \\
n' := \text{getRight}(n) \\
\{P_{S\{null/n'\}} \wedge (n' = \text{null})\} \\
\hline
\text{CONS/AUX} \\
\{\exists m, n_0. \text{H} \alpha, \beta, \delta. \diamond \alpha \leftarrow m[\beta \otimes n[\delta]] \wedge (n' = n_0) \wedge (\alpha, \beta, \delta \text{ -}\mathbb{R} P_{S\{null/n'\}})\} \\
n' := \text{getRight}(n) \\
\{P_S\} \\
\hline
\text{DISJ} \\
\left\{ \begin{array}{l} \exists m, n_0. \text{H} \alpha, \beta, \delta. \quad \diamond \alpha \leftarrow n[\delta] \otimes m[\beta] \wedge (n' = n_0) \wedge (\alpha, \beta, \delta \text{ -}\mathbb{R} P_{S\{m/n'\}}) \\ \vee \diamond \alpha \leftarrow m[\beta] \otimes n[\delta] \wedge (n' = n_0) \wedge (\alpha, \beta, \delta \text{ -}\mathbb{R} P_{S\{null/n'\}}) \end{array} \right\} \\
n' := \text{getRight}(n) \\
\{P_S\}
\end{array}$$

getFirst

$$\begin{array}{c}
\{\alpha \leftarrow n[m[\beta] \otimes \delta] \wedge (n' = n_0)\} \\
n' := \text{getFirst}(n) \\
\{(\alpha \leftarrow n[m[\beta] \otimes \delta])_{[n_0/n']}\} \wedge (n' = m) \\
\hline
\text{SEP} \\
\{(\alpha \leftarrow n[m[\beta] \otimes \delta] \wedge (n' = n_0)) * ((\alpha \leftarrow n[m[\beta] \otimes \delta])_{[n_0/n']}) \multimap (\alpha, \beta, \delta \text{ -}\mathbb{R} P_{S\{m/n'\}})\} \\
n' := \text{getFirst}(n) \\
\{((\alpha \leftarrow n[m[\beta] \otimes \delta])_{[n_0/n']}) \wedge (n' = m) * ((\alpha \leftarrow n[m[\beta] \otimes \delta])_{[n_0/n']}) \multimap (\alpha, \beta, \delta \text{ -}\mathbb{R} P_{S\{m/n'\}})\} \\
\hline
\text{CONS/REV} \\
\{\alpha, \beta, \delta \mathbb{R} (\alpha \leftarrow n[m[\beta] \otimes \delta] * ((\alpha \leftarrow n[m[\beta] \otimes \delta])_{[n_0/n']}) \multimap (\alpha, \beta, \delta \text{ -}\mathbb{R} P_{S\{m/n'\}})) \wedge (n' = n_0)\} \\
n' := \text{getFirst}(n) \\
\{\alpha, \beta, \delta \mathbb{R} ((\alpha \leftarrow n[m[\beta] \otimes \delta])_{[n_0/n']}) * ((\alpha \leftarrow n[m[\beta] \otimes \delta])_{[n_0/n']}) \multimap (\alpha, \beta, \delta \text{ -}\mathbb{R} P_{S\{m/n'\}}) \wedge (n' = m)\} \\
\hline
\text{CONS} \\
\{\alpha, \beta, \delta \mathbb{R} (\diamond (\alpha \leftarrow n[m[\beta] \otimes \delta])_{[n_0/n']} \wedge (\alpha, \beta, \delta \text{ -}\mathbb{R} P_{S\{m/n'\}}) \wedge (n' = n_0))\} \\
n' := \text{getFirst}(n) \\
\{\alpha, \beta, \delta \mathbb{R} (\alpha, \beta, \delta \text{ -}\mathbb{R} P_{S\{m/n'\}}) \wedge (n' = m)\} \\
\hline
\text{CONS/FRESH} \\
\{\forall \alpha, \beta, \delta. \alpha, \beta, \delta \mathbb{R} (\diamond (\alpha \leftarrow n[m[\beta] \otimes \delta])_{[n_0/n']} \wedge (n' = n_0) \wedge (\alpha, \beta, \delta \text{ -}\mathbb{R} P_{S\{m/n'\}}))\} \\
n' := \text{getFirst}(n) \\
\{P_{S\{m/n'\}} \wedge (n' = m)\} \\
\hline
\text{CONS/AUX} \\
\{\exists m, n_0. \text{H} \alpha, \beta, \delta. \diamond \alpha \leftarrow n[m[\beta] \otimes \delta] \wedge (n' = n_0) \wedge (\alpha, \beta, \delta \text{ -}\mathbb{R} P_{S\{m/n'\}})\} \\
n' := \text{getFirst}(n) \\
\{P_S\}
\end{array}$$

$$\begin{array}{c}
\{\alpha \leftarrow n[\varnothing_T] \wedge (n' = n_0)\} \\
n' := \text{getFirst}(n) \\
\{(\alpha \leftarrow n[\varnothing_T])_{[n_0/n']}] \wedge (n' = \text{null})\} \\
\hline
\text{SEP} \\
\{(\alpha \leftarrow n[\varnothing_T] \wedge (n' = n_0)) * ((\alpha \leftarrow n[\varnothing_T])_{[n_0/n']} \multimap (\alpha, \beta, \delta \text{-}\textcircled{R} P_{S[\text{null}/n']}))\} \\
n' := \text{getFirst}(n) \\
\{((\alpha \leftarrow n[\varnothing_T])_{[n_0/n']} \wedge (n' = \text{null})) * ((\alpha \leftarrow n[\varnothing_T])_{[n_0/n']} \multimap (\alpha, \beta, \delta \text{-}\textcircled{R} P_{S[\text{null}/n']}))\} \\
\hline
\text{CONS/REV} \\
\{\alpha, \beta, \delta \textcircled{R} (\alpha \leftarrow n[\varnothing_T] * ((\alpha \leftarrow n[\varnothing_T])_{[n_0/n']} \multimap (\alpha, \beta, \delta \text{-}\textcircled{R} P_{S[\text{null}/n']})) \wedge (n' = n_0))\} \\
n' := \text{getFirst}(n) \\
\{\alpha, \beta, \delta \textcircled{R} ((\alpha \leftarrow n[\varnothing_T])_{[n_0/n']} * ((\alpha \leftarrow n[\varnothing_T])_{[n_0/n']} \multimap (\alpha, \beta, \delta \text{-}\textcircled{R} P_{S[\text{null}/n']})) \wedge (n' = \text{null}))\} \\
\hline
\text{CONS} \\
\{\alpha, \beta, \delta \textcircled{R} (\diamond (\alpha \leftarrow n[\varnothing_T])_{[n_0/n']} \wedge (\alpha, \beta, \delta \text{-}\textcircled{R} P_{S[\text{null}/n']}) \wedge (n' = n_0))\} \\
n' := \text{getFirst}(n) \\
\{\alpha, \beta, \delta \textcircled{R} (\alpha, \beta, \delta \text{-}\textcircled{R} P_{S[\text{null}/n']}) \wedge (n' = \text{null})\} \\
\hline
\text{CONS/FRESH} \\
\{\forall \alpha, \beta, \delta, \alpha, \beta, \delta, \diamond (\alpha \leftarrow n[\varnothing_T])_{[n_0/n']} \wedge (n' = n_0) \wedge (\alpha, \beta, \delta \text{-}\textcircled{R} P_{S[\text{null}/n']})\} \\
n' := \text{getFirst}(n) \\
\{P_{S[\text{null}/n']} \wedge (n' = \text{null})\} \\
\hline
\text{CONS/AUX} \\
\{\exists m, n_0. \text{H} \alpha, \beta, \delta. \diamond \alpha \leftarrow n[\varnothing_T] \wedge (n' = n_0) \wedge (\alpha, \beta, \delta \text{-}\textcircled{R} P_{S[\text{null}/n']})\} \\
n' := \text{getFirst}(n) \\
\{P_S\} \\
\hline
\text{DISJ} \\
\left\{ \begin{array}{l} \exists m, n_0. \text{H} \alpha, \beta, \delta. \diamond \alpha \leftarrow n[m[\beta] \otimes \delta] \wedge (n' = n_0) \wedge (\alpha, \beta, \delta \text{-}\textcircled{R} P_{S - [m/n']}) \\ \vee \diamond \alpha \leftarrow n[\varnothing_T] \wedge (n' = n_0) \wedge (\alpha \text{-}\textcircled{R} P_{S[\text{null}/n']}) \end{array} \right\} \\
n' := \text{getFirst}(n) \\
\{P_S\}
\end{array}$$

getLast

$$\begin{array}{c}
\{\alpha \leftarrow n[\delta \otimes m[\beta]] \wedge (n' = n_0)\} \\
n' := \text{getLast}(n) \\
\{(\alpha \leftarrow n[\delta \otimes m[\beta]])_{[n_0/n']}] \wedge (n' = m)\} \\
\hline
\text{SEP} \\
\{(\alpha \leftarrow n[\delta \otimes m[\beta]] \wedge (n' = n_0)) * ((\alpha \leftarrow n[\delta \otimes m[\beta]])_{[n_0/n']} \multimap (\alpha, \beta, \delta \text{-}\textcircled{R} P_{S[m/n']}))\} \\
n' := \text{getLast}(n) \\
\{((\alpha \leftarrow n[\delta \otimes m[\beta]])_{[n_0/n']} \wedge (n' = m)) * ((\alpha \leftarrow n[\delta \otimes m[\beta]])_{[n_0/n']} \multimap (\alpha, \beta, \delta \text{-}\textcircled{R} P_{S[m/n']}))\} \\
\hline
\text{CONS/REV} \\
\{\alpha, \beta, \delta \textcircled{R} (\alpha \leftarrow n[\delta \otimes m[\beta]] * ((\alpha \leftarrow n[\delta \otimes m[\beta]])_{[n_0/n']} \multimap (\alpha, \beta, \delta \text{-}\textcircled{R} P_{S[m/n']})) \wedge (n' = n_0))\} \\
n' := \text{getLast}(n) \\
\{\alpha, \beta, \delta \textcircled{R} ((\alpha \leftarrow n[\delta \otimes m[\beta]])_{[n_0/n']} * ((\alpha \leftarrow n[\delta \otimes m[\beta]])_{[n_0/n']} \multimap (\alpha, \beta, \delta \text{-}\textcircled{R} P_{S[m/n']})) \wedge (n' = m))\} \\
\hline
\text{CONS} \\
\{\alpha, \beta, \delta \textcircled{R} (\diamond (\alpha \leftarrow n[\delta \otimes m[\beta]])_{[n_0/n']} \wedge (\alpha, \beta, \delta \text{-}\textcircled{R} P_{S[m/n']}) \wedge (n' = n_0))\} \\
n' := \text{getLast}(n) \\
\{\alpha, \beta, \delta \textcircled{R} (\alpha, \beta, \delta \text{-}\textcircled{R} P_{S[m/n']}) \wedge (n' = m)\} \\
\hline
\text{CONS/FRESH} \\
\{\forall \alpha, \beta, \delta, \alpha, \beta, \delta, \diamond (\alpha \leftarrow n[\delta \otimes m[\beta]])_{[n_0/n']} \wedge (n' = n_0) \wedge (\alpha, \beta, \delta \text{-}\textcircled{R} P_{S[m/n']})\} \\
n' := \text{getLast}(n) \\
\{P_{S[m/n']} \wedge (n' = m)\} \\
\hline
\text{CONS/AUX} \\
\{\exists m, n_0. \text{H} \alpha, \beta, \delta. \diamond \alpha \leftarrow n[\delta \otimes m[\beta]] \wedge (n' = n_0) \wedge (\alpha, \beta, \delta \text{-}\textcircled{R} P_{S[m/n']})\} \\
n' := \text{getLast}(n) \\
\{P_S\}
\end{array}$$

$$\begin{array}{c}
\{\alpha \leftarrow n[\emptyset_T] \wedge (n' = n_0)\} \\
n' := \text{getLast}(n) \\
\{(\alpha \leftarrow n[\emptyset_T])_{[n_0/n']}\} \wedge (n' = \text{null})\} \\
\hline
\text{SEP} \\
\{(\alpha \leftarrow n[\emptyset_T] \wedge (n' = n_0)) * ((\alpha \leftarrow n[\emptyset_T])_{[n_0/n']} \multimap (\alpha, \beta, \delta \text{-}\textcircled{R} P_S[mull/n']))\} \\
n' := \text{getLast}(n) \\
\{((\alpha \leftarrow n[\emptyset_T])_{[n_0/n']}\} \wedge (n' = \text{null})) * ((\alpha \leftarrow n[\emptyset_T])_{[n_0/n']} \multimap (\alpha, \beta, \delta \text{-}\textcircled{R} P_S[mull/n']))\} \\
\hline
\text{CONS/REV} \\
\{\alpha, \beta, \delta \textcircled{R} (\alpha \leftarrow n[\emptyset_T] * ((\alpha \leftarrow n[\emptyset_T])_{[n_0/n']} \multimap (\alpha, \beta, \delta \text{-}\textcircled{R} P_S[mull/n'])) \wedge (n' = n_0))\} \\
n' := \text{getLast}(n) \\
\{\alpha, \beta, \delta \textcircled{R} ((\alpha \leftarrow n[\emptyset_T])_{[n_0/n']}\} * ((\alpha \leftarrow n[\emptyset_T])_{[n_0/n']} \multimap (\alpha, \beta, \delta \text{-}\textcircled{R} P_S[mull/n'])) \wedge (n' = \text{null}))\} \\
\hline
\text{CONS} \\
\{\alpha, \beta, \delta \textcircled{R} (\diamond (\alpha \leftarrow n[\emptyset_T])_{[n_0/n']}\} \wedge (\alpha, \beta, \delta \text{-}\textcircled{R} P_S[mull/n'])) \wedge (n' = n_0)\} \\
n' := \text{getLast}(n) \\
\{\alpha, \beta, \delta \textcircled{R} (\alpha, \beta, \delta \text{-}\textcircled{R} P_S[mull/n']) \wedge (n' = \text{null})\} \\
\hline
\text{CONS/FRESH} \\
\{\forall \alpha, \beta, \delta, \alpha, \beta, \delta \textcircled{R} (\diamond (\alpha \leftarrow n[\emptyset_T])_{[n_0/n']}\} \wedge (n' = n_0) \wedge (\alpha, \beta, \delta \text{-}\textcircled{R} P_S[mull/n']))\} \\
n' := \text{getLast}(n) \\
\{P_S[mull/n'] \wedge (n' = \text{null})\} \\
\hline
\text{CONS/AUX} \\
\{\exists m, n_0. \text{H}\alpha, \beta, \delta. \diamond \alpha \leftarrow n[\emptyset_T] \wedge (n' = n_0) \wedge (\alpha, \beta, \delta \text{-}\textcircled{R} P_S[mull/n'])\} \\
n' := \text{getLast}(n) \\
\{P_S\} \\
\hline
\text{DISJ} \\
\left\{ \begin{array}{l} \exists m, n_0. \text{H}\alpha, \beta, \delta. \diamond \alpha \leftarrow n[\delta \otimes m[\beta]] \wedge (n' = n_0) \wedge (\alpha, \beta, \delta \text{-}\textcircled{R} P_S[m/n']) \\ \vee \diamond \alpha \leftarrow n[\emptyset_T] \wedge (n' = n_0) \wedge (\alpha \text{-}\textcircled{R} P_S[mull/n']) \end{array} \right\} \\
n' := \text{getLast}(n) \\
\{P_S\}
\end{array}$$

deleteNode

$$\begin{array}{c}
\{\alpha \leftarrow n[\beta]\} \\
\text{deleteNode}(n) \\
\{\alpha \leftarrow \beta\} \\
\hline
\text{SEP} \\
\{\alpha \leftarrow n[\beta] * (\alpha \leftarrow \beta \multimap (\alpha, \beta \text{-}\textcircled{R} P_S))\} \\
\text{deleteNode}(n) \\
\{\alpha \leftarrow \beta * (\alpha \leftarrow \beta \multimap (\alpha, \beta \text{-}\textcircled{R} P_S))\} \\
\hline
\text{CONS/REV} \\
\{\alpha, \beta \textcircled{R} ((\alpha \leftarrow \beta \multimap (\alpha, \beta \text{-}\textcircled{R} P_S)) * \alpha \leftarrow n[\beta])\} \\
\text{deleteNode}(n) \\
\{\alpha, \beta \textcircled{R} (\alpha, \beta \text{-}\textcircled{R} P_S)\} \\
\hline
\text{CONS/FRESH} \\
\{\forall \alpha, \beta. \alpha, \beta \textcircled{R} ((\alpha \leftarrow \beta \multimap (\alpha, \beta \text{-}\textcircled{R} P_S)) * \alpha \leftarrow n[\beta])\} \\
\text{deleteNode}(n) \\
\{P_S\} \\
\hline
\text{CONS} \\
\{\text{H}\alpha, \beta. ((\alpha \leftarrow \beta \multimap (\alpha, \beta \text{-}\textcircled{R} P_S)) * \alpha \leftarrow n[\beta])\} \\
\text{deleteNode}(n) \\
\{P_S\}
\end{array}$$

deleteSubTree

$$\begin{array}{c}
\frac{\{ \alpha \leftarrow n[\text{tree}(c)] \} \\ \text{deleteSubtree}(n) \\ \{ \alpha \leftarrow n[\emptyset_T] \}}{\text{SEP}} \\
\frac{\{ \alpha \leftarrow n[\text{tree}(c)] * (\alpha \leftarrow n[\emptyset_T] \multimap (\alpha \text{ -- } \textcircled{R} P_S)) \} \\ \text{deleteSubtree}(n) \\ \{ \alpha \leftarrow n[\emptyset_T] * (\alpha \leftarrow n[\emptyset_T] \multimap (\alpha \text{ -- } \textcircled{R} P_S)) \}}{\text{CONS/REV}} \\
\frac{\{ \alpha \textcircled{R} ((\alpha \leftarrow n[\emptyset_T] \multimap (\alpha \text{ -- } \textcircled{R} P_S)) * \alpha \leftarrow n[\text{tree}(c)]) \} \\ \text{deleteSubtree}(n) \\ \{ \alpha \textcircled{R} (\alpha \text{ -- } \textcircled{R} P_S) \}}{\text{CONS/FRESH}} \\
\frac{\{ \forall \alpha. \alpha \textcircled{R} ((\alpha \leftarrow n[\emptyset_T] \multimap (\alpha \text{ -- } \textcircled{R} P_S)) * \alpha \leftarrow n[\text{tree}(c)]) \} \\ \text{deleteSubtree}(n) \\ \{ P_S \}}{\text{CONS/AUX}} \\
\frac{\{ \exists c. \text{H}\alpha. ((\alpha \leftarrow n[\emptyset_T] \multimap (\alpha \text{ -- } \textcircled{R} P_S)) * \alpha \leftarrow n[\text{tree}(c)]) \} \\ \text{deleteSubtree}(n) \\ \{ P_S \}}{}
\end{array}$$

insertNodeAbove

$$\begin{array}{c}
\frac{\{ \alpha \leftarrow n[\beta] \} \\ \text{insertNodeAbove}(n) \\ \{ \alpha \leftarrow \circ [n[\beta]] \}}{\text{SEP}} \\
\frac{\{ \alpha \leftarrow n[\beta] * (\alpha \leftarrow \circ [n[\beta]] \multimap (\alpha, \beta \text{ -- } \textcircled{R} P_S)) \} \\ \text{insertNodeAbove}(n) \\ \{ \alpha \leftarrow \circ [n[\beta]] * (\alpha \leftarrow \circ [n[\beta]] \multimap (\alpha, \beta \text{ -- } \textcircled{R} P_S)) \}}{\text{CONS/REV}} \\
\frac{\{ \alpha, \beta \textcircled{R} ((\alpha \leftarrow \circ [n[\beta]] \multimap (\alpha, \beta \text{ -- } \textcircled{R} P_S)) * \alpha \leftarrow n[\beta]) \} \\ \text{insertNodeAbove}(n) \\ \{ \alpha, \beta \textcircled{R} (\alpha, \beta \text{ -- } \textcircled{R} P_S) \}}{\text{CONS/FRESH}} \\
\frac{\{ \forall \alpha, \beta. \alpha, \beta \textcircled{R} ((\alpha \leftarrow \circ [n[\beta]] \multimap (\alpha, \beta \text{ -- } \textcircled{R} P_S)) * \alpha \leftarrow n[\beta]) \} \\ \text{insertNodeAbove}(n) \\ \{ P_S \}}{\text{CONS}} \\
\frac{\{ \text{H}\alpha, \beta. ((\alpha \leftarrow \circ [n[\beta]] \multimap (\alpha, \beta \text{ -- } \textcircled{R} P_S)) * \alpha \leftarrow n[\beta]) \} \\ \text{insertNodeAbove}(n) \\ \{ P_S \}}{}
\end{array}$$

insertNodeLeft

$$\begin{array}{c}
\frac{\{ \alpha \leftarrow n[\beta] \} \\ \text{insertNodeLeft}(n) \\ \{ \alpha \leftarrow \circ [\emptyset_T] \otimes n[\beta] \}}{\text{SEP}} \\
\frac{\{ \alpha \leftarrow n[\beta] * (\alpha \leftarrow \circ [\emptyset_T] \otimes n[\beta] \multimap (\alpha, \beta \text{ -- } \textcircled{R} P_S)) \} \\ \text{insertNodeLeft}(n) \\ \{ \alpha \leftarrow \circ [\emptyset_T] \otimes n[\beta] * (\alpha \leftarrow \circ [\emptyset_T] \otimes n[\beta] \multimap (\alpha, \beta \text{ -- } \textcircled{R} P_S)) \}}{\text{CONS/REV}} \\
\frac{\{ \alpha, \beta \textcircled{R} ((\alpha \leftarrow \circ [\emptyset_T] \otimes n[\beta] \multimap (\alpha, \beta \text{ -- } \textcircled{R} P_S)) * \alpha \leftarrow n[\beta]) \} \\ \text{insertNodeLeft}(n) \\ \{ \alpha, \beta \textcircled{R} (\alpha, \beta \text{ -- } \textcircled{R} P_S) \}}{\text{CONS/FRESH}} \\
\frac{\{ \forall \alpha, \beta. \alpha, \beta \textcircled{R} ((\alpha \leftarrow \circ [\emptyset_T] \otimes n[\beta] \multimap (\alpha, \beta \text{ -- } \textcircled{R} P_S)) * \alpha \leftarrow n[\beta]) \} \\ \text{insertNodeLeft}(n) \\ \{ P_S \}}{\text{CONS}} \\
\frac{\{ \text{H}\alpha, \beta. ((\alpha \leftarrow \circ [\emptyset_T] \otimes n[\beta] \multimap (\alpha, \beta \text{ -- } \textcircled{R} P_S)) * \alpha \leftarrow n[\beta]) \} \\ \text{insertNodeLeft}(n) \\ \{ P_S \}}{}
\end{array}$$

insertNodeRight

$$\begin{array}{c}
\{\alpha \leftarrow n[\beta]\} \\
\text{insertNodeRight}(n) \\
\{\alpha \leftarrow n[\beta] \otimes \circ[\emptyset_T]\} \\
\hline
\text{SEP} \\
\{\alpha \leftarrow n[\beta] * (\alpha \leftarrow n[\beta] \otimes \circ[\emptyset_T] \multimap (\alpha, \beta \text{--}\textcircled{R} P_S))\} \\
\text{insertNodeRight}(n) \\
\{\alpha \leftarrow n[\beta] \otimes \circ[\emptyset_T] * (\alpha \leftarrow n[\beta] \otimes \circ[\emptyset_T] \multimap (\alpha, \beta \text{--}\textcircled{R} P_S))\} \\
\hline
\text{CONS/REV} \\
\{\alpha, \beta \textcircled{R} ((\alpha \leftarrow n[\beta] \otimes \circ[\emptyset_T] \multimap (\alpha, \beta \text{--}\textcircled{R} P_S)) * \alpha \leftarrow n[\beta])\} \\
\text{insertNodeRight}(n) \\
\{\alpha, \beta \textcircled{R} (\alpha, \beta \text{--}\textcircled{R} P_S)\} \\
\hline
\text{CONS/FRESH} \\
\{\forall \alpha, \beta, \alpha, \beta \textcircled{R} ((\alpha \leftarrow n[\beta] \otimes \circ[\emptyset_T] \multimap (\alpha, \beta \text{--}\textcircled{R} P_S)) * \alpha \leftarrow n[\beta])\} \\
\text{insertNodeRight}(n) \\
\{P_S\} \\
\hline
\text{CONS} \\
\{\text{H}\alpha, \beta, ((\alpha \leftarrow n[\beta] \otimes \circ[\emptyset_T] \multimap (\alpha, \beta \text{--}\textcircled{R} P_S)) * \alpha \leftarrow n[\beta])\} \\
\text{insertNodeRight}(n) \\
\{P_S\}
\end{array}$$

insertNodeFirst

$$\begin{array}{c}
\{\alpha \leftarrow n[\beta]\} \\
\text{insertNodeFirst}(n) \\
\{\alpha \leftarrow n[\circ[\emptyset_T] \otimes \beta]\} \\
\hline
\text{SEP} \\
\{\alpha \leftarrow n[\beta] * (\alpha \leftarrow n[\circ[\emptyset_T] \otimes \beta] \multimap (\alpha, \beta \text{--}\textcircled{R} P_S))\} \\
\text{insertNodeFirst}(n) \\
\{\alpha \leftarrow n[\circ[\emptyset_T] \otimes \beta] * (\alpha \leftarrow n[\circ[\emptyset_T] \otimes \beta] \multimap (\alpha, \beta \text{--}\textcircled{R} P_S))\} \\
\hline
\text{CONS/REV} \\
\{\alpha, \beta \textcircled{R} ((\alpha \leftarrow n[\circ[\emptyset_T] \otimes \beta] \multimap (\alpha, \beta \text{--}\textcircled{R} P_S)) * \alpha \leftarrow n[\beta])\} \\
\text{insertNodeFirst}(n) \\
\{\alpha, \beta \textcircled{R} (\alpha, \beta \text{--}\textcircled{R} P_S)\} \\
\hline
\text{CONS/FRESH} \\
\{\forall \alpha, \beta, \alpha, \beta \textcircled{R} ((\alpha \leftarrow n[\circ[\emptyset_T] \otimes \beta] \multimap (\alpha, \beta \text{--}\textcircled{R} P_S)) * \alpha \leftarrow n[\beta])\} \\
\text{insertNodeFirst}(n) \\
\{P_S\} \\
\hline
\text{CONS} \\
\{\text{H}\alpha, \beta, ((\alpha \leftarrow n[\circ[\emptyset_T] \otimes \beta] \multimap (\alpha, \beta \text{--}\textcircled{R} P_S)) * \alpha \leftarrow n[\beta])\} \\
\text{insertNodeFirst}(n) \\
\{P_S\}
\end{array}$$

insertNodeLast

$$\begin{array}{c}
\{\alpha \leftarrow n[\beta]\} \\
\text{insertNodeLast}(n) \\
\{\alpha \leftarrow n[\beta \otimes \circ[\emptyset_T]]\} \\
\hline
\text{SEP} \\
\{\alpha \leftarrow n[\beta] * (\alpha \leftarrow n[\beta \otimes \circ[\emptyset_T]] \multimap (\alpha, \beta \text{--}\textcircled{R} P_S))\} \\
\text{insertNodeLast}(n) \\
\{\alpha \leftarrow n[\beta \otimes \circ[\emptyset_T]] * (\alpha \leftarrow n[\beta \otimes \circ[\emptyset_T]] \multimap (\alpha, \beta \text{--}\textcircled{R} P_S))\} \\
\hline
\text{CONS/REV} \\
\{\alpha, \beta \textcircled{R} ((\alpha \leftarrow n[\beta \otimes \circ[\emptyset_T]] \multimap (\alpha, \beta \text{--}\textcircled{R} P_S)) * \alpha \leftarrow n[\beta])\} \\
\text{insertNodeLast}(n) \\
\{\alpha, \beta \textcircled{R} (\alpha, \beta \text{--}\textcircled{R} P_S)\} \\
\hline
\text{CONS/FRESH} \\
\{\forall \alpha, \beta, \alpha, \beta \textcircled{R} ((\alpha \leftarrow n[\beta \otimes \circ[\emptyset_T]] \multimap (\alpha, \beta \text{--}\textcircled{R} P_S)) * \alpha \leftarrow n[\beta])\} \\
\text{insertNodeLast}(n) \\
\{P_S\} \\
\hline
\text{CONS} \\
\{\text{H}\alpha, \beta, ((\alpha \leftarrow n[\beta \otimes \circ[\emptyset_T]] \multimap (\alpha, \beta \text{--}\textcircled{R} P_S)) * \alpha \leftarrow n[\beta])\} \\
\text{insertNodeLast}(n) \\
\{P_S\}
\end{array}$$

