MFPS 2006

Local Reasoning about Tree Update

Uri Zarfaty [1]

Department of Computing
Imperial College London
London, UK

Philippa Gardner [2]

Department of Computing
Imperial College London
London, UK

Abstract

Separation Logic and Context Logic have been used to reason locally about heap update and simple tree update. We study local reasoning based on Context Logic for a more realistic, local tree-update language which combines update commands with queries. This combination results in updates at multiple locations, which significantly affects the complexity of the reasoning.

Key words: local reasoning, context logic, tree update

1 Introduction

O'Hearn, Reynolds and Yang introduced the concept of *local* Hoare reasoning about heap update based on Separation Logic [10,13,7], a logic for reasoning about heaps which evolved from Bunched Logic of O'Hearn and Pym [?]. The idea is that, if an update only accesses part of the heap, leaving the rest unchanged, then this locality property should also be reflected in the reasoning. With Calcagno, we recently introduced Context Logic [3] to allow similar reasoning for non-flat structures such as trees. Although we justified our logic in part by reasoning about simple imperative tree-update commands, we did not apply our ideas to a full, realistic tree-update language involving the integration of update commands with queries. This turns out to be a non-trivial step, in that the combination of update at multiple locations and

[1] Email: udz@doc.ic.ac.uk

[2] Email: pg@doc.ic.ac.uk

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

the non-flat nature of trees means that we cannot directly use O'Hearn *et al.*'s small-axiom approach, where commands are specified only on the part of the heap or tree that they affect. We show in this paper that we are still able to reason locally about our tree-update language.

First, we describe our language, which to our surprise required some non-standard, but we believe natural, design choices. We focus on *local* commands. A command is local if the result of the command on a tree is the same, regardless of the context in which that tree is placed. Update commands for heaps are naturally local. We must work quite hard to find a non-local command: for example, the command 'remove all the values 25 from the heap' is non-local, but would just not be used in practice. In contrast, the current languages for updating web data are almost all non-local. For example, the tree-update command 'dispose all the nodes labelled by tag a' is non-local. A natural variant of this command would be 'dispose all the nodes labelled a under location (node identifier) n'. This is also non-local, as location n may not be in the tree but may appear later in a wider context. However, we can give a local interpretation of this command: if n is in the tree then dispose all the nodes labelled a; if n is not in the tree then return an error. Our initial motivation for focusing on such local commands was that they were essential for our exploration of local reasoning. We now also believe that local commands are good design practice, when viewing a tree as an updatable data store rather than a complete document (such as a XML document).

We present a Hoare Logic for reasoning locally about our tree-update language. Previous work on local reasoning using Separation Logic has highlighted the use of *small axioms* for specifying local commands. These axioms only mention the part of the heap affected by the command (the footprint), and are extended to the full heap using the Frame Rule. For example, consider the local command 'dispose n', which removes an address n from the heap. This only affects the address n, and its small axiom is

$$\{n \mapsto -\} \text{ dispose } n \; \{0\},$$

where the precondition states that the heap is just a cell with address n, and the postcondition states that the cell has been removed. The precondition corresponds to the minimal safety condition necessary for the command to execute.

This small axiom only describes properties about the specific cell n. To extend to properties about a larger heap, we use the Frame Rule to derive the triple

$$\{P * (n \mapsto -)\} \text{ dispose } \{P * 0\}$$

The assertion $P*(n \mapsto -)$ states that the heap can be split disjointedly into the cell n with an arbitrary value (the ordering of cells does not matter), and the rest of the heap with property P which is unaffected by the update command. The postcondition therefore has the same structure, with the removed cell specified by 0 and the rest of the heap satisfying P. The small axioms and

Frame Rule are elegant, and intuitively express the behaviour of commands. In addition, the weakest preconditions are derivable, a natural requirement which is essential for providing verification tools.

Independently, Cardelli and Gordon invented Ambient Logic for reasoning about static trees, which has a similar reasoning style to Separation Logic. A natural question is whether we can develop local Hoare reasoning for tree update (XML update) based on the Ambient Logic. With Calcagno, we have have shown that this is not possible. Instead, we had to fundamentally change the way we reason about structured data by introducing Context Logic [?]. Local data update typically identifies the portion of data to be replaced, removes it, and inserts the new data in the same place. With Context Logic, we can reason about both data and this place of insertion (contexts). We have shown that Context Logic can be used to develop local Hoare reasoning about tree update, heap update (analogous to Separation Logic reasoning, an important sanity check), and term rewriting (which escaped reasoning using Separation Logic).

The local Hoare reasoning for tree update presented in our initial paper [?] uses analogous small axioms and Frame Rule to those given for the heap case. In this paper, we show that such reasoning is not feasible for the local tree-update language presented here, because the commands act at a number of different locations in the tree rather than just one location. For example, consider the local tree command 'dispose l', where variable l has value $\{m, n\}$, typically obtained from a query. The footprint of this command is complex: m and n might be in disjoint parts of the tree, or one node might be under the other. It is therefore not possible to reason simply using small axioms. Nevertheless, local Hoare reasoning is still possible for our language, using a different axiom style where the preconditions describe arbitrary trees satisfying appropriate safety properties. For example, our axiom for 'dispose l' is

$$\{\Diamond l \wedge x\} \text{ dispose } l \ \{\text{fold } (\lambda n, Q.(\exists a, k, y.\ a_{n:k}[y] \blacktriangleright Q)(0))\ x\ l\}$$

The precondition specifies the safety property that all the locations denoted by l are in the given tree, and equates the given tree with a value x which we use in the postcondition. If l had value $\{n\}$, then the postcondition amounts to $\exists a, k, y.\, (a_{n:k}[y] \blacktriangleright x)(0)$, which specifies that it is possible to replace an empty subtree 0 by the tree $a_{n:k}[y]$ to obtain the original tree x. For an *arbitrary* location set l, the inductive fold assertion picks locations n one by one from l, replaces empty subtrees by trees of the form $a_{n:k}[y]$, to eventually obtain the original tree x. This axiom is not small, since the precondition does not specify the exact part of the tree touched by the command (the nodes denoted by l). Our reasoning is still local in that we can apply the Frame Rule to extend the reasoning to a larger tree. It is also comparatively easy to use, in that the axioms compose to provide safety conditions for complex programs and the derivations of the weakest preconditions shows a high level of regularity.

Our axiom style is undoubtedly less elegant than the small-axiom approach. The point is that we have identified an example of local update where simple small-axiom reasoning is not feasible. In this paper, we demonstrate that it is possible to do local reasoning, albeit with non-small axioms. It remains future work to investigate whether, by moving to a significantly more complex context structure, we can regain the small-axiom approach.

Related Work

We have recently seen our axiom style in Biri and Galmiche's work [2] on Hoare reasoning about a simple tree-update language without multiple locations. Partially inspired by our work on Context Logic, they propose resource trees as an alternative model of XML documents. They implement unique node addresses using sibling uniqueness, which allows for a total composition operator on trees and a partial composition operator on the resources inside the trees. They provide Hoare reasoning for their update language, based on Bunched Logic and path reasoning (with unique solutions) instead of Context Logic. Their work is based on our simple update language studied in [?], rather than a language with multiple locations as studied here. It is not clear from their discussion whether it is possible to use the small-axiom approach instead (they have the Frame Property for all but one commands) or whether, as we suspect, their path reasoning requires the non-small approach.

2 Tree Model

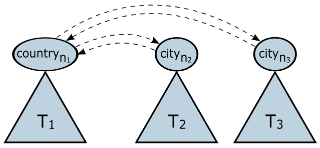
Our tree model consists of a labelled tree structure, with uniquely identified nodes and a set of pointers at each node: node identifiers allow us to update trees locally, labels allow us to traverse trees using path expressions, and pointers provide links to other parts of the data structure. This model is an adaptation of the 'trees with pointers' model studied by Cardelli, Gardner and Ghelli [4] with sets of pointers instead of the individual pointers used in [3]. It is similar to the trees studied in the book 'Data on the Web' [1], which again has individual pointers and in addition requires that the pointers are not dangling. Our tree model can be described in XML notation using ID and IDREF type attributes.

Given infinite sets of node names $\mathsf{n} \in \mathcal{N}$ and labels $\mathsf{a} \in \mathcal{A}$, we define the links at a particular node to be a finite node set $L \in \mathcal{P}_{\text{fin}}(\mathcal{N})$. Trees $T \in \mathcal{T}$ and contexts $C \in \mathcal{C}$ are defined as follows:

$$T ::= 0 \mid \mathsf{a}_{\mathsf{n}:L}[T] \;\mid\; T \,|\, T \qquad C ::= - \mid \mathsf{a}_{\mathsf{n}:L}[C] \;\mid\; T \,|\, C \;\mid\; C \,|\, T$$

The insertion of a tree in a context is written $C(T)$ and defined in the standard way. We assume a simple structural congruence on trees and contexts, denoted by $T_1 \equiv T_2$ and $C_1 \equiv C_2$, which states that parallel composition $(\_\,|\,\_)$ is commutative and associative with identity 0. An alternative choice would have been to use non-commutative composition, to fit more closely with the XML notation. This choice requires a trivial change to our reasoning.

A *well-formed* tree or context is one where the node identifiers are unique. We assume that all trees and contexts are well-formed. We write $\mathrm{Nodes}(T)$ for the set of node identifiers of a tree T. We are not directly concerned with the actual names of the node identifiers: our update language does not refer to them directly, just like standard heap update, and the append command renames the tree to be inserted to avoid name clashes, again standard practice. We say that two trees are *equivalent modulo renaming*, written $T_1 \simeq T_2$, if each can be mapped onto the other by renaming its node identifiers and any internal pointers. We give a standard example [1] of a tree with pointers, to illustrate our notation. We also use this example later in the paper to show the effect of an example update program.



$$\mathsf{country}_{\mathsf{n}_1:\{\mathsf{n}_2,\mathsf{n}_3\}}[T_1] \mid \mathsf{city}_{\mathsf{n}_2:\{\mathsf{n}_1\}}[T_2] \mid \mathsf{city}_{\mathsf{n}_3:\{\mathsf{n}_1\}}[T_3]$$

3 A Local Update Language

We provide a core update language for our tree model, integrating primitive update commands with query commands. Unlike update for relational databases, update languages for trees have rarely been studied in depth (but see [11]). We highlight the importance of *local* query and update commands for our trees. This emphasis is non-standard. Most of the languages in the literature are non-local: for example, the query command 'return all the nodes labelled a' is not local, in that its behaviour on a tree changes depending on the context of the tree. In contrast, our interpretation of the query 'return all the nodes labelled a under l' is local: either it succeeds and returns a set of nodes if the locations given by l are in the tree, or it fails and gives an error. With our view of a tree as a possibly large store of data, rather than a closed document (XML document), we believe it is good design practice to work with local commands. Locality is essential for the local reasoning studied in this paper.

3.1 Storage Model and Expressions

Our data storage model consists of two components: a store $s \in \mathcal{S}$ that maps variables to values, and a working tree T. Our command language uses three types of variables: $\mathrm{Var}_T = \{x, y, \ldots\}$ for trees; $\mathrm{Var}_L = \{k, l, \ldots\}$ for links; and $\mathrm{Var}_A = \{a, b, \ldots\}$ for labels. A store is a partial, finite function $s : (\mathrm{Var}_A \rightharpoonup_{\mathrm{fin}} \mathcal{A}) \times (\mathrm{Var}_L \rightharpoonup_{\mathrm{fin}} \mathcal{P}_{\mathrm{fin}}(\mathcal{N})) \times (\mathrm{Var}_T \rightharpoonup_{\mathrm{fin}} \mathcal{T})$. We do not require store variables for node identifiers, since our language only deals with updates at sets of locations. We write $[s|x \leftarrow v]$ to mean s overwritten with $s(x) = v$.

We define tree expressions E_T, link expressions E_L and label expressions E_A:

$$E_T ::= x \mid 0 \qquad E_L ::= l \mid \emptyset \qquad E_A ::= a \mid \mathsf{a} \in \mathcal{A}$$

We write $[\![E]\!]s$ for the valuation of E with respect to a store s. Note that expressions do not refer to explicit node identifiers, just as expressions in heap-update languages do not refer explicitly to heap addresses. Instead our language refers to identifiers only indirectly, by querying a tree to obtain sets of node identifiers or using the 'new' command to create new nodes with fresh identifiers.

3.2 Local Queries

We give an abstract account of *local queries*. All our results will be given at this abstract level. In Sect. 3.3, we describe a specific query language to help with examples and illustrate that local queries can still be expressive.

Definition. A query $q : \mathcal{S} \times \mathcal{T} \to \mathcal{P}_{\text{fin}}(\mathcal{N}) \cup \{\text{error}\}$ is a function such that $q(s,T) \in \mathcal{P}_{\text{fin}}(\mathcal{N})$ implies $q(s,T) \subseteq \text{Nodes}(T)$. A query q is *local* if and only if $\forall s, T, L, C.\ q(s,T) = L$ and $C(T)$ well-formed implies $q(s, C(T)) = L$.

Our definition is non-standard. We make an essential distinction between returning the empty set when no nodes satisfy a query, and an 'error' when the query is ill-defined on the tree: for example, if it tries to follow a dangling pointer. This follows from our decision to work with local queries: if a tree is sufficient to make a local query execute without error, then the query is guaranteed never to 'go beyond' that tree, and will return the same result on any larger tree. We explain this further in Sect. 3.3 where we give specific examples of local queries.

3.3 A Local Query Language

For our results, we will work with abstract queries. Here we briefly describe a local query language, motivated by XPATH [12], to help with examples and show that we have not lost much expressive power by moving to local queries:

$$\begin{array}{rll}
q & ::= l/\pi \mid q \cup q \mid q \cap q \mid q - q & \text{query} \\
\pi & ::= \pi_a :: \pi_f \mid \pi/\pi \mid \pi \cup \pi \mid \pi \cap \pi \mid \pi - \pi & \text{path} \\
\pi_a & ::= \text{self} \mid \text{child} \mid \text{descendant} \mid \text{parent} \mid \text{link} & \text{path axis} \\
\pi_f & ::= E_A \mid * & \text{label filter}
\end{array}$$

A basic query l/π is 'rooted' by an initial set of nodes given by a variable l. It then 'follows' in the tree the paths given by π. Each path step consists of a path axis which describes the next 'movements', dependent on a label condition being satisfied. The path axes are self-explanatory, except perhaps for the 'link' axis which 'follows' all the pointers from the current nodes.

	Trees	Links	Labels
Assign	$x := E_T$	$l' := E_L$ $l' := q$	$a := E_A$
Lookup	$x :=$ get-trees at l	$l' :=$ get-links at l	$a :=$ get-labels at l
Update	dispose at l append E_T at l	dispose-links at l append-links E_L at l	set-labels E_A at l
Move	move l to l'		
New	$l' :=$ new E_A at l		

Fig. 1. Basic Update Commands

Notice that an arbitrary ancestor axis is not possible, since it would break our locality requirement. We also require that all the initial nodes given by l are in the tree, or the query returns an error. Similarly, we obtain an error when a query tries to 'move' outside the tree, by following either a dangling pointer or the parent axis at the root of the tree. These conditions make our queries local. For example, the query l/descendant :: a returns all the nodes labelled a under the node identifiers given by l if *all* the identifiers are in the tree. If one identifier is not in the tree, then the query results in an error. This choice is analogous to local heap update, which returns an error if an address is not present in the heap. One could argue that a query should be able to return a partial answer if at least some of the nodes are in the tree. This is a reasonable point, but is just not feasible if we require the commands to be local.

The query semantics for this language is given in the Appendix. Below are some other simple examples of queries from our language, using some standard notation: π_f for child::π_f, '..' for parent::$*$ and '.' for self::$*$:

- **l/city** — all 'city'-labelled child nodes of l
- **l/city** ∪ **link::city** — all 'city'-labelled child nodes or link targets of l.
- **l/((../∗) - .)** — all the siblings of l. Note that, as a result of the locality condition, this returns an error for root-level nodes.

3.4 Update Commands

We present our high-level imperative update language for locally manipulating trees with pointers. The basic commands are given in Fig.1. They include commands for assignment, lookup and update, each with related forms for trees, links and labels. They also include a 'move' and 'new' command. We assume command sequencing $\mathbb{C};\mathbb{C}$, but do not present looping control commands since they do not contribute to the ideas presented in this paper. The extension to incorporate such looping commands, and the corresponding extension to the reasoning, is standard.

The commands should be fairly familiar. There are however some subtleties. All of the commands, apart from assignment, act at a set of locations given by a link variable l. To ensure the locality of the commands, we insist that all the locations given by l are present in the tree for the commands to execute; otherwise an error is returned. A number of the commands have several possible behavioural interpretations: for example, when referring to a tree at a node, we can refer to the whole tree including the node, or just the subforest; when appending a tree at a node, we can append it underneath the node or as a sibling. In this paper we select one behaviour for conciseness (the former in both the above cases); our choice can be trivially adapted to give the other behaviours.

The formal operational semantics of update commands is given in the Appendix. We given an informal (but precise) description here.

Assignment assigns the value of an expression to a variable of the appropriate type. There is an additional case for links, $l' := q$, which evaluates query q and assigns the resulting set of nodes to a link variable l'.

Lookup gets link, tree or label values from the nodes specified by a link variable l (where we have chosen the tree value at a node to include both the subtree at that node and the node itself). Since the result is a set of values, whereas our language works with just values, there is the issue of how to combine many values into one. We give one solution for each data type: for trees, we concatenate the results, renaming nodes as necessary; for links, we return the union of the results; for labels, we choose at random from the results. We have chosen to work with these specific solutions for simplicity, but can easily extend our reasoning to a more abstract approach.

Update updates the trees, links or labels at the nodes specified by a link variable l. In the tree case, this involves two commands: dispose, which disposes the trees located at l, or append at l, which appends the value of a tree expression E_T underneath the nodes in l. Direct appending is a partial operation. We choose to rename the node identifiers and any internal links, to obtain a total append function. This renaming is standard practice. There are similarly two link update commands, whilst label update consists of one command which simply replaces the old labels by new ones.

Move removes *one* tree from a source location, and adds the exact same tree at *one* target location, without renaming any nodes. The moving is only possible if the target location is not contained inside the source, while the non-renaming only works if the source and target destination both consist of just one node. The move command is important, since non-renaming allows us maintain inbound links. In contrast, the append command gives us no control over the inbound links: dangling pointers may be captured by renaming, or may be left dangling.

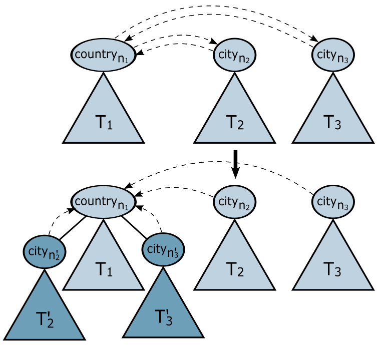
New creates new nodes at the locations specified by a link variable l. The new nodes have labels specified by E_A and fresh identifiers, which are stored in a link variable l'. The nodes have no links and no subtrees. These can be added using the other commands.

3.5 Examples

We present two simple programs showing the ease of collapsing a link structure. The first collapses all the links at a location l; the second only collapses the links given by a variable l'. On the right, we show the action of 'collapse-all-links' on the example tree from Sect. 2.

```
collapse-all-links at $l$
  $\triangleq$ $k := l/\text{link::}*$;
     $x :=$ get-trees at $k$;
     append $x$ at $l$;
     dispose-links at $l$

collapse-links $l'$ at $l$
  $\triangleq$ $k := l/\text{link::}* \cap l'$;
     $x :=$ get-trees at $k$;
     append $x$ at $l$;
     $k' := l/\text{link::}* - l'$;
     dispose-links at $l$;
     append-links $k'$ at $l$
```



collapse-all-links at l *where* $l = \{\mathsf{n}_1\}$

Note that both programs still work when l refers to more than one location.

4 Context Logic for Tree Update

We apply Context Logic reasoning to our tree model. Most of the ideas presented here are variants of previous ideas associated with Context Logic. We also introduce some simple inductive predicates for working with sets of locations.

4.1 Environment

Whilst our update language uses variables to denote node sets widely, it does not refer directly to individual node identifiers. For our Hoare reasoning, however, it is essential to both mention and quantify over these identifiers. We define a set of node variables $\text{Var}_N = \{m, n, \dots\}$ and an environment $e : \text{Var}_N \rightharpoonup_{\text{fin}} \mathcal{N}$. To allow comparisons between node sets and identifiers, we also introduce extended link expressions $\hat{E}_L$:

$$\hat{E}_L ::= E_L \mid \{n\} \mid \hat{E}_L \cup \hat{E}_L \mid \hat{E}_L \cap \hat{E}_L$$

with valuations $[\![\hat{E}_L]\!]es$ defined as expected.

4.2 Logic

Context Logic consists of two assertion languages: assertions on trees $P \in \mathcal{P}$, and assertions on contexts $K \in \mathcal{K}$. Both languages include standard assertions

Tree Assertions	**Context Assertions**
$e, s, T \vDash K(P)$ iff $\exists C, T'.(T \equiv C(T') \wedge e, s, C \vDash K \wedge e, s, T' \vDash P)$	$e, s, C \vDash P_1 \rhd P_2$ iff $\forall T.(e, s, T \vDash P_1 \wedge C(T)$ well-formed $\Rightarrow e, s, C(T) \vDash P_2)$
$e, s, T \vDash K \lhd P$ iff $\forall C.(e, s, C \vDash K \wedge C(T)$ well-formed $\Rightarrow e, s, C(T) \vDash P)$	$e, s, C \vDash -$ iff $C \equiv -$
$e, s, T \vDash [\![P]\!]$ iff $\exists T'.(T \simeq T' \wedge e, s, T' \vDash P)$	$e, s, C \vDash E_{A(n:\hat{E}_L)}[K]$ iff $\exists C'.(e, s, C' \vDash K \wedge C \equiv [\![E_A]\!]s_{([\![n]\!]e, [\![\hat{E}_L]\!]es)}[C'])$
$e, s, T \vDash E_T$ iff $T = [\![E_T]\!]s$	
$e, s, T \vDash E_A = E'_A$ iff $[\![E_A]\!]s = [\![E'_A]\!]s$	$e, s, C \vDash P \mid K$ iff $\exists T, C'.(C \equiv T \mid C' \wedge e, s, T \vDash P \wedge e, s, C' \vDash K)$
$e, s, T \vDash \hat{E}_L = \hat{E}'_L$ iff $[\![\hat{E}_L]\!]es = [\![\hat{E}'_L]\!]es$	
$e, s, T \vDash \hat{E}_L \bumpeq q(.)$ iff $q(s, T) = [\![\hat{E}_L]\!]es$	

Fig. 2. Formula Semantics for the Structural and Specific Assertions

from Boolean First-order Logic, structural assertions for analysing the tree and context structure which are novel to Context Logic, and specialised assertions associated with our tree model. The assertions are defined by the grammars:

$$\begin{array}{rll}
P ::= & K(P) \mid K \lhd P & \text{structural assertions} \\
& [\![P]\!] \mid E_T & \multirow{2}{*}{$\left.\begin{array}{l}\\ \\ \end{array}\right\}$ specific assertions} \\
& E_A = E_A \mid \hat{E}_L = \hat{E}_L \mid \hat{E}_L \bumpeq q(.) & \\
& P \Rightarrow P \mid \text{false} & \text{classical assertions} \\
& \exists x.P \mid \exists l.P \mid \exists a.P \mid \exists n.P & \text{quantifiers} \\
K ::= & P \rhd P \mid - & \text{structural assertions} \\
& E_{A(n:\hat{E}_L)}[K] \ \mid \ P \mid K & \text{specific assertions} \\
& K \Rightarrow K \mid \text{False} & \text{classical assertions} \\
& \exists x.K \mid \exists l.K \mid \exists a.K \mid \exists n.K & \text{quantifiers}
\end{array}$$

The semantics of the structural and specific assertions is given in Fig.2, using an overloaded satisfaction relation $\vDash$ defined on environments, stores, and either trees (for $\mathcal{P}$) or contexts (for $\mathcal{K}$). The structural assertions form the essence of Context Logic. The *application formula* $K(P)$ specifies that the given tree can be split into a context satisfying K applied to a subtree satisfying P. For example, if True $\triangleq$ False $\Rightarrow$ False, then the formula True(P) states that some subtree satisfies property P. We also have the two right adjoints of application. The first, $K \lhd P$, is satisfied by a given tree if, whenever we successfully insert the tree into a context satisfying K, then the resulting tree satisfies P. For example, the formula True $\lhd P$ states that, whatever context we place the given tree in, the result will satisfy P. Meanwhile, the other adjoint, $P_1 \rhd P_2$, is satisfied by a given *context* if, whenever we successfully insert in it a subtree satisfying P_1, then the resulting tree satisfies P_2. For example, if true $\triangleq$ false $\Rightarrow$ false, then the context formula true $\rhd P_2$ states that, regardless of what subtree is put in the given context, the result will satisfy P_2. This adjoint is essential for expressing weakest preconditions of update. The assertion $-$ specifies the empty context.

The assertions of Boolean First-order Logic are standard, and hence are not given in Fig.2. We use the derived connectives $\neg$, $\wedge$ and $\vee$. We extentially quantify over the value variables (trees, links, labels) and node variables, and derive $\forall$. Much of the interest in Context Logic lies in the interplay between the structural and classical assertions. For example, we will often use the following derived formulæ:

- $\Diamond P \triangleq \mathrm{True}(P)$ specifies that somewhere property P holds;
- $P_1 \blacktriangleright P_2 \triangleq \neg(P_1 \rhd \neg P_2)$ specifies that *there exists* a tree satisfying P_1 such that, when it is put in the hole of the given context, the result satisfies P_2;
- $K \blacktriangleleft P_2 \triangleq \neg(K \lhd \neg P_2)$ specifies that *there exists* a context satisfying K such that, when the given tree is put in the hole, the result satisfies P_2;
- $P_1 \vdash P_2 \triangleq (- \wedge (P_1 \rhd P_2))(\mathrm{true})$ specifies that whenever a tree satisfies P_1 then it also satisfies P_2.

The specific tree and context assertions are specialised to our application of tree update. The context assertions correspond directly to the context structure of trees: the assertion $E_{A(n:\hat{E}_L)}[K]$ specifies that the context has a top node described by $E_{A(n:\hat{E}_L)}$ and a subcontext satisfying K; the assertion $P \,|\, K$ specifies that the context can be split horizontally into a tree satisfying P and a context satisfying K. From these we define the analogous tree assertions, writing $P \,|\, Q$ for $(P \,|\, -)(Q)$ and $E_{A(n:\hat{E}_L)}[Q]$ for $(E_{A(n:\hat{E}_L)}[-])(Q)$. An additional assertion $K \,|\, P$, corresponding to parallel composition on the right, is not required as our trees are commutative; adding it, however, allows us to easily extend the logic to handle non-commutative trees.

In addition, we have the following specific tree assertions: a renaming modality $[\![P]\!]$, satisfied by any tree that is equivalent modulo renaming of node identifiers to one satisfying P (we shall use this to reason about the tree lookup and update commands); tree expressions E_T, satisfied if the current tree is structurally equal to the value of E_T; and equality assertions $E_A = E_A$, $\hat{E}_L = \hat{E}_L$ and $\hat{E}_L \doteqdot q(.)$. The equality assertions $E_A = E_A$ and $\hat{E}_L = \hat{E}_L$ are 'pure' assertions, that are independent of the current tree and therefore context-insensitive: $K((E = E') \wedge P) \Leftrightarrow (E = E') \wedge K(P)$. This property is very useful in derivation proofs. We do not have explicit assertions for equality and equality modulo renaming of tree expressions as they are derivable:

$$E_T = F_T \triangleq E_T \vdash F_T \qquad E_T \simeq F_T \triangleq E_T \vdash [\![F_T]\!].$$

Our final equality $\hat{E}_L \doteqdot q(.)$ holds if the query q returns the same value as $\hat{E}_L$ when evaluated on the current tree. Note that, although $\hat{E}_L \doteqdot q(.)$ is not pure, the locality of our queries means that if $\hat{E}_L \doteqdot q(.)$ holds for a tree, then it holds for any larger tree: that is, $K((\hat{E}_L \doteqdot q(.)) \wedge P) \Rightarrow (\hat{E}_L \doteqdot q(.)) \wedge K(P)$.

To conclude, we give some other useful derived formulæ:

- $n \in \hat{E}_L \triangleq (\hat{E}_L \cap \{n\}) = \{n\}$

- $n \in E_T \triangleq E_T \vdash \exists a, k. \Diamond a_{n:k}[\text{true}]$
- $\hat{E}_L = \hat{F}_L \uplus \hat{G}_L \triangleq (\hat{E}_L = \hat{F}_L \cup \hat{G}_L) \wedge (\hat{F}_L \cap \hat{G}_L = \emptyset)$
- $|\hat{E}_L| = |\hat{F}_L| \triangleq \exists x, y.\, x \simeq y \wedge \forall n.(n \in \hat{E}_L \Leftrightarrow n \in x \wedge n \in \hat{F}_L \Leftrightarrow n \in y)$
- $\Diamond \hat{E}_L \triangleq \forall n.(n \in \hat{E}_L \Rightarrow \exists a, k. \Diamond a_{n:k}[\text{true}])$

The first four assertions are all 'pure' and are fairly self-explanatory. The last assertion specifies that all the locations in $\hat{E}_L$ are in the tree.

4.3 Inductive Predicates

Our update commands typically act on a set of locations, which may be disjoint from each other or nested. In order to reason about such updates, we use a simple form of inductive predicate which is expressive enough for our purposes and corresponds to a fold-like operator on node sets. An alternative choice is to add full recursion to our logic, but we believe our approach is simpler for this specific task.

We define an inductive predicate *fold* with three arguments—a function $P_f : \mathcal{N} \times \mathcal{P} \to \mathcal{P}$, a base case assertion P_0, and a link expression $\hat{E}_L$ denoting the set of locations over which induction occurs:

$$\begin{aligned}\text{fold } P_f\ P_0\ \hat{E}_L \triangleq\ & (\hat{E}_L = \emptyset) \wedge P_0) \vee \\ & (\exists n, l.\ (\hat{E}_L = l \uplus \{n\}) \wedge P_f(n, \text{fold } P_f\ P_0\ l)\end{aligned}$$

The fold uses the locations from $\hat{E}_L$, one by one in any order, to expand P_f recursively, with P_0 serving as the base case for when $\hat{E}_L$ is empty. This definition is well-founded since $\hat{E}_L$ denotes a finite set of locations. Note that the arbitrariness of the expansion order means that there is an alternative, equivalent definition of fold, which expands the fold in the other direction:

$$\begin{aligned}\text{fold } P_f\ P_0\ \hat{E}_L \triangleq\ & (\hat{E}_L = \emptyset) \wedge P_0) \vee \\ & (\exists n, l.\ (\hat{E}_L = l \uplus \{n\}) \wedge \text{fold } P_f\ P_f(n, P_0)\ l)\end{aligned}$$

As an example, consider the following fold assertion which we shall later use to reason about dispose:

$$\text{fold } (\lambda n, Q.\ \exists a, k, y.(a_{n:k}[y] \blacktriangleright Q)(0))\ P_0\ l$$

This assertion is satisfied by a tree if it is possible to add to the tree subtrees with root nodes n, for each n in l, and obtain a tree satisfying base assertion P_0. Viewing this from the opposite direction, the assertion simply describes the result of disposing the subtrees at all the nodes in l, on a tree satisfying P_0.

For lookup and new, we require a more expressive fold predicate, which computes values of type τ as the recursion unfolds (where τ is either $\mathcal{T}$, $\mathcal{L}$ or $\mathcal{A}$). We define an inductive predicate *fold-val* which this time takes four arguments—a function $P_f : \tau \times \mathcal{N} \times (\tau \to \mathcal{P}) \to \mathcal{P}$, a base case assertion

$P_0 : \tau \rightarrow \mathcal{P}$, a starting value $E_0 : \tau$, and a link expression $\hat{E}_L$:

$$\begin{aligned}\text{fold-val } P_f \; P_0 \; E_0 \; \hat{E}_L \triangleq {} & ((\hat{E}_L = \emptyset) \wedge P_0(E_0)) \vee \\ & (\exists n, l. (\hat{E}_L = l \uplus \{n\}) \wedge P_f(E_0, n, \lambda x. \text{fold-val } P_f \; P_0 \; x \; l)\end{aligned}$$

Like fold, this uses the locations from $\hat{E}_L$ to expand P_f recursively, but in addition passes values between calls with a starting value E_0. As an example, consider the following assertion used to reason about tree lookup:

$$\text{fold-val } (\lambda y', n, Q. \begin{pmatrix} \exists a, k, y_0, y_1. \; \Diamond a_{n:k}[y_1] \wedge \\ (y' \vdash [\![a_{n:k}[y_1]]\!] | y_0) \wedge Q(y_0) \end{pmatrix}) \; (\lambda y'. \; y' = 0) \; y \; l$$

This specifies that the variable y denotes a concatenation (with renaming) of the trees $a_{n:k}[y]$ at the nodes n in l, with a base value of 0 for when l is empty.

5 Program Logic

We present a Hoare Logic for reasoning locally about our tree-update language. As described in the Introduction, the disjointed nature of our tree update, involving multiple, possibly nested locations, prevents us from using a small-axiom specification. Nevertheless, we show that local Hoare reasoning is still possible, using a different axiom style. We present the axioms, give the corresponding weakest preconditions, and show that the derivations of the weakest preconditions have a highly uniform structure. We also show that our axiom style is comparatively easy to use, illustrating our ideas using the ‘collapse-all-links’ program from Sect. 3.5.

5.1 Hoare Logic

Our Hoare Logic uses a fault-avoiding interpretation of Hoare triples: triple $\{P\} \; \mathbb{C} \; \{Q\}$ holds iff, when $\mathbb{C}$ is run in a state satisfying P, then $\mathbb{C}$ doesn't fault and the resulting state satisfies Q. We use four inference rules: Consequence, Auxiliary Variable Elimination, Sequencing and the Frame Rule. The first three are standard [9]. The key rule for local reasoning is the Frame Rule:

$$\textbf{Frame Rule:} \quad \frac{\{P\} \; \mathbb{C} \; \{Q\}}{\{K(P)\} \; \mathbb{C} \; \{K(Q)\}} \quad \text{Mod}(\mathbb{C}) \cap \text{FV}(K) = \{\}$$

where $\text{Mod}(\mathbb{C})$ contains the variables modified by the command $\mathbb{C}$, and $\text{FV}(K)$ is the set of free variables in K. The Frame Rule formalises the idea of local behaviour, stating that if a tree P is sufficient for the faultless execution of a command, then any additional tree structure, introduced using a context assertion K, is unaltered by that command. The soundness of the Frame Rule depends on the locality of the commands. All the commands in our update language are local.

Assign:

$$\{(m = E_L) \wedge 0\}\ l := E_L\ \{(l = m) \wedge 0\}$$

$$\{(m \mathrel{\overset{\frown}{-}} q(.)) \wedge x\}\quad l := q\quad \{(l = m) \wedge x\}$$

Update:

$$\{\Diamond l \wedge x\}\quad \text{dispose at } l \quad \{\text{fold } (\lambda n, Q.(\exists a, k, y.(a_{n:k}[y] \blacktriangleright Q)(0)))\ x\ l\}$$

$$\{\Diamond l \wedge x\}\quad \text{dispose-links at } l \quad \{\text{fold } (\lambda n, Q.(\exists a, k, y.(a_{n:k}[y] \blacktriangleright Q)(a_{n:\emptyset}[y])))\ x\ l\}$$

$$\{\Diamond l \wedge x\}\quad \text{append } E_T \text{ at } l \quad \{\text{fold } (\lambda n, Q.(\exists a, k, y.(a_{n:k}[y] \blacktriangleright Q)(a_{n:k}[y | [\![E_T]\!]])))\ x\ l\}$$

$$\{\Diamond l \wedge x\}\quad \text{append-links } E_L \text{ at } l \quad \{\text{fold } (\lambda n, Q.(\exists a, k, y.(a_{n:k}[y] \blacktriangleright Q)(a_{n:k \cup E_L}[y])))\ x\ l\}$$

$$\{\Diamond l \wedge x\}\quad \text{set-labels } E_A \text{ at } l \quad \{\text{fold } (\lambda n, Q.(\exists a, k, y.(a_{n:k}[y] \blacktriangleright Q)((E_A)_{n:k}[y])))\ x\ l\}$$

Lookup: $\mathbb{C} = y := \text{get-trees at } l$

$$\{\Diamond l \wedge x\}\ \mathbb{C} \left\{ \begin{array}{l} \text{fold-val } (\lambda y', n, Q. \left(\begin{array}{l} \exists a, k, y_0, y_1.\ \Diamond a_{n:k}[y_1] \wedge \\ \quad (y' \vdash [\![a_{n:k}[y_1]]\!] | y_0) \wedge Q(y_0) \end{array} \right)) \\ \qquad\qquad (\lambda y'.\ y' = 0)\ y\ l) \wedge x \end{array} \right\}$$

New: $\mathbb{C} = k := \text{new } E_A \text{ at } l$

$$\{\Diamond l \wedge x\}\ \mathbb{C} \left\{ \begin{array}{l} \text{fold-val } (\lambda k', n, Q. \left(\begin{array}{l} \exists m, k_0, a, k, y.\ (k' = k_0 \uplus \{m\}) \wedge \\ \quad (a_{n:k}[y] \blacktriangleright Q(k_0))(a_{n:k}[y \mid E_{A(m:\emptyset)}[0]]) \end{array} \right)) \\ \qquad\qquad (\lambda k'.\ (k' = \emptyset) \wedge x)\ k\ l \end{array} \right\}$$

Move:

$$\left\{ \begin{array}{l} (l = \{n\}) \wedge (l' = \{n'\}) \wedge \\ \quad (0 \triangleright \Diamond a'_{n':k'}[y])(a_{n:k}[z]) \wedge x \end{array} \right\} \text{ move } l \text{ to } l' \left\{ \begin{array}{l} (a'_{n':k'}[y] \blacktriangleright (a_{n:k}[z] \blacktriangleright x)(0)) \\ \quad (a'_{n':k'}[y \mid a_{n:k}[z]]) \end{array} \right\}$$

Fig. 3. (Selected) Command Axioms

5.2 Command Axioms

A representative selection of command axioms is presented in Fig.3. Compare the two axioms given for assignment. The first axiom is 'small' and standard: the precondition says that the tree is empty and that variable m has value $\hat{E}_L$; the postcondition says that the tree is still empty, and l has the value of m. The axiom for query evaluation is similar, but subtly different: the precondition now specifies that the tree has a value given by a variable x, and that variable m contains the result of executing the query q; the postcondition specifies that the tree has not changed, and that l has the value of m. The key change is that, not knowing the footprint of the query, the axiom cannot be small. Instead, the axiom is now defined on an *arbitrary* tree, and the equality $m \mathrel{\overset{\frown}{-}} q(.)$ acts as a *safety condition* ensuring that the query can be executed on that tree. The other axioms all follow this same uniform structure. The preconditions all specify a safety condition enabling the command to succeed and assign x to the given tree. The postconditions are all the strongest possible, specifying that the resulting tree has been changed as stipulated by the command: in other words, that it is possible to undo all the updates and return to the original tree x.

Consider the update commands, which we have presented in full to emphasise their uniformity. The precondition $\Diamond l$ specifies the safety property that all the update locations must be in the tree. The postcondition specifies that the resulting tree is simply the one obtained by doing the updates: in other words, that we can replace all the updated subtrees by the original subtrees $a_{n:k}[y]$, for each $n \in l$, to obtain the original tree x. The axioms for lookup and new are similar, but use fold-val instead of fold. For lookup, the postcondition states that the given tree is unchanged (the x), and that the value of y is composition of all the (renamed) subtrees $a_{n:k}[y_1]$ given by the locations $n \in l$; for new, the postcondition states that the resulting tree has fresh nodes m at each $n \in l$, with label E_A and no subtrees or links, and that these nodes are collected into a variable k.

Finally, consider the move axiom. The safety property is more complicated than the other cases (see Sect.3.4). It specifies that the source and destination variables l and l' evaluate to the single locations n and n' in the tree, and that n' cannot be under n. The postcondition states that the resulting tree has a node n under node n', and that it is possible to move n somewhere else (in fact to its original position) to obtain the original tree x.

5.3 Weakest Preconditions

We give the weakest preconditions, and derive them from the command axioms. This provides a completeness result for straightline code, and is useful in constructing verification tools. The weakest preconditions of the commands in Fig.3 are given in Fig.4 (and are explained below). Notice the strong duality between the command axioms and these weakest preconditions, with the latter often resembling the former 'in reverse'. However, whereas the axiom postconditions need only specify an existence property arising from the precondition, the weakest preconditions must describe *all* the possible trees leading to the postcondition P.

Consider the weakest preconditions for the update commands. Using fold, these specify that whenever it is possible to remove from the working tree the subtrees $a_{n:k}[y]$ for *each* $n \in l$ and replace them by the appropriate updated subtrees, then the resulting tree must satisfy the postcondition P. Note the use of the universal adjoint $R \triangleright Q$, compared with the existential adjoint $\blacktriangleright$ used in the specifications. For the dispose cases, this is not important since R is only satisfied by structurally-equal trees (it is 'exact'). However, $\triangleright$ is essential for tree-append, since the precondition must take into account all possible node renamings of the appended tree. Finally, notice that for update the fold assertion implies $\Diamond l$.

The weakest preconditions for lookup and new are slightly more complex. Unlike update, they require the explicit safety property $\Diamond l$. They must also factor in the renamings. The lookup precondition specifies that the postcondition must hold for all possible renamings y' of y, necessary since the composition of the subtrees obtained by the lookup command is defined up to renaming. The new precondition specifies that the postcondition must hold

Assign:

$$\{P[E_L/l]\}\ l := E_L\ \{P\}$$
$$\{\exists m.((m \doteqdot q(.)) \wedge P[m/l])\}\ \ l := q\ \ \{P\}$$

Update:

$$\begin{array}{rcl}
\{\text{fold } (\lambda n, Q.(\exists a,k,y.(0 \rhd Q)(a_{n:k}[y])))\ P\ l\} & \text{dispose at } l & \{P\} \\
\{\text{fold } (\lambda n, Q.(\exists a,k,y.(a_{n:\emptyset}[y] \rhd Q)(a_{n:k}[y])))\ P\ l\} & \text{dispose-links at } l & \{P\} \\
\{\text{fold } (\lambda n, Q.(\exists a,k,y.(a_{n:k}[y \mid [\![E_T]\!]] \rhd Q)(a_{n:k}[y])))\ P\ l\} & \text{append } E_T \text{ at } l & \{P\} \\
\{\text{fold } (\lambda n, Q.(\exists a,k,y.(a_{n:k\cup E_L}[y] \rhd Q)(a_{n:k}[y])))\ P\ l\} & \text{append-links } E_L \text{ at } l & \{P\} \\
\{\text{fold } (\lambda n, Q.(\exists a,k,y.((E_A)_{n:k}[y] \rhd Q)(a_{n:k}[y])))\ P\ l\} & \text{set-labels } E_A \text{ at } l & \{P\}
\end{array}$$

Lookup: $\mathbb{C} = y :=$ get-trees at l

$$\left\{ \Diamond l \wedge \forall y'. \left(\begin{array}{c} \text{fold-val } (\lambda y', n, Q. \left(\begin{array}{l} \exists a,k,y_0,y_1.\ \Diamond a_{n:k}[y_1] \wedge \\ \quad (y' \vdash [\![a_{n:k}[y_1]]\!] | y_0) \wedge Q(y_0) \end{array} \right)) \\ (\lambda y'.\ y' = 0)\ y'\ l \end{array} \right) \Rightarrow P[y'/y] \right\} \mathbb{C}\ \{P\}$$

New: $\mathbb{C} = k :=$ new E_A at l

$$\left\{ \Diamond l \wedge \forall l'. (|l'| = |l|) \Rightarrow \left(\begin{array}{c} \text{fold-val } (\lambda k', n, Q. \left(\begin{array}{l} \exists m, k_0.(k' = k_0 \uplus \{m\}) \wedge \\ \quad \exists a,k,y.(a_{n:k}[y | E_{A(m:\emptyset)}[0]] \\ \qquad \rhd Q(k_0))(a_{n:k}[y]) \end{array} \right)) \\ (\lambda k'.\ (k' = \emptyset) \wedge P[l'/k])\ l'\ l \end{array} \right) \right\} \mathbb{C}\ \{P\}$$

Move:

$$\left\{ \begin{array}{l} \exists n, n'.\ (l = \{n\}) \wedge (l' = \{n'\}) \wedge \exists a,k,y,a',k',z. \\ \quad (0 \rhd (a'_{n':k'}[y \mid a_{n:k}[z]] \rhd P)(a'_{n':k'}[y]))(a_{n:k}[z]) \end{array} \right\} \text{move } l \text{ to } l'\ \{P\}$$

Fig. 4. (Selected) Weakest Preconditions

for any fresh identifiers selected for k. The preconditions for assignment and move are straightforward.

The derivations of the weakest preconditions from the command axioms display a surprising level of regularity. As in [3], they consist of applying the right context assertion using the Frame Rule, then simplifying and eliminating auxiliary variables. For our axioms however, the right context assertion is simply the one which equates the initial tree x with the tree described in the weakest precondition: that is $(-\wedge(x \rhd P'))$, where P' is the part of the weakest precondition describing the tree. In each case, the weakest precondition follows immediately by Consequence. The derivations for the weakest preconditions in Fig.4 are given in full in the Appendix. We describe the derivations for the update cases here, as they are the most interesting.

The axiom postcondition and weakest precondition of a general update command are of the form fold $P_{\blacktriangleright}\ x\ l$ and fold $P_{\rhd}\ P\ l$, respectively, where

$$P_{\blacktriangleright} \triangleq (\lambda n, Q.(\exists a,k,y.(a_{n:k}[y] \blacktriangleright Q)(R(a,k,y))))$$
$$P_{\rhd} \triangleq (\lambda n, Q.(\exists a,k,y.(R(a,k,y) \rhd Q)(a_{n:k}[y])))$$

for an appropriate assertion $R(a,k,y)$. To derive the weakest precondition from the axiom, we equate x with the weakest precondition, applying the

$$\begin{array}{ccc}
\{\Diamond l \wedge x\} & & \{\text{fold } P_{\blacktriangleright}\ x\ l\} \\ \hline
\left\{\begin{array}{c}(- \wedge (x \triangleright \text{fold } P_{\triangleright}\ P\ l)) \\ (\Diamond l \wedge x)\end{array}\right\} & \text{Frame Rule} & \left\{\begin{array}{c}(- \wedge (x \triangleright \text{fold } P_{\triangleright}\ P\ l)) \\ (\text{fold } P_{\blacktriangleright}\ x\ l)\end{array}\right\} \\ \hline
\{\Diamond l \wedge x \wedge \text{fold } P_{\triangleright}\ P\ l\} & \Uparrow \text{Consequence} \Downarrow & \{\text{fold } P_{\blacktriangleright}\ (\text{fold } P_{\triangleright}\ P\ l)\ l\} \\ \hline
\{x \wedge \text{fold } P_{\triangleright}\ P\ l\} & \Uparrow \text{Consequence} \Downarrow & \{P\} \\ \hline
\{\text{fold } P_{\triangleright}\ P\ l\} & \text{Aux Vars} & \{P\}
\end{array}$$

Fig. 5. Derivation of Weakest Precondition for Update

frame $(- \wedge (x \triangleright \text{fold } P_{\triangleright}\ x\ l))$. We then simplify using the rules of Consequence and Auxiliary Variable Elimination. The resulting derivation is given in Fig. 5.

The key step is showing the implication $(\text{fold } P_{\blacktriangleright}\ (\text{fold } P_{\triangleright}\ P\ l)\ l) \Rightarrow P$. This follows from expanding the two folds appropriately. If $l \neq \emptyset$ then it is possible to expand the $P_{\triangleright}$ fold from the front and the $P_{\blacktriangleright}$ fold from back (as described in Sect. 4.3) to get:

$$\begin{array}{l}
\exists l_1, n_1.(l = l_1 \uplus n_1) \wedge \exists l_2, n_2.(l = l_2 \uplus n_2) \wedge \exists a, k, y, a', k', y'. \\
(\text{fold } P_{\blacktriangleright}\ (a_{n_1:k}[y] \blacktriangleright (R(a', k', y') \triangleright (\text{fold } P_{\triangleright}\ P\ l_2))(a'_{n_2:k'}[y']))(R(a,k,y))\ l_1)
\end{array}$$

In fact, it is possible to expand more intelligently so that $n_1 = n_2$. This is partly because none of the update results depend on the order of execution (though some orders may fail to execute). For most of the updates, the execution order does not matter at all. For tree dispose, however, the order does matter, since certain orders fail to execute: we cannot remove a tree after removing one of its ancestors. However, any successful executing order returns the same result, and since the last tree added (n_2) is guaranteed not to contain any of the earlier ones, it can safely be removed first (n_1).

We can therefore derive an expansion like before but with $n_1 = n_2$. Simplifying slightly further, we also get $a = a'$, $k = k'$ and $y = y'$. This gives us:

$$\begin{array}{ll}
& \exists l', n.(l = l' \uplus n) \wedge \exists a, k, y. \\
& (\text{fold } P_{\blacktriangleright}\ (a_{n:k}[y] \blacktriangleright (R(a,k,y) \triangleright (\text{fold } P_{\triangleright}\ P\ l'))(a_{n:k}[y]))(R(a,k,y))\ l') \\
\Rightarrow & \exists l', n.((l = l' \uplus n) \wedge \exists a, k, y. \\
& (\text{fold } P_{\blacktriangleright}\ (R(a,k,y) \triangleright (\text{fold } P_{\triangleright}\ P\ l'))(R(a,k,y))\ l') \\
\Rightarrow & \exists l', n.((l = l' \uplus n) \wedge \exists a, k, y.(\text{fold } P_{\blacktriangleright}\ (\text{fold } P_{\triangleright}\ P\ l')\ l')
\end{array}$$

with the first implication depending on the 'exact' nature of the formula $a_{n:k}[y]$, which can only be satisfied by one tree. Having shown that:

$$\text{fold } P_{\blacktriangleright}\ (\text{fold } P_{\triangleright}\ P\ l)\ l \wedge l \neq \emptyset \Rightarrow \exists l', n.((l = l' \uplus n) \wedge \text{fold } P_{\blacktriangleright}\ (\text{fold } P_{\triangleright}\ P\ l')\ l')$$

we conclude that $\text{fold } P_{\blacktriangleright}\ (\text{fold } P_{\triangleright}\ P\ l)\ l \Rightarrow \text{fold } P_{\blacktriangleright}\ (\text{fold } P_{\triangleright}\ P\ \emptyset)\ \emptyset$ by induction on l, which in turn implies P, by the definition of fold.

Backward Reasoning

$$\begin{array}{l}
\{P\} \\
\quad \text{dispose-links at } l \\
\{\text{dispose-links}_{\triangleright}\ l\ P\} \\
\quad \text{append } x \text{ at } l \\
\{\text{append}_{\triangleright}\ x\ l\ (\text{dispose-links}_{\triangleright}\ l\ P)\} \\
\quad x := \text{get-trees at } k \\
\left\{ \begin{array}{l} \Diamond k \land \forall y.((y = \text{get-trees } k) \Rightarrow \\ \quad \text{append}_{\triangleright}\ y\ l\ (\text{dispose-links}_{\triangleright}\ l\ P)) \end{array} \right\} \\
\quad k := l/\text{link::}{*} \\
\left\{ \begin{array}{l} \exists l'.(l' \bumpeq l/\text{link::} * (.)) \land \Diamond l' \land \\ \ \forall y.((y = \text{get-trees } l') \Rightarrow \\ \quad \text{append}_{\triangleright}\ y\ l\ (\text{dispose-links}_{\triangleright}\ l\ P)) \end{array} \right\}
\end{array}$$

Forward Reasoning

$$\begin{array}{l}
\{\exists l'.(l' \bumpeq l/\text{link::} * (.)) \land z\} \\
\quad k := l/\text{link::}{*} \\
\{(k \bumpeq l/\text{link::} * (.)) \land z\} \\
\quad x := \text{get-trees at } k \\
\left\{ \begin{array}{l} (k \bumpeq l/\text{link::} * (.)) \land (x = \text{get-trees } k) \\ \quad \land z \end{array} \right\} \\
\quad \text{append } x \text{ at } l \\
\left\{ \begin{array}{l} (k \bumpeq l/\text{link::} * (.)) \land (x = \text{get-trees } k) \\ \quad \land (\text{append}_{\blacktriangleright}\ x\ l\ z) \end{array} \right\} \\
\quad \text{dispose-links at } l \\
\left\{ \begin{array}{l} (x = \text{get-trees } k) \land \\ \quad (\text{dispose-links}_{\blacktriangleright}\ l\ (\text{append}_{\blacktriangleright}\ x\ l\ z)) \end{array} \right\}
\end{array}$$

Fig. 6. Program Reasoning on ‘collapse-all-links’

5.4 Program Reasoning Example

We now derive a weakest precondition and specification for the ‘collapse-all-links’ program of Sect.3.5. For clarity, we define syntactic sugar for the fold predicates used in the weakest preconditions of the constituent program commands:

$$x = \text{get-trees } \hat{E}_L \triangleq \text{fold-val } (\lambda y, n, Q. \begin{pmatrix} \exists a, k, y', y_0. \Diamond a_{n:k}[y'] \land \\ (y \vdash [\![a_{n:k}[y']]\!] | y_0) \land Q(y_0) \end{pmatrix}) \ (\lambda y.(y = 0))\ x\ \hat{E}_L$$

$$\text{append}_{\triangleright}\ E_T\ \hat{E}_L\ P_0 \triangleq \text{fold } (\lambda n, Q.(\exists a, k, y.(a_{n:k}[y \mid [\![E_T]\!]] \triangleright Q)(a_{n:k}[y])))\ P_0\ \hat{E}_L$$

$$\text{dispose-links}_{\triangleright}\ \hat{E}_L\ P_0 \triangleq \text{fold } (\lambda n, Q.(\exists a, k, y.(a_{n:\emptyset}[y] \triangleright Q)(a_{n:k}[y])))\ P_0\ \hat{E}_L$$

The assertion $(x = \text{get-trees } \hat{E}_L)$ specifies that x is a result of concatenating the subtrees at $\hat{E}_L$; similarly, $(\text{append}_{\triangleright}\ E_T\ \hat{E}_L\ P_0)$ and $(\text{dispose-links}_{\triangleright}\ \hat{E}_L\ P_0)$ state that whenever we append a tree E_T at the locations given by $\hat{E}_L$, or dispose of the links there, then P_0 holds.

The weakest precondition of the program follows immediately. Reasoning backwards, we obtain the left column of Fig.6. The last condition simplifies to:

$$\exists l', y.(l' \bumpeq l/\text{link::} * (.)) \land (y = \text{get-trees } l') \land \text{append}_{\triangleright}\ y\ l\ (\text{dispose-links}_{\triangleright}\ l\ P)$$

since appending gives the same result for trees that are equivalent modulo renaming. We are left with a natural statement of the weakest precondition, the safety condition $\exists l'.(l' \bumpeq l/\text{link::} * (.))$ stating that all the nodes in l, as well as any links from them, are in the tree.

It is also easy to compose the command axioms to get a specification of ‘collapse-all-links’. To illustrate this, we define more syntactic sugar, writing $(\text{append}_{\blacktriangleright}\ E_T\ \hat{E}_L\ P_0)$ and $(\text{dispose-links}_{\blacktriangleright}\ \hat{E}_L\ P_0)$ for the result of appending to or disposing the links of a tree satisfying P_0:

$$\begin{aligned}
\text{append}_{\blacktriangleright}\ E_T\ \hat{E}_L\ P_0 &\triangleq \text{fold}\ (\lambda n, Q.(\exists a, k, y.(a_{n:k}[y] \blacktriangleright Q)(a_{n:k}[y \,|\, [\![E_T]\!]]))))\ P_0\ \hat{E}_L \\
\text{dispose-links}_{\blacktriangleright}\ \hat{E}_L\ P_0 &\triangleq \text{fold}\ (\lambda n, Q.(\exists a, k, y.(a_{n:k}[y] \blacktriangleright Q)(a_{n:\emptyset}[y])))\ P_0\ \hat{E}_L
\end{aligned}$$

Like the precondition, the specification follows immediately. Reasoning forwards, and using the frame rule to derive safety preconditions, we obtain (in the right column of Fig.6) a specification in the same style as our axioms.

6 Conclusions

We have studied local Hoare reasoning based on Context Logic for a local tree-update language which combines update commands with queries. We highlighted the concept of local commands, which are standard for heap update but seem to be rarely used for tree update. We believe such local commands are a good design choice in their own right, as well as being essential for local reasoning.

Our reasoning about commands for updating multiple locations is a non-trivial extension of our initial work introducing Context Logic [3]. This is because the footprints of such commands are complex. In particular, it is not possible simply to adapt the small-axiom approach. We have shown that it is possible to do local Hoare reasoning, but our axioms are not small. Our results surprised us, since initially we were not sure that such reasoning was feasible at all.

We have not come to the end of the story. Our goal for this paper was to understand whether local Hoare reasoning was possible for our local tree-update language. Our next challenge is to see if it is possible to regain the small-axiom approach by moving to a more complex context structure. It is not possible to simply extend our reasoning to multi-holed contexts. Recall the 'dispose l' example, with variable l denoting the value $\{m, n\}$. The locations m and n might denote disjoint trees, in which case multi-holed contexts would probably work. However, m and n might also be one under the other, in which case multi-holed contexts will not work by themselves. Perhaps some simple additional 'wiring' structure might work. If so, our extended contexts could be a simple example of Milner's bigraphs [8]. Indeed, Sassone *et al.* have highlighted bigraphs as a good model for XML with additional cross-links precisely because of the additional context structure. They study BiLog [6,5], a logic for static bigraphs influenced in part by Separation Logic and Context Logic, but do not extend their reasoning to a tree-update language. In future, we will try to extend Context Logic in the simplest possible way to obtain, if possible, a small-axiom approach to local Hoare reasoning about tree update and compare this approach with a top-down approach starting with bigraphs.

References

[1] Abiteboul, S., P. Buneman and D. Suciu, “Data on the Web: from relations to semistructured data and XML,” Morgan Kaufmann, 1999.

[2] Biri, N., “Logiques spatiales de ressources, modèles d’arbres et applications,” Ph.D. thesis, Université Henri Poincaré (2005).

[3] Calcagno, C., P. Gardner and U. Zarfaty, *Context logic & tree update*, in: *POPL*, 2005.

[4] Cardelli, L., P. Gardner and G. Ghelli, *Querying trees with pointers*, unpublished notes, 2003; talk at APPSEM, 2001.

[5] Conforti, G., D. Macedonio and V. Sassone, *Bigraphical logics for XML*, in: *SEBD*, 2005, pp. 392–399.

[6] Conforti, G., D. Macedonio and V. Sassone, *Spatial logics for bigraphs*, in: *ICALP, LNCS 3520*, 2005, pp. 766–778.

[7] Isthiaq, S. and P. O’Hearn, *BI as an assertion language for mutable data structures*, in: *POPL*, 2001, pp. 14–26.

[8] Jensen, O. and P. Milner, *Bigraphs and mobile processes (revised)*, Technical report, University of Cambridge (2004).

[9] O’Hearn, P., J. Reynolds and H. Yang, *Local reasoning about programs that alter data structures*, in: *Computer Science Logic, LNCS 2142*, 2001, pp. 1–19.

[10] Reynolds, J., *Separation logic: a logic for shared mutable data structures*, in: *LICS*, 2002, Invited Paper.

[11] Tatarinov, I., Z. Ives, A. Halevy and D. Weld, *Updating XML*, in: *SIGMOD*, 2001, pp. 413–424.

[12] W3C, *XPATH: XML path language* (1999), *http://www.w3.org/TR/xpath/*.

[13] Yang, H. and P. O’Hearn, *A semantic basis for local reasoning*, in: *FOSSACS*, 2002, pp. 402–416.

$$
\begin{aligned}
\llbracket \pi_a :: \pi_f \rrbracket_{s,T}(L) &= \llbracket \pi_f \rrbracket_{s,T}(\llbracket \pi_a \rrbracket_{s,T}(L)) \\
\llbracket \pi_1 / \pi_2 \rrbracket_{s,T}(L) &= \llbracket \pi_2 \rrbracket_{s,T}(\llbracket \pi_1 \rrbracket_{s,T}(L)) \\
\llbracket \pi_1 \cup \pi_2 \rrbracket_{s,T}(L) &= \llbracket \pi_1 \rrbracket_{s,T}(L) \cup \llbracket \pi_2 \rrbracket_{s,T}(L) \\
\llbracket \pi_1 \cap \pi_2 \rrbracket_{s,T}(L) &= \llbracket \pi_1 \rrbracket_{s,T}(L) \cap \llbracket \pi_2 \rrbracket_{s,T}(L) \\
\llbracket \pi_1 - \pi_2 \rrbracket_{s,T}(L) &= \llbracket \pi_1 \rrbracket_{s,T}(L) \setminus \llbracket \pi_2 \rrbracket_{s,T}(L)
\end{aligned}
$$

$$
\begin{aligned}
\llbracket \mathrm{self} \rrbracket_{s,T}(L) &= L \\
\llbracket \mathrm{child} \rrbracket_{s,T}(L) &= \{\mathsf{n'} \mid \exists \mathsf{n} \in L.\, T \equiv C(\mathsf{a}_{\mathsf{n}:K'}[\mathsf{a'}_{\mathsf{n'}:K'}[T'] \mid T''])\} \\
\llbracket \mathrm{desc.} \rrbracket_{s,T}(L) &= \{\mathsf{n'} \mid \exists \mathsf{n} \in L.\, T \equiv C(\mathsf{a}_{\mathsf{n}:K}[C'(\mathsf{a'}_{\mathsf{n'}:K'}[T'])])\} \\
\llbracket \mathrm{parent} \rrbracket_{s,T}(L) &= \text{error if } \exists \mathsf{n} \in L.T \equiv \mathsf{a}_{\mathsf{n}:K}[T'] \mid T'' \\
&\quad\ \text{else } \{\mathsf{n'} \mid \exists \mathsf{n} \in L.\, T \equiv C(\mathsf{a'}_{\mathsf{n'}:K'}[\mathsf{a}_{\mathsf{n}:K}[T'] \mid T''])\} \\
\llbracket \mathrm{link} \rrbracket_{s,T}(L) &= \text{error if } \exists \mathsf{n'}, \mathsf{n} \in L.T \equiv C(\mathsf{a}_{\mathsf{n}:K}[T']) \wedge \mathsf{n'} \in K \wedge \mathsf{n'} \notin \mathrm{Nodes}(T) \\
&\quad\ \text{else } \{\mathsf{n'} \mid \exists \mathsf{n} \in L.\, T \equiv C(\mathsf{a}_{\mathsf{n}:K}[T']) \wedge \mathsf{n'} \in K\}
\end{aligned}
$$

$$
\begin{aligned}
\llbracket E_A \rrbracket_{s,T}(L) &= \{\mathsf{n} \in L \mid \mathsf{a} = \llbracket E_A \rrbracket s \wedge T \equiv C(\mathsf{a}_{\mathsf{n}:K}[T'])\} \\
\llbracket * \rrbracket_{s,T}(L) &= L
\end{aligned}
$$

$$
\llbracket \pi \rrbracket_{s,T}(\mathrm{error}) = \llbracket \pi_a \rrbracket_{s,T}(\mathrm{error}) = \llbracket \pi_f \rrbracket_{s,T}(\mathrm{error}) = \mathrm{error}
$$

Fig. A.1. Query Semantics

The following appendices contain a number of technical results referenced in the main text.

A Query Language Semantics

In this Appendix we present the semantics for the local query language described in Sect. 3.3. The language consists of:

$$
\begin{array}{rll}
q & ::= l/\pi \mid q \cup q \mid q \cap q \mid q - q & \text{query} \\
\pi & ::= \pi_a :: \pi_f \mid \pi/\pi \mid \pi \cup \pi \mid \pi \cap \pi \mid \pi - \pi & \text{path} \\
\pi_a & ::= \mathrm{self} \mid \mathrm{child} \mid \mathrm{descendant} \mid \mathrm{parent} \mid \mathrm{link} & \text{path axis} \\
\pi_f & ::= E_A \mid * & \text{label filter}
\end{array}
$$

The evaluation of a query is given by:

$$
(l/\pi)(s,T) = \begin{cases} \llbracket \pi \rrbracket_{s,T}(s(l)) & \text{if } s(l) \subseteq \mathrm{Nodes}(T) \\ \mathrm{error} & \text{otherwise} \end{cases}
$$

plus the obvious extension for the set operations, where $\llbracket \pi \rrbracket_{s,T}$ is a function defined in Fig.A.1. In essence, it either returns the set of identifiers obtained by following path π in the tree T starting from a given set of nodes, or it gives back an error if the path moves outside the tree. The path function $\llbracket \pi \rrbracket_{s,T}$ depends on the axis function $\llbracket \pi_a \rrbracket_{s,T}$, which finds the set of nodes associated with one step in the path, and the filter function $\llbracket \pi_f \rrbracket_{s,T}$, which filters the nodes depending on their labels.

Assign:

$$\frac{\llbracket E_L \rrbracket s = L}{l := E_L, s, T \ \rightsquigarrow\ [s|l \leftarrow L], T} \qquad \frac{q(s,T) = L}{l := q, s, T \ \rightsquigarrow\ [s|l \leftarrow L], T} \qquad \frac{q(s,T) = \text{error}}{l := q, s, T \ \rightsquigarrow\ \text{fault}}$$

Lookup: $\mathbb{C} = x :=$ get-trees at l

$$\frac{\llbracket l \rrbracket s = L \subseteq \text{Nodes}(T)}{\mathbb{C}, s, T \ \rightsquigarrow\ \mathbb{C}, L, [s|x \leftarrow 0], T} \qquad \frac{T \equiv C(\mathsf{a}_{\mathsf{n}:K}[T']) \quad T'' \simeq \mathsf{a}_{\mathsf{n}:\mathsf{K}}[T'] \quad (T'' \,|\, s(x)) \text{ well-formed}}{\mathbb{C}, \{\mathsf{n}\} \uplus L, s, T \ \rightsquigarrow\ \mathbb{C}, L, [s|x \leftarrow T'' \,|\, s(x)], T}$$

Dispose: $\mathbb{C} =$ dispose at l

$$\frac{\llbracket l \rrbracket s = L \subseteq \text{Nodes}(T)}{\mathbb{C}, s, T \ \rightsquigarrow\ \mathbb{C}, L, s, T} \qquad \frac{T \equiv C(\mathsf{a}_{\mathsf{n}:K}[T'])}{\mathbb{C}, \{\mathsf{n}\} \uplus L, s, T \ \rightsquigarrow\ \mathbb{C}, L, s, C(0)}$$

Append: $\mathbb{C} =$ append E_T at l

$$\frac{\llbracket l \rrbracket s = L \subseteq \text{Nodes}(T)}{\mathbb{C}, s, T \ \rightsquigarrow\ \mathbb{C}, L, s, T} \qquad \frac{T \equiv C(\mathsf{a}_{\mathsf{n}:K}[T']) \quad T'' \simeq \llbracket E_T \rrbracket s \quad C(\mathsf{a}_{\mathsf{n}:K}[T'] \,|\, T'') \text{ well-formed}}{\mathbb{C}, \{\mathsf{n}\} \uplus L, s, T \ \rightsquigarrow\ \mathbb{C}, L, s, C(\mathsf{a}_{\mathsf{n}:K}[T'] \,|\, T'')}$$

Move: $\mathbb{C} =$ move l to l'

$$\frac{\llbracket l \rrbracket s = \{\mathsf{n}\} \quad \llbracket l' \rrbracket s = \{\mathsf{n'}\} \quad T \equiv C(\mathsf{a}_{\mathsf{n}:K}[C'(\mathsf{a'}_{\mathsf{n'}:K'}[T'])])}{\mathbb{C}, s, T \ \rightsquigarrow\ s, C(\mathsf{a}_{\mathsf{n}:K}[C'(0) \,|\, \mathsf{a'}_{\mathsf{n'}:K'}[T']])}$$

$$\frac{\llbracket l \rrbracket s = \{\mathsf{n}\} \quad \llbracket l' \rrbracket s = \{\mathsf{n'}\} \quad T \equiv C(\mathsf{a}_{\mathsf{n}:K}[T'] \,|\, C'(\mathsf{a'}_{\mathsf{n'}:K'}[T'']))}{\mathbb{C}, s, T \ \rightsquigarrow\ s, C(\mathsf{a}_{\mathsf{n}:K}[T' \,|\, \mathsf{a'}_{\mathsf{n'}:K'}[T'']] \,|\, C'(0))} \qquad \frac{\text{otherwise}}{\mathbb{C}, s, T \ \rightsquigarrow\ \text{fault}}$$

New: $\mathbb{C} = l :=$ new E_A at l'

$$\frac{\llbracket l \rrbracket s = L \subseteq \text{Nodes}(T)}{\mathbb{C}, s, T \ \rightsquigarrow\ \mathbb{C}, L, [s|l \leftarrow \emptyset], T} \qquad \frac{T \equiv C(\mathsf{a}_{\mathsf{n}:K}[T']) \ \ \mathsf{a'} = \llbracket E_A \rrbracket s \ \ \mathsf{n'} \notin \text{Nodes}(T) \ \ L' = s(l) \cup \{\mathsf{n'}\}}{\mathbb{C}, \{\mathsf{n}\} \uplus L, s, T \ \rightsquigarrow\ \mathbb{C}, L, [s|l \leftarrow L'], C(\mathsf{a}_{\mathsf{n}:K}[T' \,|\, \mathsf{a'}_{\mathsf{n'}:\emptyset}[0]])}$$

Lookup, Dispose, Append & New: $\dfrac{\llbracket l \rrbracket s \not\subseteq \text{Nodes}(T)}{\mathbb{C}, s, T \ \rightsquigarrow\ \text{fault}} \qquad \dfrac{}{\mathbb{C}, \emptyset, s, T \ \rightsquigarrow\ s, T}$

Fig. B.1. (Selected) Operational Semantics

B Update Command Operational Semantics

The operational semantics of the update commands is given in Sect.3.4 is given in Fig.B.1. It uses an evaluation relation $\rightsquigarrow$, relating configuration triples $\mathbb{C}, s, T$ to terminal states s, T or 'fault'. For commands that act at multiple locations, we use *partial* computation states $\mathbb{C}, L, s, T$, which represent a state where the command $\mathbb{C}$ has yet to act on the locations in L. The semantics for these partial computation states is given in a non-deterministic fashion that does not specify the order of execution. Consider the rules for dispose, which removes the locations given by a query. The left-hand rule creates a partial computation state, with a location set given by the query. The right-hand rule picks a node from the location set, removes the tree at that node, and returns a new partial computation state. This continues until the location set is empty, when we return the final store and tree as the terminal state. The order in which the nodes are chosen does not matter, in that when we obtain an answer, it is always the same. Some orders never reach a termination state, since it is not possible to remove a location if any of its ancestors has already been removed. Our semantics is only concerned with the *existence* of a terminating order, and since our queries return locations in the tree, it is always possible to find at least one. The rules for append, lookup and new are similar.

C Derivations of Weakest Preconditions

The derivations for the weakest preconditions in Fig.4 are given in Figs. C.1 and C.2.

Assign:

$$
\begin{array}{rcl}
\multicolumn{3}{c}{\{(m = E_L) \wedge 0\}\ l := E_L\ \{(l = m) \wedge 0\}} \\
\hline
\{(0 \rhd P[m/l])((m = E_L) \wedge 0)\} & \mathrm{FR} & \{(0 \rhd P[m/l])((l = m) \wedge 0)\} \\
\hline
\{(m = E_L) \wedge P[m/l]\} & \mathrm{C} & \{(l = m) \wedge P[m/l]\} \\
\hline
\{(m = E_L) \wedge P[E_L/l]\} & \mathrm{C} & \{P\} \\
\hline
\{P[E_L/l]\} & \mathrm{AV} & \{P\}
\end{array}
$$

$$
\begin{array}{rcl}
\multicolumn{3}{c}{\{(m \bumpeq q(.)) \wedge x\}\ l := q\ \{(l = m) \wedge x\}} \\
\hline
\{(- \wedge (x \rhd P[m/l]))((m \bumpeq q(.)) \wedge x)\} & \mathrm{FR} & \{(- \wedge (x \rhd P[m/l]))((l = m) \wedge x)\} \\
\hline
\{(m \bumpeq q(.)) \wedge x \wedge P[m/l])\} & \mathrm{C} & \{(l = m) \wedge P[m/l]\} \\
\hline
\{(m \bumpeq q(.)) \wedge x \wedge P[m/l]\} & \mathrm{C} & \{P\} \\
\hline
\{\exists m.((m \bumpeq q(.)) \wedge P[m/l])\} & AV & \{P\}
\end{array}
$$

Lookup: $\mathbb{C} = z :=$ get-trees at l

$$
\begin{array}{rcl}
\{\Diamond l \wedge x\} & \mathbb{C} & \{(\text{fold-val } P_f\ P_0\ z\ l) \wedge x\} \\
\hline
\left\{\begin{array}{l} (- \wedge (x \rhd \forall y.(\text{fold-val } P_f\ P_0\ y\ l \\ \quad \Rightarrow P[y/z])))(\Diamond l \wedge x) \end{array}\right\} & \mathrm{FR} & \left\{\begin{array}{l} (- \wedge (x \rhd \forall y.(\text{fold-val } P_f\ P_0\ y\ l \\ \quad \Rightarrow P[y/z]))) \\ \quad ((\text{fold-val } P_f\ P_0\ z\ l) \wedge x) \end{array}\right\} \\
\hline
\left\{\begin{array}{l} \Diamond l \wedge x \wedge \forall y. \\ \quad (\text{fold-val } P_f\ P_0\ y\ l \Rightarrow P[y/z]) \end{array}\right\} & \mathrm{C} & \{P\} \\
\hline
\{\Diamond l \wedge \forall y.(\text{fold-val } P_f\ P_0\ y\ l \Rightarrow P[y/z])\} & \mathrm{AV} & \{P\}
\end{array}
$$

where $P_f \triangleq (\lambda y, n, Q.(\exists a, k, y', y_0.\ \Diamond a_{n:k}[y'] \wedge (y \vdash [\![a_{n:k}[y']]\!]|y_0) \wedge Q(y_0)))$
and $P_0 \triangleq (\lambda y.(y = 0))$

Update:

$$
\begin{array}{rcl}
\{\Diamond l \wedge x\} & \mathbb{C} & \{\text{fold } P_{\blacktriangleright}\ x\ l\} \\
\hline
\{(- \wedge (x \rhd \text{fold } P_{\rhd}\ P\ l))(\Diamond l \wedge x)\} & \mathrm{FR} & \{(- \wedge (x \rhd \text{fold } P_{\rhd}\ P\ l))(\text{fold } P_{\blacktriangleright}\ x\ l)\} \\
\hline
\{\Diamond l \wedge x \wedge \text{fold } P_{\rhd}\ P\ l\} & \mathrm{C} & \{\text{fold } P_{\blacktriangleright}\ (\text{fold } P_{\rhd}\ P\ l)\ l\} \\
\hline
\{x \wedge \text{fold } P_{\rhd}\ P\ l\} & \mathrm{C} & \{P\} \\
\hline
\{\text{fold } P_{\rhd}\ P\ l\} & \mathrm{AV} & \{P\}
\end{array}
$$

where $P_{\blacktriangleright} \triangleq (\lambda n, Q.(\exists a,k,y.(a_{n:k}[y] \blacktriangleright Q)(R(a, k, y))))$ and
$P_{\rhd} \triangleq (\lambda n, Q.(\exists a,k,y.(R(a,k,y) \rhd Q)(a_{n:k}[y])))$ for appropriate $R(a, k, y)$

Fig. C.1. Derivations (1)

Move: $\mathbb{C} = \text{move } l \text{ to } l'$

$$\begin{array}{ccc}
\left\{\begin{array}{l} (l=\{m\})\wedge(l'=\{n\})\wedge x\wedge \\ \quad (0\triangleright\Diamond a_{n:k}[\text{true}])(a'_{m:k'}[\text{true}]) \end{array}\right\} & \mathbb{C} & \left\{\begin{array}{l} (a_{n:k}[y]\blacktriangleright((a'_{m:k'}[z])\blacktriangleright x)(0)) \\ \quad (a_{n:k}[y\mid a'_{m:k'}[z]]) \end{array}\right\} \\ \hline
\left\{\begin{array}{l} (-\wedge(x\triangleright(0\triangleright((a_{n:k}[y\mid a'_{m:k'}[z]])\triangleright P) \\ \quad (a_{n:k}[y]))(a'_{m:k'}[z]))) \\ \qquad ((l=\{m\})\wedge(l'=\{n\})\wedge x\wedge \\ \qquad\quad (0\triangleright\Diamond a_{n:k}[\text{true}])(a'_{m:k'}[\text{true}]) \end{array}\right\} & \text{FR} & \left\{\begin{array}{l} (-\wedge(x\triangleright(0\triangleright((a_{n:k}[y\mid a'_{m:k'}[z]])\triangleright P) \\ \quad (a_{n:k}[y]))(a'_{m:k'}[z]))) \\ \qquad ((a_{n:k}[y]\blacktriangleright((a'_{m:k'}[z])\blacktriangleright x)(0)) \\ \qquad\quad (a_{n:k}[y\mid a'_{m:k'}[z]])) \end{array}\right\} \\ \hline
\left\{\begin{array}{l} (l=\{m\})\wedge(l'=\{n\})\wedge x\wedge \\ \quad (0\triangleright((a_{n:k}[y\mid a'_{m:k'}[z]])\triangleright P) \\ \qquad (a_{n:k}[y]))(a'_{m:k'}[z])\wedge \\ \qquad\quad (0\triangleright\Diamond a_{n:k}[\text{true}])(a'_{m:k'}[\text{true}]) \end{array}\right\} & \text{C} & \left\{\begin{array}{l} (a_{n:k}[y]\blacktriangleright((a'_{m:k'}[z])\blacktriangleright \\ \quad (0\triangleright((a_{n:k}[y\mid a'_{m:k'}[z]])\triangleright P) \\ \qquad (a_{n:k}[y]))(a'_{m:k'}[z])))(0)) \\ \qquad\quad (a_{n:k}[y\mid a'_{m:k'}[z]]) \end{array}\right\} \\ \hline
\left\{\begin{array}{l} (l=\{m\})\wedge(l'=\{n\})\wedge x\wedge \\ \quad (0\triangleright((a_{n:k}[y\mid a'_{m:k'}[z]])\triangleright P) \\ \qquad (a_{n:k}[y]))(a'_{m:k'}[z]) \end{array}\right\} & \text{C} & \left\{\begin{array}{l} (a_{n:k}[y]\blacktriangleright((a_{n:k}[y\mid a'_{m:k'}[z]])\triangleright P) \\ \quad (a_{n:k}[y]))(a_{n:k}[y\mid a'_{m:k'}[z]]) \end{array}\right\} \\ \hline
\left\{\begin{array}{l} (l=\{m\})\wedge(l'=\{n\})\wedge x\wedge \\ \quad (0\triangleright((a_{n:k}[y\mid a'_{m:k'}[z]])\triangleright P) \\ \qquad (a_{n:k}[y]))(a'_{m:k'}[z]) \end{array}\right\} & \text{C} & \{P\} \\ \hline
\left\{\begin{array}{l} \exists m,n.\,(l=\{m\})\wedge(l'=\{n\})\wedge\exists a,k,y, \\ \quad a',k',z.(0\triangleright((a_{n:k}[y\mid a'_{m:k'}[z]])\triangleright P) \\ \qquad (a_{n:k}[y]))(a'_{m:k'}[z]) \end{array}\right\} & \text{AV} & \{P\}
\end{array}$$

New: $\mathbb{C} = k := \text{new } E_A \text{ at } l$

$$\begin{array}{ccc}
\{\Diamond l\wedge x\} & \mathbb{C} & \left\{\text{fold-val } P_{\blacktriangleright}\ (\lambda l''.((l''=\emptyset)\wedge x))\ k\ l\right\} \\ \hline
\left\{\begin{array}{l} (-\wedge(x\triangleright\forall l'.(|l'|=|l|)\Rightarrow\text{fold-val } P_{\triangleright} \\ \quad (\lambda l''.((l''=\emptyset)\wedge P[l'/k]))\ l'\ l)) \\ \qquad (\Diamond l\wedge x) \end{array}\right\} & \text{FR} & \left\{\begin{array}{l} (-\wedge(x\triangleright\forall l'.(|l'|=|l|)\Rightarrow\text{fold-val } P_{\triangleright} \\ \quad (\lambda l''.((l''=\emptyset)\wedge P[l'/k]))\ l'\ l)) \\ \qquad (\text{fold-val } P_{\blacktriangleright}\ (\lambda l''.((l''=\emptyset)\wedge x))\ k\ l) \end{array}\right\} \\ \hline
\left\{\begin{array}{l} \Diamond l\wedge x\wedge\forall l'.(|l'|=|l|)\Rightarrow\text{fold-val} \\ \quad P_{\triangleright}\ (\lambda l''.((l''=\emptyset)\wedge P[l'/k]))\ l'\ l \end{array}\right\} & \text{C} & \left\{\begin{array}{l} \text{fold-val } P_{\blacktriangleright}\ (\lambda l''.((l''=\emptyset)\wedge \\ \quad \forall l'.(|l'|=|l|)\Rightarrow\text{fold-val } P_{\triangleright} \\ \qquad (\lambda l''.((l''=\emptyset)\wedge P[l'/k]))\ l'\ l))\ k\ l \end{array}\right\} \\ \hline
\left\{\begin{array}{l} \Diamond l\wedge x\wedge\forall l'.(|l'|=|l|)\Rightarrow\text{fold-val} \\ \quad P_{\triangleright}\ (\lambda l''.((l''=\emptyset)\wedge P[l'/k]))\ l'\ l \end{array}\right\} & \text{C} & \left\{\begin{array}{l} \text{fold-val } P_{\blacktriangleright}\ (\lambda l''.((l''=\emptyset)\wedge \\ \quad \text{fold-val } P_{\triangleright}\ (\lambda l''.((l''=\emptyset)\wedge P)) \\ \qquad k\ l))\ k\ l \end{array}\right\} \\ \hline
\left\{\begin{array}{l} \Diamond l\wedge x\wedge\forall l'.(|l'|=|l|)\Rightarrow\text{fold-val} \\ \quad P_{\triangleright}\ (\lambda l''.((l''=\emptyset)\wedge P[l'/k]))\ l'\ l \end{array}\right\} & \text{C} & \{P\} \\ \hline
\left\{\begin{array}{l} \Diamond l\wedge\forall l'.(|l'|=|l|)\Rightarrow\text{fold-val} \\ \quad P_{\triangleright}\ (\lambda l''.((l''=\emptyset)\wedge P[l'/k]))\ l'\ l \end{array}\right\} & \text{AV} & \{P\}
\end{array}$$

where $P_{\blacktriangleright} \triangleq (\lambda l'',n,Q.(\exists m,l_0.(l''=l_0\uplus\{m\})\wedge\exists a,k,y.(a_{n:k}[y]\blacktriangleright Q(l_0))(a_{n:k}[y\mid E_{A(m:\emptyset)}[0]])))$
and $P_{\triangleright} \triangleq (\lambda l'',n,Q.(\exists m,l_0.(l''=l_0\uplus\{m\})\wedge\exists a,k,y.(a_{n:k}[y\mid E_{A(m:\emptyset)}[0]]\triangleright Q(l_0))(a_{n:k}[y])))$

Fig. C.2. Derivations (2)