Transfer Report
Context Logic, Tree Update and Concurrency

Mark Wheelhouse

First Supervisor: Philippa Gardener
Second Supervisor: Cristiano Calcagno

August 18, 2008

1 Introduction

This Transfer Report comes at the end of the first 8 months of my PhD here at Imperial College London. The purpose of this report is to outline the work that I have carried out in my time so far, and also to give an indication of what I hope to include in my final PhD Thesis write up. The subsections of this Introduction are intended to give a brief outline of the basic structure for my final Thesis.

1.1 Featherweight DOM

The first 4 months of my time here were mainly focused on working with Smith, Gardner, and Zarfaty in our specification of Featherweight DOM, a minimal subset of Core Level 1 of the Document Object Model [25]. Our work of providing a formal specification of DOM, that could replace the existing English specification of DOM, had proved very successful and has been accepted to both the PlanX 2008 workshop, attached to the prestigious Principles of Programming Languages (POPL) conference and also to the Principles of Database Systems (PODS) 2008 conference. We have also been invited to attend a pair of workshops at Microsoft Research Cambridge on the 'Verification of Concurrent Algorithms' and 'The Rise and Rise of the Declarative Datacentre'. I attach for reference a copy of 'Local Hoare Reasoning about DOM' [16] which is the most recent paper of our work on DOM and was accepted to the PODS conference. The work in this paper extends our first paper on DOM [15] by adding textual data to the model of DOM that we provide. The direct continuation of the work on Featherweight DOM is to extend our model to cover all aspects of Core Level 1 of DOM. This will be the subject of Smith's thesis, but there is a chance for the work of my PhD to return to this area towards the completion of my own thesis as an application of my proposed work on reasoning about concurrent tree update. I am currently helping Gardner and Smith to supervise MSc student Adam Wright for his MSc project. He is going to be carrying out an investigation that will provide us with a greater insight into where best to turn our attention within the vast size of the full DOM specification and java-script.

1.2 Specifying Move

Since finishing our work on Featherweight DOM, I have moved onto concentrating on the specification of the move command in a tree structure. This command turns out to be the most complex of the basic tree update commands, and causes us to have to make several compromises in our current work. In the following sections of this report I shall be concentrating on explaining what problems our current model causes and I will also explain the initial steps I have undertaken to improve upon our work in this area. I have currently developed only a partial solution to the problems raised by the move command, but understanding how to tackle the problems that this command raises is fundamental to the successful continuation of our work on program reasoning, especially if we wish to move our reasoning into a concurrent setting. The process of refining my initial approach into a full solution is ongoing work that I hope to have finalised within the next 2 months.

1.3 Concurrent Tree Update

The rapid growth of the Internet over the last 2 decades has seen more and more programs being written in a concurrent setting. When multiple programs are running at the same time we have to take great care to ensure that they do not interfere with one another in a detrimental fashion, or mess up shared data structures. It can be very hard for people to keep all of the complex interactions of their programs in mind when writing new programs and program verification has found it is very well suited to help solve this problem. There has been a lot of research into specifying concurrent programs in the program verification field, but Context Logic's reasoning has so far been limited to the sequential setting. Once I am able to improve the specification of the move command so that it no longer requires the previous compromises in our reasoning, I will turn my attention to providing a reasoning framework for Context Logic in a concurrent setting. Separation Logic, a logic very close to Context Logic, has already made this move into concurrent settings, and I will be drawing significantly on their work to provide similar results for Context Logic. The work on specifying the move command will impact directly on our work on concurrent reasoning. The compromises that we make in our current work will severely impact on the capabilities of our concurrent reasoning, which is why we must address these issues first.

1.4 Links to Low Level Reasoning

Very recent work by Gardner and Zarfaty [17] has shown how we can translate from our high level reasoning about tree update into a low level implementation of that reasoning. Once I have provided a high level system of reasoning for a concurrent setting, it would be very interesting to see how that might be translated into a low level implementation. Care must be taken as it has already been shown that the translation strategy from [17] cannot just be applied blindly to a concurrent setting. The footprints of our commands, the parts of the data structure that they affect, may increase as we move from the high level to the low level. This means that in some cases where it is possible to run two commands in parallel at the high level, after translation to the low level implementation it is not any longer safe to run these commands in parallel. It is my suspicion that there should exist some low level implementation for which a translation from the high level does preserve the parallel computation property of programs.

If this is not the case then it should be possible to show that it is not possible to find such an implementation. Identifying whether such an implementation exists could lead to interesting results about links between high and low level reasoning.

1.5 Concurrent DOM

As I mentioned before, I am very much interested in returning to the specification of DOM during my PhD, but this time in a concurrent setting. Once we have successfully provided the concurrent specification for a simple tree update language, I intend to extend our work to the more complex language of Featherweight DOM and provide a concurrent specification of this language. I doubt that there will be sufficient time to extend a concurrent specification of Featherweight DOM to a concurrent specification of the full DOM Core Level 1, but I intend for that extension to be an obvious next step following from my thesis. One thing that we have so far left out of our reasoning about DOM, leaving it to the implementation to handle, is the existence of a garbage collection system for handling the deletion of parts of our data structure. Existing work has begun to investigate the behavior of update languages that use garbage collection [11] and within a concurrent setting we might be able to give an effective formal specification of the behavior of such a garbage handling system for DOM.

1.6 Implementation

An implementation of the current DOM specification is provided by Xerces2. This implementation handles the following sections of DOM,

- DOM Level 1 Core and HTML
- DOM Level 2 Core
- DOM Level 3 Core
- DOM Level 2 Traversal and Range
- DOM Level 2 Events: Mutation Events
- DOM Level 3 Load and Save

I am aiming to provide a specification of Concurrent Featherweight DOM that is good enough to pave the way for a future concurrent low level implementation of DOM.

1.7 Timeline

The proposed timeline for my PhD is given in Figure 1. Having already finished my part in our work on Featherweight DOM my next task is to finalise my work on specificing the tree update Move command. I expect this work to be finished by October this year, marking the end of my first years worth of work. I then intend to spend up to 3 months applying my work on move to the more complex data structure of Featherweight DOM. After revisiting Featherweight DOM, I plan to return to the world of simple tree update where I will spend about 6 months investigating applying reasoning about Concurrency to Context Logic. Once

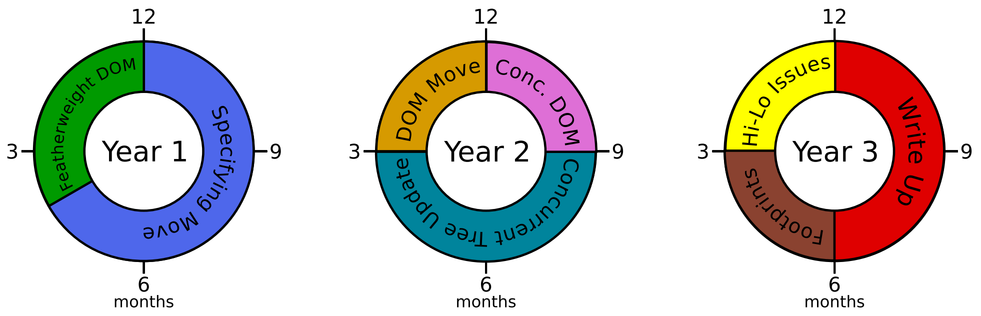

Figure 1: Proposed PhD Timeline

this stage of my work is complete, I will again spend about 3 months applying what I have learned to the more complex datastructure of Featherweight DOM to provide a concurrent specification of a DOM subsection. It is my hope that my work up to this point, especially the work with Featherweight DOM will actually take less time than I have allowed for it and any extra time gained can go towards the final parts of my PhD where I aim to spend at least 3 months apiece on looking at links between High and Low level reasoning and carrying out an evalutaion of when a command's footprint has really been captured by a specification. I plan to dedicate the last 6 months of my PhD to the write up of my Thesis.

1.8 Report Outline

I have attached a copy of 'Local Hoare Reasoning about DOM' to this report, which is our most up to date work on Featherweight DOM. The rest of this report will be arranged in the following structure. I will expand on the concepts discussed in the Introduction and provide a general overview of the background of program reasoning in section 2. In section 3 I will explain the motivation behind why I chose to begin the rest of my work by investigating the specification of the move command. I will then examine existing work on extending Context Logic in section 4. Having seen why it is necessary, I will then show the beginnings of my work on extending Context Logic to a multi-holed disjoint reasoning setting in section 5. Section 6 concludes this report with an outline of the work that I will be carrying out immediately over the next few months as well as some ideas about how my work will evolve over the next 2 years.

2 Background

Before we discuss the existing work on extending Context Logic, it is necessary to have an understanding of the concepts mentioned in the previous section. We will first give a brief outline of Program Reasoning and then discuss the introduction of Separation Logic and Context Logic into this area.

2.1 Program Reasoning

The field of Program Reasoning began with [18] in the late 60's. The idea of providing pre- and post-conditions for a command or program had been around before this work, but this was the first time an attempt had been made to provide a formal language for expressing these conditions. Moreover, the idea of then using these conditions to derive conditions of larger commands and programs was a revolutionary step. The motivation behind Hoare's pioneering work was that the cost of testing computer programs for correct performance was very high. Indeed, he points out in his first paper on the subject that,

> *'...the cost of an error in certain types of program may be almost incalculable - a lost spacecraft, a collapsed building, a crashed aeroplane, or a world war.'*

Instead, he suggested that people turn to mathematics to find ways of formally proving the properties that they want their programs to fulfill. It is from these very ideas that the field of program verification was born.

2.1.1 Axioms and Rules

The work in [18] centered around the identification of a core set of commands and the provision of axioms which described the behavior of those commands. When combined with a set of rules, this axiom set could then be used to propagate properties through the commands of a program without having to directly run its code

The axioms come in the form of a Hoare Triple $\{P\}\ \mathbb{C}\ \{Q\}$. If the assertion P is true before initiation of the program $\mathbb{C}$, then the assertion Q will be true on its completion. This general idea has held true, with a few minor changes, to this day. We shall formally define the notion of a Local Hoare Triple in section 5 when we show the first steps of our work.

The rules are then described in terms of these Hoare Triples. For example we have the Rule of Consequence:

$$\text{if } \{P\}\ \mathbb{C}\ \{Q\}\ \textit{and}\ Q \Rightarrow Q' \text{ then } \{P\}\ \mathbb{C}\ \{Q'\}$$

This simply states that if it can be shown that the post-condition Q of program $\mathbb{C}$ implies the assertion Q', then we can deduce that the assertion Q' will hold after the program has completed. This lets us weaken the post-condition of a program. There are similar rules for strengthening the pre-condition of a program and deducing the effects of running programs sequentially. We give a full set of rules for our axioms in section 5.

As an example let us consider a toy command `addOne`(x) which increments the value stored at x by 1. This would have an axiom resembling:

$$\{s(x) = v\}\ \texttt{addOne}(x)\ \{s(x) = v + 1\}$$

for some program store s. It is easy to see that $s(x) = v + 1 \Rightarrow s(x) > v$. So by the rule of consequence and the axiom above we can deduce the Hoare Triple:

$$\{s(x) = v\}\ \texttt{addOne}(x)\ \{s(x) > v\}$$

This might be a useful property to know when we are analysing the output of the program. From the above triple we can deduce that the output will always be positive so long as the input $v \geq 0$.

2.1.2 Footprint of a Command

An important thing to understand about a program is the notion of its footprint. This idea was first introduced informally in [19] describing the footprint of a program as

> *'...only those cells which are accessed by the program during execution'.*

For example, the command `delete(n)`, which deletes the subtree starting at node `n`, only accesses the nodes in that subtree. So the footprint of the `delete` command is the whole subtree at node `n`. On the other hand, we can see that a command that looks up some value at a node `m` will only access the node `m` and the value we are looking at. So a `look-up` command will have a much smaller footprint than a `delete` command. Recent work [23] has given a more mathematical definition and explored this area further, but the description given above is sufficient for this report.

2.2 Separation Logic

In 2001, the field of Program Reasoning took a new turn with the introduction of Separation Logic [19],[20]. Up until this point most formalisms had taken a global view of the whole program state when specifying programs. However, the authors of [20] had a different viewpoint, one of local reasoning. They summarise this as follows.

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

So the view here is one very much centered around specifying just the footprints of the atomic commands such that the rest of the data structure is unaffected. One can then make use of a set of rules to infer the behavior of other commands. This idea of local reasoning is only valid if the commands themselves are local, that is they do not require global information to successfully operate.

2.2.1 The Model

Separation Logic models the state of a program as the combination of two components. The first of these, the data store s, is a finite partial function that maps variables to their values. The second of these, the heap h, is an indexed set of memory cells, often viewed as tuples. For example, we would write **emp** to represent the empty heap with no cells, and $(x \mapsto 1, 2)$ to represent the single cell heap shown in figure 2.

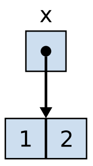

Figure 2: The heap $(x \mapsto 1, 2)$

The power of Separation Logic comes from the introduction of $*$ and $-\!\!*$, two new spatial connectives. $*$ decomposes the current resource into two separate pieces and $-\!\!*$ talks about new, or fresh, heaps. The conjunction $P * Q$ is true just when the current heap can be split into two components, one of which makes P true, and the other of which makes Q true. The implication $P -\!\!* Q$ talks about new pieces of heap that are disjoint from the current heap. The implication is true if for every new heap that makes P true, the combination by $*$ of this new heap and the current heap will result in a heap that makes Q true.

The reason that these connectives are powerful is their ability to move the reasoning from a global level to a local one. The crucial part of this step comes from the Frame Rule.

$$\text{F} \qquad \text{R} \quad : \quad \frac{\{P\}\ \mathbb{C}\ \{Q\}}{\{P * R\}\ \mathbb{C}\ \{Q * R\}} \quad \text{Modifies}(\mathbb{C}) \cap \text{Free}(R) = \{\}$$

This rule of inference simply states that if some program $\mathbb{C}$ behaves in a certain way in its local heap, then it will behave in the same way even if we extend the size of the heap it is operating in. The side conditions simply ensure that the program doesn't modify any of this extra heap that we have added. So this rule allows for program reasoning to be confined to just the cells that the program access, and indeed we can verify that the rest of the heap remains unchanged automatically.

2.2.2 Examples

To give a feeling of how Separation Logic is used when reasoning about programs and their data structures we give a few examples of common uses for separation logic formula.

a) A simple example of the $*$ connective in use is in the formula $(x \mapsto 1, y) * (y \mapsto 2, \varnothing)$. This style of formula is often used when expressing a null terminated linked list structure in the heap. Figure 3 shows what the heap for this formula looks like.

b) To see how we can reason locally, let us consider the command `set2`(x, v) which sets the second cell at x to some value v. This command would have a specification of the form:

$$\{x \mapsto a, b\}\ \texttt{set2}(x, v)\ \{x \mapsto a, v\}$$

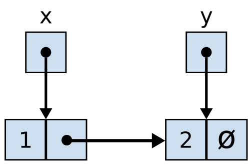

Figure 3: The heap $(x \mapsto 1, y) * (y \mapsto 2, \varnothing)$

Notice that this specification is described only over the cells that the `set2` command accesses. To use this specification in the context of a larger program we will need to be able to use the specification in a larger heap. The Frame Rule provides precisely this ability. We simply use the $*$ operator to separate out the cell at x when we wish to modify it. Figure 4 shows this in action.

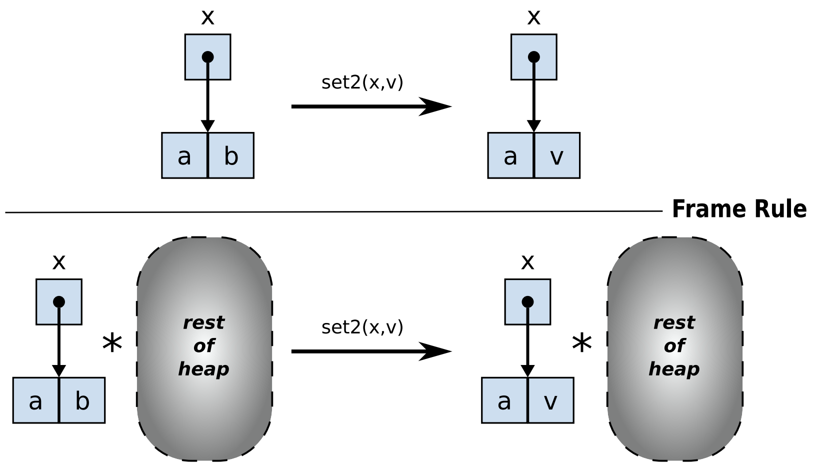

Figure 4: The Frame Rule in use.

c) The $-\!\!*$ connective is often used to talk about hypothetical properties of a heap. For example we might have the formula $(x \mapsto \_, 7) -\!\!* P$. This states that when x with second cell containing the value 7 is added to the current heap, then some property P will hold. The $\_$ simply says that we can have any value in the first cell of x. Lets assume instead that currently we have a heap that contains the cell $(x \mapsto 1, 2)$ and also has the property given above. Then we would write $(x \mapsto 1, 2) * ((x \mapsto \_, 7) -\!\!* P)$. We can see how this heap evolves when we run a `set2` command on it.

$$\{(x \mapsto 1, 2) * ((x \mapsto \_, 7) -\!\!* P)\}$$
$$\texttt{set2}(x, 7)$$
$$\{(x \mapsto 1, 7) * ((x \mapsto \_, 7) -\!\!* P)\}$$
$$\{P\}$$

This example shows the interaction between $*$ and $-\!\!*$.

2.2.3 Practical Program Verification

Local Reasoning and Separation Logic have proved very successful over the last few years and has inspired the creation of the successful reasoning tools SLAyer [24] and SpaceInvader [13] based off of their first tool the Smallfoot project [2]. Smallfoot makes use of Separation Logic to provide a system for automatic assertion checking in annotated programs. It chops up these programs into Hoare Triples for certain symbolic instructions and then checks that these triples hold true. This approach has yielded some interesting results. Most notably a subtle program termination error was found in a Windows device driver [3] and several memory leaks and memory safety bugs where found in the IEEE 1394 firewire device driver [1]. These are real program errors that had been missed by extensive testing. Finding these errors shows the practical advantages of using program verification to check that programs are correct rather than testing. The early identification of these errors has saved a great deal of time and money that would have been spent in the future when the effects of the errors were eventually noticed.

2.3 Single-Hole Context Logic

The view of a program's state, provided by the work of Separation Logic, is obviously a very low level one. The heap is a finite set of data cells, with pointers between them. Sometimes we want to think of a programs data structure at a higher level than this. A prime example is any program that deals with trees or graphs. One can use a heap structure to represent these higher level concepts, but in these cases a great deal of care must be taken to maintain the pointer structure of such objects, especially when we are moving nodes about within such structures. In 2004 Separation Logic and Ambient Logic [12] inspired the creation of Context Logic [7]. This new logic provided a way to tackle program reasoning at a higher level of abstraction whilst also maintaining the idea of local reasoning. The initial model has been expanded and improved to handle more complex data structures [8],[10] and expressivity has been analyised [9]. For this outline we shall cover just the basic model.

2.3.1 The Model

In a similar fashion to Separation Logic, Context Logic models the state of a program as the combination of two components. The first of these, just as in Separation Logic, is the data store *s*, a finite partial function mapping variables to their values. However, in the case of Context Logic, the data structure of the program is modeled in a direct high level fashion. So a language which operates over a tree carries a tree *t* with it. Similarly a graph *g* is carried by a program that operates over graphs. Much of our current work with Context Logic has concentrated on working with a tree like data structure which we define inductively as follows:

$$\text{tree} \quad \mathbf{T} \quad ::= \emptyset \mid \mathbf{n}[\mathbf{T}] \mid \mathbf{T}|\mathbf{T}$$

This is just a simple ordered tree structure, with each node having a unique identifier, but no further content. (It is a trivial step to extend the data structure, and subsequent reasoning, with some data content.) For example we write $\emptyset$ for the empty tree and $\mathbf{n}[\mathbf{m}[\emptyset] \mid \mathbf{p}[\emptyset]]$ to represent the small three node tree shown in figure 5.

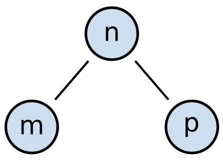

Figure 5: The tree $\mathbf{n}[\mathbf{m}[\varnothing] \mid \mathbf{p}[\varnothing]]$

When working with Single-Holed Context Logic, we also require the definition of a context structure that has the same shape as our data and contains a single context hole (_). We can place data in this context hole to obtain a complete data structure, a tree in this case.

$$\text{tree context} \quad \mathbf{C} ::= \_ \mid \mathbf{n}[\mathbf{C}] \mid \mathbf{C}|\mathbf{T} \mid \mathbf{T}|\mathbf{C}$$

Notice that we do not need to directly record the left/right sibling relationship between the nodes **m** and **p** as this information can be derived in the high level model due to the parallel composition connective |. For a formula of Separation Logic to record the same information we would have to add explicit pointers between all such sibling nodes. The Separation Logic formula $(\mathbf{n} \mapsto \mathbf{m}, \mathbf{p}) * (\mathbf{m} \mapsto \varnothing) * (\mathbf{p} \mapsto \varnothing)$ might seem to encode the same information, but consider what happens if we remove the node **n** from the the data structure. The Context Logic formula becomes $\mathbf{m}[\varnothing] \mid \mathbf{p}[\varnothing]$ and so we maintain the information that **m** and **p** are sibling nodes. However, in the Separation Logic case the formula becomes $(\mathbf{m} \mapsto \varnothing) * (\mathbf{p} \mapsto \varnothing)$ which only specifies that we have two disjoint nodes. We lose the information about the sibling relation of nodes **m** and **p**. Adding the explicit sibling pointers to the data structure would lead to a rather more complex formula for the tree, of the form $(\mathbf{n} \mapsto \varnothing, \mathbf{m}, \mathbf{p}, \varnothing) * (\mathbf{m} \mapsto \varnothing, \mathbf{p}, \varnothing) * (\mathbf{p} \mapsto \mathbf{m}, \varnothing, \varnothing)$ which requires us to carefully maintain many more pointers. This simple example shows how moving our reasoning to a higher level can help us to overcome low-level issues, such as pointer update, and concentrate on more interesting features of such data structures.

As with Separation Logic, the power of local reasoning comes from the use of spatial connectives. For Context Logic these connectives are the application connective $\circ$, and its adjoints $\multimapdotinv$ and $\multimapdot$. We use context application $\circ$ to break apart parts of the data structure by pulling out some substructure and putting a context hole in its place. So in our tree setting a formula of the form $C \circ P$ would be satisfied by a tree that could be split into some tree context satisfying C and a tree satisfying P that we place in the context hole of C. (see Figure 6)

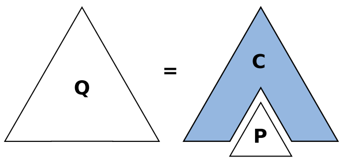

Figure 6: Context Application $Q = C \circ P$

The adjoint $C \multimapdotinv P$ would be satisfied by a tree that, whenever we insert it into a tree context satisfying C, then the resulting tree satisfies P. (see Figure 7)

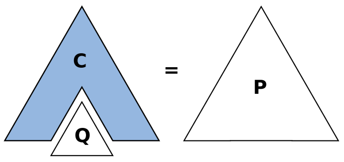

Figure 7: Left Adjoint $Q = C \circ\!- P$

Finally, the right adjoint $P \multimap Q$ would be satisfied by a context that, whenever we insert a tree satisfying P into it, then the resulting tree satisfies Q. (see Figure 8)

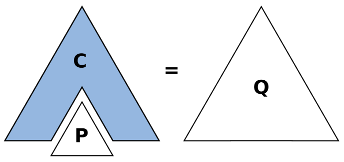

Figure 8: Right Adjoint $C = P \multimap Q$

Just as with Separation Logic, Context Logic has a Frame Rule which gives us the power to move our reasoning from the global level to a local level.

$$\text{F} \qquad \text{R} \quad : \quad \frac{\{P\}\ \mathbb{C}\ \{Q\}}{\{C \circ P\}\ \mathbb{C}\ \{C \circ Q\}} \quad \text{Modifies}(\mathbb{C}) \cap \text{Free}(C) = \{\}$$

2.3.2 Examples

To give a feeling of how we use Context Logic when carrying out program reasoning we give a few examples of common uses for Context Logic formula.

a) A simple example of the application connective $\circ$ in use is the formula $\mathbf{n}[\_] \circ (\mathbf{m}[\varnothing] \mid \mathbf{p}[\varnothing])$. This formula describes the small tree given in figure 5 where we have split off the node **n** into a tree context.

b) Our reasoning remains local when working with Context Logic. Take the command `deleteTree (n)`, which deletes the tree starting at node `n`. This command would have a specification of the form:

$$\{\mathtt{n}[T]\}\ \mathtt{deleteTree(n)}\ \{\varnothing\}$$

Notice that the footprint of this command is the entire subtree at `n`, as the command will have to access each node in this tree in order to delete it. When we wish to use this command in the context of a larger program we need to be able to extend the tree. We do this by making use of the Frame Rule to separate out the subtree that we are going to delete. We can see how this works in figure 9.

c) When reasoning about programs the right adjoint $\multimap$ is more commonly used than the left adjoint. Similar to $-\!\!*$ in Separation Logic, $\multimap$ is used to describe future properties of

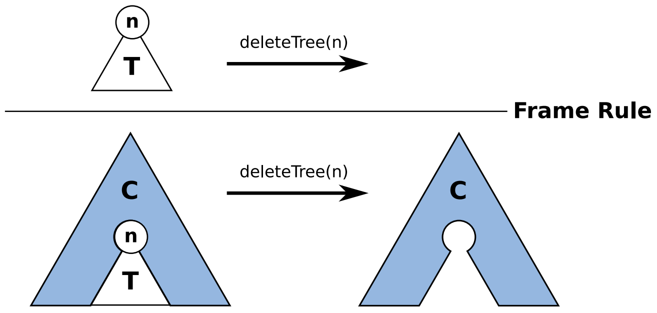

Figure 9: The Frame Rule in use.

some data structure. For example, we might write the formula $\mathtt{n}[\mathtt{p}[\emptyset]] \multimap P$ to state that when the node $\mathtt{n}$ wit a single child $\mathtt{p}$ is inserted into our context, some property P will hold. If our current tree is instead the one shown in figure 5, then we could delete the node $\mathtt{m}$ to obtain the property P. The reasoning for this action goes as follows:

$$\begin{array}{l}
\{(\mathtt{n}[\mathtt{p}[\emptyset]] \multimap P) \circ (\mathtt{n}[\mathtt{m}[\emptyset] \mid \mathtt{p}[\emptyset]])\} \\
\{(\mathtt{n}[\mathtt{p}[\emptyset]] \multimap P) \circ (\mathtt{n}[\_ \mid \mathtt{p}[\emptyset]]) \circ (\mathtt{m}[\emptyset])\} \\
\mathtt{deleteTree}(\mathtt{m}) \\
\{(\mathtt{n}[\mathtt{p}[\emptyset]] \multimap P) \circ (\mathtt{n}[\_ \mid \mathtt{p}[\emptyset]]) \circ (\emptyset)\} \\
\{(\mathtt{n}[\mathtt{p}[\emptyset]] \multimap P) \circ (\mathtt{n}[\emptyset \mid \mathtt{p}[\emptyset]])\} \\
\{(\mathtt{n}[\mathtt{p}[\emptyset]] \multimap P) \circ (\mathtt{n}[\mathtt{p}[\emptyset]])\} \\
\{P\}
\end{array}$$

This example shows the interaction between $\circ$ and $\multimap$, as well as demonstrating local reasoning.

3 Specifying Move - Motivation

In this section we shall discuss the motivating issues behind our initial choice to investigate the move command in a tree update setting. Since its very beginnings, the majority of context logic papers have been concerned with specifying and verifying update languages that operate over a tree-like data structure. From simple cases such as binary trees [4], to complicated data structures such as those seen in XML or the Document Object Model (DOM) [25]. However, one issue that has proved to be a consistent problem throughout this work is the specification of the move command.

3.1 What Is Move?

The command `move(t,n)` cuts the subtree starting at node `n` out of the data tree and reinserts it in relation to the target node `t`. This succeeds only if the node `n` is not an ancestor of the target node `t`. There are four different instances of the move command, each of which puts the removed subtree into a different location.

1. `moveBefore` reinserts the subtree as a left sibling of the target
2. `moveAfter` reinserts the subtree as a right sibling of the target
3. `prepend` reinserts the subtree as the first child of the target
4. `append` reinserts the subtree as the last child of the target

Figure 10 shows an example of the `append` command succeeding and an example of the command faulting.

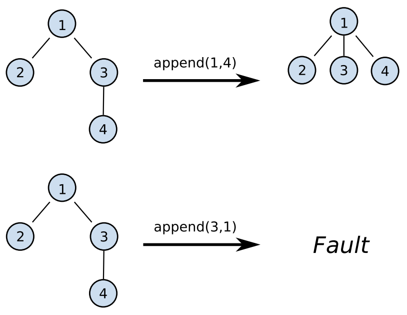

Figure 10: `append` - a case of `move`

3.2 What's the Problem?

We have recently come to the conclusion that our current Context Logic reasoning does not deal adequately with the various versions of the `move` command. They key to understanding why this is the case is to consider what separates the move command from the other tree update commands. All of the other commands that we have specified to date have only a single point of update within the tree. A command that gets, or sets, some data in a node operates only on that node. A deletion or copy command will only affect a single subtree within the data tree. However, the move command deals with two subtrees which may be disjoint within the data tree. Moreover, we are also concerned with the hierarchical relationship of the two nodes in question. (Recall that the node we are moving must not be an ancestor of the target node.) Ideally we would like to reason about these possibly disjoint sections of the data tree, but our current logic does not posses this ability. To date we have overcome this instead by specifying some arbitrary tree that connects the disjoint sections of the data tree. Thus we are back in the case where we are reasoning over a single subtree, which our logic handles very well. However, when we begin to look at our work in a Concurrent setting, where multiple commands may be run at the same time, this approach is no longer appropriate. When dealing with concurrency we expect the specification of a command to have the same scope as the commands footprint, that is the part of the data tree which might be affected by the command. Most of our work to date in fact gives specifications that are much larger than their corresponding commands footprints. If we are to tackle the challenging issues of concurrency, we must first extend our logic so that our specifications are closer to the footprints of our commands.

4 Research

There has been a wealth of research carried out in the area of Context Logic, and unsurprisingly a moderate amount of this research has been in the area of high level program reasoning. In the following section we shall highlight those works which are pertinent to providing an accurate specification of move in a tree setting.

4.1 Current Work on Move

In our recent work on formally specifying the Document Object Model (DOM) standard for web-page update [15],[16] we have begun to tackle the difficulties of the move command. In particular we look at the case of appending a subtree to a node.

4.1.1 Move - A basic command?

Early on in our work with DOM we thought that maybe thinking of the move commands as basic operations might be a mistake. Perhaps we could model these commands as the combination of a copy command, a delete command and an insert command. The idea here being that you make a copy of the subtree you want to move, you then delete the subtree from the data tree and finally you insert the copied subtree where you want to move it to. Whist this seems, on the surface, to have the correct behavior, in some implementations there is in fact a subtle difference that would make such an implementation of move incorrect. In some implementations, when we copy and insert a tree, the nodes of that tree gain fresh identifiers. They do not keep their original identifiers as this would potentially lead to identifier clashes in the tree. (Remember that our identifiers are unique) This means that variables that pointed to nodes in the moved subtree will no longer point to those nodes after the move. However, a move command which physically moves the subtree does preserve the identifiers of the nodes in the moved subtree. As an example of this consider the situation in figure 11.

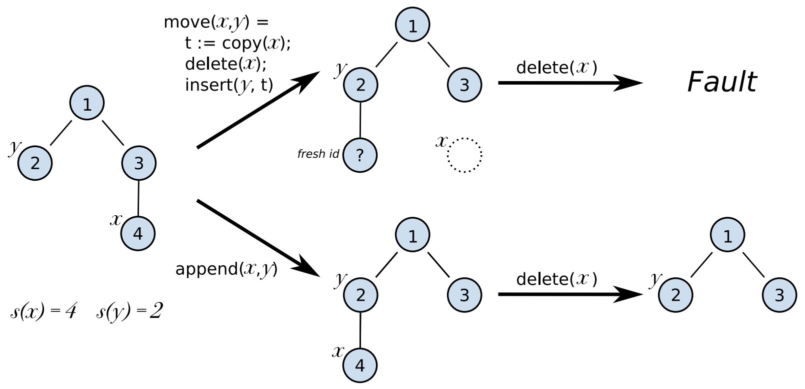

Figure 11: Copying a tree does not preserve node identifiers

Here we have a program state consisting of a small tree and a variable store s which maps the variable x to node id 4 and variable y to node id 2. Notice the difference between moving node 4 and copying and re-inserting it. In the latter case, the fresh id means that if we use the variable x in the future then our program will fault as the node 4 no longer exists. This, however, is not the only issue with specifying move as the composition of other commands. The footprints of the move commands are not the same size as the combination of the footprints of the composite commands. The move cases need the extra information of hierarchical links between the target nodes. Clearly this implies that the move commands are basic commands which must be specified, as they provide a behavior that we cannot correctly simulate with the use of other simple commands.

4.1.2 An Existing Move Specification

Whilst our logic may not currently be ideally suited to handling the specification of move commands, it is however still possible. Let us take the style of specification for the `append` command from [15] as an example of what we are currently capable of.

$$\begin{array}{l}\{(\emptyset \multimap (\mathtt{c} \circ \mathtt{parent[t']})) \circ \mathtt{child[t]}\} \\ \mathtt{append(parent, child)} \\ \{\mathtt{c} \circ \mathtt{parent[t'} \otimes \mathtt{child[t]]}\}\end{array}$$

The pre-condition for this command seems quite complicated at a first glance, but we can break it down into simpler parts. $\mathtt{c} \circ \mathtt{parent[t']}$ simply describes a context which contains the node `parent` at some point. Now $(\emptyset \multimap (\mathtt{c} \circ \mathtt{parent[t']}))$ describes a context which, if we insert the empty tree into it, contains the node `parent`. So the context hole cannot be an ancestor of the node `parent` otherwise this formula would not hold. Now the whole thing says that in fact, we fill the context hole of the context with the node `child`. So have we specified that both of the nodes exist, and that the node we are going to move, `child`, is not above the target node, `parent`, in the tree. The post-condition simply states that we have moved the node `child` to be at the end of the list of children of the node `parent`. Notice, however, that this specification for the command is not really isolated to the footprint of the command. The command itself only access the nodes `parent` and `child` requiring that `child` is not an ancestor of `parent`, but the specification can only check this relation property by mentioning some arbitrary context `c` that links these nodes as well as arbitrary subtrees `t` and `t'`. Figure 12 shows this difference in specification size and footprint size.

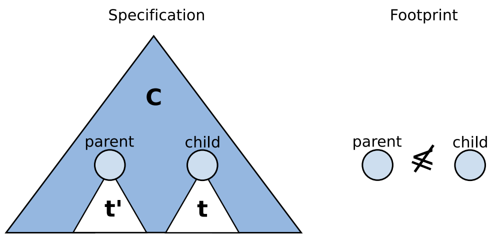

Figure 12: Append - specification size vs. footprint size

For the reasoning that we have carried out over the DOM data structure, this overly large specification is not really a problem. However, if we want to be able to use our work in a concurrent setting or claim that our reasoning is truly contained to program footprints, we will need to improve on this. We want to be able to describe the disjoint parent and child nodes without a joining context C. However, a formula that describes this is beyond the expressive power of Single-Hole Context Logic where our application connective $\circ$ cannot describe disjoint trees. Instead we turn our attention to current work on Multi-Holed Context Logic, an area that is still very much in its initial stages of development.

4.2 Multi-Holed Context Logic

The basic idea behind Multi-Holed Context Logic is to treat everything as a context. Every context can have 0 or more context holes, and data is simply a context that has no holes. Now that we have possibly many holes in our data structure, we need to take care that we keep track of where we are splitting and combining our contexts.

4.2.1 Labeled Application

The first use of Multi-Holed Context Logic [6] maintained the information of where context splitting occurred by labeling each context hole and each application connective. The multi-holed tree contexts used in this work were defined as follows:

$$\text{context} \quad c ::= \varepsilon \mid u[c] \mid c_1|c_2 \mid x$$

where the holes are labeled from some alphabet X, ranged over by x, y, z, each hole label is unique and the | operator is associative with identity ε. The set of hole labels that occur in a context c is denoted by $fn(c)$. In this setting a tree is viewed as a context that has no holes.
There is a corresponding definition of context application $ap_x(c_1, c_2) : c \times c \to c$ which is labeled by some x to denote the context hole at which the application is taking place.

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

We can abbreviate $ap_x(c_1, c_2)$ by $c_1 \circ_x c_2$.
This notion of context application gives a very precise point of application which allows the authors to prove that their application function is both associative

$$(c_1 \circ_x c_2) \circ_y c_3 = c_1 \circ_x (c_2 \circ_y c_3)$$

and partially commutative

$$(c_1 \circ_x c_2) \circ_y c_3 = (c_1 \circ_y c_3) \circ_x c_2$$

given some side conditions on on the labels of each context (see [6] for full details). The idea here is to be able to split up a tree several times, noting each time where the split has occurred. Updates can be made to individual contexts and these can be integrated back into their position in the overall context to obtain a complete tree once again. Figure 13 shows how a tree can be split up with the context structure given above.
The work in [6] was not concerned with reasoning about programs, but it would be a relatively trivial step to extend this notion of contexts to a program verification framework. One would

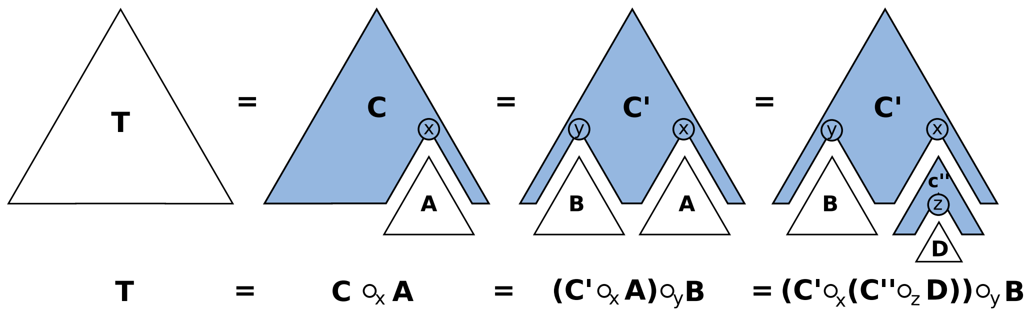

Figure 13: Multi-holed Application

need a Local Hoare Reasoning system defined over contexts, rather than the existing work which deals only with trees, where we can use the Frame Rule to add contexts both above and below our local area of reasoning.

$$\text{F} \quad \text{U} : \frac{\{P\}\ \mathbb{C}\ \{Q\}}{\{C \circ_x P\}\ \mathbb{C}\ \{C \circ_x Q\}} \quad \text{Modifies}(\mathbb{C}) \cap \text{Free}(C) = \{\},\ x \in fn(C)$$

$$\text{F} \quad \text{D} \quad : \frac{\{P\}\ \mathbb{C}\ \{Q\}}{\{P \circ_x C\}\ \mathbb{C}\ \{Q \circ_x C\}} \quad \text{Modifies}(\mathbb{C}) \cap \text{Free}(C) = \{\},\ x \in fn(P),\ x \in fn(Q)$$

Taking such an approach would solve the specification size issue for all of the simple tree update commands we are used to. For example, we could provide a specification for the getName command, mentioned above, of the form:

$$\{\mathtt{name}_{\mathtt{n}}[x]\}$$
$$\mathtt{tag} := \mathtt{getName(n)}$$
$$\{\mathtt{name}_{\mathtt{n}}[x] \wedge (\mathtt{tag} = \mathtt{name})\}$$

Notice that this specification no longer carries around the subtree of the node explicitly in the specification. Instead we use the power of the Frame Down Rule to guarantee that any subtree in the context hole x will remain unchanged by the command. However, despite the improvement to the specification of these simple commands, the logic described here does not help to reduce the size of specification required for the move commands. This is because the logic is still limited to handling only contiguous sections of a tree. We are not able to consider disjoint parts of the tree structure, which is what we need to specify the move command without carrying around some 'container' context. Notice from Figure 14, that we cannot specify the presence of disjoint subtrees *A and B* without also including enough of C' to have a single contiguous tree in the current frame.
As such, the reasoning presented here is no more local for move than what we started with in the Single-Hole case. However, the labeling of context holes is an essential practice to maintain if we are to reasoning about splitting up a data structure at multiple locations in a deterministic fashion.

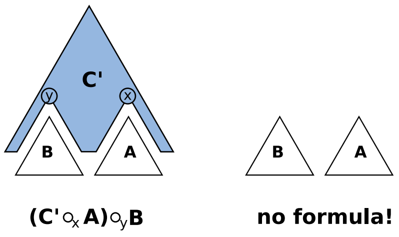

Figure 14: We can't express just the disjoint trees

4.2.2 Boxes with Wires

Having seen that labeling the application function made it impossible to specify disjoint trees, Zarfaty began to think about a different way of linking up contexts. A system capable of describing labeled boxes (contiguous sections of data), along with a way of linking (or wiring) these sections together to provide bigger structures was outlined in a diagrammatic fashion [26]. Labels where placed on the boxes and context holes and these were to be unique across the whole data structure. The only way to link structures together was to be by explicitly stating that too labels be wired together. An example of the intended wiring behavior is shown in Figure 15.

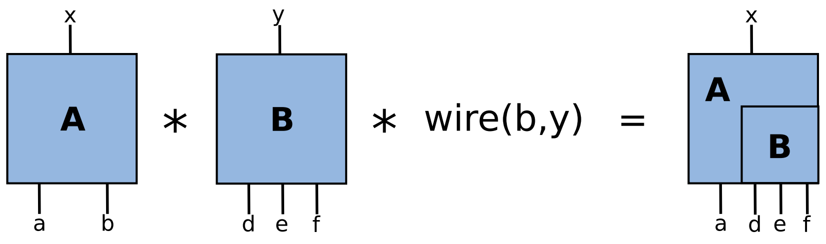

Figure 15: Wiring Up Boxes

The application system aimed for here would be far more flexible than any previous approach, but context application has gone from being a single operation to the combination of both adding a new labeled box to a collection of boxes and then describing how that box links to other boxes in the collection. We are keen to maintain a notion of context application that is similar to the one-step operation we are used to seeing. The flexible nature of the Boxes with Wiring idea should prove very powerful when specifying a command like move as the notion allows us to specify disjoint sections of a data structure without the need for a joining context between those structures. However, we are keen to also add the notion of auto-wiring, such that the boxes wire up automatically (when some conditions have been met) without the need for an explicit wire operation.

4.3 Concurrency

So far all of the program reasoning carried out with Context Logic has been restricted to the sequential case. That is where commands are carried out one at a time in a given order. However, it is fast becoming the case that people are becoming interested in the benefits of running multiple commands at the same time, concurrently, to increase the speed of programs. Separation Logic has recently begun to reason about programs that run concurrently [5],[21],[22],[14] and have introduced new rules of inference for describing concurrent program execution.

$$\text{P} \quad : \quad \frac{\{P_1\}\ \mathbb{C}_1\ \{Q_1\} \quad \{P_2\}\ \mathbb{C}_2\ \{Q_2\}}{\{P_1 * P_2\}\ \mathbb{C}_1 \parallel \mathbb{C}_2\ \{Q_1 * Q_2\}}$$

$$\text{R} \quad : \quad \frac{\{(P * RI_r) \wedge B\}\ \mathbb{C}\ \{Q * RI_r\}}{\{P\}\ \text{with } r \text{ when } B \text{ do } \mathbb{C}\ \{Q\}} \quad RI_r = \text{resource invariant for } r$$

The Parallel Rule allows multiple commands that act on disjoint data to be run at the same time, while the Resource Rule allows for commands to share data so long as they synchronise on access to that data and maintain a resource invariant property. These rules will provide the starting point for our Context Logic reasoning about concurrent programs. However, notice that in order to make efficient use of a rule like the Parallel Rule, we will first need to make sure that our command specifications are restricted to just the parts of the data structure that they access. As we have shown above this is not the case for our move specification, but it is also not the case for many of our current specifications. Take, as an example, the style of specification for the `getName` command from [15] which returns the tag data stored at a tree node `n`.

$$\{\mathtt{name}_{\mathtt{n}}[\mathtt{t}]\}$$
$$\mathtt{tag} := \mathtt{getName}(\mathtt{n})$$
$$\{\mathtt{name}_{\mathtt{n}}[\mathtt{t}] \wedge (\mathtt{tag} = \mathtt{name})\}$$

This command specification is as small as it can get with Single-Hole Context Logic. However, as shown in figure 16, this specification includes the tree beneath `n`, preventing us from using the Parallel Rule to access any cells within the subtree `t` concurrently.

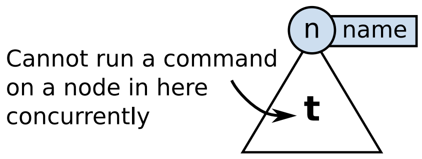

Figure 16: getName - specification size

Since this command is just a simple look-up command, accessing only one cell, the size of the data structure that is blocked by the command's specification is excessive. An ideal specification would not mention the subtree `t`, but instead use a context where a hole factors out the subtree by use of the frame rule.

5 Preliminary Results

In the following section we shall show how we can use the ideas presented in the previous section to hep us to create a formal program verification system with which we can give a local specification for all of our update commands, especially concentrating on the move commands.

5.1 Fragments and Clusters

We choose to work with a data structure similar to that described in section 4.2.2. We take fragments of our data structure, in this case tree fragments, and cluster these fragments together if we want to talk about multiple, or larger, parts of a tree. A tree fragment **F** is a contiguous section of the data tree, but a cluster **C** may be made up of fragments that are not linked together enabling us to specify disjoint sections of the data tree. Each tree fragment is uniquely labeled, and every context hole is also labeled. The formal definition of this data structure is given as follows:

5.1.1 Data Structure

For hole labels $\boldsymbol{\alpha} \in$, a countably infinite set, and node identifiers $\mathbf{n} \in$ I we have:

$$\begin{array}{rll} \text{tree fragment} & \mathbf{F} & ::= \varnothing_F \mid \boldsymbol{\alpha} \mid \mathbf{n}[\mathbf{F}] \mid \mathbf{F}|\mathbf{F} \\ \text{cluster} & \mathbf{C} & ::= \varnothing_C \mid \boldsymbol{\alpha}{\rightarrow}\mathbf{F} \mid \mathbf{C} * \mathbf{C} \end{array}$$

where the node identifiers **n** are unique, the tree fragment labels are unique and hole labels are also separately unique. Note that a tree fragment label may be equal to a hole label. There is a simple structural congruence ≡ stating that the parallel composition operator | is associative with identity $\varnothing_F$ and that fragment separation $*$ is associative and commutative with identity $\varnothing_C$. $\alpha{\rightarrow}F$ is well formed iff $\alpha \notin F$, that is that we do not allow loops in our data structure. When a tree fragment F contains no context holes we think of it as a concrete tree T.

We also provide functions holes : C → {   } and tops : C → {   } that record the hole labels of a given cluster and the tree fragment labels of a given cluster respectively.

$$\begin{array}{ll} \text{holes}(\varnothing_C) & = \{\} \\ \text{holes}(\boldsymbol{\alpha}{\rightarrow}\mathtt{F}) & = \text{holes}(\mathtt{F}) \\ \text{holes}(\mathtt{C} * \mathtt{C}') & = \text{holes}(\mathtt{C}) \cup \text{holes}(\mathtt{C}') \\ \text{holes}(\varnothing_F) & = \{\} \\ \text{holes}(\boldsymbol{\alpha}) & = \{\boldsymbol{\alpha}\} \\ \text{holes}(\mathtt{n}[\mathtt{F}]) & = \text{holes}(\mathtt{F}) \\ \text{holes}(\mathtt{F}|\mathtt{F}') & = \text{holes}(\mathtt{F}) \cup \text{holes}(\mathtt{F}') \end{array}$$

$$\begin{array}{ll} \text{tops}(\varnothing_C) & = \{\} \\ \text{tops}(\boldsymbol{\alpha}{\rightarrow}\mathtt{F}) & = \{\boldsymbol{\alpha}\} \\ \text{tops}(\mathtt{C} * \mathtt{C}') & = \text{tops}(\mathtt{C}) \cup \text{tops}(\mathtt{C}') \end{array}$$

5.1.2 Application Function

Our data structure comes with an application function that allows us to combine clusters in a way that links up tree fragments if they share a common hole and fragment label. In order to provide this application function we need to define an extra data structure for a non-commutative cluster. This is necessary to enable us to provide a terminating application function.

$$\text{non-commutative cluster} \quad \mathbf{C}_n ::= \varnothing_C \mid \boldsymbol{\alpha}{\to}\mathbf{F} \mid \mathbf{C}_n *_n \mathbf{C}_n$$

where $*_n$ is the non-commutative version of the $*$ connective. We have a function $nc : \mathrm{C} \to \mathrm{C}_n$ that transforms a commutative cluster into a non-commutative one.

$$\begin{aligned} nc(\varnothing_C) &= \varnothing_C \\ nc(\alpha{\to}\mathtt{F}) &= \alpha{\to}\mathtt{F} \\ nc(\mathtt{C}_1 * \mathtt{C}_2) &= nc(\mathtt{C}_1) *_n nc(\mathtt{C}_2) \end{aligned}$$

Using this function we can now give the definition of our cluster application function $ap : \mathrm{C} \to \mathrm{C}$ which allows us to combine $*$ separated clusters together, joining fragments together where specified by hole and fragment labels.

$$ap(\mathtt{C}) = ap_2(nc(\mathtt{C}))$$

$$\begin{aligned} ap_2(\varnothing_C) &= \varnothing_C \\ ap_2(\alpha{\to}\mathtt{F} *_n \mathtt{C}) &= \begin{cases} ap_2(\mathtt{C}[\mathtt{F}/\alpha]) & \text{if } \alpha \in \text{holes}(C) \\ ap_2(\mathtt{C} *_n \alpha{\to}\mathtt{F}) & \text{if tops}(\mathtt{C}) \cap \text{holes}(\mathtt{F}) \neq \{\} \\ \alpha{\to}\mathtt{F} * ap_2(\mathtt{C}) & \text{otherwise} \end{cases} \end{aligned}$$

We use $wf(ap(\mathtt{C}))$ to denote that $ap(\mathtt{C})$ results in a well formed cluster. An example of how this application function can break down a tree into separate parts is given in figure 17 where the tree $\alpha{\to}1[2[4[5[\varnothing_F]]] \mid 3[6[\varnothing_F] \mid 7[\varnothing_F]]]$ is broken down into the three tree fragments $\alpha{\to}1[2[\beta] \mid \delta]$, $\beta{\to}4[5[\varnothing_F]]$ and $\delta{\to}3[6[\varnothing_F] \mid 7[\varnothing_F]]$.

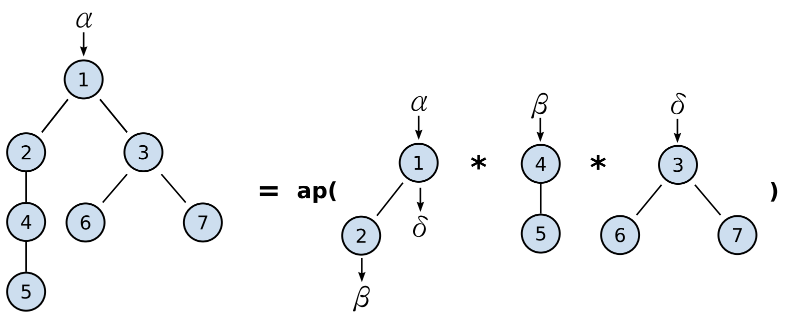

Figure 17: Application splits up a tree

Note that if we $*$ together two tree fragments which have no labels in common, then the application function is still defined on this cluster and preserves these disjoint trees, which may later be individually extended or even linked.

Examples

We give a few examples of how the application function lets us combine tree fragments and when a cluster is not well formed.

a) $ap(\alpha{\to}\mathtt{n}[\beta] * \beta{\to}\mathtt{m}[\varnothing_F]) \quad = \quad \alpha{\to}\mathtt{n}[\mathtt{m}[\varnothing_F]]$

b) $ap(\alpha{\to}\mathtt{n}[\varnothing_F] * \beta{\to}\mathtt{m}[\varnothing_F] \mid \alpha) \quad = \quad \beta{\to}\mathtt{m}[\varnothing_F] \mid \mathtt{n}[\varnothing_F]$

c) $ap(\alpha{\to}\mathtt{n}[\varnothing] * \beta{\to}\mathtt{n}[\varnothing]) \quad = \quad$ *Not well formed!*

d) $ap(\alpha{\to}\beta * \beta{\to}\alpha) \quad = \quad$ *Not well formed!*

e) $ap(\alpha{\to}\mathtt{n}[\beta] * \beta{\to}\mathtt{m}[\delta] * \gamma{\to}\mathtt{p}[\varnothing]) \quad = \quad \alpha{\to}\mathtt{n}[\mathtt{m}[\delta]] * \gamma{\to}\mathtt{p}[\varnothing]$

Notice that example e) describes a cluster which has some disjoint tree fragments, but that this is still a well formed cluster. It is this ability to describe disjoint data structures that is a depart from current work with Context Logic, and provides us with the machinery to be able to better specify the move commands.

5.1.3 Tree Shapes

In order to specify update commands that copy a tree or insert new bits of tree into the data structure we need a notion of the shape of a tree. Since the notion of tree fragments is an artifact of the reasoning, tree shapes are defined only over concrete trees which are tree fragments that do not contain context holes.

$$\text{treeshape} \quad \mathbf{T}_\circ ::= \varnothing_T \mid \circ[\mathbf{T}_\circ] \mid \mathbf{T}_\circ | \mathbf{T}_\circ$$

We write $\langle \mathtt{t} \rangle$ for the shape of a concrete tree $\mathtt{t}$ where:

$$\begin{aligned} \langle \varnothing_F \rangle &= \varnothing_T \\ \langle \mathtt{n}[\mathtt{t}] \rangle &= \circ[\langle \mathtt{t} \rangle] \\ \langle \mathtt{t} \mid \mathtt{t}' \rangle &= \langle \mathtt{t} \rangle \mid \langle \mathtt{t}' \rangle \end{aligned}$$

We write $\mathtt{t} \simeq \mathtt{t}'$ if the trees $\mathtt{t}$ and $\mathtt{t}'$ have the same shape. (i.e. $\langle \mathtt{t} \rangle = \langle \mathtt{t}' \rangle$)

5.1.4 Program State

The program state consists of a working cluster c and a variable store s sending node variables and tree shape variables to their values.

$$s : (\mathrm{Var}_\mathrm{I} \ \to \mathrm{I}\ ) \times (\mathrm{Var}_{\mathrm{T}_\circ} \to \mathrm{T}_\circ)$$

We write $s[\mathtt{x} \mapsto \mathbf{v}]$ for the variable store s overwritten with $s(\mathtt{x}) = \mathbf{v}$.

5.1.5 Expressions

We define node expressions $N \in \mathrm{Exp}_{\mathrm{I}}$, tree shape expressions $X \in \mathrm{Exp}_{\mathrm{T}_\circ}$ and boolean expressions $B \in \mathrm{Exp}_{\mathbb{B}}$:

$$\begin{array}{rll} N & ::= \mathtt{n} \mid \mathrm{nil} & \mathtt{n} \in \mathrm{Var}_{\mathrm{I}} \\ X & ::= \varnothing_T \mid \mathtt{x} \mid \circ[X] \mid X|X & \mathtt{x} \in \mathrm{Var}_{\mathrm{T}_\circ} \\ B & ::= N = N \mid X = X \mid \mathrm{false} \mid B \Rightarrow B & \end{array}$$

The valuation of an expression E in a store s is written $[\![\mathtt{E}]\!]s$ and has the obvious semantics.

5.2 Tree Update Language

We choose to work with a small tree update language, similar to that in [17] extended with a set of move commands. Unlike [17], we choose to give separate commands for each of the cases of the look-up, insert and move commands. We also provide commands for sequencing, branching and looping that would be found in any programming language. The imperative style update language we provide is capable of expressing a wide range of tree manipulation from these relatively simple commands.

5.2.1 The Commands

Our tree update language has the following commands:

$\mathbb{C}$ ::=	`skip`	*no operation*
	`x := E`	*variable assignment*
	$\mathbb{C}_{\mathrm{up}}$	*tree update commands*
	$\mathbb{C};\mathbb{C}$	*sequencing*
	`if` B `then` $\mathbb{C}$ `else` $\mathbb{C}$	*if-then-else branching*
	`while` B `do` $\mathbb{C}$	*while-do loop*

where the tree update commands are:

$\mathbb{C}_{\mathrm{up}}$ ::=	`t := copy(n)`	*copy tree shape*
	`n' := getUp(n)`	*get parent*
	`n' := getLeft(n)`	*get previous sibling*
	`n' := getRight(n)`	*get next sibling*
	`n' := getFirst(n)`	*get first child*
	`n' := getLast(n)`	*get last child*
	`delete(n)`	*delete node*
	`subDelete(n)`	*delete subtree*
	`insertBefore(n, X)`	*insert tree shape X before node n*
	`insertAfter(n, X)`	*insert tree shape X after node n*
	`insertFirst(n, X)`	*insert tree shape X as first child of n*
	`insertLast(n, X)`	*insert tree shape X as last child of n*
	`moveBefore(t, c)`	*move tree c before node t*
	`moveAfter(t, c)`	*move tree c after node t*
	`prepend(p, c)`	*prepend tree c to children of node p*
	`append(p, c)`	*append tree c to children of node p*

5.2.2 Operational Semantics

In Figure 18 we give the operational semantics of the language using an evaluation relation $\rightsquigarrow$ relating configuration triples $\mathbb{C}, s, \mathbf{c}$, terminal states $s, \mathbf{c}$, and faults, where $\mathbf{c}$ refers to a cluster, $\mathbf{f}$ refers to a tree fragment and $\mathbf{t}$ refers to a concrete tree which is a tree fragment with no holes.

$$\frac{}{\texttt{skip}, s, \mathbf{c} \rightsquigarrow s, \mathbf{c}} \qquad \frac{[\![\texttt{E}]\!]s = \mathbf{v}}{\texttt{x := E}, s, \mathbf{c} \rightsquigarrow s[\texttt{x} \mapsto \mathbf{v}], \mathbf{c}}$$

$$\frac{\mathbb{C}_1, s, \mathbf{c} \rightsquigarrow \mathbb{C}_1', s', \mathbf{c}'}{\mathbb{C}_1; \mathbb{C}_2, s, \mathbf{c} \rightsquigarrow \mathbb{C}_1'; \mathbb{C}_2, s', \mathbf{c}'} \qquad \frac{\mathbb{C}_1, s, \mathbf{c} \rightsquigarrow s', \mathbf{c}'}{\mathbb{C}_1; \mathbb{C}_2, s, \mathbf{c} \rightsquigarrow \mathbb{C}_2, s', \mathbf{c}'}$$

$$\frac{[\![\texttt{B}]\!]s = \text{true}}{\texttt{if B then } \mathbb{C}_1 \texttt{ else } \mathbb{C}_2, s, \mathbf{c} \rightsquigarrow \mathbb{C}_1, s, \mathbf{c}} \qquad \frac{[\![\texttt{B}]\!]s = \text{false}}{\texttt{if B then } \mathbb{C}_1 \texttt{ else } \mathbb{C}_2, s, \mathbf{c} \rightsquigarrow \mathbb{C}_2, s, \mathbf{c}}$$

$$\frac{[\![\texttt{B}]\!]s = \text{true}}{\texttt{while B do } \mathbb{C}, s, \mathbf{c} \rightsquigarrow \mathbb{C}; \texttt{while B do } \mathbb{C}, s, \mathbf{c}} \qquad \frac{[\![\texttt{B}]\!]s = \text{false}}{\texttt{while B do } \mathbb{C}, s, \mathbf{c} \rightsquigarrow s, \mathbf{c}}$$

$$\frac{s(\text{n}) = \mathbf{n} \quad \mathbf{c} \equiv ap(\mathbf{c}' * \alpha{\rightarrow}\mathbf{n}[\mathbf{t}]) \quad \mathbf{X} = \langle \mathbf{n}[\mathbf{t}] \rangle}{\text{t} := \texttt{copy}(\text{n}), s, \mathbf{c} \rightsquigarrow s[\text{t} \mapsto \mathbf{X}], \mathbf{c}} \qquad \frac{s(\text{n}) = \mathbf{n} \quad \mathbf{c} \equiv ap(\mathbf{c}' * \alpha{\rightarrow}\mathbf{m}[\mathbf{f}_1 \mid \mathbf{n}[\mathbf{f}_2] \mid \mathbf{f}_3])}{\text{n}' := \texttt{getUp}(\text{n}), s, \mathbf{c} \rightsquigarrow s[\text{n}' \mapsto \mathbf{m}], \mathbf{c}}$$

$$\frac{s(\text{n}) = \mathbf{n} \quad \mathbf{c} \equiv ap(\mathbf{c}' * \alpha{\rightarrow}\mathbf{m}[\mathbf{f}_1] \mid \mathbf{n}[\mathbf{f}_2])}{\text{n}' := \texttt{getLeft}(\text{n}), s, \mathbf{c} \rightsquigarrow s[\text{n}' \mapsto \mathbf{m}], \mathbf{c}} \qquad \frac{s(\text{n}) = \mathbf{n} \quad \mathbf{c} \equiv ap(\mathbf{c}' * \alpha{\rightarrow}\mathbf{n}[\mathbf{f}_1] \mid \mathbf{m}[\mathbf{f}_2])}{\text{n}' := \texttt{getRight}(\text{n}), s, \mathbf{c} \rightsquigarrow s[\text{n}' \mapsto \mathbf{m}], \mathbf{c}}$$

$$\frac{s(\text{n}) = \mathbf{n} \quad \mathbf{c} \equiv ap(\mathbf{c}' * \alpha{\rightarrow}\mathbf{n}[\mathbf{m}[\mathbf{f}_1] \mid \mathbf{f}_2])}{\text{n}' := \texttt{getFirst}(\text{n}), s, \mathbf{c} \rightsquigarrow s[\text{n}' \mapsto \mathbf{m}], \mathbf{c}} \qquad \frac{s(\text{n}) = \mathbf{n} \quad \mathbf{c} \equiv ap(\mathbf{c}' * \alpha{\rightarrow}\mathbf{n}[\mathbf{f}_1 \mid \mathbf{m}[\mathbf{f}_2]])}{\text{n}' := \texttt{getLast}(\text{n}), s, \mathbf{c} \rightsquigarrow s[\text{n}' \mapsto \mathbf{m}], \mathbf{c}}$$

$$\frac{\begin{array}{r} s(\text{n}) = \mathbf{n} \quad \mathbf{c} \equiv ap(\mathbf{c}' * \alpha{\rightarrow}\mathbf{n}[\mathbf{f}]) \\ \mathbf{c}'' \equiv ap(\mathbf{c}' * \alpha{\rightarrow}\mathbf{f}) \end{array}}{\texttt{delete}(\text{n}), s, \mathbf{c} \rightsquigarrow s, \mathbf{c}''} \qquad \frac{\begin{array}{r} s(\text{n}) = \mathbf{n} \quad \mathbf{c} \equiv ap(\mathbf{c}' * \alpha{\rightarrow}\mathbf{n}[\mathbf{t}]) \\ \mathbf{c}'' \equiv ap(\mathbf{c}' * \alpha{\rightarrow}\mathbf{n}[\varnothing_{\mathrm{F}}]) \end{array}}{\texttt{subDelete}(\text{n}), s, \mathbf{c} \rightsquigarrow s, \mathbf{c}''}$$

$$\frac{\begin{array}{c} s(\text{n}) = \mathbf{n} \quad \langle \mathbf{t} \rangle = [\![\texttt{X}]\!]s \quad \mathbf{t} \text{ has fresh ids} \\ \mathbf{c} \equiv ap(\mathbf{c}' * \alpha{\rightarrow}\mathbf{n}[\mathbf{f}]) \quad \mathbf{c}'' \equiv ap(\mathbf{c}' * \alpha{\rightarrow}\mathbf{t} \mid \mathbf{n}[\mathbf{f}]) \end{array}}{\texttt{insertBefore}(\text{n}, \texttt{X}), s, \mathbf{c} \rightsquigarrow s, \mathbf{c}''} \qquad \frac{\begin{array}{c} s(\text{n}) = \mathbf{n} \quad \langle \mathbf{t} \rangle = [\![\texttt{X}]\!]s \quad \mathbf{t} \text{ has fresh ids} \\ \mathbf{c} \equiv ap(\mathbf{c}' * \alpha{\rightarrow}\mathbf{n}[\mathbf{f}]) \quad \mathbf{c}'' \equiv ap(\mathbf{c}' * \alpha{\rightarrow}\mathbf{n}[\mathbf{f}] \mid \mathbf{t}) \end{array}}{\texttt{insertAfter}(\text{n}, \texttt{X}), s, \mathbf{c} \rightsquigarrow s, \mathbf{c}''}$$

$$\frac{\begin{array}{c} s(\text{n}) = \mathbf{n} \quad \langle \mathbf{t} \rangle = [\![\texttt{X}]\!]s \quad \mathbf{t} \text{ has fresh ids} \\ \mathbf{c} \equiv ap(\mathbf{c}' * \alpha{\rightarrow}\mathbf{n}[\mathbf{f}]) \quad \mathbf{c}'' \equiv ap(\mathbf{c}' * \alpha{\rightarrow}\mathbf{n}[\mathbf{t} \mid \mathbf{f}]) \end{array}}{\texttt{insertFirst}(\text{n}, \texttt{X}), s, \mathbf{c} \rightsquigarrow s, \mathbf{c}''} \qquad \frac{\begin{array}{c} s(\text{n}) = \mathbf{n} \quad \langle \mathbf{t} \rangle = [\![\texttt{X}]\!]s \quad \mathbf{t} \text{ has fresh ids} \\ \mathbf{c} \equiv ap(\mathbf{c}' * \alpha{\rightarrow}\mathbf{n}[\mathbf{f}]) \quad \mathbf{c}'' \equiv ap(\mathbf{c}' * \alpha{\rightarrow}\mathbf{n}[\mathbf{f} \mid \mathbf{t}]) \end{array}}{\texttt{insertLast}(\text{n}, \texttt{X}), s, \mathbf{c} \rightsquigarrow s, \mathbf{c}''}$$

$$\frac{\begin{array}{c} s(\text{p}) = \mathbf{p} \quad \mathbf{c} \equiv ap(\mathbf{c}' * \alpha{\rightarrow}\mathbf{p}[\mathbf{f}] * \beta{\rightarrow}\mathbf{d}[\mathbf{t}]) \\ s(\text{d}) = \mathbf{d} \quad \mathbf{c}'' \equiv ap(\mathbf{c}' * \alpha{\rightarrow}\mathbf{d}[\mathbf{t}] \mid \mathbf{p}[\mathbf{f}] * \beta{\rightarrow}\varnothing_{\mathrm{F}}) \end{array}}{\texttt{moveBefore}(\text{p}, \text{d}), s, \mathbf{c} \rightsquigarrow s, \mathbf{c}''} \qquad \frac{\begin{array}{c} s(\text{p}) = \mathbf{p} \quad \mathbf{c} \equiv ap(\mathbf{c}' * \alpha{\rightarrow}\mathbf{p}[\mathbf{f}] * \beta{\rightarrow}\mathbf{d}[\mathbf{t}]) \\ s(\text{d}) = \mathbf{d} \quad \mathbf{c}'' \equiv ap(\mathbf{c}' * \alpha{\rightarrow}\mathbf{p}[\mathbf{f}] \mid \mathbf{d}[\mathbf{t}] * \beta{\rightarrow}\varnothing_{\mathrm{F}}) \end{array}}{\texttt{moveAfter}(\text{p}, \text{d}), s, \mathbf{c} \rightsquigarrow s, \mathbf{c}''}$$

$$\frac{\begin{array}{c} s(\text{p}) = \mathbf{p} \quad \mathbf{c} \equiv ap(\mathbf{c}' * \alpha{\rightarrow}\mathbf{p}[\mathbf{f}] * \beta{\rightarrow}\mathbf{d}[\mathbf{t}]) \\ s(\text{d}) = \mathbf{d} \quad \mathbf{c}'' \equiv ap(\mathbf{c}' * \alpha{\rightarrow}\mathbf{p}[\mathbf{d}[\mathbf{t}] \mid \mathbf{f}] * \beta{\rightarrow}\varnothing_{\mathrm{F}}) \end{array}}{\texttt{prepend}(\text{p}, \text{d}), s, \mathbf{c} \rightsquigarrow s, \mathbf{c}''} \qquad \frac{\begin{array}{c} s(\text{p}) = \mathbf{p} \quad \mathbf{c} \equiv ap(\mathbf{c}' * \alpha{\rightarrow}\mathbf{p}[\mathbf{f}] * \beta{\rightarrow}\mathbf{d}[\mathbf{t}]) \\ s(\text{d}) = \mathbf{d} \quad \mathbf{c}'' \equiv ap(\mathbf{c}' * \alpha{\rightarrow}\mathbf{p}[\mathbf{f} \mid \mathbf{d}[\mathbf{t}]] * \beta{\rightarrow}\varnothing_{\mathrm{F}}) \end{array}}{\texttt{append}(\text{p}, \text{d}), s, \mathbf{c} \rightsquigarrow s, \mathbf{c}''}$$

$$\frac{\text{otherwise}}{\mathbb{C}, s, \mathbf{c} \rightsquigarrow \mathit{fault}}$$

Figure 18: Operational Semantics for our Tree Update Language

5.3 A Context Logic Over Clusters

In order to reason about our update language we need to define a Context Logic over our new Fragments and Clusters data structure.

5.3.1 Logical Environment

To be able to use logical variables for fragments, clusters and labels we need to carry an environment with our reasoning. A logical environment e is a function that maps logical fragment, cluster and label variables to their values:

$$e : (\mathrm{Var_F} \rightarrow \mathrm{F}) \times (\mathrm{Var_C} \rightarrow \mathrm{C}) \times (\mathrm{Var_L} \quad \rightarrow \mathrm{L} \quad)$$

Just as with the program state s we write $e[\mathtt{x} \mapsto \mathbf{v}]$ for the environment e overwritten with $e(\mathtt{x}) = \mathbf{v}$.

5.3.2 Formulae

The formulae for our Context Logic over Clusters include fragment formulae P_F and cluster formulae P_C.

$$\begin{array}{lll}
P_F ::= & & \\
& P_F \Rightarrow P_F \mid \mathrm{false}_F & \textit{Classical formulae} \\
& \emptyset_F \mid \boldsymbol{\alpha} \mid \mathtt{n}[P_F] \mid P_F | P_F & \textit{Fragment formulae} \\
& \mathtt{X} & \textit{Expression formulae} \\
& & \\
P_C ::= & & \\
& P_C * P_C \mid P_C \multimap\!\!* P_C & \textit{Structural formulae} \\
& P_C \Rightarrow P_C \mid \mathrm{false}_C & \textit{Classical formulae} \\
& \emptyset_C \mid \boldsymbol{\alpha} \rightarrow P_F & \textit{Cluster formulae} \\
& \mathtt{B} \mid \mathrm{var}_\mathrm{E} & \textit{Expression formulae } \mathtt{E} \in \{\mathrm{F}, \mathrm{C}, \mathrm{L} \quad \} \\
& \exists \alpha.\, P_C \mid \exists \mathtt{n}.\, P_C & \textit{Quantification}
\end{array}$$

Note that the commutativity of $*$ means that the $-\!\!*$ adjoint is symmetrical in this setting. Thus we do not require a separate $*\!\!-$ adjoint in our logic.

5.3.3 Satisfaction Relation

Given a logical environment e and a variable store s, the semantics of Context Logic is given by two satisfaction relations, $\models_F$ for tree fragments and $\models_C$ for clusters. These are defined as follows:

$$
\begin{array}{ll}
e, s, \mathbf{F} \vDash_F P_F \Rightarrow P_F & \Leftrightarrow e, s, \mathbf{F} \vDash_F P_F \Rightarrow e, s, \mathbf{F} \vDash_F P'_F \\
e, s, \mathbf{F} \vDash_F \mathrm{false}_F & \Leftrightarrow \mathit{never} \\
e, s, \mathbf{F} \vDash_F \varnothing_F & \Leftrightarrow \mathbf{F} \equiv \varnothing_F \\
e, s, \mathbf{F} \vDash_F \boldsymbol{\alpha} & \Leftrightarrow \mathbf{F} \equiv \boldsymbol{\alpha} \\
e, s, \mathbf{F} \vDash_F \mathrm{n}[P_F] & \Leftrightarrow \exists \mathbf{F}_1 .\, \mathbf{F} \equiv s(\mathrm{n})[\mathbf{F}_1] \wedge e, s, \mathbf{F}_1 \vDash_F P_F \\
e, s, \mathbf{F} \vDash_F P_F | P'_F & \Leftrightarrow \exists \mathbf{F}_1, \mathbf{F}_2 .\, \mathbf{F} \equiv \mathbf{F}_1 | \mathbf{F}_2 \wedge e, s, \mathbf{F}_1 \vDash_F P_F \wedge e, s, \mathbf{F}_2 \vDash_F P'_F \\
e, s, \mathbf{F} \vDash_F \mathtt{X} & \Leftrightarrow \langle \mathbf{F} \rangle = [\![\mathtt{X}]\!] s \\
 & \\
e, s, \mathbf{C} \vDash_C P_C * P'_C & \Leftrightarrow \exists \mathbf{C}_1, \mathbf{C}_2 .\, \mathbf{C} \equiv ap(\mathbf{C}_1 * \mathbf{C}_2) \wedge e, s, \mathbf{C}_1 \vDash_C P_C \wedge e, s, \mathbf{C}_2 \vDash_C P'_C \\
e, s, \mathbf{C} \vDash_C P_C \mathrel{-\!\!*} P'_C & \Leftrightarrow \forall \mathbf{C}_1 .\, wf(ap(\mathbf{C} * \mathbf{C}_1)) \wedge e, s, \mathbf{C}_1 \vDash_C P_C \Rightarrow ap(\mathbf{C} * \mathbf{C}_1) \vDash_C P'_C \\
e, s, \mathbf{C} \vDash_C P_C \Rightarrow P'_C & \Leftrightarrow e, s, \mathbf{C} \vDash_C P_C \Rightarrow e, s, \mathbf{C} \vDash_C P'_C \\
e, s, \mathbf{C} \vDash_C \mathrm{false}_C & \Leftrightarrow \mathit{never} \\
e, s, \mathbf{C} \vDash_C \varnothing_C & \Leftrightarrow \mathbf{C} \equiv \varnothing_C \\
e, s, \mathbf{C} \vDash_C \boldsymbol{\alpha} \rightarrow P_F & \Leftrightarrow \exists \mathbf{F} .\, \mathbf{C} \equiv \boldsymbol{\alpha} \rightarrow \mathbf{F} \wedge e, s, \mathbf{F} \vDash_F P_F \\
e, s, \mathbf{C} \vDash_C \mathtt{B} & \Leftrightarrow [\![\mathtt{B}]\!] s = \mathrm{true} \\
e, s, \mathbf{C} \vDash_C \mathrm{var}_\mathrm{E} & \Leftrightarrow \mathbf{C} \equiv e(\mathrm{var}_\mathrm{E}) \\
e, s, \mathbf{C} \vDash_C \exists \alpha .\, P_C & \Leftrightarrow \exists \mathbf{v} .\, e[\alpha \mapsto \mathbf{v}], s, \mathbf{C} \vDash_C P_C \\
e, s, \mathbf{C} \vDash_C \exists \mathrm{n} .\, P_C & \Leftrightarrow \exists \mathbf{v} .\, e, s[\mathrm{n} \mapsto \mathbf{v}], \mathbf{C} \vDash_C P_C
\end{array}
$$

5.3.4 Derived Formulae

We can derive the standard classical logic connectives from false and $\Rightarrow$ as usual, as well as deriving the following useful formulae for specifying when a we have a fragment which is a concrete tree, and for dropping the need to mention when a subtree is empty. Finally we introduce the notation for expressing 'somewhere in the cluster' $\Diamond P_C$ and the related concept 'everywhere in the cluster' $\Box P_C$.

$$
\begin{array}{rl}
\mathrm{tree}(P_\mathrm{F}) & \triangleq P_\mathrm{F} \wedge \mathrm{holes}(P_\mathrm{F}) = \{\} \\
\mathrm{n} & \triangleq \mathrm{n}[\varnothing_F] \\
\Diamond P_C & \triangleq \mathrm{true}_C * P_C \\
\Box P_C & \triangleq \neg \Diamond \neg P_C
\end{array}
$$

5.3.5 Local Hoare Triples

When providing the interpretation for Local Hoare Triples in this work we had two choices. Either we could define our Local Hoare Triple interpretation in terms of the operational semantics over clusters that we gave in section 5.2.2, or we could provide an interpretation of Local Hoare Triples that relates clusters back to a standard operational semantics over trees. We chose to work with the first of these cases as it leads to a more standard definition of Local Hoare Triples. So a Local Hoare Triple $\{P_C\}\ \mathbb{C}\ \{Q_C\}$ is interpreted on environments e, stores s and clusters $\mathbf{C}$ using a fault avoiding partial correctness interpretation:

$$
\{P_C\}\ \mathbb{C}\ \{Q_C\} \quad \Leftrightarrow \quad \forall e, s, \mathbf{C} .\, e, s, \mathbf{C} \vDash_C P_C \Rightarrow \begin{array}{l} \mathbb{C}, s, \mathbf{C} \not\rightsquigarrow \mathrm{fault} \wedge \\ \forall s', \mathbf{C}' .\, \mathbb{C}, s, \mathbf{C} \rightsquigarrow s', \mathbf{C}' \Rightarrow e, s', \mathbf{C}' \vDash_C Q_C \end{array}
$$

So if a cluster satisfies the pre-condition P_C, then the program $\mathbb{C}$ is guaranteed not to fault, and if $\mathbb{C}$ terminates, then the resulting cluster will satisfy post-condition Q_C. The alternative view of Local Hoare Triples would have to relate a cluster back to the tree based operational semantics. So we would instead interpret the Local Hoare Triple $\{P_C\}\ \mathbb{C}\ \{Q_C\}$ as:

$$\{P_C\}\ \mathbb{C}\ \{Q_C\} \quad \Leftrightarrow \quad \forall \mathbf{C}. \left(\begin{array}{l} \exists \alpha, \mathbf{T}.\ \mathbf{C} * P_C = \alpha{\rightarrow}\mathbf{T} \\ \Rightarrow \\ \mathbb{C}, \mathbf{T} \rightsquigarrow \mathbf{T}' \wedge \alpha{\rightarrow}\mathbf{T}' = \mathbf{C} * Q_C \end{array} \right)$$

In this case if a cluster satisfies the pre-condition P_C then when extend that cluster by any **C** to a concrete tree **T**, the program $\mathbb{C}$ will not fault, and moreover, the resulting tree $\mathbf{T}'$ is just the a cluster satisfying Q_C extended by **C**. Whilst this form of interpretation would allow us to maintain a more standard tree based operational semantics, this is achieved only by having a much more complicated Local Hoare Triple. This approach would likely draw more criticism than a change to the operational semantics as it leads to a more complex model. Moreover, the universal quantification over clusters **C** in the definition of this case would likely cause problems if we choose to attempt to automate our reasoning system. It is our hope that we can find a simple translation function that will relate operational semantics over trees to operational semantics over clusters that will show that we have not changed the behavior of our commands at the level of the operational semantics.

5.3.6 Command Axioms

We can now provide the Command Axioms for our Tree Update Language defined in section 5.2.1.

$\{\varnothing_C\}$	$\mathtt{skip}$	$\{\varnothing_C\}$
$\{\varnothing_C \wedge (\mathtt{v} = \mathtt{E})\}$	$\mathtt{x} := \mathtt{E}$	$\{\varnothing_C \wedge (\mathtt{x} = \mathtt{v})\}$
$\{\alpha{\rightarrow}\mathtt{n}[\mathrm{tree}(\mathtt{f})] \wedge (\mathtt{X} = \langle \mathtt{n}[\mathtt{f}] \rangle)\}$	$\mathtt{t} := \mathtt{copy}(\mathtt{n})$	$\{\alpha{\rightarrow}\mathtt{n}[\mathrm{tree}(\mathtt{f})] \wedge (\mathtt{t} = \mathtt{X})\}$
$\{\alpha{\rightarrow}\mathtt{p}[\beta \mid \mathtt{n}[\delta] \mid \gamma]\}$	$\mathtt{n'} := \mathtt{getUp}(\mathtt{n})$	$\{\alpha{\rightarrow}\mathtt{p}[\beta \mid \mathtt{n}[\delta] \mid \gamma] \wedge (\mathtt{n'} = \mathtt{p})\}$
$\{\alpha{\rightarrow}\mathtt{l}[\beta] \mid \mathtt{n}[\delta]\}$	$\mathtt{n'} := \mathtt{getLeft}(\mathtt{n})$	$\{\alpha{\rightarrow}\mathtt{l}[\beta] \mid \mathtt{n}[\delta] \wedge (\mathtt{n'} = \mathtt{l})\}$
$\{\alpha{\rightarrow}\mathtt{n}[\delta] \mid \mathtt{r}[\beta]\}$	$\mathtt{n}' := \mathtt{getRight}(\mathtt{n})$	$\{\alpha{\rightarrow}\mathtt{n}[\delta] \mid \mathtt{r}[\beta] \wedge (\mathtt{n'} = \mathtt{r})\}$
$\{\alpha{\rightarrow}\mathtt{n}[\mathtt{c}[\beta] \mid \delta]\}$	$\mathtt{n}' := \mathtt{getFirst}(\mathtt{n})$	$\{\alpha{\rightarrow}\mathtt{n}[\mathtt{c}[\beta] \mid \delta] \wedge (\mathtt{n'} = \mathtt{c})\}$
$\{\alpha{\rightarrow}\mathtt{n}[\delta \mid \mathtt{c}[\beta]]\}$	$\mathtt{n}' := \mathtt{getLast}(\mathtt{n})$	$\{\alpha{\rightarrow}\mathtt{n}[\delta \mid \mathtt{c}[\beta]] \wedge (\mathtt{n'} = \mathtt{c})\}$
$\{\alpha{\rightarrow}\mathtt{n}[\beta]\}$	$\mathtt{delete}(\mathtt{n})$	$\{\alpha{\rightarrow}\beta\}$
$\{\alpha{\rightarrow}\mathtt{n}[\mathrm{tree}(\mathtt{f})]\}$	$\mathtt{subDelete}(\mathtt{n})$	$\{\alpha{\rightarrow}\mathtt{n}\}$
$\{\alpha{\rightarrow}\mathtt{n}[\beta]\}$	$\mathtt{insertBefore}(\mathtt{n}, \mathtt{X})$	$\{\alpha{\rightarrow}\mathtt{X} \mid \mathtt{n}[\beta]\}$
$\{\alpha{\rightarrow}\mathtt{n}[\beta]\}$	$\mathtt{insertAfter}(\mathtt{n}, \mathtt{X})$	$\{\alpha{\rightarrow}\mathtt{n}[\beta] \mid \mathtt{X}\}$
$\{\alpha{\rightarrow}\mathtt{n}[\beta]\}$	$\mathtt{insertFirst}(\mathtt{n}, \mathtt{X})$	$\{\alpha{\rightarrow}\mathtt{n}[\mathtt{X} \mid \beta]\}$
$\{\alpha{\rightarrow}\mathtt{n}[\beta]\}$	$\mathtt{insertLast}(\mathtt{n}, \mathtt{X})$	$\{\alpha{\rightarrow}\mathtt{n}[\beta \mid \mathtt{X}]\}$
$\{\alpha{\rightarrow}\mathtt{t}[\beta] * \delta{\rightarrow}\mathtt{c}[\mathrm{tree}(\mathtt{f})]\}$	$\mathtt{moveBefore}(\mathtt{t}, \mathtt{c})$	$\{\alpha{\rightarrow}\mathtt{c}[\mathrm{tree}(\mathtt{f})] \mid \mathtt{t}[\beta] * \delta{\rightarrow}\varnothing_F\}$
$\{\alpha{\rightarrow}\mathtt{t}[\beta] * \delta{\rightarrow}\mathtt{c}[\mathrm{tree}(\mathtt{f})]\}$	$\mathtt{moveAfter}(\mathtt{t}, \mathtt{c})$	$\{\alpha{\rightarrow}\mathtt{t}[\beta] \mid \mathtt{c}[\mathrm{tree}(\mathtt{f})] * \delta{\rightarrow}\varnothing_F\}$
$\{\alpha{\rightarrow}\mathtt{p}[\beta] * \delta{\rightarrow}\mathtt{c}[\mathrm{tree}(\mathtt{f})]\}$	$\mathtt{prepend}(\mathtt{p}, \mathtt{c})$	$\{\alpha{\rightarrow}\mathtt{p}[\mathtt{c}[\mathrm{tree}(\mathtt{f})] \mid \beta] * \delta{\rightarrow}\varnothing_F\}$
$\{\alpha{\rightarrow}\mathtt{p}[\beta] * \delta{\rightarrow}\mathtt{c}[\mathrm{tree}(\mathtt{f})]\}$	$\mathtt{append}(\mathtt{p}, \mathtt{c})$	$\{\alpha{\rightarrow}\mathtt{p}[\beta \mid \mathtt{c}[\mathrm{tree}(\mathtt{f})]] * \delta{\rightarrow}\varnothing_F\}$

5.3.7 Inference Rules

We have the standard rules of inference; Sequence, Consequence, Auxiliary Variable Elimination, If-Then-Else, While-Do and we add the local reasoning Frame Rule of Context Logic. This rule permits local reasoning by allowing the inference of invariant properties implied by locality and is presented here in terms of cluster separation:

$$\text{S} \quad : \quad \frac{\{P_C\}\ \mathbb{C}_1\ \{Q_C\} \quad \{Q_C\}\ \mathbb{C}_2\ \{R_C\}}{\{P_C\}\ \mathbb{C}_1; \mathbb{C}_2\ \{R_C\}}$$

$$\text{C} \quad : \quad \frac{P'_C \Rightarrow P_C \quad \{P_C\}\ \mathbb{C}\ \{Q_C\} \quad Q_C \Rightarrow Q'_C}{\{P'_C\}\ \mathbb{C}\ \{Q'_C\}}$$

$$\text{A} \quad \text{V} \quad \text{E} \quad : \quad \frac{\{P_C\}\ \mathbb{C}\ \{Q_C\}}{\{\exists \mathtt{n}.\, P_C\}\ \mathbb{C}\ \{\exists \mathtt{n}.\, Q_C\}} \quad \mathtt{n} \notin \mathrm{FV}(\mathbb{C})$$

$$\text{I -T} \quad \text{-E} \quad : \quad \frac{\{\mathtt{B} \wedge P_C\}\ \mathbb{C}_1\ \{Q_C\} \quad \{\neg\mathtt{B} \wedge P_C\}\ \mathbb{C}_2\ \{Q_C\}}{\{P_C\}\ \mathtt{if\ B\ then}\ \mathbb{C}_1\ \mathtt{else}\ \mathbb{C}_2\ \{Q_C\}}$$

$$\text{W} \quad \text{-D} \quad : \quad \frac{\{\mathtt{B} \wedge P_C\}\ \mathbb{C}\ \{P_C\}}{\{P_C\}\ \mathtt{while\ B\ do}\ \mathbb{C}\ \{\neg\mathtt{B} \wedge P_C\}}$$

$$\text{F} \quad \text{R} \quad : \quad \frac{\{P\}\ \mathbb{C}\ \{Q\}}{\{C * P\}\ \mathbb{C}\ \{C * Q\}} \quad \mathrm{Mod}(\mathbb{C}) \cap \mathrm{FV}(C) = \{\}$$

5.3.8 Weakest Preconditions

The weakest preconditions for each of our update commands are given below.

$\{P\}$	`skip`	$\{P\}$
$\{\exists \mathtt{v}.\, P[\mathtt{v}/\mathtt{x}] \wedge (\mathtt{v} = \mathtt{E})\}$	`x := E`	$\{P\}$
$\{\exists \alpha, \mathtt{f}, \mathtt{X}.\, \alpha{\to}\mathtt{n}[\mathrm{tree}(\mathtt{f})] \wedge (\mathtt{X} = \langle \mathtt{n}[\mathtt{f}] \rangle) \wedge P[\mathtt{X}/\mathtt{t}]\}$	`t := copy(n)`	$\{P\}$
$\{\exists \alpha, \beta, \gamma, \delta, \mathtt{p}\alpha{\to}\mathtt{p}[\beta \mid \mathtt{n}[\delta] \mid \gamma] \wedge P[\mathtt{p}/\mathtt{n}'])\}$	`n' := getUp(n)`	$\{P\}$
$\{\exists \alpha, \beta, \delta, \mathtt{l}.\, \alpha{\to}\mathtt{l}[\beta] \mid \mathtt{n}[\delta] \wedge P[\mathtt{l}/\mathtt{n}']\}$	`n' := getLeft(n)`	$\{P\}$
$\{\exists \alpha, \beta, \delta, \mathtt{r}.\, \alpha{\to}\mathtt{n}[\delta] \mid \mathtt{r}[\beta] \wedge P[\mathtt{r}/\mathtt{n}']\}$	`n' := getRight(n)`	$\{P\}$
$\{\exists \alpha, \beta, \delta, \mathtt{c}.\, \alpha{\to}\mathtt{n}[\mathtt{c}[\beta] \mid \delta] \wedge P[\mathtt{c}/\mathtt{n}']\}$	`n' := getFirst(n)`	$\{P\}$
$\{\exists \alpha, \beta, \delta, \mathtt{c}.\, \alpha{\to}\mathtt{n}[\delta \mid \mathtt{c}[\beta]] \wedge P[\mathtt{c}/\mathtt{n}']\}$	`n' := getLast(n)`	$\{P\}$
$\{\exists \alpha, \beta.\, \alpha{\to}\mathtt{n}[\beta] * ((\alpha{\to}\beta) \mathrel{-\!\!*} P)\}$	`delete(n)`	$\{P\}$
$\{\exists \alpha, \mathtt{f}.\, \alpha{\to}\mathtt{n}[\mathrm{tree}(\mathtt{f})] * ((\alpha{\to}\mathtt{n}) \mathrel{-\!\!*} P)\}$	`subDelete(n)`	$\{P\}$
$\{\exists \alpha, \beta.\, \alpha{\to}\mathtt{n}[\beta] * ((\alpha{\to}\mathtt{X} \mid \mathtt{n}[\beta]) \mathrel{-\!\!*} P)\}$	`insertBefore(n, X)`	$\{P\}$
$\{\exists \alpha, \beta.\, \alpha{\to}\mathtt{n}[\beta] * ((\alpha{\to}\mathtt{n}[\beta] \mid \mathtt{X}) \mathrel{-\!\!*} P)\}$	`insertAfter(n, X)`	$\{P\}$
$\{\exists \alpha, \beta.\, \alpha{\to}\mathtt{n}[\beta] * ((\alpha{\to}\mathtt{n}[\mathtt{X} \mid \beta]) \mathrel{-\!\!*} P)\}$	`insertFirst(n, X)`	$\{P\}$
$\{\exists \alpha, \beta.\, \alpha{\to}\mathtt{n}[\beta] * ((\alpha{\to}\mathtt{n}[\beta \mid \mathtt{X}]) \mathrel{-\!\!*} P)\}$	`insertLast(n, X)`	$\{P\}$
$\{\exists \alpha, \beta, \delta, \mathtt{f}.\, \alpha{\to}\mathtt{t}[\beta] * \delta{\to}\mathtt{c}[\mathrm{tree}(\mathtt{f})] * ((\alpha{\to}\mathtt{c}[\mathrm{tree}(\mathtt{f})] \mid \mathtt{t}[\beta] * \delta{\to}\varnothing_F) \mathrel{-\!\!*} P)\}$	`moveBefore(t, c)`	$\{P\}$
$\{\exists \alpha, \beta, \delta, \mathtt{f}.\, \alpha{\to}\mathtt{t}[\beta] * \delta{\to}\mathtt{c}[\mathrm{tree}(\mathtt{f})] * ((\alpha{\to}\mathtt{t}[\beta] \mid \mathtt{c}[\mathrm{tree}(\mathtt{f})] * \delta{\to}\varnothing_F) \mathrel{-\!\!*} P)\}$	`moveAfter(t, c)`	$\{P\}$
$\{\exists \alpha, \beta, \delta, \mathtt{f}.\, \alpha{\to}\mathtt{p}[\beta] * \delta{\to}\mathtt{c}[\mathrm{tree}(\mathtt{f})] * ((\alpha{\to}\mathtt{p}[\mathtt{c}[\mathrm{tree}(\mathtt{f})] \mid \beta] * \delta{\to}\varnothing_F) \mathrel{-\!\!*} P)\}$	`prepend(p, c)`	$\{P\}$
$\{\exists \alpha, \beta, \delta, \mathtt{f}.\, \alpha{\to}\mathtt{p}[\beta] * \delta{\to}\mathtt{c}[\mathrm{tree}(\mathtt{f})] * ((\alpha{\to}\mathtt{p}[\beta \mid \mathtt{c}[\mathrm{tree}(\mathtt{f})]] * \delta{\to}\varnothing_F) \mathrel{-\!\!*} P)\}$	`append(p, c)`	$\{P\}$

5.3.9 Deriving the Weakest Preconditions

We can show that the weakest preconditions for our update commands can be derived from their corresponding command axioms (see A for full proofs). This means that our Local Hoare Reasoning is complete for straight line code.

5.3.10 Examples

We shall now give a few examples of the tree update reasoning we can carry out with this Context Logic over Clusters.

a) A simple example of the use of the fragment separation connective $*$ is in the cluster formula $\alpha \rightarrow \mathtt{n}[\beta] * \beta \rightarrow \mathtt{m} \mid \mathtt{p}$ which describes a simple 3 node tree that we have split into two tree fragments. This is shown in figure 19.

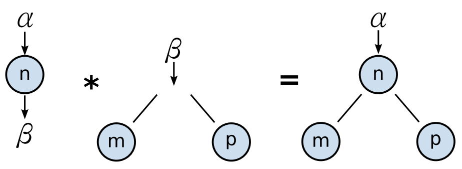

Figure 19: The cluster $\alpha \rightarrow \mathtt{n}[\beta] * \beta \rightarrow \mathtt{m} \mid \mathtt{p}$

b) Our logic allows us to reason at a more local level than we could before. Consider the `insertAfter` command. In previous work [17] the axiom of this command was given as:

$$\{\mathtt{n[t]}\}$$
$$\mathtt{insertAfter(n, X)}$$
$$\{\mathtt{n[t]} \mid \mathtt{X}\}$$

where the tree variable `t` is used to assert that the subtree beneath node `n` is not modified by the command. We are now able to give the axiom of this command as:

$$\{\alpha \rightarrow \mathtt{n}[\beta]\}$$
$$\mathtt{insertAfter(n, X)}$$
$$\{\alpha \rightarrow \mathtt{n}[\beta] \mid \mathtt{X}\}$$

which uses the context hole β instead of explicitly mentioning the entire subtree. The locality of the command, and the Frame Rule, automatically ensure that whatever is placed into the context hole β is not modified by the command.

c) The $-\!\!*$ connective allows us to specify future properties of a cluster. This is most commonly used when specifying the weakest preconditions of our commands. For example we might have the formula $(\alpha \rightarrow \mathtt{n}[\beta] \mid \mathtt{X}) -\!\!* P$ which specifies that when we add the fragment with a tree of shape `X` to the right of the node `n`, then some property P will hold. If our current cluster does not have such a tree beside node `n`, we can use the insertAfter command to put that tree in place as follows:

$$\begin{array}{l}\{\alpha{\rightarrow}\mathtt{n}[\beta] * ((\alpha{\rightarrow}\mathtt{n}[\beta] \mid \mathtt{X}) -\!\!* P)\} \\ \mathtt{insertAfter(n, X)} \\ \{\alpha{\rightarrow}\mathtt{n}[\beta] \mid \mathtt{X} * ((\alpha{\rightarrow}\mathtt{n}[\beta] \mid \mathtt{X}) -\!\!* P)\} \\ \{P\}\end{array}$$

This example shows how the interaction between $*$ and $-\!\!*$ closely resembles the behavior of these connectives in Separation Logic.

d) The principle power of local reasoning is the ability to build up the specifications of complex commands from the simple specifications of the basic commands. For example, let us consider the composite command `replaceTree` (n,X) which replaces the tree starting from node `n` with a tree of shape `X`. Rather than producing the operational semantics of this command then creating an axiom from this, we can instead simply use the specifications of our basic commands to deduce the axiom for this more complex command. Consider the following implementation of the `replaceTree` command:

```
replaceTree(n, X) ≜  insertAfter(n, X);
                     subDelete(n);
                     delete(n)
```

We can use the existing command axioms, along with the inference rules of our reasoning system, to derive the axiom of this composite command as follows:

$$\begin{array}{l}\{\alpha{\rightarrow}\mathtt{n}[\mathrm{tree}(\mathtt{f})]\} \\ \{\alpha{\rightarrow}\mathtt{n}[\beta] * \beta{\rightarrow}\mathrm{tree}(\mathtt{f})\} \\ \mathtt{insertAfter(n, X);} \\ \{\alpha{\rightarrow}\mathtt{n}[\beta] \mid \mathtt{X} * \beta{\rightarrow}\mathrm{tree}(\mathtt{f})\} \\ \{\alpha{\rightarrow}\mathtt{n}[\mathrm{tree}(\mathtt{f})] \mid \mathtt{X}\} \\ \{\alpha{\rightarrow}\beta \mid \mathtt{X} * \beta{\rightarrow}\mathtt{n}[\mathrm{tree}(\mathtt{f})]\} \\ \mathtt{subDelete(n);} \\ \{\alpha{\rightarrow}\beta \mid \mathtt{X} * \beta{\rightarrow}\mathtt{n}\} \\ \{\alpha{\rightarrow}\beta \mid \mathtt{X} * \beta{\rightarrow}\mathtt{n}[\delta] * \delta{\rightarrow}\varnothing_F\} \\ \mathtt{delete(n)} \\ \{\alpha{\rightarrow}\beta \mid \mathtt{X} * \beta{\rightarrow}\delta * \delta{\rightarrow}\varnothing_F\} \\ \{\alpha{\rightarrow}\beta \mid \mathtt{X} * \beta{\rightarrow}\varnothing_F\} \\ \{\alpha{\rightarrow}\varnothing_F \mid \mathtt{X}\} \\ \{\alpha{\rightarrow}\mathtt{X}\}\end{array}$$

So the axiom for the `replaceTree` command is as one would expect:

$$\begin{array}{l}\{\alpha{\rightarrow}\mathtt{n}[\mathrm{tree}(\mathtt{f})]\} \\ \mathtt{replaceTree(n, X)} \\ \{\alpha{\rightarrow}\mathtt{X}\}\end{array}$$

5.4 Specification Size

The specifications for our commands are now closer to the footprint size then we had before. All of the non-move commands have genuinely small axioms, as we could have managed with a small extension to the multi-holed logic discussed in 4.2.1. However, we have also managed to reduce the size of the specifications of our move commands. If we recall our previous attempt at providing a specification for the `append` command resulted in an axiom of the form:

$$\begin{array}{l}\{(\varnothing \multimap (\mathsf{c} \circ \mathtt{parent[t']})) \circ \mathtt{child[t]}\}\\ \mathtt{append(parent, child)}\\ \{\mathsf{c} \circ \mathtt{parent[t'} \otimes \mathtt{child[t]]}\}\end{array}$$

Now we are able to instead provide a smaller axiom for append:

$$\begin{array}{l}\{\alpha{\to}\mathsf{p}[\beta] * \delta{\to}\mathsf{c}[\mathrm{tree}(\mathtt{f})]\}\\ \mathtt{append(p, c)}\\ \{\alpha{\to}\mathsf{p}[\beta \mid \mathsf{c}[\mathrm{tree}(\mathtt{f})]] * \delta{\to}\varnothing_F\}\end{array}$$

Compared to our previous axiom for this command we have made a significant reduction in specification size. Most importantly we have removed the necessity to use an arbitrary context variable c to link the parent and child nodes in the initial tree. However, we have also managed to replace the subtree of the parent node with a context hole, further reducing the specification size. Unfortunately we cannot perform the same reduction in size beneath the child node, as we must ensure that the child is not an ancestor of the parent node. We enforce this property on the initial cluster by the definition of $*$. If the parent node where to appear beneath the child node then we would have a formula of the form:

$$\alpha{\to}\mathsf{p}[\beta] * \delta{\to}\mathsf{c}[\Diamond\mathsf{p}[\mathrm{true}]]$$

which has a clash of unique identifier p and so is not a well formed cluster. Figure 20 shows how the specification size we can now provide lies in between our previous attempt and the size that we would like to achieve.

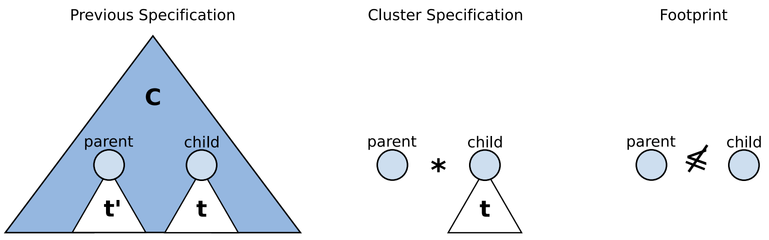

Figure 20: Comparing Specification Sizes

We have not yet reached our goal of a truly local specification of the move command, but we have taken a first step.

5.5 The Future of Move

We still have more work to do on improving the specifications we are able to provide for the move commands of our tree update language. Over the next 2-3 months we will be refining the initial approach we have taken here to provide a reasoning system that is capable of specifying the move commands in a fully local fashion.

5.5.1 Further Reductions to Move Axioms

As we mentioned above, our specifications over the move commands are significantly smaller than previous attempts, but are still not as small as the footprints of these commands. Our most pressing task is to try and find a way of extending the reasoning system given above to be able to further reduce the size of the move axioms. We have not actively pushed in this direction yet, wanting to be sure that the basic ideas of our framework were sound first, but we are hoping to introduce the idea of a 'restricted wiring'. That is some logical formula which has no physical structure, but restricts how the rest of the cluster may be extended or 'wired up'. So for example we hope to be able to express a formula at the location of a context hole that prevents our structure from looping. This would probably look something like:

$$\alpha \rightarrow \mathtt{n}[\beta \wedge (\beta \neq \alpha)]$$

Then we might be able to apply a similar strategy to the axiom of the move commands to prevent us from adding a cluster with the frame rule that would put the child above the parent. If we can succeed in this then we can have a move axiom that is the same size as the command footprint, without the need to explicitly specify the child's subtree in the axiom.

$$\{\alpha \rightarrow \mathtt{parent}[\beta] * \gamma \rightarrow \mathtt{child}[\delta] \wedge (\delta \neq \alpha)\}$$

However, we need to ensure that these restrictions are updated correctly when we use the frame rule to extend the tree fragments, otherwise we risk providing an incorrect or misleading specification.

6 Long Term Goals

Using our Context Logic for Clusters we have provided a local reasoning framework that is capable of reasoning about disjoint data structures in a local fashion. We also have a specification for move that is very nearly the same size as its footprint. In this final section we shall outline the long term goals of this PhD.

6.1 Concurrency and DOM

Undoubtedly, one the of fastest growing research areas in modern computer science is that of concurrent programming. We recently attended the 'Verification of Concurrent Algorithms' workshop at Microsoft Research Cambridge where the state of the art in concurrency theory was discussed. Separation Logic is already playing a crucial role in this area, and it is our hope that we will also be able to make significant headway at the higher level with Context Logic. We are working very hard at extending our reasoning framework to handle the concurrent setting and are beginning to tackle the initial issues of resource sharing and transactions. As we mentioned above, our reasoning is not quite fine-grained enough to be applied directly, but once we further reduce our specification sizes, we are confident that we will be able to tackle this complex area head on. We are aiming that within the next 2 years we should be able to provide a specification of a Concurrent DOM. There already exist implementations that allow for the concurrent use of DOM commands, but the whole of the existing DOM specification is limited to the sequential case. Providing a rigorous definition of this concurrent behavior would prove a good test of the power of our logic in a concurrent setting, as well as providing some clarity to what is currently a fairly vague area of understanding.

6.2 Formal Analysis of Footprints

We have a notion of a command's footprint, the parts of the data structure that the command may access, and we are able to provide specifications for our commands, but what does it mean to say that our command specifications have the correct size when compared to the command footprints? We need to carry out some investigation into what it means for our definitions to be right and we also want to be able to provide some notion of when two specifications are the same. We mentioned in section 4.1.1 that we could provide a close implementation of the move commands with the combination of 3 other basic commands, but that the specification sizes would not match up. We intend to investigate these footprint sizes further and see if it is still the case in our new cluster based reasoning model that move is in some sense a larger command than its composite command equivalent.

6.3 High and Low Level Concurrency

Following the example set out in [17] we wish to further investigate the links between High and Low level reasoning. In particular we want to investigate the translation from High level to Low level in a concurrent setting and how properties such as the ability to run commands in parallel translate between these levels. For example take the high level parallel execution of the commands `insertLast(n,X)` and `dispose(m) = subDelete(m); delete(m)`. The success of this parallel execution depends on the relation between the nodes `n` and `m`. Figure 21 shows the 3 different cases and their success or failure.

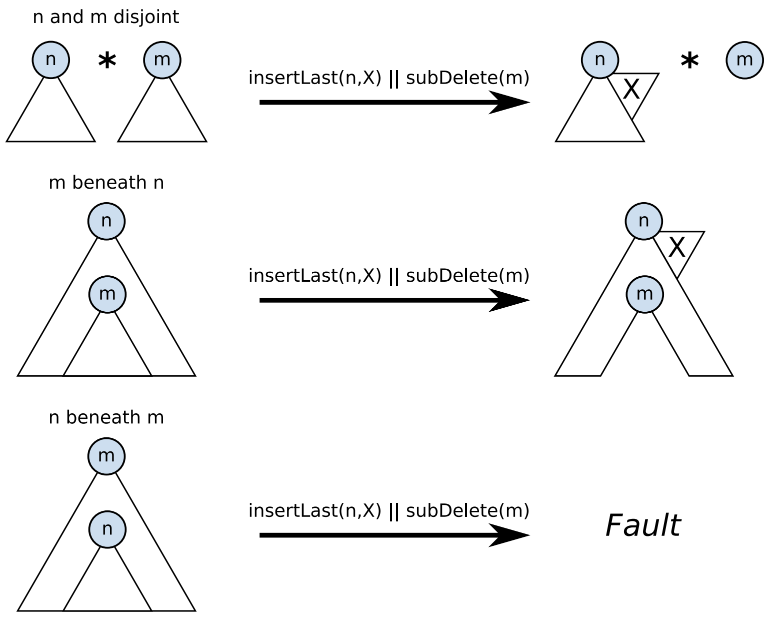
n and m disjoint
n
*
m
insertLast(n,X) || subDelete(m)
n
X
*
m
m beneath n
n
m
insertLast(n,X) || subDelete(m)
n
X
m
n beneath m
m
n
insertLast(n,X) || subDelete(m)
Fault

Figure 21: parallel execution of `insertAfter` (`n`,`X`) and `dispose` (`m`)

When we use the translation of [17] to move the specifications of these commands to the low level then we find that there is a second case where the parallel execution of the commands will fault. If the nodes `n` and `m` are siblings in the tree, then at the low level we get a footprint overlap in the commands due to the pointer maintenance that we need to carry out for the `dispose` command. This prevents us from being able to run the commands in parallel at the low level in this case. We want to investigate the occurrences of similar footprint increases and see if this is a feature of moving between our reasoning levels or if it is just a feature of the particular low level implementation that we are using.

References

[1] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. W. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. *Lecture Notes in Computer Science*, 4590, 2007.

[2] J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. *Lecture Notes in Computer Science*, 4111, 2005.

[3] J. Berdine, B. Cook, D. Distefano, and P. W. O'Hearn. Automatic termination proofs for programs with shape-shifting heaps. *Lecture Notes in Computer Science*, 4144, 2006.

[4] P. E. Black. Binary tree. in Dictionary of Algorithms and Data Structures. http://www.nist.gov.dads/HTML/binarytree.html.

[5] S. Brookes. A semantics for concurrent separation logic. *Theor. Comput. Sci.*, 375, 2007.

[6] C. Calcagno, T. Dinsdale-Young, and P. Gardner. Adjuct elimination in context logic for trees. In *APLAS*, 2007.

[7] C. Calcagno, P. Gardner, and U. Zarfaty. A context logic for tree update, 2004.

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

[9] C. Calcagno, P. Gardner, and U. Zarfaty. Context logic as modal logic: completeness and parametric inexpressivity. *SIGPLAN Not.*, 2007.

[10] C. Calcagno, P. Gardner, and U. Zarfaty. Local reasoning about data update. *Electronic Notes in Theoretical Computer Science*, 172, 2007.

[11] C. Calcagno and P. W. O'Hearn. On garbage and program logic. *Lecture Notes in Computer Science*, 2030, 2001.

[12] L. Cardelli and A. D. Gordon. Anytime, anywhere, modal logics for mobile ambients, 2000.

[13] D. Distefano, P. O'Hearn, and H. Yang. A local shape analysis based on separation logic, 2006.

[14] X. Feng, R. Ferreira, and Z. Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. *Lecture Notes in Computer Science*, 4421, 2007.

[15] P. Gardner, G. Smith, M. Wheelhouse, and U. Zarfaty. Dom: Towards a formal specification. In *Plan-X*, 2008.

[16] P. Gardner, G. Smith, M. Wheelhouse, and U. Zarfaty. Local hoare reasoning about dom. In *PODS*, 2008.

[17] P. Gardner and U. Zarfaty. Integrated reasoning about high-level tree update and a low-level implementation. 2008.

[18] C. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 1969.

[19] S. Ishtiaq and P. W. O'Hearn. Bi as an assertion language for mutable data structures. In *POPL*, 2001.

[20] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. *Lecture Notes in Computer Science*, 2142, 2001.

[21] P. W. O'Hearn. Resources, concurrency and local reasoning. *Theor. Comput. Sci.*, 375, 2007.

[22] M. Parkinson and V. Vafeiadis. A marriage of rely/guarantee and separation logic. *Lecture Notes in Computer Science*, 4703, 2007.

[23] M. Raza and P. Gardner. Footprints in local reasoning. *Lecture Notes in Computer Science*, 4962, 2008.

[24] M. Research. Slayer: automatic verification tool, 2006. http://research.microsoft.com/SLAyer/.

[25] W3C. Dom: Document object model. W3C recommendation, 2005. http://www.w3.org/DOM/.

[26] U. Zarfaty. Notes on disjoint wirings. Unpublished 2008.

Appendix

A Weakest Precondition Derivations

The derivations for the weakest preconditions for each of our update commands are given in this appendix. Several of the proofs rely on the following property.

Lemma:

$$(Q_C \mathrel{-\!\!*} P_C) * Q_C \Leftrightarrow (\Diamond Q_C) \wedge P_C$$

proof: Follows from the fact that out data structure is precise. The reverse direction of implication is not true in general, only in the case where the data structure is precise.

skip

$$\begin{array}{ll}
\{\varnothing_C\} & \\
\mathtt{skip} & \\
\{\varnothing_C\} & \text{F} \\
\hline
\{(\varnothing_C) * (\varnothing_C \mathrel{-\!\!*} P)\} & \\
\mathtt{skip} & \\
\{(\varnothing_C) * (\varnothing_C \mathrel{-\!\!*} P)\} & \text{C} \\
\hline
\{P\} & \\
\mathtt{skip} & \\
\{P\} &
\end{array}$$

assignment

$$\begin{array}{lll}
\{\varnothing_C \wedge (\mathtt{v} = \mathtt{E})\} & & \\
\mathtt{x} := \mathtt{E} & & \\
\{\varnothing_C \wedge (\mathtt{x} = \mathtt{v})\} & \text{F} & \\
\hline
\{(\varnothing_C \wedge (\mathtt{v} = \mathtt{E})) * ((\varnothing_C) \mathrel{-\!\!*} P[\mathtt{v}/\mathtt{x}]\} & & \\
\mathtt{x} := \mathtt{E} & & \\
\{(\varnothing_C \wedge (\mathtt{x} = \mathtt{v})) * ((\varnothing_C) \mathrel{-\!\!*} P[\mathtt{v}/\mathtt{x}]\} & \text{C} & \\
\hline
\{(\varnothing_C) * ((\varnothing_C) \mathrel{-\!\!*} P[\mathtt{v}/\mathtt{x}] \wedge (\mathtt{v} = \mathtt{E})\} & & \\
\mathtt{x} := \mathtt{E} & & \\
\{(\varnothing_C) * ((\varnothing_C) \mathrel{-\!\!*} P[\mathtt{v}/\mathtt{x}] \wedge (\mathtt{x} = \mathtt{v})\} & \text{C} & \\
\hline
\{P[\mathtt{v}/\mathtt{x}] \wedge (\mathtt{v} = \mathtt{E})\} & & \\
\mathtt{x} := \mathtt{E} & & \\
\{P[\mathtt{v}/\mathtt{x}] \wedge (\mathtt{x} = \mathtt{v})\} & \text{C} & /\text{E} \\
\hline
\{\exists \mathtt{v}.\, P[\mathtt{v}/\mathtt{x}] \wedge (\mathtt{v} = \mathtt{E})\} & & \\
\mathtt{x} := \mathtt{E} & & \\
\{P\} & &
\end{array}$$

copy

$$\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{l}\{\alpha\to\mathrm{n}[\mathrm{tree}(\mathtt{f})] \wedge (\mathtt{X} = \langle\mathrm{n}[\mathtt{f}]\rangle)\}\\ \mathtt{t} := \mathrm{copy(n)}\\ \{\alpha\to\mathrm{n}[\mathrm{tree}(\mathtt{f})] \wedge (\mathtt{t} = \mathtt{X})\}\end{array}}{\begin{array}{l}\{(\alpha\to\mathrm{n}[\mathrm{tree}(\mathtt{f})] \wedge (\mathtt{X} = \langle\mathrm{n}[\mathtt{f}]\rangle)) * ((\alpha\to\mathrm{n}[\mathrm{tree}(\mathtt{f})]) \mathrel{-\!\!*} P[\mathtt{X}/\mathtt{t}])\}\\ \mathtt{t} := \mathrm{copy(n)}\\ \{(\alpha\to\mathrm{n}[\mathrm{tree}(\mathtt{f})] \wedge (\mathtt{t} = \mathtt{X})) * ((\alpha\to\mathrm{n}[\mathrm{tree}(\mathtt{f})]) \mathrel{-\!\!*} P[\mathtt{X}/\mathtt{t}])\}\end{array}}\text{F}}{\begin{array}{l}\{(\alpha\to\mathrm{n}[\mathrm{tree}(\mathtt{f})]) * ((\alpha\to\mathrm{n}[\mathrm{tree}(\mathtt{f})]) \mathrel{-\!\!*} P[\mathtt{X}/\mathtt{t}]) \wedge (\mathtt{X} = \langle\mathrm{n}[\mathtt{f}]\rangle)\}\\ \mathtt{t} := \mathrm{copy(n)}\\ \{(\alpha\to\mathrm{n}[\mathrm{tree}(\mathtt{f})]) * ((\alpha\to\mathrm{n}[\mathrm{tree}(\mathtt{f})]) \mathrel{-\!\!*} P[\mathtt{X}/\mathtt{t}]) \wedge (\mathtt{t} = \mathtt{X})\}\end{array}}\text{C}}{\begin{array}{l}\{\Diamond(\alpha\to\mathrm{n}[\mathrm{tree}(\mathtt{f})]) \wedge P[\mathtt{X}/\mathtt{t}] \wedge (\mathtt{X} = \langle\mathrm{n}[\mathtt{f}]\rangle)\}\\ \mathtt{t} := \mathrm{copy(n)}\\ \{P[\mathtt{X}/\mathtt{t}] \wedge (\mathtt{t} = \mathtt{X})\}\end{array}}\text{C}}{\begin{array}{l}\{\exists\alpha, \mathtt{f}, \mathtt{X}.\, \Diamond(\alpha\to\mathrm{n}[\mathrm{tree}(\mathtt{f})]) \wedge P[\mathtt{X}/\mathtt{t}] \wedge (\mathtt{X} = \langle\mathrm{n}[\mathtt{f}]\rangle)\}\\ \mathtt{t} := \mathrm{copy(n)}\\ \{P\}\end{array}}\text{C}\quad /\text{E}$$

getUp

$$\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{l}\{\alpha\to\mathrm{p}[\beta \mid \mathrm{n}[\delta] \mid \gamma]\}\\ \mathtt{n'} := \mathrm{getUp(n)}\\ \{\alpha\to\mathrm{p}[\beta \mid \mathrm{n}[\delta] \mid \gamma] \wedge (\mathtt{n'} = \mathrm{p})\}\end{array}}{\begin{array}{l}\{(\alpha\to\mathrm{p}[\beta \mid \mathrm{n}[\delta] \mid \gamma]) * ((\alpha\to\mathrm{p}[\beta \mid \mathrm{n}[\delta] \mid \gamma]) \mathrel{-\!\!*} P[\mathrm{p}/\mathrm{n}'])\}\\ \mathtt{n'} := \mathrm{getUp(n)}\\ \{(\alpha\to\mathrm{p}[\beta \mid \mathrm{n}[\delta] \mid \gamma] \wedge (\mathtt{n'} = \mathrm{p})) * ((\alpha\to\mathrm{p}[\beta \mid \mathrm{n}[\delta] \mid \gamma]) \mathrel{-\!\!*} P[\mathrm{p}/\mathrm{n}'])\}\end{array}}\text{F}}{\begin{array}{l}\{(\alpha\to\mathrm{p}[\beta \mid \mathrm{n}[\delta] \mid \gamma]) * ((\alpha\to\mathrm{p}[\beta \mid \mathrm{n}[\delta] \mid \gamma]) \mathrel{-\!\!*} P[\mathrm{p}/\mathrm{n}'])\}\\ \mathtt{n'} := \mathrm{getUp(n)}\\ \{(\alpha\to\mathrm{p}[\beta \mid \mathrm{n}[\delta] \mid \gamma]) * ((\alpha\to\mathrm{p}[\beta \mid \mathrm{n}[\delta] \mid \gamma]) \mathrel{-\!\!*} P[\mathrm{p}/\mathrm{n}']) \wedge (\mathtt{n'} = \mathrm{p})\}\end{array}}\text{C}}{\begin{array}{l}\{\Diamond(\alpha\to\mathrm{p}[\beta \mid \mathrm{n}[\delta] \mid \gamma]) \wedge P[\mathrm{p}/\mathrm{n}']\}\\ \mathtt{n'} := \mathrm{getUp(n)}\\ \{P[\mathrm{p}/\mathrm{n}'] \wedge (\mathtt{n'} = \mathrm{p})\}\end{array}}\text{C}}{\begin{array}{l}\{\exists\alpha, \beta, \gamma, \delta, \mathrm{p}.\, \Diamond(\alpha\to\mathrm{p}[\beta \mid \mathrm{n}[\delta] \mid \gamma]) \wedge P[\mathrm{p}/\mathrm{n}']\}\\ \mathtt{n'} := \mathrm{getUp(n)}\\ \{P\}\end{array}}\text{C}\quad /\text{E}$$

getLeft

$$\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{l}\{\alpha\to\mathtt{l}[\beta] \mid \mathrm{n}[\delta]\}\\ \mathtt{n'} := \mathtt{getLeft}(\mathrm{n})\\ \{\alpha\to\mathtt{l}[\beta] \mid \mathrm{n}[\delta] \wedge (\mathtt{n'} = \mathtt{l})\}\end{array}}{\begin{array}{l}\{(\alpha\to\mathtt{l}[\beta] \mid \mathrm{n}[\delta]) * ((\alpha\to\mathtt{l}[\beta] \mid \mathrm{n}[\delta]) \mathrel{-\!\!*} P[\mathtt{l}/\mathrm{n}'])\}\\ \mathtt{n'} := \mathtt{getLeft}(\mathrm{n})\\ \{(\alpha\to\mathtt{l}[\beta] \mid \mathrm{n}[\delta] \wedge (\mathtt{n'} = \mathtt{l})) * ((\alpha\to\mathtt{l}[\beta] \mid \mathrm{n}[\delta]) \mathrel{-\!\!*} P[\mathtt{l}/\mathrm{n}'])\}\end{array}}\text{F}}{\begin{array}{l}\{(\alpha\to\mathtt{l}[\beta] \mid \mathrm{n}[\delta]) * ((\alpha\to\mathtt{l}[\beta] \mid \mathrm{n}[\delta]) \mathrel{-\!\!*} P[\mathtt{l}/\mathrm{n}'])\}\\ \mathtt{n'} := \mathtt{getLeft}(\mathrm{n})\\ \{(\alpha\to\mathtt{l}[\beta] \mid \mathrm{n}[\delta]) * ((\alpha\to\mathtt{l}[\beta] \mid \mathrm{n}[\delta]) \mathrel{-\!\!*} P[\mathtt{l}/\mathrm{n}']) \wedge (\mathtt{n'} = \mathtt{l})\}\end{array}}\text{C}}{\begin{array}{l}\{\Diamond(\alpha\to\mathtt{l}[\beta] \mid \mathrm{n}[\delta]) \wedge P[\mathtt{l}/\mathrm{n}']\}\\ \mathtt{n'} := \mathtt{getLeft}(\mathrm{n})\\ \{P[\mathtt{l}/\mathrm{n}'] \wedge (\mathtt{n'} = \mathtt{l})\}\end{array}}\text{C}}{\begin{array}{l}\{\exists\alpha, \beta, \delta, \mathtt{l}.\, \Diamond(\alpha\to\mathtt{l}[\beta] \mid \mathrm{n}[\delta]) \wedge P[\mathtt{l}/\mathrm{n}']\}\\ \mathtt{n'} := \mathtt{getLeft}(\mathrm{n})\\ \{P\}\end{array}}\text{C}\quad /\text{E}$$

getRight

$$\begin{array}{ll}
\{\alpha{\to}\mathtt{n}[\delta] \mid \mathtt{r}[\beta]\} & \\
\mathtt{n}' := \mathtt{getRight(n)} & \\
\{\alpha{\to}\mathtt{n}[\delta] \mid \mathtt{r}[\beta] \wedge (\mathtt{n'} = \mathtt{r})\} & \mathrm{F} \\ \cline{1-1}
\{(\alpha{\to}\mathtt{n}[\delta] \mid \mathtt{r}[\beta]) * ((\alpha{\to}\mathtt{n}[\delta] \mid \mathtt{r}[\beta]) \mathrel{-\!\!*} P[\mathtt{r}/\mathtt{n}'])\} & \\
\mathtt{n}' := \mathtt{getRight(n)} & \\
\{(\alpha{\to}\mathtt{n}[\delta] \mid \mathtt{r}[\beta] \wedge (\mathtt{n'} = \mathtt{r})) * ((\alpha{\to}\mathtt{n}[\delta] \mid \mathtt{r}[\beta]) \mathrel{-\!\!*} P[\mathtt{r}/\mathtt{n}'])\} & \mathrm{C} \\ \cline{1-1}
\{(\alpha{\to}\mathtt{n}[\delta] \mid \mathtt{r}[\beta]) * ((\alpha{\to}\mathtt{n}[\delta] \mid \mathtt{r}[\beta]) \mathrel{-\!\!*} P[\mathtt{r}/\mathtt{n}'])\} & \\
\mathtt{n}' := \mathtt{getRight(n)} & \\
\{(\alpha{\to}\mathtt{n}[\delta] \mid \mathtt{r}[\beta]) * ((\alpha{\to}\mathtt{n}[\delta] \mid \mathtt{r}[\beta]) \mathrel{-\!\!*} P[\mathtt{r}/\mathtt{n}']) \wedge (\mathtt{n'} = \mathtt{r})\} & \mathrm{C} \\ \cline{1-1}
\{\Diamond(\alpha{\to}\mathtt{n}[\delta] \mid \mathtt{r}[\beta]) \wedge P[\mathtt{r}/\mathtt{n}']\} & \\
\mathtt{n}' := \mathtt{getRight(n)} & \\
\{P[\mathtt{r}/\mathtt{n}'] \wedge (\mathtt{n'} = \mathtt{r})\} & \mathrm{C} \quad /\mathrm{E} \\ \cline{1-1}
\{\exists \alpha, \beta, \delta, \mathtt{r}.\, \Diamond(\alpha{\to}\mathtt{n}[\delta] \mid \mathtt{r}[\beta]) \wedge P[\mathtt{r}/\mathtt{n}']\} & \\
\mathtt{n}' := \mathtt{getRight(n)} & \\
\{P\} &
\end{array}$$

getFirst

$$\begin{array}{ll}
\{\alpha{\to}\mathtt{n}[\mathtt{c}[\beta] \mid \delta]\} & \\
\mathtt{n}' := \mathtt{getFirst(n)} & \\
\{\alpha{\to}\mathtt{n}[\mathtt{c}[\beta] \mid \delta] \wedge (\mathtt{n'} = \mathtt{c})\} & \mathrm{F} \\ \cline{1-1}
\{(\alpha{\to}\mathtt{n}[\mathtt{c}[\beta] \mid \delta]) * ((\alpha{\to}\mathtt{n}[\mathtt{c}[\beta] \mid \delta]) \mathrel{-\!\!*} P[\mathtt{c}/\mathtt{n}'])\} & \\
\mathtt{n}' := \mathtt{getFirst(n)} & \\
\{(\alpha{\to}\mathtt{n}[\mathtt{c}[\beta] \mid \delta] \wedge (\mathtt{n'} = \mathtt{c})) * ((\alpha{\to}\mathtt{n}[\mathtt{c}[\beta] \mid \delta]) \mathrel{-\!\!*} P[\mathtt{c}/\mathtt{n}'])\} & \mathrm{C} \\ \cline{1-1}
\{(\alpha{\to}\mathtt{n}[\mathtt{c}[\beta] \mid \delta]) * ((\alpha{\to}\mathtt{n}[\mathtt{c}[\beta] \mid \delta]) \mathrel{-\!\!*} P[\mathtt{c}/\mathtt{n}'])\} & \\
\mathtt{n}' := \mathtt{getFirst(n)} & \\
\{(\alpha{\to}\mathtt{n}[\mathtt{c}[\beta] \mid \delta]) * ((\alpha{\to}\mathtt{n}[\mathtt{c}[\beta] \mid \delta]) \mathrel{-\!\!*} P[\mathtt{c}/\mathtt{n}']) \wedge (\mathtt{n'} = \mathtt{c})\} & \mathrm{C} \\ \cline{1-1}
\{\Diamond(\alpha{\to}\mathtt{n}[\mathtt{c}[\beta] \mid \delta]) \wedge P[\mathtt{c}/\mathtt{n}']\} & \\
\mathtt{n}' := \mathtt{getFirst(n)} & \\
\{P[\mathtt{c}/\mathtt{n}'] \wedge (\mathtt{n'} = \mathtt{c})\} & \mathrm{C} \quad /\mathrm{E} \\ \cline{1-1}
\{\exists \alpha, \beta, \delta, \mathtt{c}.\, \Diamond(\alpha{\to}\mathtt{n}[\mathtt{c}[\beta] \mid \delta]) \wedge P[\mathtt{c}/\mathtt{n}']\} & \\
\mathtt{n}' := \mathtt{getFirst(n)} & \\
\{P\} &
\end{array}$$

getLast

$$\begin{array}{ll}
\{\alpha{\to}\mathtt{n}[\delta \mid \mathtt{c}[\beta]]\} & \\
\mathtt{n}' := \mathtt{getLast(n)} & \\
\{\alpha{\to}\mathtt{n}[\delta \mid \mathtt{c}[\beta]] \wedge (\mathtt{n'} = \mathtt{c})\} & \mathrm{F} \\ \cline{1-1}
\{(\alpha{\to}\mathtt{n}[\delta \mid \mathtt{c}[\beta]]) * ((\alpha{\to}\mathtt{n}[\delta \mid \mathtt{c}[\beta]]) \mathrel{-\!\!*} P[\mathtt{c}/\mathtt{n}'])\} & \\
\mathtt{n}' := \mathtt{getLast(n)} & \\
\{(\alpha{\to}\mathtt{n}[\delta \mid \mathtt{c}[\beta]] \wedge (\mathtt{n'} = \mathtt{c})) * ((\alpha{\to}\mathtt{n}[\delta \mid \mathtt{c}[\beta]]) \mathrel{-\!\!*} P[\mathtt{c}/\mathtt{n}'])\} & \mathrm{C} \\ \cline{1-1}
\{(\alpha{\to}\mathtt{n}[\delta \mid \mathtt{c}[\beta]]) * ((\alpha{\to}\mathtt{n}[\delta \mid \mathtt{c}[\beta]]) \mathrel{-\!\!*} P[\mathtt{c}/\mathtt{n}'])\} & \\
\mathtt{n}' := \mathtt{getLast(n)} & \\
\{(\alpha{\to}\mathtt{n}[\delta \mid \mathtt{c}[\beta]]) * ((\alpha{\to}\mathtt{n}[\delta \mid \mathtt{c}[\beta]]) \mathrel{-\!\!*} P[\mathtt{c}/\mathtt{n}']) \wedge (\mathtt{n'} = \mathtt{c})\} & \mathrm{C} \\ \cline{1-1}
\{\Diamond(\alpha{\to}\mathtt{n}[\delta \mid \mathtt{c}[\beta]]) \wedge P[\mathtt{c}/\mathtt{n}']\} & \\
\mathtt{n}' := \mathtt{getLast(n)} & \\
\{P[\mathtt{c}/\mathtt{n}'] \wedge (\mathtt{n'} = \mathtt{c})\} & \mathrm{C} \quad /\mathrm{E} \\ \cline{1-1}
\{\exists \alpha, \beta, \delta, \mathtt{c}.\, \Diamond(\alpha{\to}\mathtt{n}[\delta \mid \mathtt{c}[\beta]]) \wedge P[\mathtt{c}/\mathtt{n}']\} & \\
\mathtt{n}' := \mathtt{getLast(n)} & \\
\{P\} &
\end{array}$$

delete

$$\dfrac{\dfrac{\begin{array}{l}\{\alpha\rightarrow \mathrm{n}[\beta]\}\\ \texttt{delete(n)}\\ \{\alpha\rightarrow\beta\}\end{array}}{\begin{array}{l}\{\alpha\rightarrow \mathrm{n}[\beta] * ((\alpha\rightarrow\beta) \mathrel{-\!\!*} P)\}\\ \texttt{delete(n)}\\ \{\alpha\rightarrow\beta * ((\alpha\rightarrow\beta) \mathrel{-\!\!*} P)\}\end{array}}\mathrm{F}}{\begin{array}{l}\{\exists\alpha,\beta.\,\alpha\rightarrow \mathrm{n}[\beta] * ((\alpha\rightarrow\beta) \mathrel{-\!\!*} P)\}\\ \texttt{delete(n)}\\ \{P\}\end{array}}\mathrm{C}\quad /\mathrm{E}$$

subDelete

$$\dfrac{\dfrac{\begin{array}{l}\{\alpha\rightarrow \mathrm{n}[\mathrm{tree}(\mathtt{f})]\}\\ \texttt{subDelete(n)}\\ \{\alpha\rightarrow \mathrm{n}\}\end{array}}{\begin{array}{l}\{\alpha\rightarrow \mathrm{n}[\mathrm{tree}(\mathtt{f})] * ((\alpha\rightarrow \mathrm{n}) \mathrel{-\!\!*} P)\}\\ \texttt{subDelete(n)}\\ \{\alpha\rightarrow \mathrm{n} * ((\alpha\rightarrow \mathrm{n}) \mathrel{-\!\!*} P)\}\end{array}}\mathrm{F}}{\begin{array}{l}\{\exists\alpha,\mathtt{f}\alpha\rightarrow \mathrm{n}[\mathrm{tree}(\mathtt{f})] * ((\alpha\rightarrow \mathrm{n}) \mathrel{-\!\!*} P)\}\\ \texttt{subDelete(n)}\\ \{P\}\end{array}}\mathrm{C}\quad /\mathrm{E}$$

insertBefore

$$\dfrac{\dfrac{\begin{array}{l}\{\alpha\rightarrow \mathrm{n}[\beta]\}\\ \texttt{insertBefore(n, X)}\\ \{\alpha\rightarrow \mathtt{X} \mid \mathrm{n}[\beta]\}\end{array}}{\begin{array}{l}\{\alpha\rightarrow \mathrm{n}[\beta] * ((\alpha\rightarrow \mathtt{X} \mid \mathrm{n}[\beta]) \mathrel{-\!\!*} P)\}\\ \texttt{insertBefore(n, X)}\\ \{\alpha\rightarrow \mathtt{X} \mid \mathrm{n}[\beta] * ((\alpha\rightarrow \mathtt{X} \mid \mathrm{n}[\beta]) \mathrel{-\!\!*} P)\}\end{array}}\mathrm{F}}{\begin{array}{l}\{\exists\alpha,\beta.\,\alpha\rightarrow \mathrm{n}[\beta] * ((\alpha\rightarrow \mathtt{X} \mid \mathrm{n}[\beta]) \mathrel{-\!\!*} P)\}\\ \texttt{insertBefore(n, X)}\\ \{P\}\end{array}}\mathrm{C}\quad /\mathrm{E}$$

insertAfter

$$\dfrac{\dfrac{\begin{array}{l}\{\alpha\rightarrow \mathrm{n}[\beta]\}\\ \texttt{insertAfter(n, X)}\\ \{\alpha\rightarrow \mathrm{n}[\beta] \mid \mathtt{X}\}\end{array}}{\begin{array}{l}\{\alpha\rightarrow \mathrm{n}[\beta] * ((\alpha\rightarrow \mathrm{n}[\beta] \mid \mathtt{X}) \mathrel{-\!\!*} P)\}\\ \texttt{insertAfter(n, X)}\\ \{\alpha\rightarrow \mathrm{n}[\beta] \mid \mathtt{X} * ((\alpha\rightarrow \mathrm{n}[\beta] \mid \mathtt{X}) \mathrel{-\!\!*} P)\}\end{array}}\mathrm{F}}{\begin{array}{l}\{\exists\alpha,\beta.\,\alpha\rightarrow \mathrm{n}[\beta] * ((\alpha\rightarrow \mathrm{n}[\beta] \mid \mathtt{X}) \mathrel{-\!\!*} P)\}\\ \texttt{insertAfter(n, X)}\\ \{P\}\end{array}}\mathrm{C}\quad /\mathrm{E}$$

insertFirst

$$\dfrac{\dfrac{\begin{array}{l}\{\alpha\to\mathtt{n}[\beta]\}\\ \mathtt{insertFirst(n,X)}\\ \{\alpha\to\mathtt{n}[\mathtt{X}\mid\beta]\}\end{array}}{\begin{array}{l}\{\alpha\to\mathtt{n}[\beta]*((\alpha\to\mathtt{n}[\mathtt{X}\mid\beta])\mathrel{-\!\!*}P)\}\\ \mathtt{insertFirst(n,X)}\\ \{\alpha\to\mathtt{n}[\mathtt{X}\mid\beta]*((\alpha\to\mathtt{n}[\mathtt{X}\mid\beta])\mathrel{-\!\!*}P)\}\end{array}}\mathrm{F}}{\begin{array}{l}\{\exists\alpha,\beta.\,\alpha\to\mathtt{n}[\beta]*((\alpha\to\mathtt{n}[\mathtt{X}\mid\beta])\mathrel{-\!\!*}P)\}\\ \mathtt{insertFirst(n,X)}\\ \{P\}\end{array}}\mathrm{C}\quad /\mathrm{E}$$

insertLast

$$\dfrac{\dfrac{\begin{array}{l}\{\alpha\to\mathtt{n}[\beta]\}\\ \mathtt{insertLast(n,X)}\\ \{\alpha\to\mathtt{n}[\beta\mid\mathtt{X}]\}\end{array}}{\begin{array}{l}\{\alpha\to\mathtt{n}[\beta]*((\alpha\to\mathtt{n}[\beta\mid\mathtt{X}])\mathrel{-\!\!*}P)\}\\ \mathtt{insertLast(n,X)}\\ \{\alpha\to\mathtt{n}[\beta\mid\mathtt{X}]*((\alpha\to\mathtt{n}[\beta\mid\mathtt{X}])\mathrel{-\!\!*}P)\}\end{array}}\mathrm{F}}{\begin{array}{l}\{\exists\alpha,\beta.\,\alpha\to\mathtt{n}[\beta]*((\alpha\to\mathtt{n}[\beta\mid\mathtt{X}])\mathrel{-\!\!*}P)\}\\ \mathtt{insertLast(n,X)}\\ \{P\}\end{array}}\mathrm{C}\quad /\mathrm{E}$$

moveBefore

$$\dfrac{\dfrac{\begin{array}{l}\{\alpha\to\mathtt{t}[\beta]*\delta\to\mathtt{c}[\mathrm{tree}(\mathtt{f})]\}\\ \mathtt{moveBefore(t,c)}\\ \{\alpha\to\mathtt{c}[\mathrm{tree}(\mathtt{f})]\mid\mathtt{t}[\beta]*\delta\to\varnothing_F\}\end{array}}{\begin{array}{l}\{\alpha\to\mathtt{t}[\beta]*\delta\to\mathtt{c}[\mathrm{tree}(\mathtt{f})]*((\alpha\to\mathtt{c}[\mathrm{tree}(\mathtt{f})]\mid\mathtt{t}[\beta]*\delta\to\varnothing_F)\mathrel{-\!\!*}P)\}\\ \mathtt{moveBefore(t,c)}\\ \{\alpha\to\mathtt{c}[\mathrm{tree}(\mathtt{f})]\mid\mathtt{t}[\beta]*\delta\to\varnothing_F*((\alpha\to\mathtt{c}[\mathrm{tree}(\mathtt{f})]\mid\mathtt{t}[\beta]*\delta\to\varnothing_F)\mathrel{-\!\!*}P)\}\end{array}}\mathrm{F}}{\begin{array}{l}\{\exists\alpha,\beta,\delta,\mathtt{f}.\,\alpha\to\mathtt{t}[\beta]*\delta\to\mathtt{c}[\mathrm{tree}(\mathtt{f})]*((\alpha\to\mathtt{c}[\mathrm{tree}(\mathtt{f})]\mid\mathtt{t}[\beta]*\delta\to\varnothing_F)\mathrel{-\!\!*}P)\}\\ \mathtt{moveBefore(t,c)}\\ \{P\}\end{array}}\mathrm{C}\quad /\mathrm{E}$$

moveAfter

$$\dfrac{\dfrac{\begin{array}{l}\{\alpha\to\mathtt{t}[\beta]*\delta\to\mathtt{c}[\mathrm{tree}(\mathtt{f})]\}\\ \mathtt{moveAfter(t,c)}\\ \{\alpha\to\mathtt{t}[\beta]\mid\mathtt{c}[\mathrm{tree}(\mathtt{f})]*\delta\to\varnothing_F\}\end{array}}{\begin{array}{l}\{\alpha\to\mathtt{t}[\beta]*\delta\to\mathtt{c}[\mathrm{tree}(\mathtt{f})]*((\alpha\to\mathtt{t}[\beta]\mid\mathtt{c}[\mathrm{tree}(\mathtt{f})]*\delta\to\varnothing_F)\mathrel{-\!\!*}P)\}\\ \mathtt{moveAfter(t,c)}\\ \{\alpha\to\mathtt{t}[\beta]\mid\mathtt{c}[\mathrm{tree}(\mathtt{f})]*\delta\to\varnothing_F*((\alpha\to\mathtt{t}[\beta]\mid\mathtt{c}[\mathrm{tree}(\mathtt{f})]*\delta\to\varnothing_F)\mathrel{-\!\!*}P)\}\end{array}}\mathrm{F}}{\begin{array}{l}\{\exists\alpha,\beta,\delta,\mathtt{f}.\,\alpha\to\mathtt{t}[\beta]*\delta\to\mathtt{c}[\mathrm{tree}(\mathtt{f})]*((\alpha\to\mathtt{t}[\beta]\mid\mathtt{c}[\mathrm{tree}(\mathtt{f})]*\delta\to\varnothing_F)\mathrel{-\!\!*}P)\}\\ \mathtt{moveAfter(t,c)}\\ \{P\}\end{array}}\mathrm{C}\quad /\mathrm{E}$$

prepend

$$\dfrac{\dfrac{\begin{array}{l}\{\alpha{\to}\mathtt{p}[\beta] * \delta{\to}\mathtt{c}[\mathrm{tree}(\mathtt{f})]\}\\ \mathtt{prepend}(\mathtt{p},\mathtt{c})\\ \{\alpha{\to}\mathtt{p}[\mathtt{c}[\mathrm{tree}(\mathtt{f})] \mid \beta] * \delta{\to}\varnothing_F\}\end{array}}{\begin{array}{l}\{\alpha{\to}\mathtt{p}[\beta] * \delta{\to}\mathtt{c}[\mathrm{tree}(\mathtt{f})] * ((\alpha{\to}\mathtt{p}[\mathtt{c}[\mathrm{tree}(\mathtt{f})] \mid \beta] * \delta{\to}\varnothing_F) \mathrel{-\!\!*} P)\}\\ \mathtt{prepend}(\mathtt{p},\mathtt{c})\\ \{\alpha{\to}\mathtt{p}[\mathtt{c}[\mathrm{tree}(\mathtt{f})] \mid \beta] * \delta{\to}\varnothing_F * ((\alpha{\to}\mathtt{p}[\mathtt{c}[\mathrm{tree}(\mathtt{f})] \mid \beta] * \delta{\to}\varnothing_F) \mathrel{-\!\!*} P)\}\end{array}}\,\mathrm{F}}{\begin{array}{l}\{\exists\alpha,\beta,\delta,\mathtt{f}.\, \alpha{\to}\mathtt{p}[\beta] * \delta{\to}\mathtt{c}[\mathrm{tree}(\mathtt{f})] * ((\alpha{\to}\mathtt{p}[\mathtt{c}[\mathrm{tree}(\mathtt{f})] \mid \beta] * \delta{\to}\varnothing_F) \mathrel{-\!\!*} P)\}\\ \mathtt{prepend}(\mathtt{p},\mathtt{c})\\ \{P\}\end{array}}\,\mathrm{C}\quad /\mathrm{E}$$

append

$$\dfrac{\dfrac{\begin{array}{l}\{\alpha{\to}\mathtt{p}[\beta] * \delta{\to}\mathtt{c}[\mathrm{tree}(\mathtt{f})]\}\\ \mathtt{append}(\mathtt{p},\mathtt{c})\\ \{\alpha{\to}\mathtt{p}[\beta \mid \mathtt{c}[\mathrm{tree}(\mathtt{f})]] * \delta{\to}\varnothing_F\}\end{array}}{\begin{array}{l}\{\alpha{\to}\mathtt{p}[\beta] * \delta{\to}\mathtt{c}[\mathrm{tree}(\mathtt{f})] * ((\alpha{\to}\mathtt{p}[\beta \mid \mathtt{c}[\mathrm{tree}(\mathtt{f})]] * \delta{\to}\varnothing_F) \mathrel{-\!\!*} P)\}\\ \mathtt{append}(\mathtt{p},\mathtt{c})\\ \{\alpha{\to}\mathtt{p}[\beta \mid \mathtt{c}[\mathrm{tree}(\mathtt{f})]] * \delta{\to}\varnothing_F * ((\alpha{\to}\mathtt{p}[\beta \mid \mathtt{c}[\mathrm{tree}(\mathtt{f})]] * \delta{\to}\varnothing_F) \mathrel{-\!\!*} P)\}\end{array}}\,\mathrm{F}}{\begin{array}{l}\{\exists\alpha,\beta,\delta,\mathtt{f}.\, \alpha{\to}\mathtt{p}[\beta] * \delta{\to}\mathtt{c}[\mathrm{tree}(\mathtt{f})] * ((\alpha{\to}\mathtt{p}[\beta \mid \mathtt{c}[\mathrm{tree}(\mathtt{f})]] * \delta{\to}\varnothing_F) \mathrel{-\!\!*} P)\}\\ \mathtt{append}(\mathtt{p},\mathtt{c})\\ \{P\}\end{array}}\,\mathrm{C}\quad /\mathrm{E}$$