Imperial College London
Department of Computing

Segment Logic

Mark James Wheelhouse

May 21, 2012

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London
and the Diploma of Imperial College London

Abstract

O'Hearn, Reynolds and Yang introduced local Hoare reasoning about mutable data structures using separation logic. They reason about the local parts of the memory accessed by programs, and thus construct their smallest complete specifications. Gardner *et al.* generalised their work, using context logic to reason about structured data at the same level of abstraction as the data itself. In particular, we developed a formal specification of the Document Object Model (DOM), a W3C XML update library. Whilst we kept to the spirit of local reasoning, we were not able to retain small specifications for all of the commands of DOM: for example, our specification of the appendChild command was not small.

We show how to obtain such small specifications by developing a more fine-grained context structure, allowing us to work with arbitrary segments of a data structure. We introduce segment logic, a logic for reasoning about such segmented data structures, staring at first with a simple tree structure, but then showing how to generalise our approach to arbitrary structured data.

Using our generalised segment logic we construct a reasoning framework for abstract program modules, showing how to reason about such modules at the client level. In particular we look at modules for trees, lists, heaps and the more complex data model of DOM.

An important part of any abstraction technique is an understanding of how to link the abstraction back to concrete implementations. Building on our previous abstraction and refinement work for local reasoning, we show how to soundly implement the segment models used in our abstract reasoning. In particular we show how to implement our fine-grained list and tree modules so that their abstract specifications are satisfied by the concrete implementations. We also show how our reasoning from the abstract level can be translated to reasoning at the concrete level.

Finally, we turn our attention to concurrency and show how having genuine small axioms for our commands allows for a simple treatment of abstract level concurrency constructs.

Declaration of Originality

All of the ideas contained within this thesis are original and the product of my own work, unless otherwise stated.

- Mark James Wheelhouse

Contents

List of Figures

Acknowledgements

I would like to thank Philippa Gardner for agreeing to supervise my PhD, for pushing me to my full potential, and for showing me that there can be order within chaos. Her friendship and advice have been a constant help throughout my PhD.

I would like to thank Cristiano Calcagno for agreeing to be my second supervisor and for providing a sounding board for our development of Segment Logic.

I would also like to thank Uri Zarfaty, Gareth Smith, Mohammed Raza, Thomas Dinsdale-Young, and Adam Wright, for their willingness to discuss my work and provide useful feedback. They have each given their time without hesitation and provided a valuable testing ground for my ideas.

I would like to thank my parents Ron and Jill, and my in-laws Martin and Jan for their support and understanding over the past 4 years. I especially thank my wife Vicky for putting up with my unusual working hours, being totally supportive and even offering to proof read my work.

Finally, I would like to thank EPSRC for their financial support.

To the memory of Gabrielle Sinnaduri; a wise teacher, dedicated collegue and friend. Her sage advice and helpful push in the right direction inspried me to begin on my academic career. It is through Gabrielle that I met my supervisor Philippa Gardner, and with her encouragement I decided to undertake a PhD. I will always remember her for her upbeat attitude, no matter the situation. She is greatly missed by all who knew her.

‘When you do things right, people won’t be sure you’ve done anything at all.’

God Entity - Futurama

1 Introduction

1.1 Motivation and Objectives

This PhD was motivated by work on providing a formal specification for the W3C Document Object Model (DOM) library [68]. The DOM library provides a platform free API for manipulating XML structures on the web. However, the existing specification is written in English, leaving it somewhat ambiguous and non-compositional. In my master's thesis [70], with Smith, I looked at a small subset of the DOM commands and provided a formal specification for them in terms of Context Logic. Smith has continued this work in his thesis [64], extending the specification to cover the full structure and command set of DOM Core Level 1 [69]. During this work we observed that we could not provide Small Axioms for all of the commands of DOM. In particular, the specification for the appendChild command required a substantial over-approximation of the command's intuitive footprint. Smith's PhD was only concerned with providing a specification for DOM in a sequential setting, and so the size of a command's axioms was not a particularly relevant factor. Such an over approximation could be tolerated. However, I wanted to push our specification into a concurrent setting in the future, and in this setting it is important that a command is specified in a local way. That is, the specification for a command only describes the part of the data structure that is affected by that command. This enables us to make use of as much disjoint reasoning as possible, which has been shown to be simple and powerful in other concurrent settings [52]. Thus, investigating how to specify a command like appendChild without over approximating its footprint was the starting point for my PhD.

The principle objectives of this PhD were:

- ◇ to provide small axioms for commands that affect multiple parts of a tree at once, like appendChild;
- ◇ to provide a reasoning framework for abstract program modules that manipulate structured data;

◇ to show how to implement abstract program modules so that their abstract specifications are satisfied by the implementation;

◇ and to extended context logic so that it can be used to reason about update programs in a concurrent setting.

The main aim of this PhD has been to develop a reasoning system that can break apart data structures in a more fine-grained fashion than allowed by context logic. During my work on reasoning about DOM I found that context logic was not able to express disjointness properties natural to separation logic. Many programs, even at high levels of abstraction, work on disparate parts of a data structure, so our reasoning should be able to describe these structures without needing to fill in the connecting context. Multi-holed context logic provided a good starting point for this work, but as we shall see this was not enough on its own. In this thesis we develop an abstract reasoning system based on segment logic, a logic for reasoning about fine-grained abstract data structures.

My work was not limited to just the abstract level. An important part of any abstraction technique is to be able to show how one can refine the abstraction. In particular, I wanted to be able to prove that a given implementation satisfies our abstract specification. To this end, I worked with Dinsdale-Young to develop a theory of abstraction and refinement for local reasoning [26]. In this thesis I extend this work to incorporate the fine-grained reasoning that I have already mentioned and also to address some of the criticisms of our earlier work. In particular, I chose to modify our theory to include a more general treatment of the 'crust' of an implementation.

The eventual target of this work is to provide the techniques required to reason about an XML update library, in the style of DOM, in the concurrent setting. There are a number of services on the web whose purpose is to allow concurrent editing of tree-like structures: for example, collaborative publishing systems such as LiveJournal, Blogger and Wordpress, and collaborative editing systems such as Wikipedia, TWiki and Google Docs. These systems typically use a relational database to manage concurrency at the grain of pages or documents, mapping the data structures of the database into XML which is then shown to the users. This approach seems unsatisfactory. It should be possible to manipulate the XML structure directly to achieve a finer grain of concurrency. Taking the concurrency theory that has been developed for separation logic [52] I have applied similar techniques to our fine-grained reasoning framework.

1.2 Contributions

The main contributions of this thesis are:

◇ the development of segment logic for trees, which provides assertions that can describe fine-grained properties of tree structures;

◇ the generalisation of the segment model to handle arbitrary structured data;

◇ the development of a fine-grained abstract local reasoning framework, based on generalised segments, that allows for fine-grained high-level reasoning about program modules. In particular, the fine-grained framework allows all commands to be locally specified;

◇ the application of our fine-grained abstract local reasoning framework to reasoning about a number of program modules, including trees, lists, heaps and DOM;

◇ the development of an abstraction and refinement theory showing how to verify a concrete implementation against an abstract specification in our fine-grained reasoning setting;

◇ and the extension of our fine-grained abstract local reasoning framework to enable reasoning about concurrent programs.

1.3 Publications

The following is a list of publications I have made as part of this PhD:

◇ **DOM: Towards a Formal Specification** [33]
- Philippa Gardner, Gareth Smith, Mark Wheelhouse and Uri Zarfaty
- Programming Language Techniques for XML 2008

◇ **Local Reasoning About DOM** [34]
- Philippa Gardner, Gareth Smith, Mark Wheelhouse and Uri Zarfaty
- Principles of Database Systems 2008

◇ **Small Specifications for Tree Update** [35]
- Philippa Gardner and Mark Wheelhouse
- Web Services and Formal Methods 2010

◇ **Abstraction and Refinement for Local Reasoning** [26]
  - Thomas Dinsdale-Young, Philippa Gardner and Mark Wheelhouse
  - Verified Software: Theories, Tools and Experiments 2010

◇ **Abstract Reasoning for Concurrent Indexes** [57]
  - Pedro da Rocha Pinto, Thomas Dinsdale-Young, Philippa Gardner and Mark Wheelhouse
  - Verification of Concurrent Data Structures 2011

◇ **A Simple Abstraction for Complex Concurrent Indexes** [56]
  - Pedro da Rocha Pinto, Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner and Mark Wheelhouse
  - Object-Oriented Programming, Systems, Languages & Applications 2011

I was also involved in the supervision of two student projects:

◇ James Kearney - Concurrent Segment Logic for Trees [46]

◇ Pedro da Rocha Pinto - Reasoning about Concurrent Indexes [21]

1.4 Thesis Overview

◇ Chapter 2 introduces the background theory on which the work in this thesis is based. In particular, we give a brief introduction to program verification, separation logic, context logic and reasoning about concurrency. We give a detailed explanation of why existing context logic work is unable to provide small axioms for certain commands, and why this is a problem.

◇ Chapter 3 introduces segment logic for trees, showing how to provide fine-grained reasoning about tree structures. We also show how to generalise the segment model to handle arbitrary structured data, such as lists and the more complex W3C Document Object Model.

◇ Chapter 4 introduces the fine-grained abstract local reasoning framework, based on the generalised segment model, which uses axiomatic semantics to provide fine-grained reasoning about sequential programs for manipulating structured data such as lists and trees.

◇ Chapter 5 applies our fine-grained abstract local reasoning framework to the specification of a number of abstract program modules. We look at both simple modules, trees and heaps, as well as more complex modules for list-stores and the W3C Document Object Model.

- ◇ Chapter 6 describes data abstraction and refinement in the fine-grained abstract local reasoning framework. In particular, we show how to prove that an implementation of a module satisfies an abstract specification of that module.
- ◇ Chapter 7 discusses how our work leads towards reasoning about concurrent programs for manipulating structured data. We show how the segment model allows for simple reasoning about disjoint concurrency and also show how to reason about some additional concurrency constructs, such as conditional critical regions, in the style of concurrent separation logic.
- ◇ Finally, chapter 8 concludes this thesis with a summary of achievements and a discussion of applications and future work.

2 Background Theory

We begin by introducing the background history of program verification and recent breakthroughs in reasoning with separation logic. In 1969, Hoare introduced a formal reasoning framework, known as 'Hoare reasoning', on which this thesis is based. In §2.1 we look at this method of program verification in some detail. Hoare reasoning allowed for reasoning about programs written in a simple while language, but was not able to deal with heap manipulation. In 2001, O'Hearn, Reynolds and Yang adapted Hoare's work so that it could reason about C programs that manipulate a heap. They developed separation logic and its 'local Haore reasoning' framework, where programs can be specified just in terms of the resources they access. We discuss this in §2.2. We also look at the extension of separation logic to reasoning about concurrent programs in §2.3. Next, in §2.4.1, we turn to the work of Calcagno, Gardner and Zarfaty which generalises the ideas of separation logic to abstract structured data. Their context logic can be used for 'abstract local Hoare reasoning' and has proven particularly successful for reasoning about tree structures. Finally, in §**??** and §2.4.3, we discuss the size of a program's specification in relation to that program's footprint (the resources accessed by it). We point out the issues with the size of some program specifications in the abstract local Hoare reasoning framework and motivate the work of this thesis.

2.1 Program Verification

Since the year 2000, there has been a resurgence of interest in automatic program verification. This is largely due to the success of several verification technologies on a number of carefully chosen systems and carefully chosen properties. For example Microsofts Static Driver Verifier [2] (and its precursor, SLAM [3]) are able to automatically prove that device drivers follow certain API usage rules. Other prominent tools in this line include Blast [7] and Magic [65], which target open-source code. Full automation is achieved in these tools by combining ideas from program verification with those from static program analysis.

However, there is still a large gap between verifying carefully selected properties of carefully chosen programs, to more the general properties required to verify general code. The problem lies not just with the size of the target code, but also with the tracking of specialised programming patterns, such as resource sharing and allocation. Handling resource is a central problem for program verification and we aim to develop the theory that will allow allow more and more resource manipulating programs to be automatically verified. With resource in mind we choose to focus our attention on the verification technique known as 'Hoare reasoning'.

Hoare was one of the first computer scientists to turn his attention to the field of program verification. In the late 60's he developed a reasoning system, known as 'Hoare reasoning' [40], that used logical pre- and post-conditions to specify a program's behaviour. Moreover, his reasoning system provided a way of using these specifications to derive the specifications of larger programs. 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 wanted their programs to fulfil. It is from these very ideas that several logical reasoning systems, such as separation logic and context logic were born.

2.1.1 Hoare Reasoning

Hoare developed a static reasoning system that allows for properties to be propagated though a program without having to directly run its code. Hoare's reasoning technique centred around the identification of a core set of commands and the provision of axioms which described the behaviour of those commands. These basic axioms are then combined with a set of reasoning rules that allow us to derive properties of larger composite programs.

The axioms of core commands are given as *Hoare triples* of the form $\{P\}\ \mathbb{C}\ \{Q\}$ where $\mathbb{C}$ is a program and P and Q are logical assertions that describe the pre- and post-conditions of the program respectively. A Hoare triple may have either a partial or total correctness interpretation. The partial correctness interpretation of the triple $\{P\}\ \mathbb{C}\ \{Q\}$ says that if the assertion P is true of the program state before initiation of the program $\mathbb{C}$, then the program will not fault and the assertion Q will be true of the program state if $\mathbb{C}$ terminates. The total correctness interpretation in addition guarantees that $\mathbb{C}$ will terminate. We often choose to work with the partial correctness interpretation, as program termination proofs tend to be non-trivial, especially when we have loops in our programs. When termination is considered important, it is common to use Hoare reasoning with the partial correctness interpretation and prove program termination independently via other techniques.

As an example of a Hoare triple consider the following axiom for the assignment statement $\mathtt{x} := \mathtt{E}$:

$$\{P[\mathtt{E}/\mathtt{x}]\} \quad \mathtt{x} := \mathtt{E} \quad \{P\}$$

where $\mathtt{x}$ is a program variable and $\mathtt{E}$ is an expression of a programming language without side effects, but possibly containing the variable $\mathtt{x}$. Any assertion $P(\mathtt{x})$ which is true of $\mathtt{x}$ *after* the assignment is made must also have been true of the value of expression $\mathtt{E}$ taken *before* the assignment is made.

In truth, this is really an axiom schema, it describes an infinite set of axioms which all share a common pattern (described purely in syntactic terms). As an example of a concrete axiom from this schema, consider the Hoare triple describing the behaviour of the decrement command $\mathtt{x} := \mathtt{x} - 1$ (also referred to as `x--`):

$$\{\sigma(\mathtt{x}) = v\} \quad \mathtt{x} := \mathtt{x} - 1 \quad \{\sigma(\mathtt{x}) = v - 1\}$$

where we denote the value stored at $\mathtt{x}$ in the variable store σ by $\sigma(\mathtt{x})$.

The *Hoare reasoning rules* are described in terms of Hoare triples. For example, we have the rule of consequence:

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

This rule states that if it can be shown that P' implies the precondition P of the program $\mathbb{C}$, then P' is also a valid precondition of the program $\mathbb{C}$. The rule also states that if it can be shown that the postcondition Q of program $\mathbb{C}$ implies the assertion Q', then we can deduce that the assertion Q' will hold for the program state after the program has completed. This rule lets us strengthen the precondition and weaken the postcondition of a program.

For example, it is easy to see that $\sigma(\mathtt{x}) = v \land \sigma(\mathtt{x}) > 0 \Rightarrow \sigma(\mathtt{x}) = v$ and given $v > 0$ that $\sigma(\mathtt{x}) = v - 1 \Rightarrow \sigma(\mathtt{x}) = v' \land \sigma(\mathtt{x}) \geq 0$. Thus, by applying the rule of consequence to the specification of $\mathtt{x} := \mathtt{x} - 1$ given above, we can deduce the Hoare triple:

$$\{\sigma(\mathtt{x}) = v \wedge \sigma(\mathtt{x}) > 0\} \quad \mathtt{x} := \mathtt{x} - 1 \quad \{\sigma(\mathtt{x}) = v' \wedge \sigma(\mathtt{x}) \geq 0\}$$

In addition to the logical reasoning rules there are rules for deducing the effects of running compound commands. For example, the rule of sequential composition is given as:

$$\text{if } \{P\}\ \mathbb{C}_1\ \{R\} \text{ and } \{R\}\ \mathbb{C}_2\ \{Q\} \text{ then } \{P\}\ \mathbb{C}_1\ ;\ \mathbb{C}_2\ \{Q\}$$

If, starting from P, the proven result R of the first program $\mathbb{C}_1$ is identical to the precondition under which the second program $\mathbb{C}_2$ produces the result Q, then the whole program will produce this result. We can also give rules for more complex compound commands such as iteration using a `while` loop:

$$\text{if } \{P \wedge B\}\ \mathbb{C}\ \{P\} \text{ then } \{P\}\ \mathtt{while}\ B\ \mathtt{do}\ \mathbb{C}\ \{\neg B \wedge P\}$$

Here we need to establish an invariant P that is true on entry to the loop and at the end of each loop iteration. The rule is strengthened in that we can assume that the condition B is true at the start of the loop and false on exit from the loop.

We shall give a full set of inference rules for our reasoning framework, described in detail in chapter 4, but the rules given above are enough to reason about a small example program. Consider the following small program that takes some positive variable and reduces it to zero in a loop:

$$\mathtt{while}(\mathtt{x} > 0)\ \mathtt{do}\ \mathtt{x} := \mathtt{x} - 1$$

We can provide a proof that the variable `x` will indeed be reduced to 0 by the end of the loop. We sketch the proof below:

```
{σ(x) = v ∧ v > 0}
{σ(x) = v ∧ σ(x) ≥ 0}
while(x > 0)
  {σ(x) = v ∧ σ(x) ≥ 0 ∧ σ(x) > 0}
  {σ(x) = v ∧ σ(x) > 0}
  x := x − 1
  {σ(x) = v′ ∧ σ(x) ≥ 0}
{σ(x) = v′ ∧ σ(x) ≥ 0 ∧ σ(x) ≤ 0}
{σ(x) = 0}
```

In the first step we use the rule of consequence to weaken the precondition to generate an invariant for our `while` loop. We then need to show that with this invariant the body of the loop is satisfied and we can re-establish the invariant. Inside the loop we add the loop condition to the loop invariant and use the rule of consequence to weaken this to the precondition of the assignment command. Using the assignment axiom discussed earlier in this section we can then re-establish the loop invariant. Finally, outside the loop, we add the negation of the loop condition to the loop invariant and use the rule of consequence to establish the overall postcondition.

Hoare reasoning has been studied extensively, but it is poorly equipped to deal with resources, such as heap manipulation. Thus, the Hoare reasoning system does not scale well to realistic programs and has seen little practical use. The assertions used above are all written in first-order logic and thus describe the global program state. It is common in realistic programs to have to deal with pointers or dynamically modified data structures. Such constructs introduce complex aliasing relationships between pointers which need to be expressed to correctly specify and prove program properties. Even in only moderate sized pointer manipulating programs, it often takes more effort to describe the pointer aliasing than the actual effect of the program.

2.2 Separation Logic

In 2001, the field of program verification took a new turn when O'Hearn, Reynolds and Yang introduced separation logic [43][53]. Up until this point, most formalisms had taken a global view of the whole program state when specifying programs. However, O'Hearn, Reynolds and Yang had a different viewpoint, one of *local* reasoning. They summarise this idea 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.*'

Separation logic focuses on specifying the local behaviour of a set of basic commands, such that the rest of the data structure is unaffected. One can then make use of a set of inference rules to infer the behaviour of these commands on larger data structures, and to combine the effects of multiple commands into more complex programs. This idea of local reasoning is only valid if the basic commands as well compound commands, such as `if` and `while`, behave in a local way, that is they must not require global information to successfully operate.

Separation logic was originally introduced to reason about the standard RAM model. The RAM model describes the state of a program as the combination of two components. The first of these, the *data store* σ, is a finite partial function that maps variables to their values. The second of component, the *heap* h, is a finite partial function mapping heap addresses to their values. The empty heap is modelled by the empty function and when we want to reason about multiple heap cells we take the disjoint union of their functions. Thus, the disjoint union of heaps is only defined if they have disjoint sets of heap addresses.

The assertion language of separation logic is used to express properties of the heap. We write `emp` to represent the empty heap, and $(x \mapsto 1)$ to represent the single cell heap shown in Figure 2.1.

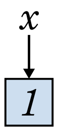

Figure 2.1: The single celled heap satisfying $(x \mapsto 1)$.

It is frequently useful to make use of two or more heap cells that are grouped together in memory. For example, we might want a cell x to carry multiple values, or pointers to other cells. To enable this we introduce a cons cell notation $(x \mapsto 1{,}2)$ that represents the two celled heap show in Figure 2.2 which describes the disjoint union of $(x \mapsto 1)$ and $(x + 1 \mapsto 2)$. This idea can be generalised to heap cells of arbitrary size.

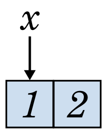

Figure 2.2: The two celled heap satisfying $(x \mapsto 1{,}2)$.

The true power of the separation logic assertion language comes from the introduction of two novel spatial connectives: the separating conjunction $*$ and its right adjoint $-\!\!*$. The separating conjunction $*$ decomposes the current heap into two separate pieces of heap, whilst its right adjoint $-\!\!*$ talks about properties of the current heap when extended with certain new, or fresh, heaps. The separating conjunction $P * Q$ is true just when the current heap can be split into two disjoint components, one of which makes P true, and the other of which makes Q true. The separating implication $P -\!\!* Q$ talks about new pieces of heap that are disjoint from the current heap. This implication is true if for every new heap that makes P true, the disjoint union of this new heap and the current heap will result in a heap that makes Q true.

An assertion P is said to be *precise* if for all program states s there is at most one substate $s' \subseteq s$ where s' satisfies the assertion P. For separation logic this property can be characterised as: for all s, there is at most one s' satisfying P, such that $\exists s_0.\, s = s' \uplus s_0$ (where $\uplus$ is the disjoint union of heaps).

Separation logic's new connectives make it easy to express disjointness and aliasing properties in a concise fashion. For example. the separating conjunction $*$ can be used in the formula $(x \mapsto 1{,}y) * (y \mapsto 2{,}\mathsf{null})$ to describe the heap shown in Figure 2.3. The use of $*$ ensures that the cells x and y are disjoint, and so the cell x does not reference itself. Notice that the classical logic assertion $(x \mapsto 1{,}y) \wedge (y \mapsto 2{,}\mathsf{null})$ only describes the presence of two cells in the heap and that these may, or may not, be the same cell (i.e. it is not known if $x = y$).

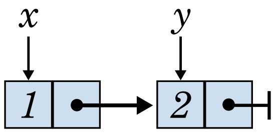

Figure 2.3: The heap satisfying $(x \mapsto 1{,}y) * (y \mapsto 2{,}\mathsf{null})$.

Being able to express these disjointness properties in a simple way makes reasoning about pointer-manipulating programs far more tractable than with traditional Hoare reasoning techniques. In particular, when working with separation logic we do not have to consider aliasing between $*$ separated resources, as they are forced to be disjoint (as in the example of Figure 2.3).

The $-\!\!*$ connective is commonly used to talk about hypothetical properties of a heap. For example, if we describe a heap with the formula $(x \mapsto -{,}7) -\!\!* P$, then this states that when a cons cell at x, with the second cell containing the value 7, is added to the current heap, then some property P will hold. Here we use $-$ to state that we can have any value in the first cell at x. We shall shortly see how such hypothetical properties can be utilised in the Hoare reasoning setting.

2.2.1 Local Hoare Reasoning

There is an intuitive notion of the footprint that a program touches. This idea was first introduced informally in [43] describing the footprint of a program as

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

For example, the program $\mathtt{x} := \mathtt{x} - 1$ given in §2.1 only accesses the variable x. The footprint of this program is, therefore, just the variable x within the store σ.

In separation logic the idea of program footprints is taken to heart by giving a *local fault-avoiding partial-correctness* interpretation of a Hoare triple. In this interpretation, the Hoare triple $\{P\}\ \mathbb{C}\ \{Q\}$ says that if the state satisfies assertion P before the program runs, then either the program $\mathbb{C}$ does not terminate, or if it does $\mathbb{C}$ does not fault and the terminating state satisfies the assertion Q. This interpretation allows for one to give *small specifications* for programs, where the precondition describes only the footprint of the command and not the rest of the program state. For example, consider the double assignment program $\mathtt{set2}(x, v) = [x] := v\ ;\ [x+1] := v$, which sets the contents of both cells of a binary cons cell x to some value v. This can be specified using a small specification as follows:

$$\{x \mapsto -,-\} \quad \mathtt{set2}(x,v) \quad \{x \mapsto v,v\}$$

This specification is local in the sense that it only mentions the binary cons cell at x which is modified by the command. To be able to use this specification in a larger heap, separation logic introduces an inference rule called the *frame rule*:

$$\textsc{Frame Rule:} \quad \frac{\{P\}\ \mathbb{C}\ \{Q\}}{\{P * R\}\ \mathbb{C}\ \{Q * R\}} \quad \mathrm{mod}(\mathbb{C}) \cap \mathrm{free}(R) = \{\}$$

The frame rule states that if some program $\mathbb{C}$ run on a heap satisfying the assertion P results in a heap satisfying the assertion Q, then it will still behave in the same way if we extend this heap and, moreover, this extra heap, which satisfies the assertion R, will not be affected by the program. The rule's side condition ensures that the program does not modify any of the extra heap that is added.

Using local Hoare triples and the frame rule allows program reasoning to be confined to the cells that a program accesses. We can automatically derive that the rest of the heap remains unchanged. Consider again the $\mathtt{set2}(x, v)$ program described above. Our specification only mentions the cons cell x which is updated by the program. To use this specification in the proof of a larger program we would need to extend it to a larger heap. The frame rule provides precisely this ability, allowing us to infer the specification of the program in a larger heap by adding on the extra disjoint heap with the $*$ operator. Figure 2.4 shows this in action.

To see the interaction between $-\!\!*$ and $*$ assume we have a heap that contains the cell $(x \mapsto 1,2)$ and disjointly satisfies the property $(x \mapsto -,7) -\!\!* P$. That is, we have a heap satisfying the assertion $(x \mapsto 1,2) * ((x \mapsto -,7) -\!\!* P)$. If we run the

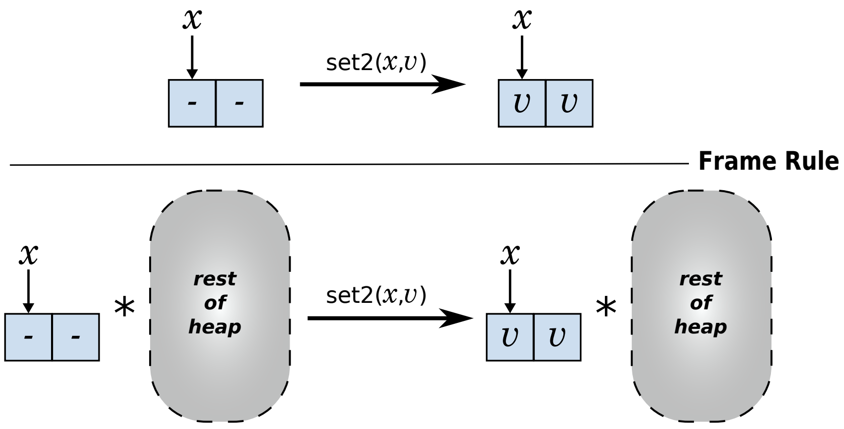

Figure 2.4: Separation logic frame rule example.

$\texttt{set2}(x, 7)$ program on this heap then the heap will now contain a cell satisfying the assertion $x \mapsto -,7$ and so, by the definition of $-\!\!*$, the whole heap will satisfy the assertion P. We give a sketch of the proof of this below:

$$\begin{array}{l}
\left\{ \; (x \mapsto 1,2) * ((x \mapsto -,7) -\!\!* P) \; \right\} \\
\quad \left\{ \; x \mapsto 1,2 \; \right\} \\
\quad \texttt{set2}(x, 7) \\
\quad \left\{ \; x \mapsto 7,7 \; \right\} \\
\left\{ \; (x \mapsto 7,7) * ((x \mapsto -,7) -\!\!* P) \; \right\} \\
\left\{ \; (x \mapsto -,7) * ((x \mapsto -,7) -\!\!* P) \; \right\} \\
\left\{ \; P \; \right\}^{\dagger}
\end{array}$$

The first step of this proof is to use the frame rule to frame off the heap not affected by the $\texttt{set2}(x, 7)$ program. Notice that the footprint of the program $\texttt{set2}(x, 7)$ is just the single cons cell at address x. We then apply the small axiom for $\texttt{set2}(x, 7)$ and bring back the framed off heap. Finally, by applying the rule of consequence, we can establish the postcondition of the program.

Calcagno, O'Hearn and Yang generalised separation logic by developing abstract separation logic based on separation algebras [17]. A separation algebra $(S, \star, u)$ is a partial commutative monoid with unit u where the $\star$ operator provides a way of disjointly splitting up structures. This algebra can then be used to provide a general theory and semantic basis for separation logics based on variants of the heap model.

Recent work by Gardner and Raza [60] has given a more mathematical definition of a command's footprint in terms of local functions and limits on these functions. They have also formally investigated what it means to provide small specifications given their footprint definition. However, the informal description give above is sufficient to understand the concepts presented in this thesis.

2.2.2 Abstract Predicates

What we have seen so far gives quite a low-level view of the program state. In practice many programmers provide clients with an abstract view of the program state and allow access to the state via some abstract interface. Parnas [55] first described the principles of information hiding and abstraction, showing that without it seemingly independent program components could become tied together. Hoare provided a logic for data abstraction [39] that used abstraction functions to hide internal implementation details from the client. These ideas were later developed further by Liskov [47] and Guttag [37] to provide what we know know as abstract datatypes. In 2002 Reynolds informally introduced *abstract predicates* to separation logic [63] to provide a mechanism for abstracting program specifications. Later, in 2005, Parkinson and Bierman gave a formal treatment of abstract predicates in separation logic [54] combing the ideas of abstract datatypes and abstraction functions into a single definition.

Abstract predicates are useful for providing abstractions that shield a client from the full details of how a data structure is implemented. Consider a list-deletion program that traverses a list, deleting each node in turn.

```
disposeList($i$)  ::=  $j$ := null ;
                       while $i \neq$ null do
                         ( $j$ := [$i$] ; dispose($i$) ; $i$ := $j$ )
```

In order to provide a specification for this program we must have a loop invariant which states that the heap contains a linked list with first node i representing some sequence α. Before the invention of separation logic this would have been described by a predicate of the form:

$$\begin{aligned}\mathsf{list}(i,\alpha) \overset{\text{def}}{=}\ & (\alpha = \emptyset \wedge i = \mathsf{null}) \\ & \vee (\alpha = a : \alpha' \wedge \exists j.\, i \mapsto a{,}j \wedge \mathsf{list}(j,\alpha'))\end{aligned}$$

This seems simple enough. The $\mathsf{list}(i,\alpha)$ predicate can be unfolded inductively, based on the input α to determine the exact structure of the list. However, this predicate only tells us that the list *exists* in the program state. It does not tell us anything else about how it may be connected with the rest of the program state. For example,

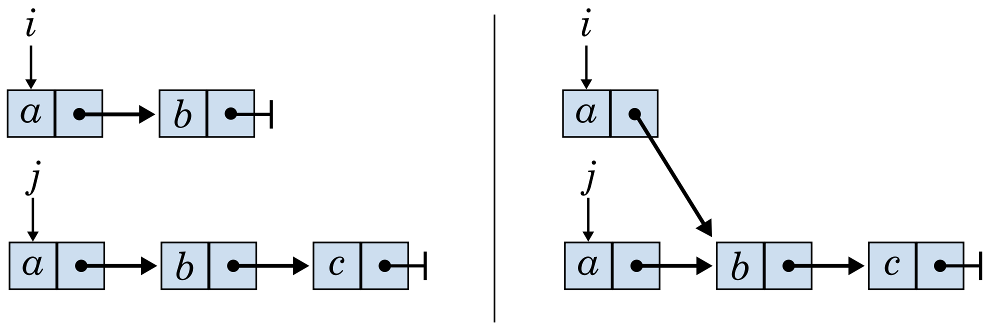

Figure 2.5: Some possible states that satsify $\exists \alpha, \beta.\, list(i, \alpha) \wedge list(j, \beta)$.

let us add to our assertion the knowledge that some other list j, representing some sequence β is also in the program state. The obvious assertion for the program state is now:

$$\exists \alpha, \beta.\, \mathsf{list}(i, \alpha) \wedge \mathsf{list}(j, \beta)$$

However, this assertion makes no mention of the possible sharing of heap cells between the two lists. Both of the cases shown in Figure 2.5 satisfy the above assertion. So deleting the list i might have some effect on the list j.

If we wanted to be sure that the two lists are fully disjoint (so we are in the first case of Figure 2.5), then the assertion must be extended to assert that the only heap address reachable from both i and j is null.

$$\begin{array}{l} \exists \alpha, \beta.\, \mathsf{list}(i, \alpha) \wedge \mathsf{list}(j, \beta) \\ \wedge\ (\forall x.\, reach(i, x) \wedge reach(j, x) \Rightarrow x = \mathsf{null}) \end{array}$$

where

$$reach(i, x) \ \stackrel{\text{def}}{=}\ (i = x) \vee (\exists a, y.\, i \mapsto a,y \wedge reach(y, x))$$

It is clearly undesirable to have to explicitly describe the reachable sets of addresses of each list as this breaks the abstraction. However, the reality is worse than this. If we wanted to add the knowledge of a third list k representing some sequence γ that is disjoint from the lists i and j, then the assertion would have to become:

$$\begin{array}{l} (\exists \alpha, \beta, \gamma.\, \mathsf{list}(i, \alpha) \wedge \mathsf{list}(j, \beta) \wedge \mathsf{list}(k, \gamma)) \\ (\wedge \forall x.\, reach(i, x) \wedge reach(j, x) \Rightarrow x = \mathsf{null}) \\ (\wedge \forall x.\, reach(i, x) \wedge reach(k, x) \Rightarrow x = \mathsf{null}) \\ (\wedge \forall x.\, reach(j, x) \wedge reach(k, x) \Rightarrow x = \mathsf{null}) \end{array}$$

Each time we wish to consider another disjoint list we have to add in reachability statements that describe that this additional list is disjoint from all of the other lists we have considered so far. Adding a fourth list would require six reachability statements, a fifth list would require ten, and so on. As such, this approach clearly will not scale well to large programs which work with many data structures.

Thankfully, separation logic provides the technology to reason about disjointness without the need to provide these kinds of reachability assertions. Using separation logic we can give an abstract predicate for a list as:

$$\begin{aligned} \mathsf{dlist}(i, \alpha) \overset{\text{def}}{=} & \ (\alpha = \emptyset \land i = \mathsf{null} \land \mathsf{emp}) \\ & \lor (\alpha = a : \alpha' \land \exists j.\, i \mapsto a, j * \mathsf{dlist}(j, \alpha')) \end{aligned}$$

As before, this predicate can be inductively unfolded based on the input sequence α. However, the use of the separating conjunction in the predicate definition ensures that each cell in the list is disjoint from the others. Moreover, expressing that we have two disjoint lists i and j is now simple:

$$\exists \alpha, \beta.\, \mathsf{dlist}(i, \alpha) * \mathsf{dlist}(j, \beta)$$

The use of $*$, in both the definition of the dlist predicate and in the assertion itself, ensures that each heap cell in each list is separate. The assertion is not satisfiable by any program state where this is not the case. This approach scales well to larger programs working with multiple data structures. For example, when we add another disjoint list k the assertion becomes:

$$\exists \alpha, \beta, \gamma.\, \mathsf{dlist}(i, \alpha) * \mathsf{dlist}(j, \beta) * \mathsf{dlist}(k, \gamma)$$

Again, the use of $*$ in the assertion tells us for free that this extra list is disjoint from both of the previous lists. Describing such disjoint data structures is simple using separation logic. Moreover, it is possible to specify programs in terms of our abstract predicates so that clients of the program do not need to understand the programs internals. Considering again our list deletion program given above, we can specify the behaviour of this program in terms of our list predicate as follows:

$$\left\{ \mathsf{dlist}(x) \right\} \ \texttt{disposeList(x)} \ \left\{ \mathsf{emp} \right\}$$

It is up to the program implementer to show that the body of the `disposeList` program does indeed satisfy this specification. For the `disposeList` program the proof is quite simple, so we will not go into details here.

As we have seen, abstract predicates inherit some of the benefits of locality from separation logic: an operation on one abstract predicate does not affect other abstract predicates. However, clients cannot take advantage of the local behaviour that is provided by the abstraction itself.

Consider, for example, a set module. At the abstract level, the operation of removing some value from the set is local; it is independent of whether any other value is in the set. However, a typical set implementation is that of a sorted singly-linked list in the heap staring at some address h. The operation of removing a value from the set will have to traverse the list from h. The footprint of this operation, therefore, consists of the entire list segment from h up to the node with the desired value. When using abstract predicates, the abstract footprint corresponds to the concrete footprint and so, in this case, includes all elements of the set less than or equal to the value to be removed. Consequently, abstract predicates cannot be used to present local abstract specifications.

The generalisation to abstract separation logic [17] allows for abstract local reasoning for other separation algebras, such as sets, but is still unable to deal with more complex structured data, such as trees and graphs. The recent development of concurrent abstract predicates [25] gets a lot closer to solving the problem. We will discuss our relation with this work at the end of chapter 7.

Using abstract predicates it is possible to hide some of the implementation details of a program from a client. Filipović, O'Hearn, Torp-Smith and Yang have also considered data refinement for local reasoning [31]. However, in both cases the client still has to work with the low-level program model provided by Separation Logic. In chapter 6 we shall see how a slightly different abstraction/refinement technique can be used to obtain similar results with a more fine-grained abstract model. In particular we will see how to make use of the locality provided by the abstraction. We will also see the proof of a procedure that is very similar to `disposeList`.

2.2.3 Practical Verification Tools

Local reasoning with separation logic has proved very successful and inspired the creation of the reasoning tools SLAyer [61] and SpaceInvader [28] based on the Smallfoot project [5]. 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 [6] and several memory leaks and memory safety bugs where found in the IEEE 1394 firewire device driver [4]. These are real program errors that had been missed by extensive testing. Finding these errors shows the practical advantages of using program verification to prove that programs are correct rather than relying on 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, or even if, the effects of the errors were eventually noticed.

The number of tools based on of separation logic has continued to grow in recent years. In particular techniques such as bi-abduction [13] have been developed to try and remove the need to provide program annotations. The more recent tools have also been tackling more complex program languages, such as jStar [29] which principally deals with Java programs, and others, such as VeriFast [44], have been designed to provide interactive proof assistants which can be used on the fly to prove as you code.

In this thesis we focus on the backing theory behind such verification tools, rather than on their development. As such, we are not going to give a detailed account of symbolic execution techniques here.

2.3 Concurrent Separation Logic

Separation logic has been extended by O'Hearn [52] and Brookes [10] to incorporate reasoning for concurrent programs. Their approach centered around two key ideas: *ownership* and *separation*. The *ownership hypothesis* states that:

> *'A code fragment can access only those portions of state that it owns.'*

With this idea the *separation property* can then be stated as:

> *'At any time, the state can be partitioned into that owned by each process and each grouping of mutual exclusion.'*

This property is key in establishing a setting that allows for independent reasoning about components of concurrent programs. Additionally, the meaning of a Hoare Triple is extended so that when $\{P\}\ \mathbb{C}\ \{Q\}$ holds, not only is the program $\mathbb{C}$ free of faults, but the program is also free on any race conditions. A race condition occurs when two or more processes attempt to access the same memory location at the same time. It is possible that these accesses may interfere with one another and lead to unexpected behaviours.

The first step in reasoning about concurrent programs was to deal with disjoint concurrency. In disjoint concurrency, programs are constructed so that they do not ever attempt to access the same memory locations. The parallel thread programming construct $\mathbb{C}_1 \mid\mid \mathbb{C}_2$ is used to denote the creation of two threads, $\mathbb{C}_1$ and $\mathbb{C}_2$, which are then executed in parallel. The reasoning rule for disjoint concurrency can then be given as:

$$\textsc{Parallel Rule}: \frac{\{P_1\}\ \mathbb{C}_1\ \{Q_1\} \quad \{P_2\}\ \mathbb{C}_2\ \{Q_2\}}{\{P_1 * P_2\}\ \mathbb{C}_1 \mid\mid \mathbb{C}_2\ \{Q_1 * Q_2\}}$$

with the side condition that $\mathbb{C}_1$ does not modify any variables free in $P_2, \mathbb{C}_2, Q_2$ and $\mathbb{C}_2$ does not modify any variables free in $P_1, \mathbb{C}_1, Q_1$. We can use this rule to prove that programs which act on separate parts of the heap are safe to run in parallel. For example, consider the following proof sketch for the program $[x] := 5 \mid\mid [y] := 6$,

$$\begin{array}{c}
\{x \mapsto - * y \mapsto -\} \\
\begin{array}{l||l}
\{x \mapsto -\} & \{y \mapsto -\} \\
[x] := 5 & [y] := 6 \\
\{x \mapsto 5\} & \{y \mapsto 6\}
\end{array} \\
\{x \mapsto 5 * y \mapsto 6\}
\end{array}$$

The overall precondition states that the cells x and y are disjoint. We make use of the Parallel Rule to split the state across the two program threads and update the cells appropriately. If the assignments were not being made to disjoint cells then the program would have a race condition. For example, in the program $[x] := 5 \mid\mid [x] := 6$ we would not know the value stored in x at the end of the program. Moreover, depending on how the heap assignments interact, the value may not necessarily even be one of 5 or 6. Such a program is not provable in this setting as the assertion $x \mapsto -$ cannot be split so that it is sent to both threads, as is required to satisfy the precondition of each assignment axiom.

Reasoning about disjoint concurrency alone would not be very interesting. The next step was to introduce a simple model of process interaction based around the declaration of shared resources and then restricting all accesses to these resources to be with mutual exclusion. The resource declaration statement `res` r `in` $\mathbb{C}$ creates a new resource r which can then be used in the rest of the program. This resource will be associated with a resource invariant in the reasoning and will initially own the portion of the program state described by this invariant. It is important, for soundness, that these resource invariants be precise assertions so that they describe an exact part of the program state.

In order to use a resource a program must make use of a *conditional critical region* (or CCR) command `with` r `when` B `do` $\mathbb{C}$. Two `with` commands for the same region r cannot be executed at the same time. Additionally, in order to enter the region some Boolean expression B must evaluate to true. If the expression B is not true, then the process must wait until the condition is satisfied. The proof rules provided for these commands are given as follows:

$$\textsc{Resource Rule}: \quad \frac{\{P\}\,\mathbb{C}\,\{Q\}}{\{P * RI\}\,\texttt{res}\ r\ \texttt{in}\ \mathbb{C}\,\{Q * RI\}}$$

$$\text{CCR }\textsc{Rule}: \quad \frac{\{(P * RI) \wedge B\}\,\mathbb{C}\,\{Q * RI\}}{\{P\}\,\texttt{with}\ r\ \texttt{when}\ B\ \texttt{do}\ \mathbb{C}\,\{Q\}}$$

where the CCR rule has the side condition that no other process modifies variables free in P or Q and in both rules the resource invariant RI is required to be precise.

With these new rules, we can now reason about programs which share some program state. For example, consider the simple producer/consumer program below.

```
x := alloc(a, b) ;               ||  with buf when full do
with buf when ¬full do           ||    y := c ;
  c := x ;                       ||    full := false
  full := true                   ||  dispose(y)
```

Here we have two treads running in parallel with a shared buffer buf. The left-hand thread produces a cell and then, when the buffer is empty, passes a reference to this cell into the buffer setting the buffer's flag to full. The right-hand thread waits for the buffer to be full, then copies the cell reference out of the buffer and sets the buffer's flag to empty. It then disposes this cell as an example of consuming the data obtained form the buffer. In practice this kind of code is likely to be encased in a looping structure of some kind, but here we consider just a single use of the buffer. The reasoning can easily scale to more complex examples that make use of the buffer more than once. For simplicity, lets assume that this code is operating in a setting where the shared buffer buf has already been initialised with resource invariant RI_{buf} given as:

$$(full \wedge c \mapsto -,-) \vee (\neg full \wedge \mathsf{emp})$$

We can then provide a proof sketch for the program as shown in Figure 2.6. Notice how the CCRs transfer the ownership of the cell x through the buffer buf from

```
                         $\{\mathsf{emp}\}$
                    $\{\mathsf{emp} * \mathsf{emp}\}$
$\{\mathsf{emp}\}$                                   ||  $\{\mathsf{emp}\}$
$x :=$ alloc$(a, b)$ ;                               ||  with $buf$ when $full$ do
$\{x \mapsto -,-\}$                                  ||    $\{(\mathsf{emp} * RI_{buf}) \wedge full\}$
with $buf$ when $\neg full$ do                       ||    $\{full \wedge c \mapsto -,-\}$
  $\{(x \mapsto -,- * RI_{buf}) \wedge \neg full\}$  ||    $y := c$ ;
  $\{(\neg full \wedge \mathsf{emp}) * x \mapsto -,-\}$ ||  $full := \mathsf{false}$ ;
  $c := x$ ;                                         ||    $\{y \mapsto -,- \wedge \neg full\}$
  $full := \mathsf{true}$                            ||    $\{(\neg full \wedge \mathsf{emp}) * y \mapsto -,-\}$
  $\{full \wedge c \mapsto -,-\}$                    ||    $\{RI_{buf} * y \mapsto -,-\}$
  $\{RI_{buf}\}$                                     ||  $\{y \mapsto -,-\}$
  $\{RI_{buf} * \mathsf{emp}\}$                      ||  dispose$(y)$
$\{\mathsf{emp}\}$                                   ||  $\{\mathsf{emp}\}$
                    $\{\mathsf{emp} * \mathsf{emp}\}$
                         $\{\mathsf{emp}\}$
```

Figure 2.6: Proof outline for the simple producer/consumer program.

the left-hand thread to the right-hand thread. It is also important to note that the `dispose` and `alloc` commands are able to run outside of the CCRs in this program, despite accessing the same cell. The synchronisation provided by the CCRs ensures that there is no possibility of a race condition occurring between these two commands.

Proving the soundness of these new concurrent separation logic inference rules is not a simple matter. Originally Brookes provided an operational semantics for the language of concurrent separation logic [10], but the resulting proof of soundness was very complex and hard to understand. Calcagno, O'Hearn and Yang used their abstract separation logic [17] and a denotational trace semantics to provided a more readable soundness proof. More recently, Vafeiadis has given a much simpler proof of soundness for concurrent separation logic [66] in terms of its original operational semantics.

With the approach taken in concurrent separation logic it is easy to verify simple concurrent programs. However, this style of reasoning with resource invariants is only really suited to programs that have statically declared critical regions. To handle more dynamic uses of concurrency it is necessary to use concepts such as rely-guarantee [45], or the more recent development of deny-guarantee [30]. Each of these approaches to reasoning about concurrency have been applied to the work of separation logic to produce novel concurrent reasoning systems [67][25].

In this thesis, we carry out an initial investigation into the realms of abstract concurrency following the simple style of concurrent separation logic. Linking our work to more recent developments in concurrency reasoning is probably one of the more interesting future steps to undertake. We will discuss this further in chapter 8.

2.4 Abstract Reasoning

As we have already seen, the view of a program's state provided by Separation Logic is a low-level one. The heap is often considered as a finite set of data cells with pointers between them and we build up more complex structures out of this heap spaghetti. However, we sometimes want to think of a program's data structure at a higher level of abstraction than this. We have seen that it is possible to abstract away from the low-level details using abstract predicates, but it is also possible to directly base our reasoning on more abstract data models.

Consider, as an example, a program library that manipulates trees. The specification of such a library should be independent of its underlying implementation. Any implementation of such a library will have to take great care in maintaining the correct pointer structure of the tree, especially when moving or deleting nodes within a tree. However, so long as the implementations of such commands satisfy the library's abstract specification, a client of the library need not be concerned with these implementation details. To them, all of this pointer update is occurring 'under the hood' of the library.

2.4.1 Context Logic

In 2004, inspired by the successes of separation logic and ambient logic [19], Calcagno, Gardner and Zarfaty created context logic [11]. 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 key idea was to reason at the level of abstraction provided to the client. Rather than reasoning about modules in terms of their internal details, we can instead reason about them in terms of their abstract interfaces. The initial work was on a simple tree model, but this has since been expanded and improved to handle more realistic data structures [14][16] and the logic's expressivity has been analyised [15]. Additionally, work on *abstraction and refinement* [26] has shown how to prove whether a given implementation satisfies a library's abstract specification.

For this overview we concentrate on a simple tree model of context logic. In a similar fashion to separation logic, context logic for trees 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 σ: a finite partial function mapping variables to their values. However, in context logic for trees, the second component of the program state models the tree structure our programs manipulate in a direct, high-level, fashion. The context logic tree model is defined in terms of trees and their associated contexts.

Trees $t \in \mathrm{T}_{\mathrm{ID}}$ are defined inductively as:

$$\text{tree } t \quad ::= \quad \varnothing \mid n[t] \mid t \otimes t$$

where $\varnothing$ is the empty tree, the node identifiers $n \in \textsc{Id}$ are unique in the tree and the $\otimes$ operator is associative, but not commutative, with identity $\varnothing$. We work with unique node identifiers, in the style of DOM, allowing us to specify commands that take node identifiers as arguments. Another option would be to work with paths, as in [51], but we choose to focus on a simple model here.

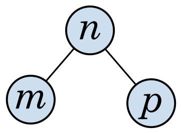

Figure 2.7: The tree $n[m[\varnothing] \otimes p[\varnothing]]$.

The small three node tree, shown in Figure 2.7, is represented as $n[m[\varnothing] \otimes p[\varnothing]]$. Notice that we do not need to directly record the left/right sibling relationship between the nodes m and p as this information is encoded in the abstract model by the $\otimes$ connective. To encode the same information in a separation logic heap model we would have to add explicit pointers between all such sibling nodes. The assertion $(n \mapsto m,p) * (m \mapsto \varnothing) * (p \mapsto \varnothing)$ might seem to describe a heap with the same structure as the tree, but consider what happens if we remove the node n from the data structure, say when working with the frame rule. In the high-level tree model we are left with $m[\varnothing] \otimes p[\varnothing]$ which still contains the information that m and p are siblings. However, the separation logic assertion $(m \mapsto \varnothing) * (p \mapsto \varnothing)$ only specifies that we have two disjoint nodes. We have lost all information about the sibling relation of nodes m and p. We could add explicit sibling pointers to the heap representation, resulting in a rather more complex low-level model of the tree. For example, we could represent the tree by the heap $(n \mapsto \mathsf{null},\mathsf{null},p,\mathsf{null}) * (m \mapsto \mathsf{null},n,\mathsf{null},p) * (p \mapsto m,n,\mathsf{null},\mathsf{null})$, as shown in Figure 2.8. Either heap representation, however, requires us to make a choice about how the tree structure is implemented, breaking the abstraction. Moving our reasoning to a higher level can help us to overcome low-level and implementation specific issues, such as pointer update, and concentrate on more interesting features of such data structures.

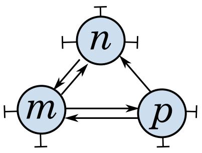

Figure 2.8: The heap $(n \mapsto \mathsf{null},\mathsf{null},p,\mathsf{null}) * (m \mapsto \mathsf{null},n,\mathsf{null},p) * (p \mapsto m,n,\mathsf{null},\mathsf{null})$.

In our example tree structure, the tree nodes do not have any contents besides their child nodes. It is trivial to extend the data structure, and the reasoning to follow, such that the tree nodes carry some extra data such as labels, colours or integers. We will see some examples of other data structures in chapter 3.

As mentioned above, context logic for trees also requires the definition of a tree context structure. Tree contexts have the same shape as trees, but can also contain a single context hole $(-)$ at some point. We can place data into this context hole and obtain a complete tree.

Tree contexts $c \in \mathrm{C}_{\mathrm{ID}}$ are defined inductively as:

$$\text{tree context } c \quad ::= \quad - \mid n[c] \mid c \otimes t \mid t \otimes c$$

where node identifiers $n \in \textsc{Id}$ are unique in the tree context and the $\otimes$ operator is associative, but not commutative, with identity $\varnothing$ as before.

This context structure comes with a notion of context composition and context application. *Context composition* $\bullet$ is the combination of two tree contexts, resulting in another tree context where the second tree context is put in place of the hole of the first tree context. This operation is associative. *Context application* $\circ$ is the combination of a tree context and a complete tree, resulting in a complete tree where the hole of the tree context has been filled. This operation associates with context composition, i.e. $c_1 \circ (c_2 \circ t) = (c_1 \bullet c_2) \circ t$. Context composition $\bullet : \mathrm{C}_{\mathrm{ID}} \times \mathrm{C}_{\mathrm{ID}} \to \mathrm{C}_{\mathrm{ID}}$ and context application $\circ : \mathrm{C}_{\mathrm{ID}} \times \mathrm{T}_{\mathrm{ID}} \to \mathrm{T}_{\mathrm{ID}}$ are each defined inductively on the structure of tree contexts as:

$$\begin{array}{rcl@{\qquad\qquad}rcl}
- \bullet c_2 & \overset{\text{def}}{=} & c_2 & - \circ t_2 & \overset{\text{def}}{=} & t_2 \\
n[c_1] \bullet c_2 & \overset{\text{def}}{=} & n[c_1 \bullet c_2] & n[c] \circ t_2 & \overset{\text{def}}{=} & n[c \circ t_2] \\
(c_1 \otimes t) \bullet c_2 & \overset{\text{def}}{=} & (c_1 \bullet c_2) \otimes t & (c \otimes t_1) \circ t_2 & \overset{\text{def}}{=} & (c \circ t_2) \otimes t_1 \\
(t \otimes c_1) \bullet c_2 & \overset{\text{def}}{=} & t \otimes (c_1 \bullet c_2) & (t_1 \otimes c) \circ t_2 & \overset{\text{def}}{=} & t_1 \otimes (c \circ t_2)
\end{array}$$

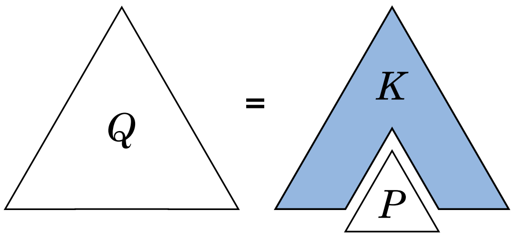

Figure 2.9: Context application $Q = K \circ P$.

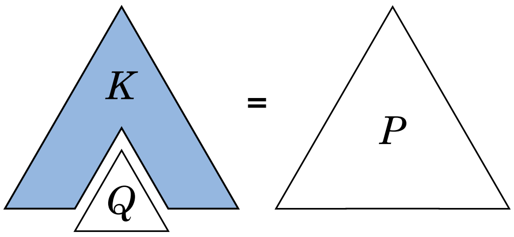

Figure 2.10: Right adjoint $Q = K \lhd P$.

The assertion language of context logic for trees is used to express properties of a tree. Our assertions for concrete trees and contexts use the same syntax as our model, for example, the assertion $\varnothing$ describes the empty tree, the assertion $n[m[\varnothing] \otimes p[\varnothing]]$ describes the small tree from 2.7, and so on. For simplicity, we sometimes drop the $\varnothing$ from our assertions, for example, writing $n[m \otimes p]$ for the assertion $n[m[\varnothing] \otimes p[\varnothing]]$.

As with separation logic, the power of context logic's assertion language comes from the use of new spatial connectives. For context logic these connectives are the application connective $\circ$ (the lifting of context application to the logical level), and its right adjoints $\lhd$ and $\rhd$. In our reasoning system we always work with complete trees, so our assertion language only includes application $\circ$ and not also composition $\bullet$ (although it is simple to extend the assertions to include the composition connective).

We use context application $\circ$ to break apart the tree by pulling out some subtree and putting a context hole in its place. The assertion $K \circ P$ is satisfied by any tree that can be split into some tree context satisfying K and a tree satisfying P (see Figure 2.9). The adjoint assertion $K \lhd P$ is satisfied by any tree that, when inserted it into a tree context satisfying K, results in a tree satisfying P (see Figure 2.10). Finally, the adjoint assertion $P \rhd Q$ is satisfied by any tree context that, when applied to a tree satisfying P, results in a tree satisfying Q (see Figure 2.11).

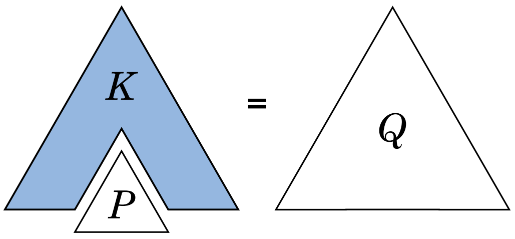

Figure 2.11: Right adjoint $K = P \triangleright Q$.

Abstract Local Hoare Reasoning

Context logic, like separation logic, uses the *local fault avoiding* interpretation of a Hoare Triple, often considering just partial correctness. However, the Hoare triples are now defined directly in terms of tree assertions. As with separation logic we give small (or local) specifications for our basic commands (we shall see that in some cases we cannot give completely small specifications).

When working at the high-level we no longer think of our data structures in terms of heap cells, but we still want our specifications to be given over just the structures that are accessed by a program. The context logic assertion language allows us to specify commands and programs directly at the high-level. For example, the tree deletion command `deleteTree`(n), which deletes the node n and its entire subtree, only access nodes within the tree structure at n. The footprint of this command is, therefore, subtree from the node n and can be specified as follows:

$$\left\{ \; n[t] \; \right\} \quad \texttt{deleteTree}(n) \quad \left\{ \; \varnothing \; \right\}$$

This local specification mentions just the subtree that is affected buy the command.

As with separation logic, to be able to use this specification as part of the proof of a larger program, context logic includes a frame rule.

$$\textsc{Abstract Frame Rule:} \quad \frac{\{P\}\ \mathbb{C}\ \{Q\}}{\{K \circ P\}\ \mathbb{C}\ \{K \circ Q\}} \quad \mathrm{mod}(\mathbb{C}) \cap \mathrm{free}(K) = \{\}$$

The abstract frame rule lets us frame on a context to both the pre- and postcondition of a program's specification using the application connective $\circ$. This added context will not be affected by the program $\mathbb{C}$, as ensured by the rule's side condition. For example, in Figure 2.12 we apply the frame rule to the `deleteTree` command given before.

The first right adjoint of application $\triangleleft$ is used for hypothetical reasoning and so does not seem to have a role in reasoning about programs. The second right adjoint

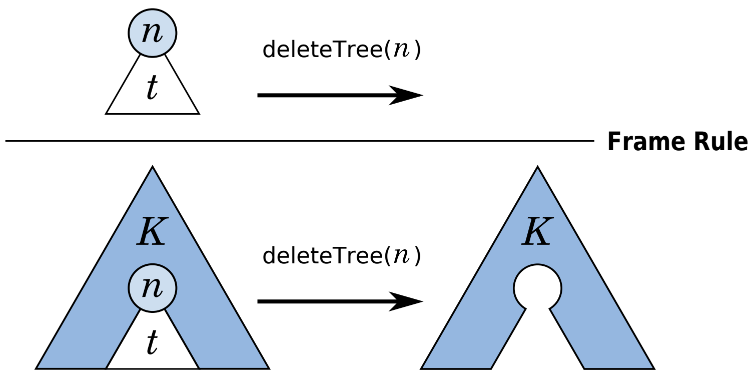

Figure 2.12: Context logic abstract frame rule example.

of application $\triangleright$ is quite similar to $-\!\!*$, the right adjoint of separation logic, and is often used to describe future properties of the data structure. The application connective $\circ$ and its right adjoint $\triangleright$ interact in much the same way as $*$ and $-\!\!*$.

For example, the formula $n[p[\varnothing]] \triangleright P$ states that if the node n with a single child p is inserted into the current context, then some property P will hold. Assume that we have the tree $n[p[\varnothing] \otimes m[\varnothing]]$, from Figure 2.7, and that this tree is in a context that satisfies property P if its hole is filled with the tree with node m removed. That is we have an overall tree that satisfies the assertion $(n[p[\varnothing]] \triangleright P) \circ (n[m[\varnothing] \otimes p[\varnothing]])$. If we run the `deleteTree`$(m)$ command on this tree then the subtree at m will be removed and the subtree at n will then satisfy property $n[p[\varnothing]]$. Thus, by the definition of $\triangleright$, the overall tree will satisfy the assertion P. This precondition is similar to the weakest, or most general, precondition of the `deleteTree`(m) command. We give a sketch of the proof of this below:

$$
\begin{array}{l}
\left\{ \; (n[p[\varnothing]] \triangleright P) \circ (n[m[\varnothing] \otimes p[\varnothing]]) \; \right\} \\
\quad \left\{ \; n[m[\varnothing] \otimes p[\varnothing]] \; \right\} \\
\quad \left\{ \; n[- \otimes p[\varnothing]] \circ m[\varnothing] \; \right\} \\
\qquad \left\{ \; m[\varnothing] \; \right\} \\
\qquad \texttt{deleteTree}(m) \\
\qquad \left\{ \; \varnothing \; \right\} \\
\quad \left\{ \; n[- \otimes p[\varnothing]] \circ \varnothing \; \right\} \\
\quad \left\{ \; n[p[\varnothing]] \; \right\} \\
\left\{ \; (n[p[\varnothing]] \triangleright P) \circ (n[p[\varnothing]]) \; \right\} \\
\left\{ \; P \; \right\}
\end{array}
$$

The first step of the proof is to use the abstract frame rule to frame off the parts of the tree that are not affected by the command. We do this in two steps, first framing of the context and then breaking apart the subtree at m from the subtree at n and framing off the new context at n. Notice that the footprint of the `deleteTree`(m) command is just the subtree at m. We then apply the small axiom for `deleteTree`(m) and bring back the framed of context at n, collapsing this back into a complete tree. Finally, by applying the rule of consequence with the definition of $\triangleright$ we establish the postcondition of the command.

Context logic provides a useful tool for reasoning about data structures at the level of abstraction provided to the client. Probably the most notable application of context logic to date has been its use in providing a formal specification of the W3C Document Object Model (DOM) [33][34] a library for manipulating XML structure on the web. In this project, a core subset of DOM commands, called Featherweight DOM, was identified and given a Hoare style context logic specification in place of its existing English specification. The extension of this work to the full DOM Core Level 1 specification [69] was the topic of Smith's thesis [64].

2.4.2 Multi-Holed Context Logic

So far we have seen a model where each context has *exactly* one context hole. Context logic has been extended to the multi-holed case, where a context can have any number of holes (including possibly none), by Calcagno, Dinsdale-Young and Gardner [12]. This allows for a uniform treatment of contexts and data, as data is just a context that has no holes. Using multi-holed context logic, Calcagno, Dinsdale-Young and Gardner where able to give a number of adjunct elimination properties that it was not possible to prove in the original context logic.

To keep track of the context holes each is labeled from a set of hole identifiers X. Multi-holed tree contexts $c \in \mathrm{T}_{\mathrm{ID},\mathrm{X}}$ are then defined inductively as:

$$\text{tree context } c \quad ::= \quad \varnothing \mid x \mid n[c] \mid c \otimes c$$

where each hole identifier $x \in \mathrm{X}$ and node identifier $n \in \textsc{Id}$ occur at most once in a tree context c, and the $\otimes$ operator is associative, but not commutative, with identity $\varnothing$. The set of hole identifiers that occur in a tree context c is denoted by $fn(c)$. We use $t, t_1, t_2...$ to denote tree contexts with no holes, i.e. complete trees.

Just as in the single-holed context model the multi-holed context structure comes with a notion of an associative context composition. Since we work only with contexts (recall that data is just a context with no holes) we do not require a notion of context application in the multi-holed setting. Context composition is defined as a set of partial functions $\bullet_x : \mathrm{T}_{\mathrm{ID},\mathrm{X}} \times \mathrm{T}_{\mathrm{ID},\mathrm{X}} \rightarrow \mathrm{T}_{\mathrm{ID},\mathrm{X}}$ indexed by hole identifiers $x \in \mathrm{X}$.

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

where $c_1[c_2/x]$ denotes the tree context c_1 with tree context c_2 in place of the context hole x in c_1.

The assertion language of multi-holed context logic follows much the same style as that of context logic. As before our assertions for concrete tree contexts use the same syntax as our model, except that we replace each occurrence of a hole label with a logical label variable $\alpha, \beta, ...$ For example, the assertion $n[\alpha]$ describes a multi-holed tree context of the form $n[x]$ where the logical environment e maps α to the hole label x. Additionally, we lift context composition $\bullet$ into the assertion language, for example, writing $n[\alpha] \bullet_\alpha t$ to describe the complete tree $n[t]$ split into a context $n[x]$ and tree t at some hole label x where $e(\alpha) = x$.

Using multi-holed contexts as the basis for defining a Hoare reasoning system enables a finer level of reasoning at the high-level. Whereas before our reasoning system dealt with tree assertions, we now have a reasoning system that treats context assertions as first class citizens. Consider, for example, a command `deleteNode`(n) that deletes just a single node n from the tree, promoting all of its children up to n's original level. We could specify this as follows:

$$\left\{\ n[\alpha]\ \right\}\ \texttt{deleteNode}(n)\ \left\{\ \alpha\ \right\}$$

Notice that we use a logical context hole variable α in both the pre- and postconditions, so this specification only mentions the node n which is being deleted. This matches the footprint of the command, which only accesses the node and not the subtree beneath it. As expected the `deleteNode`(n) command has a much smaller specification footprint than the `deleteTree`(n) command. Similarly, when specifying commands that read data from a node or look up sibling or parent information, the natural footprint does not contain the subtree beneath these nodes.

If we want to specify the behaviour of the `deleteNode`(n) command on a larger tree, we need to use an abstract frame rule. In the multi-holed case there are, in fact, two frame rules: one for wrapping a context around the current context, and one for filling context holes in the current context. These rules are, naturally, indexed by hole identifier variables α. Multi-holed context logic was introduced to investigate a number of meta-theoretical results and has never seem much use in terms of program verification. Therefore, we omit a detailed discussion of these abstract frame rules of multi-holed context logic here.

Using multi-holed context logic we can still reason about programs that affect entire subtrees. For example, we specify the `deleteTree` command just as in the single-holed case:

$$\left\{ n[t] \right\} \; \texttt{deleteTree}(n) \; \left\{ \varnothing \right\}$$

Note that it is important in this specification that t in the precondition be a complete tree, a context with no context holes, otherwise this axiom would not be sound under the frame rule. If the precondition were allowed to contain a context hole, for example $n[\alpha]$, then this hole would not be present in the postcondition $\varnothing$. So a frame composition that would be defined on the precondition, such as $n[\alpha] \bullet_\alpha t = n[t]$, would be not be defined on the postcondition, since $\varnothing \bullet_\alpha t$ is undefined for all α and t. Thus, using the frame rule, we would be able to deduce the specification:

$$\left\{ n[t] \right\} \; \texttt{deleteTree}(n) \; \left\{ \mathsf{false} \right\}$$

This could only be true if the `deleteTree` command were to diverge, which it does not.

We shall discuss the multi-holed context model and multi-holed context logic in more detail in chapter 3.

2.4.3 Introducing Segment Logic

Context logic works well for reasoning about simple tree update. When reasoning about DOM we realised that the axiom we gave for the `appendChild` command was not small. The `appendChild`(n, m) command removes the subtree at m from the tree and reinserts it as the last child of the node n. There are three possible relationships between the nodes n and m: either they are in completely disjoint parts of the tree; n is an ancestor of m; or m is an ancestor of n. If the two nodes are completely disjoint, then the `appendChild`(n, m) command simply pulls the subtree at m out of its current position and inserts it as the last child of n (see Figure 2.13). If n is an ancestor of m, then m is contained somewhere within the subtree beneath n and the effect of the `appendChild`(n, m) command is to pull the subtree at m further up the tree to the level below n (see Figure 2.14). However, if m is an ancestor of n, then n is contained somewhere within the subtree beneath m and the effect of the `appendChild`(n, m) command is to pull the subtree at m within itself, introducing a cycle into the tree structure (see Figure 2.15). When specifying the

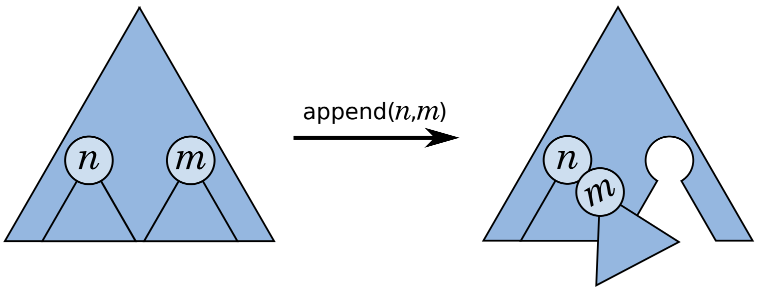

Figure 2.13: `appendChild` disjoint case.

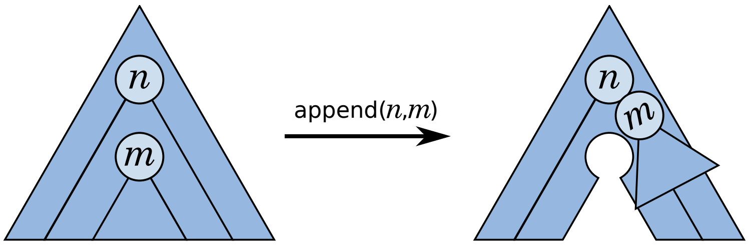

Figure 2.14: `appendChild` subtree case.

`appendChild`(n, m) command, we need to ensure that our precondition rules out the case where m is an ancestor of n, as this results in an invalid program state.

Moving away from the complexities of the DOM data structure, and concentrating on a simple tree structure, let us investigate the specification of the `appendChild`(n, m) command. Using context logic for trees, as introduced in §2.4.1, the best specification we can provide for the `appendChild` command is:

$$\left\{ \ (\varnothing \rhd (c \circ n[t_1])) \circ m[t_2] \ \right\} \ \texttt{appendChild}(n, m) \ \left\{ \ c \circ n[t_1 \otimes m[t_2]] \ \right\}$$

Concentrating on the precondition, we have a formula which describes a tree that

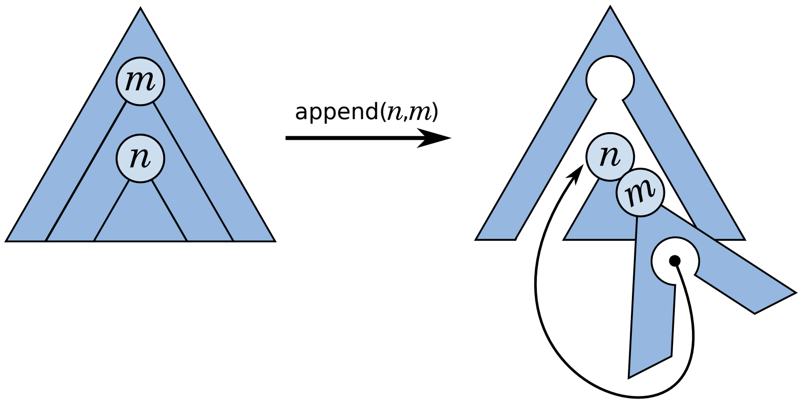

Figure 2.15: `appendChild` faulting case.

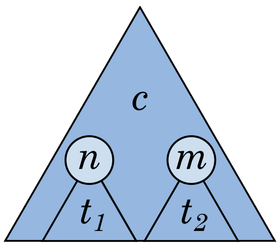

Figure 2.16: `appendChild` specification size (disjoint case).

must be able to be split up in a particular way. The formula $(\varnothing \rhd (c \circ n[t_1])) \circ m[t_2]$ states that the tree can be split into a subtree $m[t_2]$, which has top node m, and a context that satisfies $(\varnothing \rhd (c \circ n[t_1]))$. If we fill this context's hole (where the tree $m[t_2]$ was just removed from) with the empty tree $\varnothing$, then the resulting tree satisfies $c \circ n[t_1]$. What this means is if we were to replace the subtree $m[t_2]$ with $\varnothing$, then the resulting tree can be split into some arbitrary context c and a tree $n[t_1]$ with top node n. In other words, if we remove the subtree at m the remaining context still contains the node n. This means that n and m can be disjoint or n can be an ancestor of m, but m cannot be an ancestor of n. The postcondition describes the structure of the tree after the `appendChild`(n, m) command has been executed. The formula $c \circ n[t_1 \otimes m[t_2]]$ states that the tree can be split into some context c (the same context c as from the precondition) and a tree $n[t_1 \otimes m[t_2]]$ which has top node n who has had the tree $m[t_2]$ added to the end of its children (note that these are the same trees t_1 and t_2 as from the precondition).

The complex precondition is necessary to avoid the possibility of the command breaking the tree structure. However, it also makes a substantial over-approximation of the command's footprint. The precondition describes the resource that is necessary for `appendChild`(n, m) not to result in a fault. However, the specification we have given is not small. The precondition additionally describes some arbitrary linking context c (see Figure 2.16). Intuitively, we shouldn't need to reason about this extra context as it is not modified by the command.

We saw in §2.4.2 that a multi-holed context model could be used in place of the single-holed model in order to refine our specifications. However, even if we use a multi-holed tree context model, we still cannot obtain a small specification for the `appendChild`(n, m) command. The best we can manage with the multi-holed context logic is a specification of the form:

$$\Big\{\ (c_1 \bullet_\alpha n[c_2]) \bullet_\beta m[t]\ \Big\}\ \ \texttt{appendChild}(n, m)\ \ \Big\{\ (c_1 \bullet_\alpha n[c_2 \otimes m[t]) \bullet_\beta \varnothing\ \Big\}$$

This specification is certainly simpler and, due to the context c_2 beneath node n, may include less of the subtree at n. However, the specification still requires the connecting context variable c_1, so the specification has the same significant over-approximation as in the single-holed case.

Ideally, we want to be able to frame off the context c_1 and still be left with a meaningful specification. The issue here is that the formula $n[c_2] \bullet_\beta m[t]$ is only able to describe a tree context where the tree $m[t]$ is connected to the tree context $n[c_2]$. As we have already seen, the nodes n and m may be in disjoint parts of the tree, but our existing logic can only make such an assertion by describing the whole of the tree context that connects $n[c_2]$ with $m[t]$. We could add additional assertions to the formula to force the context connecting $n[c_2]$ with $m[t]$ to be minimal. For example:

$$\left\{ \; (c_1 \bullet_\alpha n[c_2]) \bullet_\beta m[t] \land \neg\exists\gamma.\, ((\neg\gamma) \bullet_\gamma (\mathsf{true} \bullet_\alpha n[c_2]) \bullet_\beta m[t]) \; \right\}$$

This formula describes the same state as before, but places an additional constraint on the form of the context c. In particular it states that this context cannot be split into a part that contains both $n[c_2]$ and $m[t]$ and some non-trivial (non hole) context. Thus, the context must be the minimal structure that connects $n[c_2]$ with $m[t]$. However, this specification still has to describe more of the tree than is being affected by the command and is significantly more complex than we would wish.

In specifying DOM, the only place that this problem arises is with the `appendChild` command. However, if we consider specifying other tree libraries then we may encounter other commands that behave in a similar way. Consider, for example, a double-deletion program `delete2Trees` that performs two tree deletion commands one after the other.

$$\texttt{delete2Trees}(n, m) ::= \texttt{deleteTree}(n) \; ; \texttt{deleteTree}(m)$$

This command should not fault unless there is some overlap between the two trees at n and m. We know how to specify the individual `deleteTree` commands in a local fashion:

$$\left\{ \; n[t_1] \; \right\} \;\; \texttt{deleteTree}(n) \;\; \left\{ \; \varnothing \; \right\}$$
$$\left\{ \; n[t_2] \; \right\} \;\; \texttt{deleteTree}(m) \;\; \left\{ \; \varnothing \; \right\}$$

However, we do not have compositional way of generating a local specification for the `delete2Trees` command from the local specifications of the individual `deleteTree` commands. Any specification would have to mention some connecting context c.

Just as with appendChild, the issue is that we cannot locally express when the two trees $n[t_1]$ and $m[t_2]$ are in disjoint parts of the tree.

The kind of disjoint behaviour we have been trying to describe so far is fairly uncommon in sequential programs, but if we look at concurrent programs we see that this pattern of manipulating multiple disjoint locations at the same time is incredibly common. A number of concurrent algorithms, such as merge-sort, parallel deletion and map-reduce all use the idea of disjointness at the very core of their design.

As an example consider an algorithm that deletes a binary tree using parallel recursive calls:

```
parTreeDelete(n)  ::=  local l, r in
                         if n $\neq$ null then
                           l := n.left ;
                           r := n.right ;
                           parTreeDelete(l) $\|$ parTreeDelete(r)
                           dispose(n)
```

This algorithm carries out some local work to set up the left and right subtrees and then makes a pair of parallel recursive calls to itself to delete these subtrees. Once both of the parallel calls have completed both subtrees will have been deleted and all that remains is to remove the top node.]

Such algorithms ensure correctness by operating on completely disjoint parts of the data structure. If threads were to attempt to access the the same structures at the same time, there would be a race to determine which gets access to the structure first. Later accesses to the same structure might not be tolerant to earlier changes and this may cause faults or undesired program behaviour. One could easily reason about such an example by breaking into the implementation of the tree structure and using concurrent separation logic, but we want to be able to reason about this at the high-level.

Our current high-level reasoning techniques are poorly equipped to handle reasoning about disjoint portions of a data structure that are not contiguously connected. The context composition and application connectives are suited to expressing containment relationships. This issue with disjointness had not appeared before our work on DOM, and in particular the appendChild command, as all of our previous commands had only acted on individual parts of a data structure. The appendChild command, however, effectively operates on two pieces of the tree at the same time. Being unable to reason about disjoint structures in the sequential setting merely leads to some inelegant specifications. However, being unable to reason about disjoint structures in the concurrent setting is totally unpractical.

In the next chapter, we introduce the segment model in order to express disjointness of trees in a local way, without having to mention connecting contexts. We will use the `appendChild` command as the driving motivation for our development of segment logic. We shall see several interesting example programs in chapter 5 which make use of the basic `appendChild` command. In chapter 7, we will consider how to extend our reasoning system to deal with high-level concurrency.

3 Segment Logic

Segment logic provides a fine-grained analysis of abstract data structures. We take the idea of disjoint reasoning, introduced by separation logic, and apply it to abstract data structures. Our disjoint reasoning for abstract data structures allows for a more fine-grained analysis of data than context logic. This allows us to naturally express properties that may hold over disparate parts of a data structure. In particular, we are able to describe properties of disjoint sub-structures. This will enable us to provide small axioms for commands which current techniques are forced to over-specify. In addition, it will open the door for reasoning about disjoint concurrency at the abstract level.

We introduce segment logic for trees, first giving tree segments in §3.1 and then giving the logic itself in §3.2. Our tree segments provide an instrumented model of trees that enriches the tree structure with additional information that aids our reasoning. From segment logic for trees, we generalise to arbitrary segment algebras and a general segment logic in §3.3. In chapters 4 and 5 we show how segment logic can be used to provide fine-grained local reasoning about structured data.

3.1 Tree Segments

Tree structures are one of the most common data structures encountered in computing. For example, trees are typically used to store ordered data for quick retrieval. However, of more interest to us is their use in recording structured data such as XML or DOM objects. Web-based and distributed programs often communicate with, or manipulate, tree structured data such as XML. If we want to be able to reason about these programs, then we are going to need to understand what it means to manipulate tree structures correctly.

We define multi-holed tree contexts and tree segments following the informal presentation of multi-holed tree contexts given in chapter 2. Here, we work with a simple tree structure. In chapter 5 we will extend these ideas to the complex tree structure of DOM.

Throughout this section, we use the countably infinite, disjoint sets $\textsc{Id} = \{m, n, ...\}$ for location names and $\mathrm{X} = \{x, y, z, ...\}$ for hole labels.

3.1.1 Trees

We model trees as finite, uniquely-labelled, unranked and ordered forests.

- ◇ They are *finite* since their branching and their depth are both required to be finite.
- ◇ They are *uniquely-labelled* since each node in a tree has an associated label which is unique to that node, similar to node identifiers in DOM.
- ◇ They are *unranked* since a node can have any number of children, regardless of its label. The number of children of a node can change as the tree structure is updated.
- ◇ They are *ordered* since the children of each node occur in a fixed sequence, from first to last, that can only be changed by updating the tree.
- ◇ They are really *forests* since any number of nodes can occur at the root level of the tree. We call them trees, in part to link with DOM which has a set of trees at the root level.

Definition 3.1 (Trees)**.** The set of *trees* $\mathrm{T}_{\textsc{Id}}$, ranged over by $t, t_1, ...$, is defined inductively as:

$$t \quad ::= \quad \varnothing \mid n[t] \mid t \otimes t$$

with the restriction that each location name $n \in \textsc{Id}$ occurs at most once in a tree, and the assumption that $\otimes$ is associative with identity $\varnothing$.

Example 3.2 (Trees)**.** The following are all examples of trees:

$$\begin{gathered}\varnothing \\ n[m[\varnothing]] \\ n[\varnothing] \otimes m[\varnothing] \\ p[n[\varnothing] \otimes m[\varnothing]] \\ p[n[\varnothing] \otimes m[\varnothing]] \otimes q[r[\varnothing] \otimes s[\varnothing] \otimes t[\varnothing]]\end{gathered}$$

whereas $n[\varnothing] \otimes n[\varnothing]$ and $n[n[\varnothing]]$ are not examples of trees as they do not have unique location names.

3.1.2 Multi-holed Tree Contexts

We have already introduced the idea of multi-holed tree contexts in chapter 2. Here we give the formal definition.

Definition 3.3 (Multi-holed Tree Contexts)**.** The set of *multi-holed tree contexts* $\mathrm{T}_{\textsc{Id},\mathrm{X}}$, ranged over by $ct, ct_1, ...$, is defined inductively as:

$$ct \quad ::= \quad \varnothing \mid x \mid n[ct] \mid ct \otimes ct$$

with the restriction that each hole label, $x \in \mathrm{X}$, and location name, $n \in \textsc{Id}$, occur at most once in a tree context ct, and the assumption that $\otimes$ is associative with identity $\varnothing$.

Example 3.4 (Tree Contexts)**.** The following are all examples of multi-holed tree contexts:

$$\begin{gathered}\varnothing\\ x\\ n[m[\varnothing]]\\ n[x]\\ n[x]\otimes y\\ p[x\otimes n\otimes y]\\ p[x\otimes m[\varnothing]]\otimes y\otimes q[z]\end{gathered}$$

whereas $p[x \otimes p[\varnothing]]$, $n[x \otimes x]$ and $q[x] \otimes x$ are not examples of multi-holed tree contexts as they do not have unique location names or unique hole labels.

Notation: Notice that a tree is just a multi-holed tree context that has no context holes. We use t, t_1, t_2 to denote complete trees. We often omit the $\varnothing$ leaves from a tree context to make it more readable, for example writing $n[m \otimes p]$ instead of $n[m[\varnothing] \otimes p[\varnothing]]$.

We provide a function that keeps track of the free hole labels in a multi-holed tree context.

Definition 3.5 (Context Hole labels)**.** The *free holes function*

$$fh_{\mathrm{T}} : \mathrm{T}_{\textsc{Id},\mathrm{X}} \to \mathcal{P}(\mathrm{X}),$$

is defined by induction on the structure of multi-holed tree contexts as:

$$\begin{array}{rcl} fh_{\mathrm{T}}(\varnothing) & \stackrel{\text{def}}{=} & \emptyset \\ fh_{\mathrm{T}}(x) & \stackrel{\text{def}}{=} & \{x\} \\ fh_{\mathrm{T}}(n[ct]) & \stackrel{\text{def}}{=} & fh_{\mathrm{T}}(ct) \\ fh_{\mathrm{T}}(ct_1 \otimes ct_2) & \stackrel{\text{def}}{=} & fh_{\mathrm{T}}(ct_1) \cup fh_{\mathrm{T}}(ct_2) \end{array}$$

We provide a similar free names function which keeps track of the location names that are assigned in a multi-holed tree context.

Definition 3.6 (Location Names)**.** The *free names function*

$$fn_{\mathrm{T}} : \mathrm{T}_{\mathrm{ID,X}} \to \mathcal{P}(\mathrm{ID})$$

is defined by induction on the structure of multi-holed tree contexts as:

$$\begin{array}{rcl} fn_{\mathrm{T}}(\varnothing) & \stackrel{\text{def}}{=} & \emptyset \\ fn_{\mathrm{T}}(x) & \stackrel{\text{def}}{=} & \emptyset \\ fn_{\mathrm{T}}(n[ct]) & \stackrel{\text{def}}{=} & \{n\} \cup fn_{\mathrm{T}}(ct) \\ fn_{\mathrm{T}}(ct_1 \otimes ct_2) & \stackrel{\text{def}}{=} & fn_{\mathrm{T}}(ct_1) \cup fn_{\mathrm{T}}(ct_2) \end{array}$$

Notation: We write $ct_1 \# ct_2$ to denote when two tree contexts are disjoint: that is, $fh_{\mathrm{T}}(ct_1) \cap fh_{\mathrm{T}}(ct_2) = \emptyset$ and $fn_{\mathrm{T}}(ct_1) \cap fn_{\mathrm{T}}(ct_2) = \emptyset$.

Multi-holed tree contexts come with a notion of context composition which allows us to compose two tree contexts. Context composition takes three arguments: a label $x \in \mathrm{X}$, and two tree contexts $ct_1, ct_2 \in \mathrm{T}_{\mathrm{ID,X}}$. The composition replaces the label x in ct_1 with the tree context ct_2. If the label x is not in the tree context ct_1 then the composition is undefined.

Definition 3.7 (Context Composition)**.** The *context composition* operator

$$\bullet : \mathrm{X} \times \mathrm{T}_{\mathrm{ID,X}} \times \mathrm{T}_{\mathrm{ID,X}} \rightharpoonup \mathrm{T}_{\mathrm{ID,X}}$$

is defined by induction on the structure of multi-holed tree contexts as:

$$\begin{aligned}
\bullet(x, \varnothing, ct) &\overset{\text{def}}{=} \text{undefined} \\
\bullet(x, y, ct) &\overset{\text{def}}{=} \begin{cases} ct & \text{if } y = x \\ \text{undefined} & \text{otherwise} \end{cases} \\
\bullet(x, n[ct'], ct) &\overset{\text{def}}{=} \begin{cases} n[\bullet(x, ct', ct)] & \text{if } \bullet(x, ct', ct) \text{ is defined} \\ \text{undefined} & \text{otherwise} \end{cases} \\
\bullet(x, (ct_1 \otimes ct_2), ct) &\overset{\text{def}}{=} \begin{cases} (\bullet(x, ct_1, ct) \otimes ct_2 & \text{if } \bullet(x, ct_1, ct) \text{ is defined} \\ ct_1 \otimes (\bullet(x, ct_2, ct) & \text{if } \bullet(x, ct_2, ct) \text{ is defined} \\ \text{undefined} & \text{otherwise} \end{cases}
\end{aligned}$$

Notation: We write $ct_1 \bullet_x ct_2$ to mean $\bullet(x, ct_1, ct_2)$.

It is possible for the composition $ct_1 \bullet_x ct_2$ to be defined even if the tree context ct_2 contains the hole x, since the composition will fill (replace) the hole x in ct_1. Thus the unique label constraint will not be violated.

Under certain conditions, and taking undefined terms to be equal, context composition is associative and commutative.

Lemma 3.8 (Semi-Associativity)**.** For all $x, y \in \mathrm{X}$ and $ct_1, ct_2, ct_3 \in \mathrm{T}_{\mathrm{ID},\mathrm{X}}$, if $y = x$ or $y \notin fh_{\mathrm{T}}(ct_1)$ then

$$ct_1 \bullet_x (ct_2 \bullet_y ct_3) \quad = \quad (ct_1 \bullet_x ct_2) \bullet_y ct_3$$

Lemma 3.9 (Semi-Commutativity)**.** For all $x, y \in \mathrm{X}$ and $ct_1, ct_2, ct_3 \in \mathrm{T}_{\mathrm{ID},\mathrm{X}}$, if $y \neq x$ and $x, y \in fh_{\mathrm{T}}(ct_1)$ and $y \notin fh_{\mathrm{T}}(ct_2)$ and $x \notin fh_{\mathrm{T}}(ct_3)$ then

$$(ct_1 \bullet_x ct_2) \bullet_y ct_3 \quad = \quad (ct_1 \bullet_y ct_3) \bullet_x ct_2$$

Both of these lemmas follow trivially from the definition of context composition.

Definition 3.10 (Substitution)**.** We write $ct[y/x]$ to denote the substitution of hole label y for hole label x in tree context ct. We define substitution in terms of context composition:

$$ct[y/x] \overset{\text{def}}{=} \begin{cases} ct \bullet_x y & \text{if } x \in fh_{\mathrm{T}}(ct) \\ ct & \text{otherwise} \end{cases}$$

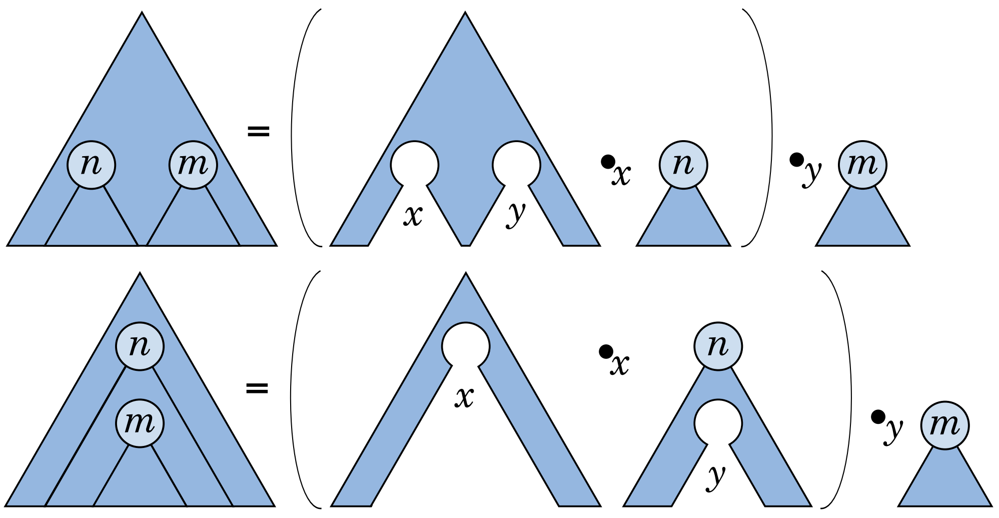

Figure 3.1: Splitting of a tree into contexts.

3.1.3 Tree Segments

In order to provide fine-grained reasoning about trees, we need a refined notion of what it means to decompose trees. Tree contexts give us a way of breaking up a tree structure into a context and a subtree. We can update this subtree and then join it together with the context to get the overall updated tree. However, the example of `appendChild` shows that this is not enough and that we need a finer way of breaking up the tree structure. We introduce tree segments which allow this fine-grained separation of tree structures.

The intuition behind the tree segment model is appealingly simple. Rather than modeling complete trees or subtrees, we instead model pieces (or segments or fragments) of a tree. In contrast to the multi-holed tree context model, we do not require that these pieces of tree be connected. That is, the pieces may describe completely disjoint parts of the tree.

When we work with multi-holed tree contexts we use composition to split the working tree into contexts and subtrees (see Figure 3.1). However, when we do this, the structures lose information about where they originated from. It is the composition function that determines which holes get filled when contexts are composed.

When we work with tree segments we split the working tree into a commutative structure (or bag) of pieces, each of which knows how it joins up with the other pieces. In Figure 3.2 we consider a splitting of a tree, using tree segments, which is similar to that obtained using contexts in Figure 3.1. The hole labels (in the holes) and the hole addresses (on the arrows) determine which segments fill which holes.

The tree segment model is also able to break up a tree structure in a more fine-

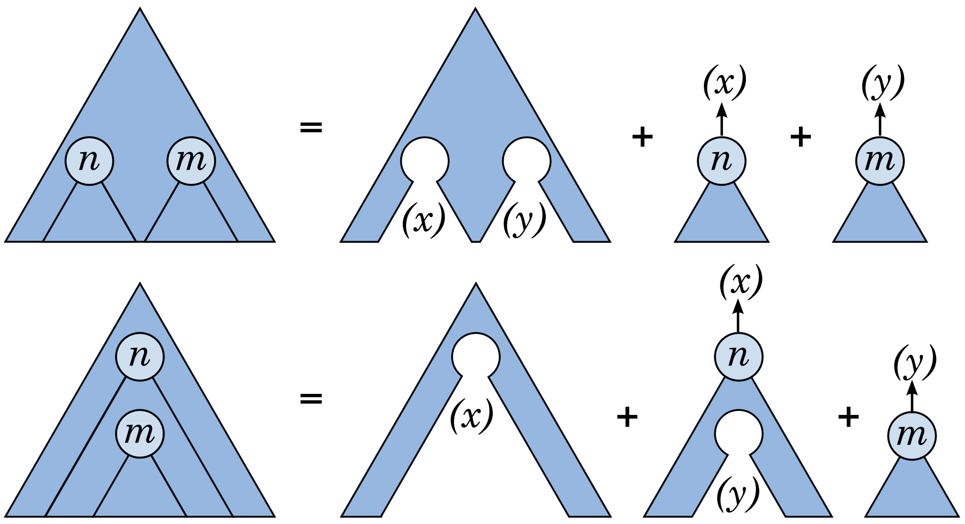

Figure 3.2: Splitting of a tree into segments.

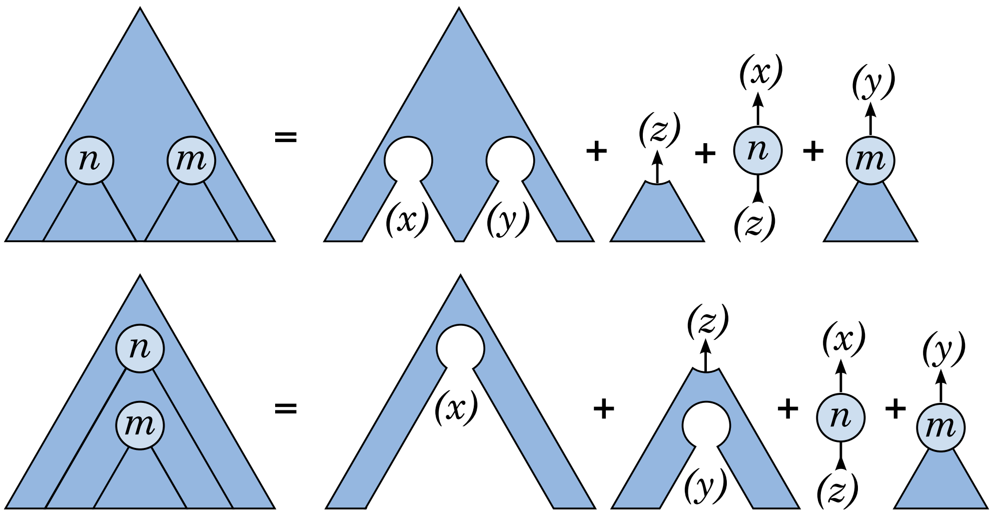

Figure 3.3: Fine-grained splitting of a tree into segments.

grained fashion. In Figure 3.3 we show how with segments we can do more than just mimic the context splitting of Figure 3.1. In particular, we can break apart the tree into disjoint chunks that can be viewed in any combination (note that lack of bracketing) even if they are not connected. In both cases of Figure 3.3 the tree segment with top node n can be split into the single node segment n at address x with hole label z, plus a tree segment at address z which contains the children of node n. As discussed in §2.4.3, the node n and the tree with top node m form the intuitive footprint of the `appendChild` command. Thus, Figure 3.3 demonstrates how we may uniformly extract the minimal data required to reason about the `appendChild` command.

It is possible to take our notion of separation to the extreme, by cutting up the tree structure into individual nodes, with hole labels and addresses showing how the nodes are joined together. (This spaghetti of wires is not far off of a heap representation of a tree.) However, such an approach does not make full use of the abstraction available here. Our instrumented view of the program state allows us to minimally cut up the tree and get at exactly the data about which we wish to reason.

We can think of tree segments as abstract heaps mapping addresses to pieces of tree. However, notice that our segment addresses are not the same as heap addresses. Heap addresses are real values that a program can access and manipulate. Segment addresses do not really exist, they are merely an instrumentation that allows us to reason about the tree. Later, in chapter 6, we shall see that our abstract addresses correspond to stability requirements on a data structure.

When working with general data, the notion of nesting of data (e.g. in trees), or ordering of data (e.g. in lists) is important for describing particular properties. For example, in our tree model the trees $n[m_1[\varnothing] \otimes m_2[\varnothing]]$ and $n[m_2[\varnothing] \otimes m_1[\varnothing]]$ are distinct. It is important to keep track of such relations when we break apart our data structures. This is achieved in the segment model by introducing hole labels and addresses whenever we split data apart. It is important that such labels be disjoint from the internal identifiers of the model, as we may wish to capture arbitrary shapes of data with a single address. For example, in the tree case we may wish to describe a segment that contains a forest structure $x \leftarrow (m_1[\varnothing] \otimes m_2[\varnothing])$. We could probably capture the same information with the tree identifiers, but such an approach would be ad-hoc and would not generalise to arbitrary data structures.

Recall the definition of multi-holed tree contexts $\mathrm{T}_{\mathrm{ID},\mathrm{X}}$ from Definition 3.3. Informally, tree segments consist of sets of labelled tree contexts (as illustrated above). Tree contexts can either be labelled with some label $x \in \mathrm{X}$, or with a special label 0 used to indicate that a tree is rooted. A rooted tree context does not have a parent and may never acquire one thorough an extension of the tree segment. We write X_0 for the set of labels X extended by the special empty label 0. Note that we do not allow 0 to be used as a hole label.

Definition 3.11 (Tree Segments)**.** The set of *tree segments* $\mathrm{S_T}$, ranged over by $st, st_1, ...$, is defined inductively as:

$$st \ ::= \ \emptyset \mid \{(x, ct)\} \mid st \uplus st$$

with tree contexts $ct \in \mathrm{T_{ID,X}}$, addresses $x \in \mathrm{X}_0$, the restriction that addresses, hole labels and location names are unique across the set st, and the restriction that for each $(x, ct) \in st$, $x \notin fh_{\mathrm{T}}(ct)$ (that is tree segments are cycle free). The disjoint union of tree segments $\uplus$ is defined only when the segments have disjoint addresses, hole labels and location names. The operation is both associative and commutative.

In this definition we require that the empty label 0 only occurs once in the tree segment. That is, our segments are representing parts of a single rooted tree. In particular, this means that $(0, ct) \uplus (0, ct')$ is undefined regardless of the choices of ct and ct'. Later, when we consider modelling DOM, we will relax the uniqueness condition for the empty label 0 to allow for there to be multiple rooted trees. We will see that the concept of an empty label is useful for describing rooted structures and that different segment models place different restrictions upon the use of the empty label.
Notation: We write $x{\leftarrow}ct$ for $\{(x, ct)\}$ when $x \neq 0$ and $\lceil ct \rceil$ as shorthand for $\{(0, ct)\}$.

Example 3.12 (Tree Segments)**.** The following are all examples of tree segments:

$$\begin{gathered}
\emptyset \\
\lceil x \rceil \\
x{\leftarrow}n[m[\varnothing]] \\
\lceil n[x] \rceil \uplus x{\leftarrow}m[\varnothing] \\
z{\leftarrow}p[n[\varnothing] \otimes x] \uplus x{\leftarrow}y \uplus y{\leftarrow}m[\varnothing] \\
\lceil p[x] \otimes q[r[\varnothing] \otimes y] \rceil \uplus x{\leftarrow}n[\varnothing] \otimes m[\varnothing] \uplus y{\leftarrow}s[\varnothing] \otimes t[\varnothing]
\end{gathered}$$

whereas $x{\leftarrow}n[\varnothing] \uplus x{\leftarrow}m[\varnothing]$, $x{\leftarrow}z \uplus y{\leftarrow}z$ and $x{\leftarrow}n[\varnothing] \uplus y{\leftarrow}n[\varnothing]$ are not examples of tree segments as they do not have unique addresses, unique hole labels, or unique location names.

We provide a free addresses function that keeps track of the addresses in a tree segment, a free holes function that keeps track of the hole labels in a tree segment and a free names function which keeps track of the location names that are currently assigned in a tree segment

Definition 3.13 (Segment Addresses, Hole Labels and Location Names)**.** The *free addresses function*

$$fa_{\mathrm{T}} : \mathrm{S}_{\mathrm{T}} \to \mathcal{P}(\mathrm{X})$$

is defined by induction on the structure of tree segments as:

$$\begin{aligned}
fa_{\mathrm{T}}(\emptyset) &\stackrel{\text{def}}{=} \emptyset \\
fa_{\mathrm{T}}(x \leftarrow ct) &\stackrel{\text{def}}{=} \begin{cases} \emptyset & \text{if } x = 0 \\ \{x\} & \text{otherwise} \end{cases} \\
fa_{\mathrm{T}}(st_1 \uplus st_2) &\stackrel{\text{def}}{=} fa_{\mathrm{T}}(st_1) \cup fa_{\mathrm{T}}(st_2)
\end{aligned}$$

The *free holes function*

$$fh_{\mathrm{T}} : \mathrm{S}_{\mathrm{T}} \to \mathcal{P}(\mathrm{X})$$

is defined by induction on the structure of tree segments as:

$$\begin{aligned}
fh_{\mathrm{T}}(\emptyset) &\stackrel{\text{def}}{=} \emptyset \\
fh_{\mathrm{T}}(x \leftarrow ct) &\stackrel{\text{def}}{=} fh_{\mathrm{T}}(ct) \\
fh_{\mathrm{T}}(st_1 \uplus st_2) &\stackrel{\text{def}}{=} fh_{\mathrm{T}}(st_1) \cup fh_{\mathrm{T}}(st_2)
\end{aligned}$$

The *free names function*

$$fn_{\mathrm{T}} : \mathrm{S}_{\mathrm{T}} \to \mathcal{P}(\textsc{Id})$$

is defined by induction on the structure of tree segments as:

$$\begin{aligned}
fn_{\mathrm{T}}(\emptyset) &\stackrel{\text{def}}{=} \emptyset \\
fn_{\mathrm{T}}(x \leftarrow ct) &\stackrel{\text{def}}{=} fn_{\mathrm{T}}(ct) \\
fn_{\mathrm{T}}(st_1 \uplus st_2) &\stackrel{\text{def}}{=} fn_{\mathrm{T}}(st_1) \cup fn_{\mathrm{T}}(st_2)
\end{aligned}$$

The overloading of functions fh_{T} and fn_{T} for contexts and segments is intentional, as they are analogous definitions.

Notation: We write $st_1 \# st_2$ to denote that two tree segments are disjoint, that is, $fa_{\mathrm{T}}(st_1) \cap fa_{\mathrm{T}}(st_2) = \emptyset$, $fh_{\mathrm{T}}(st_1) \cap fh_{\mathrm{T}}(st_2) = \emptyset$ and $fn_{\mathrm{T}}(st_1) \cap fn_{\mathrm{T}}(st_2) = \emptyset$. We write $st[y/x]$ to denote the substitution of label y for label x in tree segment st. This substitution replaces both address and hole labels and has the obvious definition.

Tree segments can be viewed as sets of labeled tree contexts. It is natural to combine tree segments when their addresses, hole labels and location names are disjoint. The union of disjoint tree segments allows us to describe disjoint tree structures without having to include any connecting context.

Definition 3.14 (Tree Segment Combination)**.** The segment combination operator

$$+ : \mathrm{S_T} \times \mathrm{S_T} \rightharpoonup \mathrm{S_T}$$

is defined as:

$$st_1 + st_2 \overset{\text{def}}{=} \begin{cases} st_1 \uplus st_2 & st_1 \# st_2 \\ \text{undefined} & \text{otherwise} \end{cases}$$

Tree segment combination is associative and commutative with identity $\emptyset$.

As well as being able to combine tree segments when their addresses, hole labels and location names are disjoint, we also require a method of compressing tree segments when one contains hole label x and the other has address x. Notice that the hole labels and addresses in Figure 3.2 and Figure 3.3 are bracketed. We shall see that this bracketing corresponds to compressing the segments at those labels.

Just as context application is defined on top of tree contexts, so segment compression is defined on top of tree segments. Segment compression takes two arguments: a label $x \in \mathrm{X}$ and a tree segment $st \in \mathrm{S_T}$. If the label x does not occur in the tree segment st, then the compression leaves st unmodified. If the label x occurs as both an address and a hole label in the tree segment st, then the compression removes the segment $x \leftarrow ct$ from st and replaces the hole label x by the tree context ct. This is illustrated in Figure 3.5. If the label x occurs only as an address in the tree segment st, then the compression replaces x by 0, creating a new rooted tree context. This is illustrated in Figure 3.4. Finally, if the label x occurs only as a hole label in the tree segment st, then the compression results in an undefined tree segment. In our model we choose to interpret this compression as preventing the hole from being filled, which means the segment would never be able to represent a complete tree. It is possible to interpret this compression in other ways, such as filling the hole with an empty tree.

In this last case the compression operation is analogous to context composition for multi-holed tree contexts. At the end of this chapter we will give a more detailed discussion of our choice to have a segment compression operator in our model. We now give the formal definition of tree segment compression.

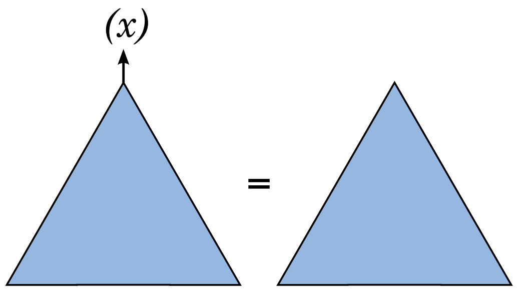

Figure 3.4: Compressing a tree segment with just address x.

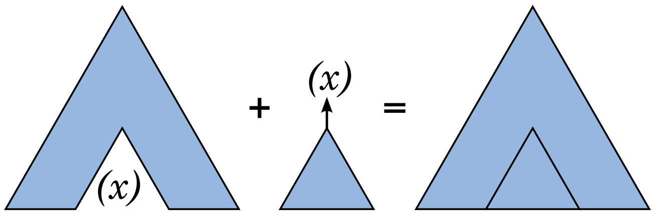

Figure 3.5: Compressing a tree segment with address x and hole label x.

Definition 3.15 (Tree Segment Compression)**.** The segment compression operator

$$\mathsf{comp} : \mathrm{X} \times \mathrm{S}_{\mathrm{T}} \rightharpoonup \mathrm{S}_{\mathrm{T}}$$

is defined as:

$$\mathsf{comp}(x, st) \overset{\text{def}}{=} \begin{cases} st & \text{if } x \notin fa_{\mathrm{T}}(st) \text{ and } x \notin fh_{\mathrm{T}}(st) \\ st' + z{\leftarrow}(ct \bullet_x ct') & \text{if } \exists st', z, ct, ct'.\, st = st' + z{\leftarrow}ct + x{\leftarrow}ct' \\ & \text{and } x \in fh_{\mathrm{T}}(ct) \\ st' + \lceil ct \rceil & \text{if } \exists st', ct.\, st = st' + x{\leftarrow}ct \\ & \text{and } x \notin fh_{\mathrm{T}}(st') \\ \text{undefined} & \text{otherwise} \end{cases}$$

The segment compression function allows us to describe when two tree segments are actually connected by some label. Conversely it can also be thought of as giving us a way of breaking apart a contiguous tree segment into two tree segments.

Notation: We write $(x)(st)$ as shorthand for $\mathsf{comp}(x, st)$. This is analogous to the restriction operator of Milner's π-calculus [48].

Lemma 3.16 (Compression Properties)**.** Tree segment compression satisfies the following properties: for all $x, y, z \in \mathrm{X}$, $st \in \mathrm{S_T}$ and $ct, ct' \in \mathrm{T_{ID,X}}$,

$$\begin{aligned}
(x)(\emptyset) &= \emptyset && (3.1)\\
(x)(y)(st) &= (y)(x)(st) && (3.2)\\
(x)(st) &= (y)(st[y/x]) \qquad \text{if } y \notin fa_{\mathrm{T}}(st) \cup fh_{\mathrm{T}}(st) && (3.3)\\
(x)(st + st') &= (x)(st) + st' \qquad \text{if } x \notin fa_{\mathrm{T}}(st') \text{ and } x \notin fh_{\mathrm{T}}(st') && (3.4)\\
y \leftarrow ct &= (x)(y \leftarrow ct_1 + x \leftarrow ct_2) \qquad \text{if } ct = ct_1 \bullet_x ct_2 \text{ and } x \neq y && (3.5)
\end{aligned}$$

The first four properties (3.1, 3.2, 3.3 and 3.4) should not be surprising and are analogous to properties for restriction from the π-calculus [48]. The final property (3.5), known as the collapse-expand property, allows us to use compression to expand a tree segment into two disjoint tree segments. The label x, introduced to be the splitting point, cannot occur in the current segment due to the properties of $\bullet$. In a right to left reading it also allows us to collapse two tree segments when they are connected by a common label.

Segment compression is a natural concept, but it also greatly simplifies our reasoning. In our segment model we have two important operators for describing data. The segment combination operator $+$ allows us to describe disjoint portions of program state (just as in separation logic). The compression function comp allows us to join together (and split apart) pieces of the program state when necessary. Both of these operators provide an instrumentation of the program state that helps our reasoning.

We could consider an abstract model that did not include a compression operator, instead enriching the segment combination operator $+$ with the behaviour of compression, i.e. $x \leftarrow y + y \leftarrow z = x \leftarrow z$. However, extending $+$ with this extra behaviour stops it from being associative. For example,

$$\begin{aligned}
(y \leftarrow \varnothing + x \leftarrow y) + y \leftarrow z &= x \leftarrow \varnothing + y \leftarrow z\\
y \leftarrow \varnothing + (x \leftarrow y + y \leftarrow z) &= y \leftarrow \varnothing + x \leftarrow z
\end{aligned}$$

The intuitive meaning of $st_1 + st_2$ is that the tree segments st_1 and st_2 are disjoint. When we talk about the disjointness of multiple objects we should not care about the order in which we consider them. Thus, it is natural for our $+$ operator to be both associative and commutative, as $\star$ is in a separation algebra. Using a non-associative $+$ operator would seem to be somewhat unnatural.

Compression helps to control the bracketing that would otherwise be required of the model, and ensures that every element of the model describes a unique structure. Moreover, compression is a local and compositional property. If we want to try and move a segment over a compression operator, rather than having to check for label name clashes in the whole segment, we only need to check that the segment we are moving does not mention the label being compressed. Notice, that in the penultimate compression property (3.4) discussed above, to move s' across the compression of x we only have to check that the label x is not contained within s'. This is a simple property to check. If we had a model without compression and with a non-associative $+$, we would need to replace this property with one that describes when it is safe to re-bracket a segment:

$$st_1 + (st_2 + st_3) \quad = \quad (st_1 + st_2) + st_3 \quad \text{if } fv(st_1) \cap fv(st_2) \cap fv(st_3) = \emptyset$$

where $fv(st) = fa_{\mathrm{T}}(st) \cup fh_{\mathrm{T}}(st)$. Notice that to switch the brackets we have to check that none of the labels in st_2 occur in both st_1 and st_3, otherwise we would be changing how these segments are compressed together when we re-bracket the segment. This property still isn't very complex, but it requires much more work to check that it holds each time we wish to change our view of the model.

Another reason to choose compression over non-associativity, is that it leads to a simple notion of alpha equivalence (property 3.3 discussed above). Compression gives us a natural bound on the occurrence of a particular label. Outside of the compression the label is hidden from the rest of the data structure, which means that its actual value is not important. If we work with a model that does not include compression then the concept of a bound name becomes more complicated. In general it is not safe to rename a free variable. However if a label occurred as both a free address *and* a free hole label, then it would be possible to rename this label. For example, we can think of $x{\leftarrow}n[y] + y{\leftarrow}\varnothing$ as $x{\leftarrow}n[z] + z{\leftarrow}\varnothing$, because of the linearity of hole labels.

Work by Back [1] uses a refinement calculus which has a similar definition to our tree segments, except that it allows for cycles and does not include the notion of a compression function. Back's work provides refinement diagrams as a way of representing the architecture of large software systems. Here we represent the abstract program state and how that state is affected by state update operations. In particular, we view segments as an instrumented view of the program state (in this case trees), not as the program state itself. The compression function is important as a tool that relates our instrumented segment model back to the real abstract data model.

The cost of using compression is a slightly more complicated model, but the reward is a more intuitive way of handling the update of structured data. We have to take care when introducing our general reasoning framework, in chapter 4, that our reasoning rules work well with compression. For the sequential case we can follow the style of Gabby and Pitts [20]. However, we shall see that for our concurrent reasoning, in chapter 7, we have to be more cautious.

3.2 Segment Logic for Trees

We have given a model for tree segments. We now introduce segment logic for trees in order to reasoning about this model. First, we present the logical environment which contains logical variables for tree contexts, tree segments and labels.

Definition 3.17 (Logical Environments)**.** A *logical environment* maps logical variables to their concrete values. Given distinct sets of tree context variables $\text{LV}\textsc{ar}_\text{T}$ ranged over by $ct, ...$, tree segment variables $\text{LV}\textsc{ar}_\text{S}$ ranged over by $st, ...$ and label variables $\text{LV}\textsc{ar}_\text{X}$ ranged over by $\alpha, \beta, ...$, the set of *logical environments* $\text{E}\textsc{nv}$, ranged over by $e, ...$, consists of functions defined by:

$$e : (\text{LV}\textsc{ar}_\text{T} \rightharpoonup_\text{fin} \text{T}_{\text{ID},\text{X}}) \times (\text{LV}\textsc{ar}_\text{S} \rightharpoonup_\text{fin} \text{S}_\text{T}) \times (\text{LV}\textsc{ar}_\text{X} \rightharpoonup_\text{fin} \text{X})$$

Notation: We write $e[v \mapsto u]$ for the logical environment e overwritten with $e(v) = u$, where v is a generic logical variable and u is a generic value. We also write $x\#e, st$ to mean that the label x is fresh with respect to the logical environment e and the tree segment st, that is, $x \notin fh_\text{T}(st)$, $x \notin fa_\text{T}(st)$ and there does not exists v such that $e(v) = x$.

Definition 3.18 (Logical Formulae)**.** The formulae of segment logic for trees are divided into two sets: the segment formulae $P, Q, ...$ and the tree context formulae $P_T, Q_T, ...$ The segment formulae P are defined inductively as:

$$\begin{array}{lll} P ::= & P \Rightarrow P \mid \mathsf{false} & \textit{Classical Assertions} \\ & \mid \alpha{\leftarrow}P_T & \textit{Tree Segment Assertions} \\ & \mid \mathsf{emp} \mid P * P \mid \alpha \circledR P \mid P {-\!\!*} P \mid P \oslash \alpha & \textit{Structural Assertions} \\ & \mid \exists v.\, P \mid \text{И}\alpha.\, P & \textit{Quantification} \end{array}$$

The tree context formulae P_T are defined inductively as:

$$\begin{array}{rcll} P_T & ::= & P_T \Rightarrow P_T \mid \mathsf{false}_{\mathrm{T}} & \textit{Classical Assertions} \\ & & \mid \exists v.\, P_T & \textit{Quantification} \\ & & \mid \varnothing \mid \alpha \mid n[P_T] \mid P_T \otimes P_T \mid @\alpha & \textit{Tree Specific Assertions} \end{array}$$

Notation: We write $free(P)$ for the set of variables that occur free in the formula P. Note that α is free in $\alpha \leftarrow P_T$, $\alpha \circledR P$ and $P \oslash \alpha$, but bound in $\text{И}\alpha.\, P$.

Just as in context logic and the BI heap logic underpinning separation logic, the segment formulae consist of classical formulae, structural formulae and specific formulae for describing the structure of data (in this case trees).

The standard separation conjunction $*$, its unit emp and its right adjoint (magic wand) $-\!*$, are structural formulae which are, by now, well known from the separation logic literature: the formula $P * Q$ describes a tree segment that can be split into two disjoint parts, one satisfying P and the other satisfying Q; the formula emp describes an empty tree segment; and the formula $P -\!* Q$ describes a tree segment that, when combined (disjointly) with a tree segment satisfying P, results in a tree segment satisfying Q.

The revelation connective $\circledR$ and its right adjoint (also called hiding) $\oslash$, are also structural formulae and, as far as we are aware, have not been used in the local reasoning setting. They have been used in the Ambient Logic [20] following the work of Pitts and Gabbay [32]. The formula $\alpha \circledR P$ describes a tree segment which has been compressed at the label stored in variable α and where the uncompressed tree segment satisfies P. The formula $P \oslash \alpha$ describes a tree segment which satisfies P if it is compressed at the label stored in variable α. We shall see in Example 3.27 that revelation, and its right adjoint, are important for giving the weakest preconditions of commands.

In addition we have the quantification formulae $\exists v.\, P$ and $\text{И}\alpha.\, P$. The formula $\exists v.\, P$ describes a tree segment that, with some value stored in variable v, satisfies P. The formula $\text{И}\alpha.\, P$ describes a tree segment that, with a fresh label stored in variable α, satisfies P. Both existential quantification and freshness quantification serve to allow us to forget about actual values of location names and labels. Existential quantification is sufficient for most properties, but to be able to describe certain properties of labels we also need the freshness quantification. In particular, when we split a tree segment into two tree segments we need to ensure that the label at which the splitting takes place is a fresh label.

$$
\begin{array}{lll}
e, st \models P \Rightarrow Q & \Leftrightarrow & e, st \models P \;\Rightarrow\; e, st \models Q \\
e, st \models \mathsf{false} & & \textit{never} \\
e, st \models \mathsf{emp} & \Leftrightarrow & st = \emptyset \\
e, st \models \alpha{\leftarrow}P_T & \Leftrightarrow & \exists ct, x.\; e(\alpha) = x \wedge\; st = x{\leftarrow}ct \;\wedge\; e, ct \models_{\mathrm{T}} P_T \\
e, st \models P * Q & \Leftrightarrow & \exists st_1, st_2.\; st = st_1 + st_2 \;\wedge\; e, st_1 \models P \;\wedge\; e, st_2 \models Q \\
e, st \models \alpha \circledR P & \Leftrightarrow & \exists x, st'.\; e(\alpha) = x \;\wedge\; st = (x)(st') \;\wedge\; e, st' \models P \\
e, st \models P \mathrel{-\!\!*} Q & \Leftrightarrow & \forall st'.\; e, st' \models P \;\wedge\; \exists st''.\, st'' = st + st' \;\Rightarrow\; e, st'' \models Q \\
e, st \models P \obslash \alpha & \Leftrightarrow & \exists x.\; e(\alpha) = x \;\wedge \forall st'.\; st' = (x)(st) \;\Rightarrow\; e, st' \models P \\
e, st \models \exists v.\, P & \Leftrightarrow & \exists u.\; e[v \mapsto u], st \models P \\
e, st \models \mathsf{\reflectbox{N}}\alpha.\, P & \Leftrightarrow & \exists x.\; x \# e, st \;\wedge\; e[\alpha \mapsto x], st \models P
\end{array}
$$

Figure 3.6: Satisfaction relation for segment formulae.

In his thesis [23] Dinsdale-Young shows that, in multi-holed context logic, it is possible to replace existential quantification with freshness quantification. However, the analogous result does not seem to hold in segment logic. The details are subtle, but we will illustrate them when we look at some example formulae in §3.2.1.

We use a segment specific formula `emp` to describe the empty tree segment $\emptyset$. We also use a specific segment formula $\alpha{\leftarrow}P_T$ to describe a tree segment $x{\leftarrow}ct$ where x is the value of the variable α and the tree context ct satisfies the tree context formula P_T. The majority of the remaining tree formulae simply describe the structure of a tree context. However, the tree formula $@\alpha$ states that the hole stored in the variable α occurs in the tree context. This formula will be needed in our specification `appendChild` which requires an assertion expressing that a tree context is complete (has no holes). We will show how to derive such an assertion shortly.

To keep our tree specific formulae simple, rather than using context logic to describe the tree contexts, we instead we chose to use tree context formulae in the style of Ambient Logic [20]. However, we could also have chosen that P_T be a context logic formula, a first-order logical formula for describing trees or even XDuce types [41]. Note that the multi-holed context logic formula $P \bullet_\alpha Q$ can be expressed by the segment logic formula $\mathsf{\reflectbox{N}}\alpha.\, \alpha\circledR(P * \alpha{\leftarrow}Q)$. Similarly the context logic formula $P \circ Q$ can be expressed by the segment logic formula $\mathsf{\reflectbox{N}}\alpha.\, \alpha\circledR(P[\alpha/-] * \alpha{\leftarrow}Q)$.

Definition 3.19 (Satisfaction Relations)**.** Given a logical environment e, the semantics of segment logic for trees is given in Figure 3.6 and Figure 3.7 by two satisfaction relations $e, st \models P$ and $e, ct \models_{\mathrm{T}} P_T$ defined on tree segments and tree contexts respectively.

$$\begin{array}{lll}
e, ct \vDash_{\mathrm{T}} P_T \Rightarrow Q_T & \Leftrightarrow & e, ct \vDash_{\mathrm{T}} P_T \ \Rightarrow \ e, ct \vDash_{\mathrm{T}} Q_T \\
e, ct \vDash_{\mathrm{T}} \mathsf{false}_{\mathrm{T}} & & \textit{never} \\
e, ct \vDash_{\mathrm{T}} \varnothing & \Leftrightarrow & ct = \varnothing \\
e, ct \vDash_{\mathrm{T}} \alpha & \Leftrightarrow & \exists x.\, e(\alpha) = x \ \wedge \ ct = x \\
e, ct \vDash_{\mathrm{T}} n[P_T] & \Leftrightarrow & \exists ct'.\ ct = n[ct'] \ \wedge \ e, ct' \vDash_{\mathrm{T}} P_T \\
e, ct \vDash_{\mathrm{T}} P_T \otimes Q_T & \Leftrightarrow & \exists ct_1, ct_2.\ ct = ct_1 \otimes ct_2 \ \wedge \ e, ct_1 \vDash_{\mathrm{T}} P_T \ \wedge \ e, ct_2 \vDash_{\mathrm{T}} Q_T \\
e, ct \vDash_{\mathrm{T}} @\alpha & \Leftrightarrow & e(\alpha) = x \ \wedge \ x \in fv(ct) \\
e, ct \vDash_{\mathrm{T}} \exists v.\, P_T & \Leftrightarrow & \exists u.\ e[v \mapsto u], ct \vDash_{\mathrm{T}} P_T
\end{array}$$

Figure 3.7: Satisfaction relation for tree formulae.

Derived Formulae

The classical logic connectives $\neg P$, true, $P \vee Q$, $P \wedge Q$ and $\forall v.\, P$, are derived from false, $\Rightarrow$ and $\exists$ as normal. We derive the hidden label quantification of Ambient logic [20] $\mathsf{H}\alpha.\, P$ form freshness $\mathsf{И}$ and revelation ®:

$$\mathsf{H}\alpha.\, P \ ::= \ \mathsf{И}\alpha.\, \alpha ® P$$

The hidden label quantification allows us to talk about restricted labels in a tree segment. We also give a number of notational shorthands for freshness, revelation and hiding:

$$\begin{array}{rcl}
\mathsf{И}\alpha, \beta.\, P & ::= & \mathsf{И}\alpha.\, (\mathsf{И}\beta.\, P) \\
\alpha, \beta ® P & ::= & \alpha ® (\beta ® P) \\
P \oslash \alpha, \beta & ::= & (P \oslash \alpha) \oslash \beta
\end{array}$$

Finally we give two further derived formulae that describe structural properties of tree contexts:

$$\begin{array}{rcll}
\mathsf{tree}(P_T) & ::= & P_T \wedge \neg\exists\alpha.\, @\alpha & \\
\circ[P_T] & ::= & \exists m.\, m[P_T] & \quad \text{if } m \notin free(P_T)
\end{array}$$

The complete tree formula $\mathsf{tree}(P_T)$ describes a tree context ct satisfying P_T where there are no context holes in ct, i.e $fv(ct) = \emptyset$. Notice that $\mathsf{tree}(P_T) \otimes \mathsf{tree}(Q_T) \Leftrightarrow \mathsf{tree}(P_T \otimes Q_T)$ follows from the definitions of $\otimes$ and tree. We use $\circ[P_T]$ to drop the identifier of a tree node when it is not necessary to know its value.

The binding convention of our connectives, from strongest to weakest, is given by:

$$\neg, \leftarrow, ®, *, \wedge, \vee, \oslash, -\!*, \Rightarrow, \Leftrightarrow, \mathsf{И}, \forall, \exists.$$

Notice that the structure of the segment formulae is orthogonal to the structure of the tree context formulae. Segment logic can easily be tailored to reason about other data structures, such as lists and heaps, by replacing the tree context formulae with some other formulae. In §3.3 we will look at formally generalising the segment model so that it may be used to reason about any structured data.

3.2.1 Segment Logic Examples

We give a number of examples that illustrate how segment logic can be used to capture properties about trees.

Example 3.20 (Simple Segments)**.** The simplest type of tree segment, other than the empty tree segment emp, is that describing a single labelled tree context. The formula $\alpha \leftarrow n[\gamma]$ describes a segment consisting of a node n with address α and context hole γ.

Example 3.21 (Disjointness)**.** Our segment formulae allow us to express properties about disjoint parts of a tree. The formula $\alpha \leftarrow n[\gamma] * \beta \leftarrow m[\delta]$ describes a tree segment consisting of a node n with address α and context hole γ, and node m with address β and context hole δ. The use of the separating conjunction means that n and m cannot be the same location, α and β cannot be the same address and γ and δ cannot be the same hole label.

Example 3.22 (Tree Contexts)**.** Our tree formulae are mostly used to describe the exact structure of some piece of the tree. The formula $n[m \otimes p]$ describes a tree with top node n that has just two children m and p. Our use of multi-holed contexts also lets us capture more fine-grained properties. The formula $n[m[\alpha] \otimes \beta]$ describes a tree with top node n with first child m. The children of m and any further children of n have been replaced by context holes. This formula tightly captures the information that m is the first child of n.

Example 3.23 (Complete Trees)**.** Our tree predicate allows us to describe properties of complete trees. The formula $n[\mathsf{tree}(ct)]$ describes a complete tree (a tree context with no holes) with top node n. Being able to describe complete trees is essential if we want to describe the safety preconditions of programs that manipulate complete trees. If we allowed such trees to contain context holes then some arbitrary amount of the tree would remain unaffected by the program.

Example 3.24 (Rooted Trees)**.** Our segment formulae allow us to express properties about the root of a tree. The formula $\mathsf{H}\alpha.\,(\alpha \leftarrow n[\beta])$ describes a tree with a single node n at the root level. Being able to describe rooted trees is essential if we want to describe the safety preconditions of programs whose behaviour may be modified at the root level. For example, a program that looks up the parent of a node will return the node's parent, or null if the node is at the root level.

Example 3.25 (Specifying Append)**.** Using properties of complete trees and disjointness we can construct the safety precondition of the `appendChild(n,m)` command discussed in §2.4.3. Assume that we have a variable store σ with $\sigma(\mathtt{n}) = n$ and $\sigma(\mathtt{m}) = m$. The segment formula $\alpha{\leftarrow}n[\gamma] * \beta{\leftarrow}m[\mathsf{tree}(ct)]$ describes a tree segment consisting of a single node n at address α and a complete tree with top node m at address β. In particular, the formula states that m is not an ancestor of n, since n is required to be disjoint from the tree $m[ct]$. This elegantly captures both the case where the trees at n and m are disjoint and the case where n is an ancestor of m.

Example 3.26 (Revelation)**.** We use revelation to compose and decompose tree segments. The formula $\alpha, \beta Ⓡ (\delta{\leftarrow}r[\alpha \otimes \beta] * \alpha{\leftarrow}n[\gamma] * \beta{\leftarrow}m[\mathsf{tree}(ct)])$ describes a tree segment consisting of a node n with address α and context hole γ and a complete tree with top node m, where in addition the holes α and β are the only siblings beneath node r at address δ. The use revelation tells us that the labels stored in α and β are compressed in this tree segment. This means that the nodes n and m are in fact siblings beneath node r. The formula logically implies the formula $\delta{\leftarrow}r[n[\gamma] \otimes m[\mathsf{tree}(ct)]]$ which describes the same tree segment. When a label is revealed we can choose to collapse the logical description of a tree segment. Working backwards through this example we can see how to split up (or expand) a tree segment into multiple segments, although in this case the labels in variables α and β would need to be fresh.

Example 3.27 (Adjoints)**.** To describe hypothetical properties of a tree, such as weakest preconditions, we need to make use of the revelation adjoint (hiding) $\oslash$ as well as the separating conjunction adjoint (magic wand) $-\!\!*$, which is standard. Consider the formula $\exists n, ct.\, \mathsf{H}\alpha.\, (\, (\alpha{\leftarrow}\varnothing -\!\!* (P \oslash \alpha)) * \alpha{\leftarrow}n[\mathsf{tree}(ct)]\,)$. This describes a tree segment which can be separated into a complete tree, with top node n at an address x denoted by the bound label α, and a tree segment st satisfying $\alpha{\leftarrow}\varnothing -\!\!* (P \oslash \alpha)$. If this tree segment is extended to a segment $st' = (x)(x{\leftarrow}\varnothing + st)$ it will satisfy P. Assuming that we have a variable store σ with $\sigma(\mathtt{n}) = n$, this formula describes the weakest precondition of a program that deletes the subtree at `n`. The effect of running such a program is to take a state satisfying $\alpha{\leftarrow}n[\mathsf{tree}(ct)]$ to a state satisfying $\alpha{\leftarrow}\varnothing$. When called on a state satisfying the weakest precondition $\exists n, ct.\, \mathsf{H}\alpha.\, (\, (\alpha{\leftarrow}\varnothing -\!\!* (P \oslash \alpha)) * \alpha{\leftarrow}n[\mathsf{tree}(ct)]\,)$ the tree deletion program will result in a state satisfying $\mathsf{H}\alpha.\,(\,(\alpha{\leftarrow}\varnothing \mathrel{-\!\!*} (P \oslash \alpha)) * \alpha{\leftarrow}\varnothing\,)$ which is logically equivalent to the formula P.

Example 3.28 (Existential Quantification)**.** Our main use of existential quantification is to allow us to forget the actual values of location names. For example, the formula $\exists m.\, \alpha{\leftarrow}n[m[\beta] \otimes \gamma]$ describes a node n that has at least one child (although we do not know its name). Such a formula is useful for describing the precondition of a program that identifies if a node has any children.

Example 3.29 (Freshness Quantification)**.** Our main use of the freshness quantification is to ensure that when we split apart a tree segment we do so using a fresh label. In particular, our logic includes the following equivalence $\mathsf{H}\alpha.\,(\beta{\leftarrow}P_T * \alpha{\leftarrow}Q_T) \Leftrightarrow \beta{\leftarrow}P_T[Q_T/\alpha]$ if $\alpha \in free(P_T)$. The left to right reading of this equivalence, the collapse, follows without the freshness part included in the hiding quantification over α. However, the right to left reading of the equivalence, the expansion, is only possible with some quantification over label α (which does not occur free on the right-hand-side). The choice of freshness quantification ensures that the label used to perform the splitting is fresh.

Example 3.30 (Existential vs. Freshness)**.** In our logic we find it useful to have both existential and freshness quantification. Consider the formula $\exists\alpha.\,(\alpha \circledR (\beta{\leftarrow}n[\gamma] * \alpha{\leftarrow}\varnothing))$. The use of existential quantification means that it is possible to choose $e(\alpha)$ to be equal to $e(\gamma)$ and thus have the two segments collapse together. This means that the tree segment $y{\leftarrow}n[\varnothing]$ satisfies the formula if $e(\beta) = y$. Replacing the existential quantification with a freshness quantification gives the formula $\text{И}\alpha.\,(\alpha \circledR (\beta{\leftarrow}n[\gamma] * \alpha{\leftarrow}\varnothing))$. The use of the freshness quantification means that it is not possible to choose $e(\alpha)$ to be equal to $e(\gamma)$, so we know that the two segments are separate. This means that the tree segment $y{\leftarrow}n[\varnothing]$ does not satisfy the formula. In most cases we use existential quantification for location names and the freshness quantification for labels.

3.3 Generalising Segment Logic

The fundamental property of being able to split up data structures into different pieces, or segments, is not unique to trees, but can be applied to many other data structures, such as lists and heaps. To generalise our approach we define a segment algebra for arbitrary data structures and a general segment logic for reasoning about such structures. We will see that tree segments and segment logic for trees form a special case of this approach.

3.3.1 Multi-holed Context Algebras

We build up our notion of a segment algebra from the existing concept of a multi-holed context algebra, first introduced in Dinsdale-Young's thesis [23]. A multi-holed context algebra generalises the idea of a multi-holed tree contexts (Definition 3.3) to arbitrary structured data.

Definition 3.31 (Multi-holed Context Algebra)**.** A *multi-holed context algebra* $\mathcal{M} = (\mathcal{C}, \mathcal{X}, fh, \bullet)$ consists of:

- ◇ a set of multi-holed contexts, $\mathcal{C}$;
- ◇ a countably infinite set of hole labels, $\mathcal{X}$ with $\mathcal{X} \subseteq \mathcal{C}$;
- ◇ a free holes function $fh : \mathcal{C} \to \mathcal{P}_{\mathsf{fin}}(\mathcal{X})$;
- ◇ a partial context composition operator $\bullet : \mathcal{X} \times \mathcal{C} \times \mathcal{C} \rightharpoonup \mathcal{C}$;

where $\mathcal{P}_{\mathsf{fin}}(\mathcal{X})$ is the finite power set of labels in $\mathcal{X}$.

Notation: Recall that we write $c_1 \bullet_x c_2$, instead of $\bullet(x, c_1, c_2)$ for the composition of contexts c_1 and c_2 at label x.

Multi-holed context algebras are required to satisfy the following properties: for all $c, c_1, c_2, c_3 \in \mathcal{C}$ and $x, y \in \mathcal{X}$,

- ◇ $fh(x) = \{x\}$;
- ◇ if context composition $c_1 \bullet_x c_2$ is defined then $x \in fh(c_1)$, $fh(c_1) \cap fh(c_2) \subseteq \{x\}$ and $fh(c_1 \bullet_x c_2) = (fh(c_1) \setminus \{x\}) \cup fh(c_2)$;
- ◇ $x \bullet_x c = c$
  (that is x behaves as the left identity of $\bullet_x$);
- ◇ $c \bullet_x x = c$ if $x \in fh(c)$
  (that is, x behaves as the right identity of $\bullet_x$ when $x \in fh(c)$);
- ◇ $(c_1 \bullet_x c_2) \bullet_y c_3 = c_1 \bullet_x (c_2 \bullet_y c_3)$ if $x = y$ or $y \notin fh(c_1)$
  (we say that composition is *semi-associative*);
- ◇ $(c_1 \bullet_x c_2) \bullet_y c_3 = (c_1 \bullet_y c_3) \bullet_x c_2$ if $x \neq y$, $x \notin fh(c_3)$ and $y \notin fh(c_2)$
  (we say that composition is *semi-commutative*).

(Undefined terms are considered equal.)

Our multi-holed context algebras do not necessarily have to contain an unit, or empty element. This allows us to expresses a greater number of models, including terms in term rewriting. The examples considered in this thesis, however, do tend to include a unit element.

3.3.2 Multi-holed Context Algebra Examples

We give a number of examples of multi-holed context algebras that represent common data structures, including trees, lists and heaps. We will later extend these context structures to segment structures following the style of tree segments.

Example 3.32 (Multi-holed Tree Context Algebra)**.** We have already seen how to define multi-holed tree contexts in §3.1.2. The multi-holed tree context algebra is defined by $\mathcal{M}_{\mathrm{T}} = (\mathrm{T}_{\mathrm{ID,X}}, \mathrm{X}, fh_{\mathrm{T}}, \bullet)$ where $\bullet : \mathrm{X} \times \mathrm{T}_{\mathrm{ID,X}} \times \mathrm{T}_{\mathrm{ID,X}} \rightharpoonup \mathrm{T}_{\mathrm{ID,X}}$ and $fh_{\mathrm{T}} : \mathrm{T}_{\mathrm{ID,X}} \to \mathcal{P}_{\mathsf{fin}}(\mathrm{X})$ are as defined in Definition 3.5 and Definition 3.7 respectively. It is not difficult to show that the conditions of a multi-holed context algebra are satisfied by these definitions.

Example 3.33 (Multi-holed List Context Algebra)**.** Lists are finite sets of elements where ordering is important. They can also be view as flat trees (trees in which all the nodes are at the root level) and so can be thought of as a special case of the tree model. The multi-holed list context algebra is defined by $\mathcal{M}_{\mathrm{L}} = (\mathrm{L}_{\mathrm{VAL,X}}, \mathrm{X}, fh_{\mathrm{L}}, \bullet)$ where,

◇ the set of multi-holed list contexts $\mathrm{L}_{\mathrm{VAL,X}}$, ranged over by $cl, cl_2, ...$, is defined inductively as:

$$cl \;::=\; \varepsilon \mid x \mid u \mid cl : cl$$

with the restriction that $u \in \mathrm{VAL}$, each hole label $x \in \mathrm{X}$ occurs at most once in a list context cl and the assumption that : is associative with identity ε.

◇ the free holes function

$$fh_{\mathrm{L}} : \mathrm{L}_{\mathrm{VAL,X}} \to \mathcal{P}_{\mathsf{fin}}(\mathrm{X})$$

is defined by induction on the structure of multi-holed list contexts as:

$$\begin{aligned} fh_{\mathrm{L}}(\varepsilon) &= \emptyset \\ fh_{\mathrm{L}}(x) &= \{x\} \\ fh_{\mathrm{L}}(u) &= \emptyset \\ fh_{\mathrm{L}}(cl_1 : cl_2) &= fh_{\mathrm{L}}(cl_1) \cup fh_{\mathrm{L}}(cl_2) \end{aligned}$$

◇ and the context composition operator

$$\bullet : \mathrm{X} \times \mathrm{L}_{\mathrm{VAL},\mathrm{X}} \times \mathrm{L}_{\mathrm{VAL},\mathrm{X}} \rightharpoonup \mathrm{L}_{\mathrm{VAL},\mathrm{X}}$$

is defined by induction on the structure of multi-holed list contexts as:

$$\begin{aligned}
\varepsilon \bullet_x cl &\overset{\text{def}}{=} \text{undefined} \\
y \bullet_x cl &\overset{\text{def}}{=} \begin{cases} cl & \text{if } y = x \\ \text{undefined} & \text{otherwise} \end{cases} \\
u \bullet_x cl &\overset{\text{def}}{=} \text{undefined} \\
(cl_1 : cl_2) \bullet_x cl &\overset{\text{def}}{=} \begin{cases} (cl_1 \bullet_x cl) : cl_2 & \text{if } x \in fh_{\mathrm{L}}(cl_1) \\ cl_1 : (cl_2 \bullet_x cl) & \text{if } x \in fh_{\mathrm{L}}(cl_2) \\ \text{undefined} & \text{otherwise} \end{cases}
\end{aligned}$$

Again, it is not difficult to show that the conditions of a multi-holed context algebra are satisfied by these definitions.

The model we have given above is for arbitrary lists, but we can also place additional constraints upon the lists, such as uniqueness of elements or ordering in increasing size of elements. These additional constraints can be useful for representing lists with certain assumed properties. For example, in chapter 5 we will be reasoning about lists of unique addresses.

Example 3.34 (Multi-holed Heap Context Algebra)**.** The heap model of separation logic views heaps as finite partial functions from addresses to values. Disjoint heap union is then the union of heaps with disjoint domains. Here, we define heaps syntactically. The set of heap addresses ADR, ranged over by $a, a_1, a', ...$, is typically taken to be the positive integers (i.e. $\mathrm{ADR} = \mathbb{Z}^+$). The set of values VAL, rnged over by $u, u', ...$, can be arbitrary, but is taken to include the set of heap address (i.e. $\mathrm{ADR} \subseteq \mathrm{VAL}$). We add holes $x, y, ... \in \mathrm{X}$ to the heap structure to be used as place-holders for missing parts of the heap. The multi-holed heap context algebra is defined by $\mathcal{M}_{\mathrm{H}} = (\mathrm{H}_{\mathrm{ADR},\mathrm{X}}, \mathrm{X}, fh_{\mathrm{H}}, \bullet)$ where,

◇ the set of multi-holed heap contexts $\mathrm{H}_{\mathrm{ADR},\mathrm{X}}$, ranged over by $ch, ch_1, ...$, is defined inductively as:

$$ch \ ::= \ \mathsf{emp} \mid x \mid a \mapsto u \mid ch \star ch$$

with the restriction that each hole label $x \in \mathrm{X}$ and address $a \in \mathrm{ADR}$ occur at most once in a heap context ch, $u \in \mathrm{VAL}$ ranges over values, and the assumption that $\star$ is associative and commutative with identity emp.

◇ the free holes function

$$fh_{\mathrm{H}} : \mathrm{H}_{\mathrm{ADR,X}} \to \mathcal{P}_{\mathsf{fin}}(\mathrm{X})$$

is defined by induction over the structure of multi-holed heap contexts as:

$$\begin{aligned}
fh_{\mathrm{H}}(\mathsf{emp}) &= \emptyset \\
fh_{\mathrm{H}}(x) &= \{x\} \\
fh_{\mathrm{H}}(a \mapsto u) &= \emptyset \\
fh_{\mathrm{H}}(ch_1 \star ch_2) &= fh_{\mathrm{H}}(ch_1) \cup fh_{\mathrm{H}}(ch_2)
\end{aligned}$$

◇ and the context composition operator

$$\bullet : \mathrm{X} \times \mathrm{H}_{\mathrm{ADR,X}} \times \mathrm{H}_{\mathrm{ADR,X}} \rightharpoonup \mathrm{H}_{\mathrm{ADR,X}}$$

is defined by induction on the structure of multi-holed heap contexts as:

$$\begin{aligned}
\mathsf{emp} \bullet_x ch &\stackrel{\text{def}}{=} \text{undefined} \\
y \bullet_x ch &\stackrel{\text{def}}{=} \begin{cases} ch & \text{if } y = x \\ \text{undefined} & \text{otherwise} \end{cases} \\
(a \mapsto u) \bullet_x ch &\stackrel{\text{def}}{=} \text{undefined} \\
(ch_1 \star ch_2) \bullet_x ch &\stackrel{\text{def}}{=} \begin{cases} (ch_1 \bullet_x ch) \star ch_2 & \text{if } x \in fh_{\mathrm{H}}(ch_1) \\ ch_1 \star (ch_2 \bullet_x ch) & \text{if } x \in fh_{\mathrm{H}}(ch_2) \\ \text{undefined} & \text{otherwise} \end{cases}
\end{aligned}$$

Due to the associativity and commutativity of $\star$ we can contract all holes to the end of the heap. It is this uniformity that allows separation logic to work without explicitly tracking the holes. In effect, every heap can be thought to have a hole in it. We choose to track the holes in our model of heaps in order to have a uniform treatment of data. In chapter 4 we will see that this allows us to provide a single framework for reasoning about imperative programs, regardless of the data structures they manipulate.

Example 3.35 (Separation Algebras as Multi-holed Context Algebras). In [17], Calcagno, O'Hearn and Yang consider abstract models for separation logic, of which the heap model is an instance. Separation algebras are defined to be partial commutative monoids $(\mathcal{S}, \star, u)$. Any such separation algebra gives rise to a multi-holed context algebra $\mathcal{M}_{\mathrm{S}} = (\mathrm{S}_{\mathrm{X}}, \mathrm{X}, fh_{S}, \bullet)$ where,

◇ the set of multi-holed contexts $\mathrm{S_X}$ is defined as:

$$\mathrm{S_X} \stackrel{\text{def}}{=} \{(h, \bar{x}) \mid h \in \mathcal{S}, \bar{x} \in \mathcal{P}_{\mathsf{fin}}(\mathrm{X})\}$$

◇ the free holes function

$$fh_S : \mathrm{S_X} \to \mathcal{P}_{\mathsf{fin}}(\mathrm{X})$$

is defined as:

$$fh_S((h, \bar{x})) \stackrel{\text{def}}{=} \bar{x}$$

◇ and the context composition operator

$$\bullet : \mathrm{X} \times \mathrm{S_X} \times \mathrm{S_X} \rightharpoonup \mathrm{S_X}$$

is defined as:

$$(h_1, \bar{x}) \bullet_x (h_2, \bar{y}) \stackrel{\text{def}}{=} \begin{cases} (h_1 \star h_2, (\bar{x} \backslash \{x\}) \cup \bar{y}) & \text{if } x \in \bar{x} \text{ and } \bar{x} \cap \bar{y} \subseteq \{x\} \\ \text{undefined} & \text{otherwise} \end{cases}$$

The context elements $(h, \bar{x})$ can be thought of as adding the hole labels $\bar{x}$ onto the end of h with $\star$. As with the multi-holed heap context model, this allows us to treat arbitrary segment algebras in a uniform fashion.

Example 3.36 (Multi-holed List Pair Context Algebra)**.** As an example of a somewhat more unusual context algebra we consider representing a pair of lists. In chapter 5 we will extend this idea to provide a model of a list store that can contain an arbitrary number of lists. This will allow us to define a list module, containing a number of lists, which we use to implement a tree model. The multi-holed list pair algebra is defined by $\mathcal{M}_{\mathrm{LP}} = (\mathrm{LP_{VAL,X}}, \mathrm{X} \times \mathrm{X}, fh_{\mathrm{LP}}, \bullet)$ where,

◇ the set of multi-holed list pair contexts $\mathrm{LP_{VAL,X}}$, ranged over by $clp, clp_1, ...$, is defined as:

$$clp \ ::= \ (cl, cl)$$

with $cl \in \mathrm{L_{VAL,X}}$ as defined in Example 3.33.

◇ the free holes function

$$fh_{\mathrm{LP}} : \mathrm{LP_{VAL,X}} \to \mathcal{P}_{\mathsf{fin}}(\mathrm{X}) \times \mathcal{P}_{\mathsf{fin}}(\mathrm{X})$$

is defined as:

$$fh_{\mathrm{LP}}((cl_1, cl_2)) \stackrel{\text{def}}{=} fh_{\mathrm{L}}(cl_1) \times fh_{\mathrm{L}}(cl_2)$$

where fv_{L} is the free holes function for multi-holed list contexts as defined in Example 3.33.

- ◇ and the context composition operator

$$\bullet : (\mathrm{X} \times \mathrm{X}) \times \mathrm{LP}_{\mathrm{VAL,X}} \times \mathrm{LP}_{\mathrm{VAL,X}} \rightharpoonup \mathrm{LP}_{\mathrm{VAL,X}}$$

is defined as:

$$(cl_1, cl_2) \bullet_{(x,y)} (cl_1', cl_2') \overset{\text{def}}{=} (cl_1 \bullet_x cl_1', cl_2 \bullet_y cl_2')$$

where $\bullet_z$ is the context composition operator for multi-holed list contexts as defined in Example 3.33. If either of the list compositions is undefined then the entire list pair composition is undefined.

Example 3.37 (Multi-holed Context Algebra Composition)**.** If we are given a pair of multi-holed context algebras $\mathcal{M}_1 = (\mathcal{C}_1, \mathcal{X}_1, fh_1, \bullet_1)$ and $\mathcal{M}_2 = (\mathcal{C}_2, \mathcal{X}_2, fh_2, \bullet_2)$, then their *direct product* $\mathcal{M}_1 \times \mathcal{M}_2 = (\mathcal{C}_1 \times \mathcal{C}_2, \mathcal{X}_1 \times \mathcal{X}_2, fh_1 \times fh_2, \bullet_1 \times \bullet_2)$ is also a multi-holed context algebra[1]. For example $\mathcal{M}_{\mathrm{H}} \times \mathcal{M}_{\mathrm{LP}}$ combines heaps with list pairs. In Chapter 6 we will combine a heap structure with a list store structure (an addressable set of lists) in order to implement a tree structure.

Context Hole Uniqueness

In many examples of multi-holed context algebras, hole labels occur uniquely in the context structure, such as in Examples 3.32 - 3.34. However, in the general case, hole labels need not be unique. Notice that in the list pair context algebra (Example 3.36) hole labels may occur in both of the lists. For example, the list pair context $(a : x, b : x)$ is well formed. The hole x is unique within each list, so there is never any confusion about which hole is being filled by a composition. In order for context composition for some pair of labels (x, y) to be defined in this model, the lists must contain labels x and y respectively. However, despite having non-unique hole labels, the list pair structure still satisfies all of the properties for a multi-holed context algebra.

The list pair example shows that context holes do not need to be syntactically unique. However, they must still satisfy some uniqueness conditions with respect to context composition. That is, context composition should behave deterministically if defined. This is captured by the following lemma.

[1]The product of partial functions is defined pointwise in the natural fashion.

Lemma 3.38 (Filling Context Holes)**.** Given an arbitrary multi-holed context algebra $\mathcal{M} = (\mathcal{C}, \mathcal{X}, fh, \bullet)$, for all $c \in \mathcal{C}$ and $x, y \in \mathcal{X}$,

(a) $c \bullet_x y$ is only defined if $y \notin fh(c)$ or $x = y$,

(b) $(c \bullet_x y) \bullet_x z$ is *undefined* if $x \neq y$

Part (a) states that we cannot add duplicate holes to a context with context composition. Part (b) states that a context hole may only be filled once.

Proof. (a) If $c \bullet_x y$ is defined then $fh(c) \cap fh(y) \subseteq \{x\}$ and $fh(y) = \{y\}$ by definition. Thus either $y \notin fh(c)$ or $y = x$.

(b) If $x = y$ then the result is trivial, so assume that $x \neq y$. If $x \notin fh(c)$ then $c \bullet_x y$ is undefined by definition so the result holds. If $x \in fh(c)$ and $y \in fh(c)$ then $c \bullet_x y$ is undefined by definition and the result holds. If $x \in fh(c)$ and $y \notin fh(c)$ then $c \bullet_x y$ is defined and then $fh(c \bullet_x y) = (fh(c) \setminus \{x\}) \cup \{y\}$. Since $x \neq y$ we know that $x \notin fh(c \bullet_x y)$ and thus $(c \bullet_x y) \bullet_x z$ is undefined and the result holds. □

Hole Substitution

It is natural to define the substitution of hole labels in multi-holed contexts. Rather than having to define this operation directly, we can use context composition to encode the standard substitution of free labels in multi-holed contexts. We will see that this treatment of substitution still satisfies the standard properties of substitution.

Definition 3.39 (Hole Substitution)**.** Given an arbitrary multi-holed context algebra $\mathcal{M} = (\mathcal{C}, \mathcal{X}, fh, \bullet)$, $c_1, c_2 \in \mathcal{C}$ and $x \in \mathcal{X}$, label substituion is defined as:

$$c_1[c_2/x] \overset{\text{def}}{=} \begin{cases} c_1 \bullet_x c_2 & \text{if } x \in fh(c_1) \\ c_1 & \text{otherwise} \end{cases}$$

We now prove that this definition of substitution satisfies the following standard substitution lemmas, given for example in [38].

Lemma 3.40. Given an arbitrary multi-holed context algebra $\mathcal{M} = (\mathcal{C}, \mathcal{X}, fh, \bullet)$, for all $c, c_1, c_2 \in \mathcal{C}$ and $x \in \mathcal{X}$,

(a) $c[x/x] = c$,

(b) $c_1[c_2/x] = c_1$ if $x \notin fh(c_1)$,

(c) $fh(c_1[c_2/x]) = (fh(c_1) \setminus \{x\}) \cup fh(c_2)$ if $x \in fh(c_1)$ and $fh(c_1) \cap fh(c_2) \subseteq \{x\}$.

Proof. (a) There are two cases to consider. If $x \notin fh(c)$ then the result follows from the definition of substitution. If $x \in fh(c)$, we can show:

$$\begin{array}{rcll} c[x/x] & = & c \bullet_x x & \text{(substitution definition)} \\ & = & c & \text{(right identity of } \bullet_x\text{)} \end{array}$$

(b) This follows immediately from the definition of substitution.

(c) By the definition of substitution, if $x \in fh(c_1)$ then $c_1[c_2/x] = c_1 \bullet_x c_2$. Since $x \in fh(c_1)$ and $fh(c_1) \cap fh(c_2) \subseteq \{x\}$, $c_1 \bullet_x c_2$ is defined and thus $fh(c_1 \bullet c_2) = (fh(c_1)\backslash\{x\}) \cup fh(c_2)$ as required. □

Lemma 3.41. Given an arbitrary multi-holed context algebra $\mathcal{M} = (\mathcal{C}, \mathcal{X}, fh, \bullet)$, for all $c, c_1, c_2, c_3 \in \mathcal{C}$ and $x, y \in \mathcal{X}$ with $x \neq y$,

(a) $c_1[y/x][c_2/y] = c_1[c_2/x]$ if $y \notin fh(c_1)$

(b) $c[y/x][x/y] = c$ if $y \notin fh(c)$

(c) $c_1[c_2/x][c_3/y] = c_1[c_2[c_3/y]/x]$ if $y \notin fh(c_1)$

(d) $c_1[c_2/x][c_3/y] = c_1[c_3/y][c_2/x]$ if $x \notin fh(c_3)$ and $y \notin fh(c_2)$

(e) $c_1[c_2/x][c_3/x] = c_1[c_2[c_3/x]/x]$

Proof. (a) There are two cases to consider. If $x \notin fh(c_1)$, then $c_1[y/x][c_2/y] = c_1[c_2/y]$ and since $y \notin fh(c_1)$, $c_1[c_2/y] = c_1$. Similarly $c_1[c_2/x] = c_1$ so the result holds. Otherwise, if $x \in fh(c_1)$ then we can show:

$$\begin{array}{rcll} c_1[y/x][c_2/y] & = & (c_1 \bullet_x y)[c_2/y] & \text{(definition)} \\ & = & (c_1 \bullet_x y) \bullet_y c_2 & \text{(definition)} \\ & = & c_1 \bullet_x (y \bullet_y c_2) & \text{(semi-associativity)} \\ & = & c_1 \bullet_x c_2 & \text{(left identity of } \bullet_y\text{)} \\ & = & c_1[c_2/x] & \text{(definition)} \end{array}$$

(b) Using (a) with $c_2 = x$ we have $c[y/x][x/y] = c[x/x]$ and, by Lemma 3.40, $c[x/x] = c$ as required.

(c) There are three cases to consider. If $x \notin fh(c_1)$ then both sides are equal to c_1 since $y \notin fh(c_1)$. If $x \in fh(c_1)$ and $y \notin fh(c_2)$ then we can show:

$$\begin{array}{rcll} c_1[c_2/x][c_3/y] & = & c_1[c_2/x] & \text{Lemma 3.40} \\ & = & c_1[c_2[c_3/y]/x] & \text{Lemma 3.40} \end{array}$$

Otherwise $x \in fh(c_1)$ and $y \in fh(c_2)$ and we can show:

$$\begin{array}{rcll}
c_1[c_2/x][c_3/y] & = & (c_1 \bullet_x c_2)[c_3/y] & \text{substitution definition} \\
& = & (c_1 \bullet_x c_2) \bullet_y c_3 & \text{substitution definition} \\
& = & c_1 \bullet_x (c_2 \bullet_y c_3) & \text{semi-associativity} \\
& = & c_1 \bullet_x c_2[c_3/y] & \text{substitution definition} \\
& = & c_1[c_2[c_3/y]/x] & \text{substitution definition}
\end{array}$$

(d) There are three cases to consider. If $x \notin fh(c_1)$ then by Lemma 3.40 both sides are equal to $c_1[c_3/y]$ since $fh(c_1[c_3/y]) = (fh(c_1)\backslash y) \cup fh(c_3)$ and $x \notin fh(c_3)$. If $y \notin fh(c_1)$ then by Lemma 3.40 both sides are equal to $c_1[c_2/x]$ since $y \notin fh(c_2)$. Otherwise if $x \in fh(c_1)$ and $y \in fh(c_1)$ then we can show:

$$\begin{array}{rcll}
c_1[c_2/x][c_3/y] & = & (c_1 \bullet_x c_2)[c_3/y] & \text{substitution definition} \\
& = & (c_1 \bullet_x c_2) \bullet_y c_3 & \text{substitution definition} \\
& = & (c_1 \bullet_y c_3) \bullet_x c_2 & \text{semi-commutativity} \\
& = & (c_1 \bullet_y c_3)[c_2/x] & \text{substitution definition} \\
& = & c_1[c_3/y][c_2/x] & \text{substitution definition}
\end{array}$$

(e) There are three cases to consider. If $x \notin fh(c_1)$ then both sides are equal to c_1. Similarly, if $x \notin fh(c_2)$ then both sides are equal to $c_1[c_2/x]$. Otherwise, if $x \in fh(c_1)$ and $x \in fh(c_2)$ then we can show:

$$\begin{array}{rcll}
c_1[c_2/x][c_3/x] & = & (c_1 \bullet_x c_2)[c_3/x] & \text{substitution definition} \\
& = & (c_1 \bullet_x c_2) \bullet_x c_3 & \text{substitution definition} \\
& = & c_1 \bullet_x (c_2 \bullet_x c_3) & \text{semi-associativity} \\
& = & c_1 \bullet_x (c_2[c_3/x]) & \text{substitution definition} \\
& = & c_1[c_2[c_3/x]/x] & \text{substitution definition}
\end{array}$$

□

3.3.3 Segment Algebras

Recall the definition of a multi-holed context algebra $\mathcal{M}$ from Definition 3.31. We build up segment algebras from multi-holed context algebras in a similar fashion to how we generated a tree segment model from a tree context model in §3.1.3

Definition 3.42 (Segment Algebra). Given a multi-holed context algebra $\mathcal{M} = (\mathcal{C}, \mathcal{X}, fv, \bullet)$, a *segment algebra* $\mathcal{S}(\mathcal{M}) = (S, \mathsf{emp}, \leftarrow, fa, fh, +, \mathsf{comp})$ consists of:

◇ a set of segments S;

- ◇ an empty segment $\mathsf{emp} \in S$;
- ◇ a partial context addressing function $\leftarrow\ : \mathcal{X} \times \mathcal{C} \rightharpoonup S$;
- ◇ a free addresses function $fa : S \to \mathcal{P}_{\mathsf{fin}}(\mathcal{X})$;
- ◇ a free holes function $fh : S \to \mathcal{P}_{\mathsf{fin}}(\mathcal{X})$;
- ◇ a partial segment combination function $+ : S \times S \rightharpoonup S$;
- ◇ a partial compression function $\mathsf{comp} : \mathcal{X} \times S \rightharpoonup S$.

satisfying the properties given in Definition 3.43.

Notation: We write $(x)(s)$ in place of $\mathsf{comp}(x, s)$. This intentionally mirrors the restriction notation from the π-calculus [48]. We also write $x{\leftarrow}c$ in place of ${\leftarrow}(x, c)$.

The substitution of free labels in segments $s[y/x]$, which is necessary for defining the properties of segment algebras, is defined inductively as:

$$\begin{aligned}
\mathsf{emp}[y/x] &\overset{\text{def}}{=} \mathsf{emp} \\
(z{\leftarrow}c)[y/x] &\overset{\text{def}}{=} \begin{cases} y{\leftarrow}c & \text{if } z = x \\ z{\leftarrow}c[y/x] & \text{otherwise} \end{cases} \\
(s_1 + s_2)[y/x] &\overset{\text{def}}{=} s_1[y/x] + s_2[y/x] \\
((z)(s))[y/x] &\overset{\text{def}}{=} (z')(s[z'/z][y/x]) \text{ with } z' \notin fa(s) \cup fh(s) \cup \{x, y\}
\end{aligned}$$

where hole substitution for contexts $c[y/x]$ is as given in Definition 3.39.

Definition 3.43 (Segment Algebra Properties)**.** Segment algebras are required to satisfy the following properties: given a segment algebra $\mathcal{S}(\mathcal{M}) = (S, \mathsf{emp}, \leftarrow, fa, fh, +, \mathsf{comp})$ then for all $s, s_1, s_2, s_3 \in S$, $c, c_1, c_2 \in \mathcal{C}$ and $x, y, z \in \mathcal{X}$,

- ◇ if $x{\leftarrow}c \in \mathcal{S}$ then $x \notin fh(c)$ (that is, segments are cycle free);
- ◇ if $s_1 + s_2$ is defined, then $fa(s_1) \cap fa(s_2) = \emptyset$ and $fh(s_1) \cap fh(s_2) = \emptyset$ (that is, free addresses and free hole labels are unique in a segment);
- ◇ $s + \mathsf{emp} = s$ (that is, emp is the identity of $+$);
- ◇ $s_1 + s_2 = s_2 + s_1$ (that is, $+$ is commutative);
- ◇ $s_1 + (s_2 + s_3) = (s_1 + s_2) + s_3$ (that is, $+$ is associative);
- ◇ if $(x)(s)$ is defined, then $fa((x)(s)) = fa(s) \backslash \{x\}$ and $fh((x)(s)) = fh(s) \backslash \{x\}$;

- ◇ $(x)(\mathsf{emp}) = \mathsf{emp}$;
- ◇ $(x)(y)(s) = (y)(x)(s)$;
- ◇ $(x)(s) = (y)(s[y/x])$ if $y \notin fa(s) \cup fh(s)$;
- ◇ $(x)(s + s') = (x)(s) + s'$ if $x \notin fa(s')$ and $x \notin fh(s')$;
- ◇ $y \leftarrow c = (x)(y \leftarrow c_1 + x \leftarrow c_2)$ if $c = c_1 \bullet_x c_2$ and $x \neq y$ (this is called the *collapse-expand property*);

(Undefined terms are considered equal.)

Restriction is well known as a mechanism for hiding names in Milner's π-calculus [48] and similarly for hiding wires in process graphs [49]. Compression satisfies all of the properties of restriction from the π-calculus.

The collapse-expand property is new. Figure 3.5 introduces the intuition of collapsing and expanding a segment. When we expand a segment, we break it into two pieces and introduce a fresh label to track the location at which the splitting took place. This label is added as a hole in one segment and as the address of the other segment. Conversely, collapsing a segment allows us to join together two pieces that share a common restricted label, as a hole in one piece and as the address of the other. We shall see that these concepts are crucial in our reasoning. Recall that $c_1 \bullet_x c_2$ is only defined if $x \in fn(c_1)$. We say that a segment is in its compressed form if it cannot be compressed further using the collapse-expand property in a right-to-left reading.

Notation: Since the ordering of compression is not important (the 8th property) we write $(\bar{x})(s)$ where $\bar{x} \subseteq \mathrm{X}$ to mean the compression of the segment s by each of the labels $x \in \bar{x}$.

3.3.4 Segment Algebra Examples

We give a number of examples of segment algebras, used to provide fine-grained representations of some common data structures, including trees, lists and heaps. These extend the multi-holed context algebras that we introduced in §3.3.2.

Example 3.44 (Tree Segment Algebra)**.** Recall the multi-holed tree context algebra $\mathcal{M}_{\mathrm{T}} = (\mathrm{T}_{\mathrm{ID,X}}, \mathrm{X}, fv_{\mathrm{T}}, \bullet)$ from Example 3.32. The tree segment algebra is defined by $\mathcal{S}(\mathcal{M}_{\mathrm{T}}) = (\mathrm{S}_{\mathrm{T}}, \emptyset, \leftarrow, fa_{\mathrm{T}}, fh_{\mathrm{T}}, +, \mathsf{comp}_{\mathrm{T}})$, where the context addressing function

$$\leftarrow\ : \mathrm{X}_0 \times \mathrm{T}_{\mathrm{ID,X}} \rightharpoonup \mathrm{S}_{\mathrm{T}}$$

is defined as:

$$x \leftarrow ct \overset{\text{def}}{=} \begin{cases} \{(x, ct)\} & \text{if } x \notin fh_{\mathrm{T}}(ct) \\ \text{undefined} & \text{otherwise} \end{cases}$$

and $\mathrm{S_T}$, $fa_{\mathrm{T}} : \mathrm{S_T} \to \mathcal{P}_{\mathsf{fin}}(\mathrm{X})$, $fh_{\mathrm{T}} : \mathrm{S_T} \to \mathcal{P}_{\mathsf{fin}}(\mathrm{X})$, $+ : \mathrm{S_T} \times \mathrm{S_T} \rightharpoonup \mathrm{S_T}$ and $\mathsf{comp} : \mathrm{X} \times \mathrm{S_T} \rightharpoonup \mathrm{S_T}$ are defined as in Definitions 3.11, 3.13, 3.14 and 3.15 respectively. These definitions satisfy all of the properties of a segment algebra.

Example 3.45 (List Segment Algebra)**.** Recall the multi-holed list context algebra $\mathcal{M}_{\mathrm{L}} = (\mathrm{L_{VAL,X}}, \mathrm{X}, fh_{\mathrm{L}}, \bullet)$ from Example 3.33. Informally, list segments consist of sets of labelled list contexts, where labels can either be some $x \in \mathrm{X}$ or the special label 0 used to indicate that a list context is rooted, as in tree segments (Definition 3.11). A rooted list context cannot be extended to the left or the right. We write X_0 for the set of labels X extended with the special empty label 0. Recall that we do not allow 0 to be used as a hole label.

The list segment algebra is defined by $\mathcal{S}(\mathcal{M}_{\mathrm{L}}) = (\mathrm{S_L}, \emptyset, \leftarrow, fa_{\mathrm{L}}, fh_{\mathrm{L}}, +, \mathsf{comp}_{\mathrm{L}})$ where:

- ◇ The set of list segments S_L, ranged over by $sl, sl_1, ...$, is defined inductively as:

  $$sl \ ::= \ \emptyset \mid \{(x, cl)\} \mid sl \uplus sl$$

  with list contexts $cl \in \mathrm{L_{VAL,X}}$ as defined in Example3.33, addresses $x \in \mathrm{X}_0$, the restriction that addresses, hole labels and list values are unique across the set sl and the restriction that for each $(x, cl) \in sl$, $x \notin fh_{\mathrm{L}}(cl)$ (that is list segments are cycle free). The disjoint union of list segments $\uplus$ is defined only when the segments have disjoint addresses and hole labels. The operation is both associative and commutative.

- ◇ The context addressing function

  $$\leftarrow \ : \mathrm{X}_0 \times \mathrm{L_{VAL,X}} \rightharpoonup \mathrm{S_L}$$

  is defined as:

  $$x \leftarrow cl \overset{\text{def}}{=} \begin{cases} \{(x, cl)\} & \text{if } x \notin fh_{\mathrm{L}}(cl) \\ \text{undefined} & \text{otherwise} \end{cases}$$

- ◇ The free addresses function

  $$fa_{\mathrm{L}} : \mathrm{S_L} \to \mathcal{P}_{\mathsf{fin}}(\mathrm{X})$$

is defined by induction on the structure of list segments as:

$$\begin{aligned}
fa_{\mathrm{L}}(\emptyset) &\overset{\text{def}}{=} \emptyset \\
fa_{\mathrm{L}}(\{(x, cl)\}) &\overset{\text{def}}{=} \begin{cases} \emptyset & \text{if } x = 0 \\ \{x\} & \text{otherwise} \end{cases} \\
fa_{\mathrm{L}}(sl_1 \uplus sl_2) &\overset{\text{def}}{=} fa_{\mathrm{L}}(sl_1) \cup fa_{\mathrm{L}}(sl_2)
\end{aligned}$$

◇ The free holes function

$$fh_{\mathrm{L}} : \mathrm{S_L} \to \mathcal{P}_{\mathsf{fin}}(\mathrm{X})$$

is defined by induction on the structure of list segments as:

$$\begin{aligned}
fh_{\mathrm{L}}(\emptyset) &\overset{\text{def}}{=} \emptyset \\
fh_{\mathrm{L}}(\{(x, cl)\}) &\overset{\text{def}}{=} fh_{\mathrm{L}}(cl) \\
fh_{\mathrm{L}}(sl_1 \uplus sl_2) &\overset{\text{def}}{=} fh_{\mathrm{L}}(sl_1) \cup fh_{\mathrm{L}}(sl_2)
\end{aligned}$$

◇ The segment combination operator

$$+ : \mathrm{S_L} \times \mathrm{S_L} \rightharpoonup \mathrm{S_L}$$

is defined as:

$$sl_1 + sl_2 \overset{\text{def}}{=} \begin{cases} sl_1 \uplus sl_2 & \text{if } fa_{\mathrm{L}}(sl_1) \cap fa_{\mathrm{L}}(sl_2) = \emptyset \\ & \text{and } fh_{\mathrm{L}}(sl_1) \cap fh_{\mathrm{L}}(sl_2) = \emptyset \\ \text{undefined} & \text{otherwise} \end{cases}$$

◇ The segment compression operator

$$\mathsf{comp}_{\mathrm{L}} : \mathrm{X} \times \mathrm{S_L} \rightharpoonup \mathrm{S_L}$$

is defined as:

$$\mathsf{comp}_{\mathrm{L}}(x, sl) \overset{\text{def}}{=} \begin{cases} sl & \text{if } x \notin fa_{\mathrm{L}}(sl) \text{ and } x \notin fh_{\mathrm{L}}(sl) \\ sl' + \{(z, cl \bullet_x cl')\} & \text{if } \exists sl', z, cl, cl'. \\ & sl = sl' + \{(z, cl), (x, cl')\} \\ & \text{and } x \in fh_{\mathrm{L}}(cl) \\ sl' + \{(0, cl)\} & \text{if } \exists sl', cl.\, sl = sl' + \{(x, cl)\} \\ & \text{and } x \notin fh_{\mathrm{L}}(sl') \\ \text{undefined} & \text{otherwise} \end{cases}$$

Notation: We write $x{\leftarrow}cl$ for $\{(x, cl)\}$ and $\lceil cl \rceil$ as shorthand for $\{(0, cl)\}$.

Example 3.46 (Heap Segment Algebra)**.** Recall the multi-holed heap context algebra $\mathcal{M}_{\mathrm{H}} = (\mathrm{H}_{\mathrm{ADR,X}}, \mathrm{X}, fh_{\mathrm{H}}, \bullet)$ from Example 3.34. Informally, heap segments consist of sets of labelled heap contexts, where labels $x \in \mathrm{X}_0$ as above.

The heap segment algebra is defined by $\mathcal{S}(\mathcal{M}_{\mathrm{H}}) = (\mathrm{S}_{\mathrm{H}}, \emptyset, \leftarrow, fa_{TH}, fh_{\mathrm{H}}, +, \mathsf{comp}_{\mathrm{H}})$ where:

- ◇ The set of heap segments S_H, ranged over by $sh, sh_1, ...$, is defined inductively as:

$$sh \quad ::= \quad \emptyset \mid \{(x, ch)\} \mid sh \uplus sh$$

  with heap contexts $ch \in \mathrm{H}_{\mathrm{ADR,X}}$, addresses $x \in \mathrm{X}_0$, the restriction that non-zero segment addresses, hole labels and heap addresses are unique across the set sh and the restriction that for each $(x, ch) \in sh$, $x \notin fh_{\mathrm{H}}(ch)$ (that is heap segments are cycle free). The disjoint union of list segments $\uplus$ is defined only when the segments have disjoint addresses, hole labels and heap addresses. The operation is both associative and commutative.

- ◇ The context addressing function, free addresses function, free hole function, segment combination operator and segment compression operator are all analogous to their corresponding definitions in the tree and list segment algebra examples.

Notice that in heap segments we do not require that the address 0 is unique in the heap, as heaps can always be joined so long as their heap addresses are disjoint. This means that we can have multiple rooted heaps. Moreover, we chose to add the extra property that the combination of disjoint rooted heaps is equivalent to disjoint heap union. That is, $\{(0{\leftarrow}ch_1)\} \uplus \{(0{\leftarrow}ch_2)\} = \{(0{\leftarrow}ch_1 \star ch_2)\}$. This means that it is natural to associate the rooted empty heap with the empty segment: $\emptyset = \{(0{\leftarrow}\mathsf{emp})\}$. We justify this by observing the following:

$$\begin{aligned}\{(0{\leftarrow}ch)\} \uplus \{(0{\leftarrow}\mathsf{emp})\} &= \{(0{\leftarrow}ch \star \mathsf{emp})\} \\ &= \{(0{\leftarrow}ch)\} \\ &= \{(0{\leftarrow}ch)\} \uplus \emptyset\end{aligned}$$

Notation: We write $x{\leftarrow}ch$ for $\{(x, ch)\}$ and $\lceil ch \rceil$ as shorthand for $\{(0, ch)\}$.

Example 3.47 (Separation Algebras as Segment Algebras)**.** Given a separation algebra $(S, \star, u)$ we have seen how this gives rise to a multi-holed context algebra

$\mathcal{M}_\mathrm{S} = (\mathrm{S}_\mathrm{X}, \mathrm{X}, fh_S, \bullet)$ in Example 3.35. As with the heap segment model see above, we label contexts with some $x \in \mathrm{X}_0$.

We can lift this multi-holed context algebra to a segment algebra defined by $\mathcal{S}(\mathcal{M}_\mathrm{S}) = (\mathrm{S}_S, \emptyset, \leftarrow, fa_S, fh_S, +, \mathsf{comp}_S)$ where:

- ◇ The set of segments S_S, ranged over by $ss, ss_1, ...$, is defined inductively as:

  $$ss \ ::= \ \emptyset \mid \{(x, cs)\} \mid ss \uplus ss$$

  with contexts $cs \in \mathrm{S}_\mathrm{X}$, addresses $x \in \mathrm{X}_0$, the restriction that non-zero segment addresses and hole labels are unique across the set ss and the restriction that for each $(x, cs) \in ss$, $x \notin fh_\mathrm{S}(cs)$. The disjoint union of segments $\uplus$ is defined only when the segments have disjoint addresses, hole labels and contents. The operation is both associative and commutative. Note that disjointness of the contents will depend on the disjoint properties associated with the original separation algebra.

- ◇ The context addressing function, free addresses function, free holes function, segment combination operator and segment compression operator are all analogous to their corresponding definitions on the tree and list segment algebra examples.

As with heap segments we choose to add the additional property that the combination of rooted contexts is equivalent to combining those contexts with the $\star$ operator. That is, $\{(0{\leftarrow}cs_1)\} \uplus \{(0{\leftarrow}cs_2)\} = \{(0{\leftarrow}cs_1 \star cs_2)\}$. Again, we also choose to associate the rooted unit with the empty segment: $\emptyset = \{(0{\leftarrow}u)\}$.

Notation: We write $x{\leftarrow}cs$ for $\{(x, cs)\}$ and $\lceil cs \rceil$ as shorthand for $\{(0, cs)\}$.

Example 3.48 (Segment Algebra Composition)**.** Given a pair of segment algebras

$$\mathcal{S}(\mathcal{M}_1) \ = \ (S_1, \mathsf{emp}_1, \leftarrow_1, fa_1, fh_1, +_1, \mathsf{comp}_1)$$

and

$$\mathcal{S}(\mathcal{M}_2) \ = \ (S_2, \mathsf{emp}_2, \leftarrow_2, fa_2, fh_2, +_2, \mathsf{comp}_2)$$

based on the multi-holed context algebras

$$\mathcal{M}_1 = (\mathcal{C}_1, \mathcal{X}_1, fh_1, \bullet_1) \quad \text{and} \quad \mathcal{M}_2 = (\mathcal{C}_2, \mathcal{X}_2, fh_2, \bullet_2)$$

their composition is defined as

$$\mathcal{S}(\mathcal{M}_1) \times \mathcal{S}(\mathcal{M}_2) \quad = \quad (S, \mathsf{emp}, \leftarrow, fa, fh, +, \mathsf{comp})$$

where,

$$\begin{aligned}
S &\overset{\text{def}}{=} S_1 \times S_2 \\
\mathsf{emp} &\overset{\text{def}}{=} (\mathsf{emp}_1, \mathsf{emp}_2) \\
(x_1, x_2){\leftarrow}(c_1, c_2) &\overset{\text{def}}{=} (x_1{\leftarrow_1}c_1, x_2{\leftarrow_2}c_2) \\
fa(s_1, s_2) &\overset{\text{def}}{=} (fa_1(s_1), fa_2(s_2)) \\
fh(s_1, s_2) &\overset{\text{def}}{=} (fh_1(s_1), fh_2(s_2)) \\
(s_1, s_2) + (s_1', s_2') &\overset{\text{def}}{=} (s_1 +_1 s_1', s_2 +_2 s_2') \\
\mathsf{comp}((x_1, x_2), (s_1, s_2)) &\overset{\text{def}}{=} (\mathsf{comp}_1(x_1, s_1), \mathsf{comp}_2(x_2, s_2))
\end{aligned}$$

for $x_1 \in \mathcal{X}_1$, $x_2 \in \mathcal{X}_2$, $c_1 \in \mathcal{C}_1$, $c_2 \in \mathcal{C}_2$, $s_1, s_1' \in S_1$ and $s_2, s_2' \in S_2$. The result of this composition is also a segment algebra.

In Chapter 6 we will be combining a heap segment algebra with a list-store segment algebra in order to implement the structure of a tree segment algebra.

Sub-Separation Algebra

Given an arbitrary segment algebra $\mathcal{S}(\mathcal{M}) = (S, \mathsf{emp}, \leftarrow, fa, fh, +, \mathsf{comp})$, the sub-algebra $(S, +, \mathsf{emp})$ is a separation algebra. All of the properties required of a separation algebra follow from the properties required of $+$ and emp for a segment algebra.

4 Fine-grained Abstract Local Reasoning

We have shown how to refine context logic to obtain a more fine-grained analysis of abstract data. We shall now introduce a general framework for reasoning about fine-grained abstract data structures, building on similar work for context logic [26]. In particular we will introduce local Hoare reasoning based on segment logic.

First, in §4.1, we introduce the simple imperative programming language about which we are going to reason. This language will be parameterised by some choice of basic commands, allowing us to tailor the language to different domains. In §4.2 we give the operational and axiomatic semantics of this programming language. The operational semantics provide us with a computational model for our programming language. By contrast, the axiomatic semantics, given in the style of local Hoare reasoning, allows us to express abstract properties of programs written in our language. Finally, in §4.3, we show that our axiomatic semantics is sound with respect to our operational semantics. This means that any properties we prove in our local Hoare reasoning system are also true of the underlying computational model.

4.1 Programming Language

We introduce our imperative programming language, which includes mutable variables and standard control-flow constructs, such as while loops and procedure calls. As well as manipulating variables, our programs also operate on a mutable data store. Our programming language is parametrised by a set of basic commands $\textsc{Cmd}$, ranged over by φ, that manipulate this data store. The choice of these basic commands depends on the domain over which the language is to be used: for instance, to work with a tree the commands lookup, node insertion, subtree deletion and subtree movement are natural; to work with a list the commands lookup, element insertion and element removal are natural; and to work with the heap the commands allocation, mutation, lookup and heap cell disposal are natural.

We assume a fixed set of program variables $\textsc{Var}$ which are interpreted over a set of values $\textsc{Val}$ that at least includes integers ($\mathbb{Z} \subseteq \textsc{Val}$). Our value expressions are similarly assumed to include syntax for basic arithmetic and comparisons, as well as variables and the standard Boolean operators. The actual definition of expression syntax is open-ended, allowing us to extend them to include values besides just integers. When no additional values are necessary, we implicitly work with the minimal expression definitions meeting our assumptions.

Assumption 1 (Expression Syntax)**.** Assume we have a set of *value expressions* $\textsc{Expr}$ ranged over by $E, E_1, ...$, such that, for all $E_1, E_2 \in \textsc{Expr}$,

$$\begin{array}{rcl} \textsc{Val} & \subseteq & \textsc{Expr} \\ \textsc{Var} & \subseteq & \textsc{Expr} \\ E_1 + E_2 & \in & \textsc{Expr} \\ E_1 - E_2 & \in & \textsc{Expr} \end{array}$$

Also assume we have a set of *Boolean expressions* $\textsc{BExpr}$ ranged over by $B, B_1, ...$, such that, for all $E_1, E_2 \in \textsc{Expr}$ and $B_1, B_2 \in \textsc{BExpr}$,

$$\begin{array}{rcl} E_1 = E_2 & \in & \textsc{BExpr} \\ E_1 < E_2 & \in & \textsc{BExpr} \\ \texttt{false} & \in & \textsc{BExpr} \\ B_1 \Rightarrow B_2 & \in & \textsc{BExpr} \end{array}$$

The remaining standard Boolean expressions for $\neg$, $\texttt{true}$, $\vee$, $\wedge$, $>$, $\leq$ and $\geq$ can be derived.

Definition 4.1 (Programming Language Syntax)**.** Given a set of basic commands $\textsc{Cmd}$ ranged over by φ, the set of commands of language $\mathcal{L}_{\textsc{Cmd}}$, ranged over by $\mathbb{C}, \mathbb{C}_1, ...$, is defined as:

$$\begin{array}{rcl} \mathbb{C} & ::= & \varphi \mid \texttt{skip} \mid \texttt{x} := E \mid \mathbb{C};\mathbb{C} \\ & & \mid \texttt{if } B \texttt{ then } \mathbb{C} \texttt{ else } \mathbb{C} \mid \texttt{while } B \texttt{ do } \mathbb{C} \\ & & \mid \texttt{procs } \overrightarrow{\texttt{r}_1} := \texttt{f}_1(\overrightarrow{\texttt{x}_1})\{\mathbb{C}\}, ..., \overrightarrow{\texttt{r}_k} := \texttt{f}_k(\overrightarrow{\texttt{x}_k})\{\mathbb{C}\} \texttt{ in } \mathbb{C} \\ & & \mid \texttt{call } \overrightarrow{\texttt{r}} := \texttt{f}(\overrightarrow{E}) \mid \texttt{local x in } \mathbb{C} \end{array}$$

where $\texttt{x}, \texttt{r}, \ldots \in \textsc{Var}$ range over program variables, $\overrightarrow{\texttt{x}_i}, \overrightarrow{\texttt{r}_i}, \overrightarrow{\texttt{r}} \in \textsc{Var}^*$ range over lists of program variables, $E, E_1, \ldots \in \textsc{Expr}$ range over value expressions, $\overrightarrow{E} \in \textsc{Expr}^*$ ranges over lists of value expressions, $B \in \textsc{BExpr}$ ranges over boolean expressions, and $\texttt{f}, \texttt{f}_1, \ldots \in \textsc{PName}$, where $\textsc{PName}$ is the set of procedure names. The names $\texttt{f}_1, \ldots, \texttt{f}_k$ of procedures defined in a single $\texttt{procs} - \texttt{in}$ block are required to be pairwise distinct. The parameter and return variables are also required to be pairwise distinct within each procedure definition.

4.2 Semantics

We give two different ways of providing the semantics of our programming language, one in the operational style and one in the axiomatic style. In §4.3, we show that our axiomatic semantics is sound with respect to our operational semantics.

Both styles of semantics will need a way of representing the current valuation of the accessible program variables at each point in the program. We model this using a variable store.

Definition 4.2 (Variable Stores)**.** The set of *variable stores* Σ, ranged over by $\sigma, \sigma_1, ...$, is the set of finite partial functions $\sigma : \textsc{Var} \rightharpoonup_{\text{fin}} \textsc{Val}$ mapping program variables to values. The disjoint union of variable stores $\uplus$ is defined only when the variable stores have disjoint domains.

Notation: We write $\emptyset$ for the empty variable store, $\sigma[\mathtt{x} \mapsto u]$ for the variable store σ overwritten with $\sigma(\mathtt{x}) = u$ and $dom(\sigma)$ for the domain of σ.

We define the semantics of expressions in terms of partial functions so that our expression semantics may be open ended. This allows us to have expressions in our syntax that do not evaluate in a meaningful way. For example, comparing a string value to an integer value or subtracting a Boolean value from an integer value are not typically well defined operations. Of course, if we do decide to give these kinds of expressions some meaning, then our framework is flexible enough to allow us to do so.

Assumption 2 (Expression Semantics)**.** The semantics of value expressions is given by the function $\mathcal{E}[\![(\cdot)]\!] : \textsc{Expr} \rightarrow (\Sigma \rightharpoonup \textsc{Val})$. The semantics of boolean expressions is given by the function $\mathcal{B}[\![(\cdot)]\!] : \textsc{Expr} \rightarrow (\Sigma \rightharpoonup \textsc{Bool})$, where $\textsc{Bool} = \{\mathsf{true}, \mathsf{false}\}$. These functions are required to satisfy the following conditions:

for all $\sigma \in \Sigma$, $n, n_1, n_2 \in \mathbb{Z}$, $\mathtt{x} \in \textsc{Var}$, $E_1, E_2 \in \textsc{Expr}$ and $B_1, B_2 \in \textsc{BExpr}$,

$$\begin{aligned}
\mathcal{E}[\![n]\!]\sigma &= n \\
\mathcal{E}[\![\mathtt{x}]\!]\sigma &= \begin{cases} \sigma(\mathtt{x}) & \text{if } x \in dom(\sigma) \\ \text{undefined} & \text{otherwise} \end{cases} \\
\mathcal{E}[\![E_1 + E_2]\!]\sigma &= \begin{cases} n_1 + n_2 & \text{if } \mathcal{E}[\![E_1]\!]\sigma = n_1 \text{ and } \mathcal{E}[\![E_2]\!]\sigma = n_2 \\ \text{undefined} & \text{otherwise} \end{cases} \\
\mathcal{E}[\![E_1 - E_2]\!]\sigma &= \begin{cases} n_1 - n_2 & \text{if } \mathcal{E}[\![E_1]\!]\sigma = n_1 \text{ and } \mathcal{E}[\![E_2]\!]\sigma = n_2 \\ \text{undefined} & \text{otherwise} \end{cases}
\end{aligned}$$

$$\begin{aligned}
\mathcal{B}[\![E_1 = E_2]\!]\sigma &= \begin{cases} \mathsf{true} & \text{if } \mathcal{E}[\![E_1]\!]\sigma = \mathcal{E}[\![E_2]\!]\sigma \\ \mathsf{false} & \text{if } \mathcal{E}[\![E_1]\!]\sigma \neq \mathcal{E}[\![E_2]\!]\sigma \\ \text{undefined} & \text{if } \mathcal{E}[\![E_1]\!]\sigma \text{ or } \mathcal{E}[\![E_2]\!]\sigma \text{ is undefined} \end{cases} \\
\mathcal{B}[\![E_1 < E_2]\!]\sigma &= \begin{cases} \mathsf{true} & \text{if } \mathcal{E}[\![E_1]\!]\sigma < \mathcal{E}[\![E_2]\!]\sigma \\ \mathsf{false} & \text{if } \mathcal{E}[\![E_1]\!]\sigma \geq \mathcal{E}[\![E_2]\!]\sigma \\ \text{undefined} & \text{if } \mathcal{E}[\![E_1]\!]\sigma \notin \mathbb{Z} \text{ or } \mathcal{E}[\![E_2]\!]\sigma \notin \mathbb{Z} \end{cases} \\
\mathcal{B}[\![\mathtt{false}]\!]\sigma &= \mathsf{false} \\
\mathcal{B}[\![B_1 \Rightarrow B_2]\!]\sigma &= \begin{cases} \mathsf{true} & \text{if } \mathcal{B}[\![B_1]\!]\sigma = \mathsf{true} \Rightarrow \mathcal{B}[\![B_2]\!]\sigma = \mathsf{true} \\ \mathsf{false} & \text{if } \mathcal{B}[\![B_1]\!]\sigma = \mathsf{true} \not\Rightarrow \mathcal{B}[\![B_2]\!]\sigma = \mathsf{true} \\ \text{undefined} & \text{if } \mathcal{B}[\![B_1]\!]\sigma \text{ or } \mathcal{B}[\![B_2]\!]\sigma \text{ is undefined} \end{cases}
\end{aligned}$$

Notice that the semantics of an expression can be undefined for a particular variable store, for instance if some variable in the expression is not assigned in the variable store.

4.2.1 Operational Semantics

We now introduce a big-step operational semantics for our programming language. The semantics will depend on the interpretation of the set of basic commands $\textsc{Cmd}$. In general, the state of a program will not only consist of a variable store, but also of some other data structure that is accessed exclusively through the basic commands, such as a tree, list or heap. We assume an arbitrary set of complete data structures $\mathcal{D}$, ranged over by $d, d', d_1, ...$

The set of program states is $\textsc{PState} = \mathcal{D} \times \Sigma$, the set of pairs of complete data structures and variable stores. We assume that the basic commands of our language have a semantic interpretation over these program states.

Assumption 3 (Semantics of Basic Commands)**.** Assume we have a semantic interpretation function for basic commands,

$$\mathcal{C}[\![(\cdot)]\!] : \textsc{Cmd} \to (\textsc{PState} \rightharpoonup \mathcal{P}(\textsc{PState})).$$

Furthermore, assume that for each $\varphi \in \textsc{Cmd}$, $\mathcal{C}[\![\varphi]\!]$ preserves the domain of the variable store. That is, for all $(d, \sigma), (d', \sigma') \in \textsc{PState}$ if $(d', \sigma') \in \mathcal{C}[\![\varphi]\!](d, \sigma)$ then $dom(\sigma) = dom(\sigma')$.

The semantics of a basic command is a partial function. When $\mathcal{C}[\![\varphi]\!](d, \sigma)$ is undefined, we say that the command faults when run on program state (d, σ). When $\mathcal{C}[\![\varphi]\!](d, \sigma)$ is defined, then either the command does not terminate, in which case $\mathcal{C}[\![\varphi]\!](d, \sigma) = \emptyset$, or the command nondeterministically results in one of the states in the non-empty set $\mathcal{C}[\![\varphi]\!](d, \sigma)$.

In order to define our operational semantics, it is necessary to provide two additional definitions. The first of these is for *procedure definition environments* which are used to interpret procedure calls. When a `procs` − `in` block is encountered, the semantics will create a procedure definition environment for each of the procedures defined in that block. This environment is then added to the stack of procedure definitions that are used to interpret the procedure calls within the block. This method of managing procedure calls allows us to provide a semantics for programs with recursive procedure calls.

Definition 4.3 (Procedure Definition Environments)**.** The set of *procedure definition environments* $\textsc{PDef}$, ranged over by $\mu, \mu', \mu_1, ...$, is the set of partial functions $\mu : \textsc{PName} \rightharpoonup_{\text{fin}} (\textsc{Var}^* \times \mathcal{L}_{\textsc{Cmd}} \times \textsc{Var}^*)$ from procedure names to triples of a list of input variables, a program and a list of output variables.

Notation: We write $\textsc{Var}^i$ when we know that the list of variables is of length i.

Definition 4.4 (Procedure Definition Stacks)**.** The set of *procedure definition stacks* $\textsc{PDef}^*$, ranged over by $\gamma, \gamma', \gamma_1, ...$, is the set of finite sequences of procedure definition environments.

When we look up a procedure in a procedure definition stack we want to return the most recent definition of that procedure. However, we also want to ensure that any procedure calls made by this procedure have the behaviour as defined at the point the procedure was defined. To ensure that this is the case we also return the procedure definition environment that was available to that procedure at the time it was defined.

Definition 4.5 (Procedure Lookup)**.** The operation of looking up a procedure in a procedure definition stack,

$$\mathsf{lookup} : \text{PN}\textsc{ame} \times (\text{P}\textsc{def}^*) \rightharpoonup (\text{V}\textsc{ar}^* \times \mathcal{L}_{\text{C}\textsc{md}} \times \text{V}\textsc{ar}^*) \times (\text{PD}\textsc{ef}^*)$$

is defined as:

$$\mathsf{lookup}(\mathtt{f}, \mu : \gamma) \overset{\text{def}}{=} \begin{cases} (\mu(\mathtt{f}), \mu : \gamma) & \text{if } \mathtt{f} \in \mathit{dom}(\mu) \\ \mathsf{lookup}(\mathtt{f}, \gamma) & \text{otherwise} \end{cases}$$

where $\mu \in \text{PD}\textsc{ef}$ and $\gamma \in \text{PD}\textsc{ef}^*$.

The lookup procedure returns a pair consisting of the procedure definition and the procedure definition stack that should be used in executing the procedure. This procedure definition stack contains the procedure definitions that were in scope at the point when the procedure in question was defined, as well as the procedure definitions that were defined at the same time as the procedure in question. This last point is key in allowing for the definition of mutually recursive procedures.

Using procedure definition stacks, our operational semantics provides static (lexical) scoping for procedure calls. For example, if some procedure `f` calls a procedure named `g` in its body, the procedure invoked will always be the most recently defined procedure named `g` at the point `f` was defined. By contrast, dynamic typing would instead invoke the most recently defined procedure named `g` at the point `f` was invoked.

Finally, we define the set of outcomes $\text{O}\textsc{ut}$, ranged over by $o, o', ...$, generated by executing a program. The result of a successful program execution will always result in some program state, if it terminates. However, not every program execution is necessarily successful. For example, a execution that tries to dereference a variable that is not defined is considered to fail. Such executions are called *faulting* executions, and we denote them with the symbol $\lightning$. The set of outcomes is then taken to be the set of program states plus the faulting outcome: $\text{O}\textsc{ut} = \text{PS}\textsc{tate} \cup \{\lightning\}$

Notice that we do not know if a program will terminate from a given initial state. It is possible for programs to loop forever. However, we are primarily concerned with terminating executions, so non-terminating executions are ignored by the semantics.

We define the big-step semantics for programs, given by judgments of the form $\mathbb{C}, \gamma, d, \sigma \Downarrow o$ denoting that, when run in the context of a procedure definition stack γ, data structure d and variable store σ, the program $\mathbb{C}$ results in the outcome o.

Definition 4.6 (Operational Semantics)**.** The big-step operational semantics for the language $\mathcal{L}_{\text{CMD}}$ is defined by the relation $\Downarrow$,

$$\Downarrow : (\mathcal{L}_{\text{CMD}} \times \text{PD}\textsc{ef}^* \times \mathcal{D} \times \Sigma) \times \textsc{Out}$$

which is given by the rules given in Figure 4.1 and Figure 4.2.

Notation: We write $|\overrightarrow{\mathtt{x}}|$ for the length of the list of variables $\overrightarrow{\mathtt{x}}$, and similarly for lists of expressions.

The operational semantics of our programming language is given in terms of a complete data structure d. In the next section, §4.2.2, we define an axiomatic semantic for our programming language that treats the data structure as a segment algebra.

4.2.2 Axiomatic Semantics

We define an axiomatic semantics for the language $\mathcal{L}_{\text{CMD}}$ using local Hoare reasoning. This semantics treats the space of program states as pairs consisting of a segment algebra $\mathcal{S}(\mathcal{M}) = (S, \mathsf{emp}, \leftarrow, fa, fh, +, \mathsf{comp})$, as defined in Definition 3.42, and a variable store Σ, as defined in Definition 4.2. That is, $\textsc{State} = S \times \Sigma$. Recall that segments logic can describe properties of incomplete data structures, whereas our operational semantics can only describe the effects of programs on complete data structures. In §4.3 we will show how to relate our axiomatic semantics on segments to our operational semantics on complete data structures.

The axiomatic semantics is parametrised by both the choice of $\mathcal{S}(\mathcal{M})$ and the axioms given for the basic commands. This gives us a fixed way of treating program variables, but allows for a flexible choice of the remaining data structure.

Before we define our axiomatic semantics, we shall take a moment to discuss the treatment of program variables. In the background chapter we were quite relaxed about punning program variables and logical variables. However, such a pun does mean that some inference rules, the frame rule in particular, need side conditions and the axiom for assignment is more complex than we would like. From this point forward, we choose to be more formal and treat the variable store as another program resource. The idea of 'variables as resource' was first introduced in separation logic by Bornat, Calcagno and Yang [8]. The main advantage of working with variables as resource is that it removes the side condition from the frame rule and simplifies the assignnment axiom.

$$\frac{(d', \sigma') \in \mathcal{C}[\![\varphi]\!](d, \sigma)}{\varphi, \gamma, d, \sigma \Downarrow d', \sigma'} \qquad \frac{}{\texttt{skip}, \gamma, d, \sigma \Downarrow d, \sigma}$$

$$\frac{\mathcal{E}[\![E]\!](\sigma[\texttt{x} \mapsto u]) = u'}{\texttt{x} := E, \gamma, d, \sigma[\texttt{x} \mapsto u] \Downarrow d, \sigma[\texttt{x} \mapsto u']}$$

$$\frac{\mathbb{C}_1, \gamma, d, \sigma \Downarrow d', \sigma' \qquad \mathbb{C}_2, \gamma, d', \sigma' \Downarrow d'', \sigma''}{\mathbb{C}_1 \ ; \mathbb{C}_2, \gamma, d, \sigma \Downarrow d'', \sigma''}$$

$$\frac{\mathcal{B}[\![B]\!]\sigma = \mathsf{true} \qquad \mathbb{C}_1, \gamma, d, \sigma \Downarrow d', \sigma'}{\texttt{if } B \texttt{ then } \mathbb{C}_1 \texttt{ else } \mathbb{C}_2, \gamma, d, \sigma \Downarrow d', \sigma'}$$

$$\frac{\mathcal{B}[\![B]\!]\sigma = \mathsf{false} \qquad \mathbb{C}_2, \gamma, d, \sigma \Downarrow d', \sigma'}{\texttt{if } B \texttt{ then } \mathbb{C}_1 \texttt{ else } \mathbb{C}_2, \gamma, d, \sigma \Downarrow d', \sigma'}$$

$$\frac{\mathcal{B}[\![B]\!]\sigma = \mathsf{true} \qquad \mathbb{C} \ ; \texttt{while } B \texttt{ do } \mathbb{C}, \gamma, d, \sigma \Downarrow d', \sigma'}{\texttt{while } B \texttt{ do } \mathbb{C}, \gamma, d, \sigma \Downarrow d', \sigma'}$$

$$\frac{\mathcal{B}[\![B]\!]\sigma = \mathsf{false}}{\texttt{while } B \texttt{ do } \mathbb{C}, \gamma, d, \sigma \Downarrow d, \sigma}$$

$$\frac{\mathbb{C}, [\texttt{f}_1 \mapsto (\overrightarrow{\texttt{x}_1}, \mathbb{C}_1, \overrightarrow{\texttt{r}_1}), ..., \texttt{f}_k \mapsto (\overrightarrow{\texttt{x}_k}, \mathbb{C}_k, \overrightarrow{\texttt{r}_k})] : \gamma, d, \sigma \Downarrow d', \sigma'}{(\texttt{procs } \overrightarrow{\texttt{r}_1} := \texttt{f}_1(\overrightarrow{\texttt{x}_1})\{\mathbb{C}_1\}, ..., \overrightarrow{\texttt{r}_k} := \texttt{f}_k(\overrightarrow{\texttt{x}_k})\{\mathbb{C}_k\} \texttt{ in } \mathbb{C}), \gamma, d, \sigma \Downarrow d', \sigma'}$$

$$\frac{\begin{array}{c} \mathsf{lookup}(\texttt{f}, \gamma) = ((\overrightarrow{\texttt{x}}, \mathbb{C}, \overrightarrow{\texttt{y}}), \gamma') \qquad \mathcal{E}[\![\overrightarrow{E}]\!]\sigma = \overrightarrow{v} \\ \overrightarrow{\texttt{r}} \in dom(\sigma) \qquad |\overrightarrow{E}| = |\overrightarrow{\texttt{x}}| \qquad |\overrightarrow{\texttt{r}}| = |\overrightarrow{\texttt{y}}| \\ \mathbb{C}, \gamma', d, \emptyset[\overrightarrow{\texttt{y}} \mapsto \overrightarrow{w}][\overrightarrow{\texttt{x}} \mapsto \overrightarrow{v}] \Downarrow d', \sigma' \qquad \sigma[\overrightarrow{\texttt{r}} \mapsto \mathcal{E}[\![\overrightarrow{\texttt{y}}]\!]\sigma'] = \sigma'' \end{array}}{\texttt{call } \overrightarrow{\texttt{r}} := \texttt{f}(\overrightarrow{E}), \gamma, d, \sigma \Downarrow d', \sigma''}$$

$$\frac{x \notin dom(\sigma) \qquad x \notin dom(\sigma') \qquad \mathbb{C}, \gamma, d, \sigma[\texttt{x} \mapsto v] \Downarrow d', \sigma'[\texttt{x} \mapsto w]}{\texttt{local x in } \mathbb{C}, \gamma, d, \sigma \Downarrow d', \sigma'}$$

$$\frac{\mathbb{C}, \gamma, d, \sigma[\texttt{x} \mapsto v] \Downarrow d', \sigma'[\texttt{x} \mapsto w]}{\texttt{local x in } \mathbb{C}, \gamma, d, \sigma[\texttt{x} \mapsto u] \Downarrow d', \sigma'[\texttt{x} \mapsto u]}$$

Figure 4.1: Operational semantics for $\mathcal{L}_{\text{CMD}}$ (non-faulting cases).

$$\frac{\mathcal{C}[\![\varphi]\!](d,\sigma) \text{ undefined}}{\varphi,\gamma,d,\sigma \Downarrow \lightning} \qquad \frac{\mathcal{E}[\![E]\!]\sigma \text{ undefined}}{\mathtt{x} := E,\gamma,d,\sigma \Downarrow \lightning} \qquad \frac{\mathtt{x} \notin \mathit{dom}(\sigma)}{\mathtt{x} := E,\gamma,d,\sigma \Downarrow \lightning}$$

$$\frac{\mathbb{C}_1,\gamma,d,\sigma \Downarrow \lightning}{\mathbb{C}_1 \;;\; \mathbb{C}_2,\gamma,d,\sigma \Downarrow \lightning} \qquad \frac{\mathbb{C}_1,\gamma,d,\sigma \Downarrow d',\sigma' \qquad \mathbb{C}_2,\gamma,d',\sigma' \Downarrow \lightning}{\mathbb{C}_1 \;;\; \mathbb{C}_2,\gamma,d,\sigma \Downarrow \lightning}$$

$$\frac{\mathcal{B}[\![B]\!]\sigma = \mathsf{true} \qquad \mathbb{C}_1,\gamma,d,\sigma \Downarrow \lightning}{\mathtt{if}\ B\ \mathtt{then}\ \mathbb{C}_1\ \mathtt{else}\ \mathbb{C}_2,\gamma,d,\sigma \Downarrow \lightning} \qquad \frac{\mathcal{B}[\![B]\!]\sigma = \mathsf{false} \qquad \mathbb{C}_2,\gamma,d,\sigma \Downarrow \lightning}{\mathtt{if}\ B\ \mathtt{then}\ \mathbb{C}_1\ \mathtt{else}\ \mathbb{C}_2,\gamma,d,\sigma \Downarrow \lightning}$$

$$\frac{\mathcal{B}[\![B]\!]\sigma \text{ undefined}}{\mathtt{if}\ B\ \mathtt{then}\ \mathbb{C}_1\ \mathtt{else}\ \mathbb{C}_2,\gamma,d,\sigma \Downarrow \lightning}$$

$$\frac{\mathcal{B}[\![B]\!]\sigma = \mathsf{true} \qquad \mathbb{C} \;;\; \mathtt{while}\ B\ \mathtt{do}\ \mathbb{C},\gamma,d,\sigma \Downarrow \lightning}{\mathtt{while}\ B\ \mathtt{do}\ \mathbb{C},\gamma,d,\sigma \Downarrow \lightning} \qquad \frac{\mathcal{B}[\![B]\!]\sigma \text{ undefined}}{\mathtt{while}\ B\ \mathtt{do}\ \mathbb{C},\gamma,d,\sigma \Downarrow \lightning}$$

$$\frac{\mathbb{C}, [\mathtt{f}_1 \mapsto (\overrightarrow{\mathtt{x}_1},\mathbb{C}_1,\overrightarrow{\mathtt{r}_1}),...,\mathtt{f}_k \mapsto (\overrightarrow{\mathtt{x}_k},\mathbb{C}_k,\overrightarrow{\mathtt{r}_k})] : \gamma,d,\sigma \Downarrow \lightning}{\mathtt{procs}\ \overrightarrow{\mathtt{r}_1} := \mathtt{f}_1(\overrightarrow{\mathtt{x}_1})\{\mathbb{C}_1\},...,\overrightarrow{\mathtt{r}_k} := \mathtt{f}_k(\overrightarrow{\mathtt{x}_k})\{\mathbb{C}_k\}\ \mathtt{in}\ \mathbb{C},\gamma,d,\sigma \Downarrow \lightning}$$

$$\frac{\mathsf{lookup}(\mathtt{f},\gamma) \text{ undefined}}{\mathtt{call}\ \overrightarrow{\mathtt{r}} := \mathtt{f}(\overrightarrow{E}),\gamma,d,\sigma \Downarrow \lightning}$$

$$\frac{|\overrightarrow{E}| = i \qquad |\overrightarrow{\mathtt{r}}| = j \qquad \mathsf{lookup}(\mathtt{f},\gamma) \notin ((\mathrm{V\scriptstyle AR}^i \times \mathcal{L}_{\mathrm{CMD}} \times \mathrm{V\scriptstyle AR}^j) \times \mathrm{PD\scriptstyle EF}^*)}{\mathtt{call}\ \overrightarrow{\mathtt{r}} := \mathtt{f}(\overrightarrow{E}),\gamma,d,\sigma \Downarrow \lightning}$$

$$\frac{1 \leq k \leq |\overrightarrow{E}| \qquad \mathcal{E}[\![E_k]\!]\sigma \text{ undefined}}{\mathtt{call}\ \overrightarrow{\mathtt{r}} := \mathtt{f}(\overrightarrow{E}),\gamma,d,\sigma \Downarrow \lightning}$$

$$\frac{\begin{array}{c}\mathsf{lookup}(\mathtt{f},\gamma) = ((\overrightarrow{\mathtt{x}},\mathbb{C},\overrightarrow{\mathtt{y}}),\gamma') \qquad \mathcal{E}[\![\overrightarrow{E}]\!]\sigma = \overrightarrow{v} \qquad |\overrightarrow{E}| = |\overrightarrow{\mathtt{x}}| \\ \mathbb{C},\gamma',d,\emptyset[\overrightarrow{\mathtt{y}} \mapsto \overrightarrow{w}][\overrightarrow{\mathtt{x}} \mapsto \overrightarrow{v}] \Downarrow \lightning\end{array}}{\mathtt{call}\ \overrightarrow{\mathtt{r}} := \mathtt{f}(\overrightarrow{E}),\gamma,d,\sigma \Downarrow \lightning}$$

$$\frac{1 \leq k \leq |\overrightarrow{\mathtt{r}}| \qquad \mathtt{r}_k \notin \mathit{dom}(\sigma)}{\mathtt{call}\ \overrightarrow{\mathtt{r}} := \mathtt{f}(\overrightarrow{E}),\gamma,d,\sigma \Downarrow \lightning}$$

$$\frac{\mathbb{C},\gamma,d,\sigma[\mathtt{x} \mapsto v] \Downarrow \lightning}{\mathtt{local}\ \mathtt{x}\ \mathtt{in}\ \mathbb{C},\gamma,d,\sigma \Downarrow \lightning}$$

Figure 4.2: Operational semantics for $\mathcal{L}_{\mathrm{CMD}}$ (faulting cases).

Assertion Language

For simplicity, rather than working with satisfaction relations, as in §3.2, we instead choose for our logical assertions to describe sets of program states, similar to the practice of Calcagno, O'Hearn and Yang [17]. We call such assertions 'predicates' and interpret them over a generalised logical environment $e \in \textsc{Env}$ that maps logical variables, including label variables $(\alpha, \beta, \gamma ...)$, to their values. The definition of our predicates, and their semantics, is parametric on the choice of multi-holed context algebra $\mathcal{M} = (\mathcal{C}, \mathcal{X}, fh, \bullet)$ and context formula $P_{\mathcal{C}}$. It is necessary for these context formulae to at least include the assertion α which describes a hole label with the value $e(\alpha)$.

Definition 4.7 (Predicates)**.** The set of *state predicates* $\textsc{Pred}$, ranged over by $P, Q, R, P', P_1, ...$, is defined inductively as:

$$\begin{array}{lll} P ::= & P \Rightarrow P \mid \mathsf{false} & \textit{Classical Assertions} \\ & \mid \alpha{\leftarrow}P_{\mathcal{C}} & \textit{Segment Specific Assertions} \\ & \mid \mathsf{emp} \mid \mathtt{x} \Rrightarrow v \mid P * P \mid \alpha \circledR P \mid P {-\!\!*} P \mid P \oslash \alpha & \textit{Structural Assertions} \\ & \mid \exists v.\, P \mid \text{И}\alpha.\, P & \textit{Quantification} \end{array}$$

where $\alpha \in \textsc{LVar}_{\mathcal{X}}$ the set of of logical label variables, $\mathtt{x} \in \textsc{Var}$ the set of program variables and $v \in \textsc{LVar}_{\textsc{Val}}$ the set of logical value variables.

Definition 4.8 (Predicate Semantics)**.** The semantics of predicates is given by the function $\mathcal{P}[\![(\cdot)]\!] : \textsc{Pred} \to (\textsc{Env} \to \mathcal{P}(\textsc{State}))$ which is defined as:

$$\begin{array}{rcl} \mathcal{P}[\![P \Rightarrow Q]\!]e & \overset{\text{def}}{=} & \{(s, \sigma) \mid (s, \sigma) \in \mathcal{P}[\![P]\!]e \cap \mathcal{P}[\![Q]\!]e \text{ or } (s, \sigma) \notin \mathcal{P}[\![P]\!]e\} \\ \mathcal{P}[\![\mathsf{false}]\!]e & \overset{\text{def}}{=} & \emptyset \\ \mathcal{P}[\![\alpha{\leftarrow}P_{\mathcal{C}}]\!]e & \overset{\text{def}}{=} & \{(x{\leftarrow}c, \emptyset) \mid e(\alpha) = x \text{ and } e, c \vDash_{\mathcal{C}} P_{\mathcal{C}}\} \\ \mathcal{P}[\![\mathsf{emp}]\!]e & \overset{\text{def}}{=} & \{(\mathsf{emp}, \emptyset)\} \\ \mathcal{P}[\![\mathtt{x} \Rrightarrow v]\!]e & \overset{\text{def}}{=} & \{(\mathsf{emp}, \sigma) \mid dom(\sigma) = \{x\} \text{ and } \sigma(x) = e(v)\} \\ \mathcal{P}[\![P * Q]\!]e & \overset{\text{def}}{=} & \{(s_1 + s_2, \sigma_1 \uplus \sigma_2) \mid (s_1, \sigma_1) \in \mathcal{P}[\![P]\!]e \text{ and } (s_2, \sigma_2) \in \mathcal{P}[\![Q]\!]e\} \\ \mathcal{P}[\![\alpha \circledR P]\!]e & \overset{\text{def}}{=} & \{((x)(s), \sigma) \mid e(\alpha) = x \text{ and } (s, \sigma) \in \mathcal{P}[\![P]\!]e\} \\ \mathcal{P}[\![P {-\!\!*} Q]\!]e & \overset{\text{def}}{=} & \{(s, \sigma) \mid (s', \sigma') \in \mathcal{P}[\![P]\!]e \text{ and } (s + s', \sigma \uplus \sigma') \in \mathcal{P}[\![Q]\!]e\} \\ \mathcal{P}[\![P \oslash \alpha]\!]e & \overset{\text{def}}{=} & \{(s, \sigma) \mid e(\alpha) = x \text{ and } ((x)(s), \sigma) \in \mathcal{P}[\![P]\!]e\} \\ \mathcal{P}[\![\exists v.\, P]\!]e & \overset{\text{def}}{=} & \{(s, \sigma) \mid (s, \sigma) \in \mathcal{P}[\![P]\!]e[v \mapsto u] \text{ and } u \in \textsc{Val}\} \\ \mathcal{P}[\![\text{И}\alpha.\, P]\!]e & \overset{\text{def}}{=} & \{(s, \sigma) \mid (s, \sigma) \in \mathcal{P}[\![P]\!]e[\alpha \mapsto x] \text{ and } x \# e, s \text{ and } x \in \mathrm{X}\} \end{array}$$

As with segment logic for trees, we can derive the standard classical connectives $\neg P$, true, $P \vee Q$, $P \wedge Q$ and $\forall v.\, P$, from false, $\Rightarrow$ and $\exists$. We denote an arbitrary variable store with the assertion σ (punning the variable store syntax) defined as:

$$\sigma \quad ::= \quad \mathsf{emp} \mid \mathtt{x} \Rrightarrow v \mid \sigma * \sigma$$

We also derive the hidden label quantification and several notational short-hands as follows:

$$\begin{aligned}
\mathtt{x} \Rrightarrow - &\overset{\text{def}}{=} \exists v.\, \mathtt{x} \Rrightarrow v \\
\mathsf{H}\alpha.\, P &\overset{\text{def}}{=} \text{И}\alpha.\, \alpha \circledR P \\
\text{И}\alpha, \beta.\, P &\overset{\text{def}}{=} \text{И}\alpha.\, (\text{И}\beta.\, P) \\
\mathsf{H}\alpha, \beta.\, P &\overset{\text{def}}{=} \mathsf{H}\alpha.\, (\mathsf{H}\beta.\, P) \\
\alpha, \beta \circledR P &\overset{\text{def}}{=} \alpha \circledR (\beta \circledR P) \\
P \oslash \alpha, \beta &\overset{\text{def}}{=} (P \oslash \alpha) \oslash \beta
\end{aligned}$$

The binding convention of our assertions, from strongest to weakest, is given by:

$$\neg,\ \leftarrow,\ \circledR,\ *\ ,\ \wedge\ ,\ \vee\ ,\ \oslash,\ -\!\!*\ ,\ \Rightarrow\ ,\ \Leftrightarrow\ ,\ \text{И},\ \forall,\ \exists.$$

From the semantics of our predicates and the properties of the segment algebra $\mathcal{S}(\mathcal{M}) = (S, \mathsf{emp}, \leftarrow, fa, fh, +, \mathsf{comp})$ we have a number of equivalences that we make use of in our reasoning framework. All of the standard classical equivalences hold. The associativity and commutativity of $+$ with identity emp gives rise to a number of logical equivalences that are analogous to those of separation logic:

$$\begin{aligned}
P * \mathsf{emp} &\Leftrightarrow P \\
P * Q &\Leftrightarrow Q * P \\
P * (Q * R) &\Leftrightarrow (P * Q) * R \\
(P \vee Q) * R &\Leftrightarrow (P * R) \vee (Q * R) \\
(P \wedge Q) * R &\Rightarrow (P * R) \wedge (Q * R)
\end{aligned}$$

The last property only holds in one direction as the state described by R is not necessarily the same in the assertions $P * R$ and $Q * R$. The definition of $-\!\!*$ is also analogous to that of separation logic, and so leads to the following equivalence:

$$P * (P -\!\!* Q) \quad \Leftrightarrow \quad Q$$

The properties of compression from Definition 3.43 give rise to the following equivalences, analogous to those from ambient logic [20]:

$$\begin{array}{rcll}
\alpha \circledR \mathsf{emp} & \Leftrightarrow & \mathsf{emp} & \\
\alpha \circledR (\beta \circledR P) & \Leftrightarrow & \beta \circledR (\alpha \circledR P) & \\
\alpha \circledR P & \Leftrightarrow & \beta \circledR P[\beta/\alpha] & \text{if } \beta \notin \mathit{free}(P) \\
\alpha \circledR (P * Q) & \Leftrightarrow & \alpha \circledR (P) * Q & \text{if } \forall \beta .\, \beta \in \mathit{free}(Q) \Rightarrow \alpha \neq \beta \\
\alpha \circledR (\beta \leftarrow P_{\mathcal{C}} * \alpha \leftarrow Q_{\mathcal{C}}) & \Rightarrow & \beta \leftarrow P_{\mathcal{C}}[Q_{\mathcal{C}}/\alpha] & \text{if } \alpha \in \mathit{free}(P_{\mathcal{C}}) \\
\mathsf{H}\alpha .\, (\beta \leftarrow P_{\mathcal{C}} * \alpha \leftarrow Q_{\mathcal{C}}) & \Leftrightarrow & \beta \leftarrow P_{\mathcal{C}}[Q_{\mathcal{C}}/\alpha] & \text{if } \alpha \in \mathit{free}(P_{\mathcal{C}})
\end{array}$$

Notice that in the penultimate case, where the label α is not certainly fresh, that the property is only one way. When we collapse a segment we are forgetting about a label. However, when we expand a segment we introduce a new label and we must ensure that this label does not clash with any existing labels. Thus, the property can only be an equivalence if the label α is known to be fresh.

The revelation connective $\circledR$ has a right adjoint $\oslash$, just as in §sec:SLtrees. This leads to the following equivalence, analogous to that of ambient logic:

$$\alpha \circledR (P \oslash \alpha) \quad \equiv \quad P$$

Finally, the properties of address and hole label uniqueness result in the following equivalences:

$$\begin{array}{rcll}
\alpha \leftarrow P_{\mathcal{C}} * \alpha \leftarrow Q_{\mathcal{C}} & \equiv & \mathsf{false} & \\
\alpha \leftarrow P_{\mathcal{C}} * \beta \leftarrow Q_{\mathcal{C}} & \equiv & \mathsf{false} & \text{if } \mathit{free}(P_{\mathcal{C}}) \cap \mathit{free}(Q_{\mathcal{C}}) \neq \emptyset
\end{array}$$

Hoare Reasoning

We now introduce our Hoare reasoning framework. The judgements of our proof system make assertions about the program state and have the form $e, \Gamma \vdash \{P\}\ \mathbb{C}\ \{Q\}$, where $P, Q \in \textsc{Pred}$ are predicates, $\mathbb{C} \in \mathcal{L}_{\text{CMD}}$ is a program, $e \in \textsc{Env}$ is a logical environment and Γ is a *procedure specification environment.* A procedure specification environment associates procedure names with pairs of pre- and postconditions (parameterised by the arguments and return values of the procedure respectively). The interpretation of judgements is that, in environment e, in the presence of procedures satisfying Γ, when executed from a state satisfying P, the program $\mathbb{C}$ will either diverge or terminate in a state satisfying Q.

When we define a procedure in our framework, we introduce a set of specifications for that procedure, which the procedure body must satisfy. These specifications are then used to determine the behaviour of calls to that procedure.

Definition 4.9 (Procedure Specifications)**.** A *procedure specification* $\mathtt{f} : \mathsf{P} \rightarrowtail \mathsf{Q}$ consists of:

- ◇ a procedure name $\mathtt{f} \in \text{PName}$;
- ◇ a parameterised precondition $\mathsf{P} : \text{Val}^i \to (\text{Env} \to \mathcal{P}(S_{\text{Store}}))$;
- ◇ a parameterised postcondition $\mathsf{Q} : \text{Val}^j \to (\text{Env} \to \mathcal{P}(S_{\text{Store}}))$;

where $i = |\overrightarrow{\mathtt{x}}|$ is the number of input values of $\mathtt{f}$ and $j = |\overrightarrow{\mathtt{r}}|$ is the number of return values of $\mathtt{f}$. The set of procedure specifications is denoted PSpec.

In a procedure specification, the precondition is parameterised by the arguments with which the procedure is called, whilst the postcondition is parameterised by the return values of the procedure. The number of parameters and return values used when the procedure is called must match the number expected by the specification when the procedure is defined, otherwise the program will fault. We do not allow procedures to access variables outside of their own scope (we do not provide global variables) so the pre- and post-conditions of a procedure specification are given as predicates over just the segment algebra part of the program state.

Definition 4.10 (Procedure Specification Environments)**.** A *procedure specification environment* $\Gamma \in \mathcal{P}(\text{PSpec})$ is a set of procedure specifications. The set of procedure specification environments is denoted PSEnv.

Notation: In our proof judgements, we write Γ, Γ' to stand for the set union $\Gamma \cup \Gamma'$.

To simplify the presentation of our inference rules we define a predicate-valued semantics for boolean expressions. This semantics interprets a boolean expression as a predicate describing the set of states in which that boolean expression holds.

Definition 4.11 (Predicate-Valued Semantics of Boolean Expressions)**.** The *predicate-valued semantics of Boolean expressions* $\mathcal{P}[\![(\cdot)]\!] : \text{BExpr} \to (\text{Env} \to \mathcal{P}(\text{State}))$ is defined by:

$$\mathcal{P}[\![\mathsf{B}]\!]e \overset{\text{def}}{=} \{(s, \sigma) \mid \mathcal{B}[\![\mathsf{B}]\!]\sigma = \mathsf{true}\}$$

Since the semantics of expressions is partial, it is also convenient to define safety predicates, which simply assert that the state permits the evaluation of a given expression.

Definition 4.12 (Safety Predicates)**.** Given a value expression $\mathtt{E} \in \textsc{Expr}$, the *expression safety predicate for* $\mathtt{E}$, denoted $\mathsf{vsafe}(\mathtt{E})$, is defined as:

$$\mathsf{vsafe}(\mathtt{E}) \stackrel{\text{def}}{=} \{(s, \sigma) \mid \mathcal{E}[\![\mathtt{E}]\!]\sigma \text{ is defined}\}$$

Similarly, given a Boolean expression $\mathtt{B} \in \textsc{BExpr}$, the *expression safety predicate for* $\mathtt{B}$, denoted $\mathsf{bsafe}(\mathtt{B})$, is defined as:

$$\mathsf{bsafe}(\mathtt{B}) \stackrel{\text{def}}{=} \{(s, \sigma) \mid \mathcal{B}[\![\mathtt{B}]\!]\sigma \text{ is defined}\}$$

Finally, in order to define the axiomatic semantics, we need axioms for the basic commands of the language. We often have just one axiom for each basic command, but some basic commands have multiple axioms that describe disjoint cases of the command. For this reason we have a set of axioms for each basic command.

Assumption 4 (Axioms for Basic Commands)**.** Assume a set of axioms for the basic commands,

$$\text{Ax}[\![(\cdot)]\!] : \textsc{Cmd} \to \mathcal{P}_{\mathsf{fin}}(\textsc{Pred} \times \textsc{Pred}).$$

Definition 4.13 (Inference Rules)**.** The Hoare Logic Rules for $\mathcal{L}_{\text{CMD}}$ are given in Figure 4.3 and Figure 4.4.

The axiom rule (Axiom) allows us to use the specifications given for our basic commands in Assumption 4.

The separating frame rule (Sep Frame) is analogous to the frame rule from separation logic [53] and embodies the basic principle of local reasoning: if a program runs without faulting on some state, then we can extend that state with additional, disjoint state as long as it is not affected by the current program. In order for the separation frame rule to work the precondition must include all of the state that is accessed while running the program, otherwise adding additional state may change the program's behaviour. Our treatment of variables as resource removes the requirement for a side-condition on the separation frame rule. This is because, with variables as resource, the variables in the frame are automatically disjoint from the variables used in the program. In order for $e, \Gamma \vdash \{P\}\, \mathbb{C}\, \{Q\}$ to hold, P must include assertions about every variable that occurs free in $\mathbb{C}$. The separation frame rule can be used to add assertions about program variables that have the same name as locally scoped variables within the program. However, since their scopes are different, the variables themselves are also considered to be different, so this does not cause any problems.

$$\textsc{Axiom}: \quad \frac{(P, Q) \in \mathrm{Ax}[\![\varphi]\!]}{e, \Gamma \vdash \{\ P\ \}\ \varphi\ \{\ Q\ \}}$$

$$\textsc{Sep Frame}: \quad \frac{e, \Gamma \vdash \{\ P\ \}\ \mathbb{C}\ \{\ Q\ \}}{e, \Gamma \vdash \{\ P * R\ \}\ \mathbb{C}\ \{\ Q * R\ \}}$$

$$\textsc{Rev Frame}: \quad \frac{e, \Gamma \vdash \{\ P\ \}\ \mathbb{C}\ \{\ Q\ \}}{e, \Gamma \vdash \{\ \alpha \circledR P\ \}\ \mathbb{C}\ \{\ \alpha \circledR Q\ \}}$$

$$\textsc{Cons}: \quad \frac{\mathcal{P}[\![P']\!]e \subseteq \mathcal{P}[\![P]\!]e \quad e, \Gamma \vdash \{\ P\ \}\ \mathbb{C}\ \{\ Q\ \} \quad \mathcal{P}[\![Q]\!]e \subseteq \mathcal{P}[\![Q']\!]}{e, \Gamma \vdash \{\ P'\ \}\ \mathbb{C}\ \{\ Q'\ \}}$$

$$\textsc{Disj}: \quad \frac{\text{for all } i \in I.\, e, \Gamma \vdash \{\ P_i\ \}\ \mathbb{C}\ \{\ Q_i\ \}}{e, \Gamma \vdash \{\ \bigvee_{i \in I} P_i\ \}\ \mathbb{C}\ \{\ \bigvee_{i \in I} Q_i\ \}}$$

$$\textsc{Exsts}: \quad \frac{\text{there exists } u \in \textsc{Val}.\, e[v \mapsto u], \Gamma \vdash \{\ P\ \}\ \mathbb{C}\ \{\ Q\ \}}{e, \Gamma \vdash \{\ \exists v.\, P\ \}\ \mathbb{C}\ \{\ \exists v.\, Q\ \}}$$

$$\textsc{Fresh}: \quad \frac{\text{there exists fresh } x \in \mathrm{X}.\, e[\alpha \mapsto x], \Gamma \vdash \{\ P\ \}\ \mathbb{C}\ \{\ Q\ \}}{e, \Gamma \vdash \{\ \text{И}\alpha.\, P\ \}\ \mathbb{C}\ \{\ \text{И}\alpha.\, Q\ \}}$$

Figure 4.3: Generic reasoning rules for $\mathcal{L}_{\text{CMD}}$.

The revelation rule (Rev Frame) can also be viewed as a frame rule. This is because revealing a label in the data structure does not change the behaviour of a program over that structure, it simply changes our view of the program state. The revelation corresponds to compression of a label at the model level. This either takes a segment and roots it (cutting off its addresses label) or it compresses together two pieces of the segment. It does not add or remove any program state, so the behaviour of the program can not change.

The consequence rule (Cons), disjunction rule (Disj), existential quantification rule (Exsts), freshness quantification rule (Fresh), skip rule (Skip) and sequencing rule (Seq) are all standard.

The if statement rule (If) requires a precondition from which we can derive the precondition of the first branch when the expression B evaluates to true and for the second branch when the expression B evaluates to false. The condition $\mathcal{P}[\![P]\!]e \subseteq \mathsf{bsafe}(B)$ ensures that the expression B can be evaluated without the program faulting.

The while statement rule (While) requires us to prove that P is a loop invariant. This means that the loop body reestablishes P when run from P in a state where the expression B evaluates to true. If P holds before the loop starts, then it will

$$\textsc{Skip}: \quad \frac{}{e,\Gamma \vdash \{\ \mathsf{emp}\ \}\ \mathtt{skip}\ \{\ \mathsf{emp}\ \}}$$

$$\textsc{Seq}: \quad \frac{e,\Gamma \vdash \{\ P\ \}\ \mathbb{C}_1\ \{\ R\ \} \quad e,\Gamma \vdash \{\ R\ \}\ \mathbb{C}_2\ \{\ Q\ \}}{e,\Gamma \vdash \{\ P\ \}\ \mathbb{C}_1;\mathbb{C}_2\ \{\ Q\ \}}$$

$$\textsc{If}: \quad \frac{\mathcal{P}[\![P]\!]e \subseteq \mathsf{bsafe}(\mathtt{B}) \quad \begin{array}{c} e,\Gamma \vdash \{\ P \wedge \mathcal{P}[\![\mathtt{B}]\!]\ \}\ \mathbb{C}_1\ \{\ Q\ \} \\ e,\Gamma \vdash \{\ P \wedge \neg\mathcal{P}[\![\mathtt{B}]\!]\ \}\ \mathbb{C}_2\ \{\ Q\ \} \end{array}}{e,\Gamma \vdash \{\ P\ \}\ \mathtt{if}\ \mathtt{B}\ \mathtt{then}\ \mathbb{C}_1\ \mathtt{else}\ \mathbb{C}_2\ \{\ Q\ \}}$$

$$\textsc{While}: \quad \frac{\mathcal{P}[\![P]\!]e \subseteq \mathsf{bsafe}(\mathtt{B}) \quad e,\Gamma \vdash \{\ P \wedge \mathcal{P}[\![\mathtt{B}]\!]\ \}\ \mathbb{C}\ \{\ P\ \}}{e,\Gamma \vdash \{\ P\ \}\ \mathtt{while}\ \mathtt{B}\ \mathtt{do}\ \mathbb{C}\ \{\ P \wedge \neg\mathcal{P}[\![\mathtt{B}]\!]\ \}}$$

$$\textsc{Assgn}: \quad \frac{\mathcal{P}[\![\mathtt{x} \Rightarrow v * \sigma]\!]e \subseteq \mathsf{vsafe}(\mathtt{E})}{e,\Gamma \vdash \{\ \mathtt{x} \Rightarrow v * \sigma\ \}\ \mathtt{x} := \mathtt{E}\ \{\ \mathtt{x} \Rightarrow \mathcal{E}[\![\mathtt{E}]\!]\sigma[\mathtt{x} \mapsto v] * \sigma\ \}}$$

$$\textsc{Local}: \quad \frac{\mathcal{P}[\![P]\!]e \cap \mathsf{vsafe}(\mathtt{x}) \equiv \emptyset \quad e,\Gamma \vdash \{\ \mathtt{x} \Rightarrow - * P\ \}\ \mathbb{C}\ \{\ \mathtt{x} \Rightarrow - * Q\ \}}{e,\Gamma \vdash \{\ P\ \}\ \mathtt{local}\ \mathtt{x}\ \mathtt{in}\ \mathbb{C}\ \{\ Q\ \}}$$

$$\textsc{PDef}: \quad \frac{\begin{array}{c} \forall(\mathtt{f}_i : \mathsf{P}_i \rightarrowtail \mathsf{Q}_i) \in \Gamma.\, e,\Gamma',\Gamma \vdash \begin{array}{c} \{\ \exists\overrightarrow{v_i}.\, \mathsf{P}_i(\overrightarrow{v_i}) * \overrightarrow{\mathtt{x}_i} \Rightarrow \overrightarrow{v_i} * \overrightarrow{\mathtt{r}_i} \Rightarrow -\ \} \\ \mathbb{C}_i \\ \{\ \exists\overrightarrow{w_i}.\, \mathsf{Q}_i(\overrightarrow{w_i}) * \overrightarrow{\mathtt{x}_i} \Rightarrow - * \overrightarrow{\mathtt{r}_i} \Rightarrow \overrightarrow{w_i}\ \} \end{array} \\ \text{for all } \mathtt{f} : \mathsf{P} \rightarrowtail \mathsf{Q} \in \Gamma, \text{ there exists } i \text{ s.t } \mathtt{f} = \mathtt{f}_i \\ \text{for all } \mathtt{f} : \mathsf{P} \rightarrowtail \mathsf{Q} \in \Gamma', \text{ for all } i, \mathtt{f} \neq \mathtt{f}_i \\ e,\Gamma',\Gamma \vdash \{\ P\ \}\ \mathbb{C}\ \{\ Q\ \} \end{array}}{\begin{array}{c} \{\ P\ \} \\ e,\Gamma' \vdash\ \mathtt{procs}\ \overrightarrow{\mathtt{r}_1} := \mathtt{f}_1(\overrightarrow{\mathtt{x}_1})\{\mathbb{C}_1\}, \ldots, \overrightarrow{\mathtt{r}_k} := \mathtt{f}_k(\overrightarrow{\mathtt{x}_k})\{\mathbb{C}_k\}\ \mathtt{in}\ \mathbb{C} \\ \{\ Q\ \} \end{array}}$$

$$\textsc{PCall}: \quad \frac{\mathcal{P}[\![\overrightarrow{\mathtt{r}} \Rightarrow \overrightarrow{v} * \sigma]\!]e \subseteq \mathsf{vsafe}(\overrightarrow{\mathtt{E}})}{e,\Gamma,(\mathtt{f} : \mathsf{P} \rightarrowtail \mathsf{Q}) \vdash \begin{array}{c} \left\{\ \mathsf{P}\left(\mathcal{E}[\![\overrightarrow{\mathtt{E}}]\!]\sigma[\overrightarrow{\mathtt{r}} \mapsto \overrightarrow{v}]\right) * \overrightarrow{\mathtt{r}} \Rightarrow \overrightarrow{v} * \sigma\ \right\} \\ \mathtt{call}\ \overrightarrow{\mathtt{r}} := \mathtt{f}(\overrightarrow{\mathtt{E}}) \\ \{\ \exists\overrightarrow{w}.\, \mathsf{Q}\,(\overrightarrow{w}) * \overrightarrow{\mathtt{r}} \Rightarrow \overrightarrow{w} * \sigma\ \} \end{array}}$$

$$\textsc{PWeak}: \quad \frac{e,\Gamma \vdash \{\ P\ \}\ \mathbb{C}\ \{\ Q\ \}}{e,\Gamma,\Gamma' \vdash \{\ P\ \}\ \mathbb{C}\ \{\ Q\ \}}$$

Figure 4.4: Language specific reasoning rules for $\mathcal{L}_{\text{CMD}}$.

also hold on termination of the loop and the expression $\mathtt{B}$ must then evaluate to false. As with the if statement rule, the condition $\mathcal{P}[\![P]\!]e \subseteq \mathsf{bsafe}(\mathtt{B})$ ensures that the expression $\mathtt{B}$ can be evaluated without the program faulting.

The assignment rule (Assgn) requires that the target variable is in scope and that it is safe to evaluate the expression $\mathtt{E}$ in the current state. In the postcondition, the target variable is updated so that its value is now that of the evaluated expression and the rest of the state is left unchanged. The evaluation of the expression may depend on other variables in the store, but the vsafe condition ensures that the expression can be evaluated without the program faulting.

The local variable rule (Local) allows us to declare local variables in a program. Recall that the predicate P is evaluated to a set of segment-store pairs (s, σ). The predicate $\mathtt{x} \Rrightarrow -$ can only be evaluated to the pair $(\mathsf{emp}, \emptyset[\mathtt{x} \mapsto v])$ for some choice of v, the initial value of x. We can only extend the predicate P with this predicate if the variable x is not already in P, which is ensured by the condition $\mathcal{P}[\![P]\!]e \cap \mathsf{vsafe}(\mathtt{x}) \equiv \emptyset$. However, it is possible for the variable x to already be in scope, in which case the separating frame rule (Sep Frame) can be used to frame off this variable before we apply the local variable rule. The outer scoped x then has no effect on the inner scoped x and its value will be unchanged after the inner scope is closed.

The procedure definition rule (PDef) uses the procedure specifications Γ to specify a set of procedures. Each procedure specification for $\mathtt{f}_i$ gives it a parameterised precondition P_i and postcondition Q_i. For each specification, the corresponding procedure body must, for each instantiation of the parameters $\overrightarrow{\mathtt{x}_i}$ with arguments $\overrightarrow{v_i}$, take a state with segment $\mathsf{P}_i(\overrightarrow{v_i})$ to one with segment $\mathsf{Q}_i(\overrightarrow{w_i})$ and return variables $\overrightarrow{\mathtt{r}_i}$ holding values $\overrightarrow{w_i}$. The procedure bodies are verified using the procedure specifications in scope, as well as their own procedure specifications, making it possible to verify mutually recursive procedure definitions. The procedure specification environment Γ must only specify the procedures that are defined in the procs block under consideration, and these procedures must have different names to any that occur in the existing procedure specification environment Γ'. To deal with procedures that redefine existing procedures we have to use the procedure weakening rule (PWeak) to forget the old specification.

The procedure call rule (PCall) allows us to reason about procedure calls. The arguments for the procedure call are obtained by evaluating the expressions $\overrightarrow{\mathtt{E}}$ and the vsafe condition ensures that these expressions can be evaluated without the program faulting. The precondition required by the procedure's specification must hold initially, and afterwards its postcondition holds for the values returned in the result variables $\overrightarrow{\mathtt{r}}$.

The procedure weakening rule (PWeak) allows for the procedure specification environment to be weakened (i.e. more procedure specifications can be added). This rule can also be used in conjunction with the procedure definition rule (PDef) to redefine an existing procedure in an inner procedure scope.

The Conjunction Rule

You may have noticed that the conjunction rule (Conj) is absent from our reasoning framework's inference rules given in Figure 4.3 and Figure 4.4.

$$\textsc{Conj}: \frac{\text{for all } i \in I, e, \Gamma \vdash \left\{\ P_i\ \right\}\ \mathbb{C}\ \left\{\ Q_i\ \right\}}{e, \Gamma \vdash \left\{\ \bigwedge_{i\in I} P_i\ \right\}\ \mathbb{C}\ \left\{\ \bigwedge_{i\in I} Q_i\ \right\}}$$

Sometimes the conjunction rule is admissible, but in general this is not always the case. Consider a command `allocEither`() that operates on a double-heap data store $\mathcal{S}(\mathcal{M}_{\text{Store}}) = \mathcal{S}(\mathcal{M}_{\mathbb{H}}) \times \mathcal{S}(\mathcal{M}_{\mathbb{H}})$ and allocates a single cell in either of the two heaps. There are two specifications for this command:

$$\{\mathtt{x} \Rrightarrow -\} \quad \mathtt{x} := \mathtt{allocEither}() \quad \{\exists y.\, (y \mapsto - \times \mathsf{emp}) * \mathtt{x} \Rrightarrow y\}$$
$$\{\mathtt{x} \Rrightarrow -\} \quad \mathtt{x} := \mathtt{allocEither}() \quad \{\exists y.\, (\mathsf{emp} \times y \mapsto -) * \mathtt{x} \Rrightarrow y\}$$

The choice of which heap the new cell is allocated in is not made by the implementation, but is instead made at the discretion of the prover. This command exhibits *angelic nondeterminism*: that is, it is possible to prove both that the command allocates the cell in the left heap and that the command allocates the cell in the right heap.

This is somewhat paradoxical, as the program cannot hope to guess which heap the prover chooses to allocate the new cell in. However, it is possible to resolve this paradox since, from the program's perspective, the two cases are the same and the distinction is only a logical abstraction.

The problem is that the conjunction rule is not compatible with angelic nondeterminism. If we have the conjunction rule then we can construct the following derivation for the `allocEither`() command:

$$\dfrac{\dfrac{}{\begin{array}{c}\{\mathtt{x} \Rrightarrow -\}\\ \mathtt{x} := \mathtt{allocEither}()\\ \{\exists y.\, (y \mapsto - \times \mathsf{emp}) * \mathtt{x} \Rrightarrow y\}\end{array}}\textsc{Ax} \qquad \dfrac{}{\begin{array}{c}\{\mathtt{x} \Rrightarrow -\}\\ \mathtt{x} := \mathtt{allocEither}()\\ \{\exists y.\, (\mathsf{emp} \times y \mapsto -) * \mathtt{x} \Rrightarrow y\}\end{array}}\textsc{Ax}}{\{\mathtt{x} \Rrightarrow -\}\ \mathtt{x} := \mathtt{allocEither}()\ \{\mathsf{false}\}}\textsc{Conj}$$

This derivation tells us that the `allocEither`() command must diverge, which it clearly does not. Without the conjunction rule we cannot come to the same conclusion.

The following two conditions on basic commands $\varphi \in \textsc{Cmd}$ are sufficient to establish that the commands do not exhibit angelic nondeterminism:

- for all $(P, Q), (P', Q') \in \mathrm{Ax}[\![\varphi]\!]$ with $(P, Q) \neq (P', Q')$, $P \wedge P' \Leftrightarrow \mathsf{false}$; and
- the predicate $\bigvee\{P \mid (P, Q) \in \mathrm{Ax}[\![\varphi]\!]\}$ is precise.

A segment logic predicate P is precise if, for every $e \in \textsc{Env}$ and $(s, \sigma) \in \textsc{State}$ there is at most one $(s', \sigma') \in \textsc{State}$ such that $(s', \sigma') \in \mathcal{P}[\![P]\!]e$, $s = (\bar{x})(s_0 + s')$ and $\sigma = \sigma_o \uplus \sigma'$ for some $\bar{x} \subseteq \mathcal{X}$, $s_0 \in \mathrm{S}$ and $\sigma_0 \in \Sigma$.

Together, these conditions imply that at most one axiom describes the behaviour of the command from any given state, and hence the conjunction rule cannot be used to derive a stronger postcondition for any of the basic commands. These conditions hold for all of the basic module commands we consider in this thesis and none of our program constructs introduce angelic nondeterminism.

The conjunction rule is, therefore, admissible for all of the abstract modules we consider. Its omission is then justified since nothing would be gained from its inclusion in our set of inference rule.

4.3 Soundness

In §4.2.1 we defined an operational semantics defined for compete data structures. In §4.2.2 we gave an axiomatic semantics defined over segments of a data structure. In order to prove that our axiomatic semantics is sound with respect to our operational semantics we need to be able to relate segments of a data structure to complete data structures. Recall that, in the multi-holed context setting, complete data is treated as a context that contains no context holes. We can relate segments to complete data in a similar way. Complete data can be treated as a segment that contains no holes, is fully compressed (i.e. consists of just one piece) and has its address label restricted (so it cannot be extended).

Definition 4.14 (Segment/Complete Data Relation)**.** Given a segment algebra $\mathcal{S}(\mathcal{M}) = (S, \mathsf{emp}, \leftarrow, fa, fh, +, \mathsf{comp})$ parametrised by a multi-holed context algebra $\mathcal{M} = (\mathcal{C}, \mathcal{X}, fh, \bullet)$ we define the complete data set $\mathcal{D}$ as:

$$\mathcal{D} \stackrel{\text{def}}{=} \{d \mid d \in \mathcal{C} \text{ and } fh(d) = \emptyset\}$$

and we define the relation $\simeq_S \in S \times \mathcal{D}$ as follows:

$$(x)(x \leftarrow d) \quad \simeq_S \quad d$$

where $x \in \mathcal{X}$ and $d \in \mathcal{D}$.

Note that under this definition we are not, in general, able to relate the segment $(x, y)(x \leftarrow d + y \leftarrow d')$ with any piece of complete data. This is because the disjoint union operator $+$ does not necessarily have an interpretation in the complete data structure. For example, in the tree model from §3.1.1 we can represent trees that are siblings, but have no notion disjoint trees.

We choose to interpret the behaviour of a program on a segment as the behaviour of that program on any complete data that is obtained by extending the segment.

Definition 4.15 (Segment Completion)**.** The *completion* of a segment-store pair (s, σ) is any program state $(d, \sigma') \in \mathcal{D} \times \Sigma$ such that there exists $\bar{x} \in \mathcal{P}_{\mathsf{fin}}(\mathcal{X})$, $s' \in \mathrm{S}$ and $\sigma_0 \in \Sigma$ with $(\bar{x})(s' + s) \simeq_S d$ and $\sigma' = \sigma \uplus \sigma_0$.

More formally, we introduce a *local Hoare triple* judgement $e, \Gamma \vDash \{P\}\, \mathbb{C}\, \{Q\}$ which holds in exactly this case.

Definition 4.16 (Local Hoare Triples)**.** Take an arbitrary segment algebra $\mathcal{S}(\mathcal{M}) = (S, \mathsf{emp}, \leftarrow, fa, fh, +, \mathsf{comp})$ parameterised by a multi-holed context algebra $\mathcal{M} = (\mathcal{C}, \mathcal{X}, fh, \bullet)$. Let $e \in \textsc{Env}$, $\gamma \in \textsc{PDef}^*$, $P, Q \in \mathcal{P}(\textsc{State})$, $\mathbb{C} \in \mathcal{L}_{\textsc{Cmd}}$ and $\Gamma \in \textsc{PSEnv}$.

$$\begin{array}{rcl}
e, \gamma \vDash \{P\}\, \mathbb{C}\, \{Q\} & \Leftrightarrow & \text{for all } (s, \sigma) \in \mathcal{P}[\![P]\!]e,\ o \in \textsc{Out},\ d_1 \in \mathcal{D},\ s_0 \in S,\ \sigma_0 \in \Sigma,\ \bar{x} \subseteq \mathrm{X} \\
& & \text{whenever } (\bar{x})(s_0 + s) \simeq_S d_1 \text{ and there exists } \sigma_1 \text{ s.t } \sigma_0 \uplus \sigma = \sigma_1 \\
& & \quad \Longrightarrow \\
& & \quad \mathbb{C}, \gamma, d_1, \sigma_1 \Downarrow o \implies o \neq \lightning \text{ and} \\
& & \quad \text{there exist } (s', \sigma') \in \mathcal{P}[\![Q]\!]e,\ d_2 \in \mathcal{D},\ \sigma_2 \in \Sigma \text{ s.t.} \\
& & \quad\quad o = (d_2, \sigma_2),\ (\bar{x})(s_0 + s') \simeq_S d_2 \text{ and } \sigma_0 \uplus \sigma' = \sigma_2 \\
e, \gamma \vDash \Gamma & \Leftrightarrow & \text{for all } (\mathtt{f} : \mathsf{P} \rightarrowtail \mathsf{Q}) \in \Gamma \\
& & \text{there exist } \overrightarrow{\mathtt{x}}, \overrightarrow{\mathtt{r}} \in \textsc{Var}^*,\ \mathbb{C} \in \mathcal{L}_{\textsc{Cmd}},\ \gamma' \in \textsc{PDef}^* \text{ s.t.} \\
& & \quad ((\overrightarrow{\mathtt{x}}, \mathbb{C}, \overrightarrow{\mathtt{r}}), \gamma') = \mathsf{lookup}(\mathtt{f}, \gamma) \text{ and} \\
& & \quad e, \gamma' \vDash \begin{array}{c} \left\{\ \exists \overrightarrow{v}.\, \mathsf{P}(\overrightarrow{v}) * \overrightarrow{\mathtt{x}} \Rightarrow \overrightarrow{v} * \overrightarrow{\mathtt{r}} \Rightarrow -\ \right\} \\ \mathbb{C} \\ \left\{\ \exists \overrightarrow{w}.\, \mathsf{Q}(\overrightarrow{w}) * \overrightarrow{\mathtt{x}} \Rightarrow - * \overrightarrow{\mathtt{r}} \Rightarrow \overrightarrow{w}\ \right\} \end{array} \\
e, \Gamma \vDash \{P\}\, \mathbb{C}\, \{Q\} & \Leftrightarrow & \text{for all } \gamma \in \textsc{PDef}^*,\ e, \gamma \vDash \Gamma \implies e, \gamma \vDash \{P\}\, \mathbb{C}\, \{Q\}
\end{array}$$

In this interpretation the informal meaning of $e, \Gamma \vDash \{P\}\ \mathbb{C}\ \{Q\}$ is that for every segment-store pair $(s, \sigma) \in \mathcal{P}[\![P]\!]e$ the program $\mathbb{C}$ will not fault when run in the context of procedures satisfying Γ on any program state d, σ_1 that is a completion of (s, σ) and, assuming the command terminates, the resulting program state (d', σ_2) will be a completion of a segment-store pair $(s', \sigma') \in \mathcal{P}[\![Q]\!]e$.

Our operational semantics assumes a semantic interpretation function for the basic commands. Our axiomatic semantics assumes a set of axioms for the basic commands. In order for our reasoning system to be sound, these assumed semantics must be compatible. We require that every basic command behaves operationally in the same way as described by its axioms.

Assumption 5 (Axiom Soundness). For all $e \in \textsc{Env}$, $\varphi \in \textsc{Cmd}$, $(P, Q) \in \textsc{Ax}[\![\varphi]\!]$, $(s, \sigma) \in \mathcal{P}[\![P]\!]e$, $d_1 \in \mathcal{D}$, $s_0 \in S$, $\sigma_0 \in \Sigma$ and $\bar{x} \subseteq X$, if $(\bar{x})(s_0 + s) \simeq_S d_1$ and there exists $\sigma_1 \in \Sigma$ such that $\sigma_0 \uplus \sigma = \sigma_1$ then $\mathcal{C}[\![\varphi]\!](d_1, \sigma_1)$ is defined and there exist $(s', \sigma') \in \mathcal{P}[\![Q]\!]e$, $d' \in \mathcal{D}$ and $\sigma_2 \in \Sigma$ such that $\mathcal{C}[\![\varphi]\!](d_1, \sigma_1) = (d_2, \sigma_2)$, $(\bar{x})(s_0 + s') \simeq_S d_2$ and $\sigma_0 \uplus \sigma' = \sigma_2$.

The above assumption also captures the property that every basic command must behave in a local fashion. That is, their behaviour does not change when run with additional program state and, moreover, they leave this additional state unchanged. This is essential for the soundness of the separation and revelation frame rules.

In order for the set of basic command axioms to be sound it is necessary that each axiom preserves the free addresses and free labels of a segment from its precondition to its postcondition. The axioms describe the behaviour of our basic commands and these are not aware of the segment structure, which exists only at the logic level. Therefore, the effects of the axioms should be limited to the data contained within the segments and should not modify the segment structure. As an example, consider a skip like command `foo` specified as:

$$\left\{\ \alpha \leftarrow \varnothing\ \right\}\ \ \texttt{foo()}\ \ \left\{\ \beta \leftarrow \varnothing\ \right\}$$

This specification may seem innocent enough, but if we apply the separation frame rule to add the frame $\beta \leftarrow \varnothing$, then we end up with the following derivation:

$$\dfrac{\dfrac{\left\{\ \alpha \leftarrow \varnothing\ \right\}\ \ \texttt{foo()}\ \ \left\{\ \beta \leftarrow \varnothing\ \right\}}{\left\{\ \alpha \leftarrow \varnothing * \beta \leftarrow \varnothing\ \right\}\ \ \texttt{foo()}\ \ \left\{\ \beta \leftarrow \varnothing * \beta \leftarrow \varnothing\ \right\}}\textsc{Sep Frame}}{\left\{\ \alpha \leftarrow \varnothing * \beta \leftarrow \varnothing\ \right\}\ \ \texttt{foo()}\ \ \left\{\ \mathsf{false}\ \right\}}\textsc{Cons}$$

This resulting specification can only be satisfied if the `foo` command were to diverge, which simply isn't the case. The issue here is not with the soundness of the frame rule, but with the definition of the axiom for the `foo` command. This axiom changes the segment identifier α to β which cause two problems. Firstly, we have no guarantee that the label β is not already in use in the wider segment and this could clash as in the example above. However, the wider segment could also contain an α hole that is expecting to be filled by the α addressed segment in the precondition. In the postcondition this segment no longer exists, so when α is later compressed we will have α occurring as just a segment hole, so the compression would be undefined. What we have seen here is that label preservation is part of the requirement that our basic command axioms describe some local behaviour (Assumption 5).

We also require that the semantics of expression evaluation behave locally.

Assumption 6 (Expression Locality)**.** For all value expressions $\mathtt{E} \in \textsc{Expr}$ and variable stores $\sigma, \sigma' \in \Sigma$ with $\mathcal{E}[\![\mathtt{E}]\!]\sigma$ and $\sigma \uplus \sigma'$ both defined, $\mathcal{E}[\![\mathtt{E}]\!](\sigma \uplus \sigma') = \mathcal{E}[\![\mathtt{E}]\!]\sigma$. Similarly, for all Boolean expressions $\mathtt{B} \in \text{B}\textsc{Expr}$ and variable stores $\sigma, \sigma' \in \Sigma$ with $\mathcal{B}[\![\mathtt{B}]\!]\sigma$ and $\sigma \uplus \sigma'$ both defined, $\mathcal{B}[\![\mathtt{B}]\!](\sigma \uplus \sigma') = \mathcal{B}[\![\mathtt{B}]\!]\sigma$.

In practice, this last assumption is trivial to check as most expression constructors are indifferent to the variable store. The only case that might be affected by the variable store is variable lookup, but treating variables as resource does not allow for an extension to the variable store to overwrite the value of any existing variables.

Theorem 4.17 (Soundness)**.** For all $e \in \textsc{Env}$, $\Gamma \in \text{PS}\textsc{Env}$, $P, Q \in \textsc{Pred}$ and $\mathbb{C} \in \mathcal{L}_{\textsc{Cmd}}$,

$$e, \Gamma \vdash \{P\}\,\mathbb{C}\,\{Q\} \quad \Longrightarrow \quad e, \Gamma \vDash \{P\}\,\mathbb{C}\,\{Q\}.$$

4.3.1 Proof of Soundness

Much of our soundness proof follows along similar lines as other soundness proofs of this kind. The cases for the majority of our inference rules are standard. There are two noticeable exceptions to this: the separating frame rule Sep Frame and the revelation frame rule Rev Frame. Due to the nature of our Hoare triple interpretation, the soundness of the rules follows almost by definition. In effect, when we reason about the behaviour of a program over a segment of the data structure, we are actually considering the behaviour of the program over all possible completions of this segment. When we apply either of the frame rules, we are simply reducing the space of possible completions that are now valid.

Proof of Theorem 4.17

The proof is by induction on the structure of the derivation of $e, \Gamma \vdash \{P\}\,\mathbb{C}\,\{Q\}$. In each case we consider the last inference rule applied.

Axiom case:

Fix $e \in \textsc{Env}$. In this case $\mathbb{C} = \varphi$ for some $\varphi \in \textsc{Cmd}$ and $(P, Q) \in Ax[\![\varphi]\!]$. Suppose that $e, \gamma \vDash \Gamma$, $(s, \sigma) \in \mathcal{P}[\![P]\!]e$, $o \in Out$, $d_1 \in \mathcal{D}$, $s_0 \in S$, $\sigma_0, \sigma_1 \in \Sigma$ and $\bar{x} \subseteq \mathrm{X}$ such that $d_1 \simeq_S (\bar{x})(s_0 + s)$, $\sigma_1 = \sigma_0 \uplus \sigma$ and $\varphi, \gamma, d_1, \sigma_1 \Downarrow o$. If $o = ↯$ then our operational semantics requires that $\mathcal{C}[\![\varphi]\!](d_1, \sigma_1)$ is undefined, which violates assumption 5 (Axiom Soundness). Thus $o = (d_2, \sigma_2)$ for some $(d_2, \sigma_2) \in \mathcal{C}[\![\varphi]\!](d_1, \sigma_1)$. Furthermore, Assumption 5 implies that $d_2 \simeq_S (\bar{x})(s_0 + s')$, $\sigma_2 = \sigma_0 \uplus \sigma'$ and $(s', \sigma') \in \mathcal{P}[\![Q]\!]e$, as required.

Sep Frame case:

Fix $e \in \textsc{Env}$. In this case $P = P' * R$ and $Q = Q' * R$ for some P', Q', R with $e, \Gamma \vDash \{P'\}\,\mathbb{C}\,\{Q'\}$ by the inductive hypothesis. Suppose that $e, \gamma \vDash \Gamma$ and $(s, \sigma) \in \mathcal{P}[\![P' * R]\!]e$. It follows that $(s, \sigma) = (s_p + s_r, \sigma_p \uplus \sigma_r)$ for some $(s_p, \sigma_p) \in \mathcal{P}[\![P']\!]e$ and $(s_r, \sigma_r) \in \mathcal{P}[\![R]\!]e$.

Now choose any $d_1 \in \mathcal{D}$, $s_0 \in S$, $\sigma_0, \sigma_1 \in \Sigma$ and $\bar{x} \subseteq \mathrm{X}$ with $d_1 \simeq_S (\bar{x})(s_0 + s) = (\bar{x})(s_0 + s_r + s_p)$ and $\sigma_1 = \sigma_0 \uplus \sigma_p$. Since $e, \Gamma \vDash \{P'\}\,\mathbb{C}\,\{Q'\}$ we know that $\mathbb{C}, \gamma, d_1, \sigma_1 \not\Downarrow ↯$. Moreover, $\mathbb{C}, \gamma, d_1, \sigma_1 \Downarrow d_2, \sigma_2$ for some $d_2 \in \mathcal{D}$ and $\sigma_2 \in \Sigma$ where $d_2 \simeq_S (\bar{x})(s_0 + s_r + s_q)$, $\sigma_2 = \sigma_0 \uplus \sigma_q$ and $(s_q, \sigma_q) \in \mathcal{P}[\![Q']\!]e$. Since $(s_r, \sigma_r) \in \mathcal{P}[\![R]\!]e$ it follows that $(s_q + s_r, \sigma_q \uplus \sigma_r) \in \mathcal{P}[\![Q' * R]\!]e$, as required.

Rev Frame case:

Fix $e \in \textsc{Env}$. In this case $P = \alpha \circledR P'$ and $Q = \alpha \circledR Q'$ for some P', Q', α with $e, \Gamma \vDash \{P'\}\,\mathbb{C}\,\{Q'\}$ by the inductive hypothesis. Suppose that $e, \gamma \vDash \Gamma$ and $(s, \sigma) \in \mathcal{P}[\![\alpha \circledR P']\!]e$. It follows that $(s, \sigma) = ((x)(s_p), \sigma)$ with $e(\alpha) = x$ and $(s_p, \sigma) \in \mathcal{P}[\![P']\!]e$.

Now choose any $d_1 \in \mathcal{D}$, $s_0 \in S$, $\sigma_0, \sigma_1 \in \Sigma$ and $\bar{y} \subseteq \mathrm{X}$ with $d_1 \simeq_S (\bar{y})(s_0 + (x)(s_p))$ and $\sigma_1 = \sigma_0 \uplus \sigma$. We have to be careful as x could be free in s_0. Choose x' fresh with respect to s_0 and s_p. Given the properties of a segment algebra (Definition 3.43) it follows that $d_1 \simeq_S (\bar{y})(s_0 + (x)(s_p)) = (\bar{y})(s_0 + (x')(s_p[x'/x])) = (x')(\bar{y})(s_0 + s_p[x'/x])$.

Since $e, \Gamma \vDash \{P'\}\,\mathbb{C}\,\{Q'\}$, and we do not allow our programs to manipulate abstract addressess or holes, we also know that $e[\alpha \mapsto x'], \Gamma \vDash \{P'\}\,\mathbb{C}\,\{Q'\}$. This means that $\mathbb{C}, \gamma, d_1, \sigma_1 \not\Downarrow ↯$. Moreover, $\mathbb{C}, \gamma, d_1, \sigma_1 \Downarrow d_2, \sigma_2$ for some $d_2 \in \mathcal{D}$ and $\sigma_2 \in \Sigma$ where $d_2 \simeq_S (x')(\bar{y})(s_0 + s_q[x'/x])$, $\sigma_2 = \sigma_0 \uplus \sigma_q$ and $(s_q, \sigma_q) \in \mathcal{P}[\![Q']\!]e$.

Now $(x')(\bar{y})(s_0 + s_q[x'/x]) = (\bar{y})(s_0 + (x')(s_q[x'/x])) = (\bar{y})(s_0 + (x)(s_q))$ and since $e(\alpha) = x$ it follows that $((x)(s_q), \sigma_q) \in \mathcal{P}[\![\alpha \circledR Q']\!]e$, as required.

Cons case:

Fix $e \in \text{Env}$. In this case $\mathcal{P}[\![P]\!]e \subseteq \mathcal{P}[\![P']\!]e$ and $\mathcal{P}[\![Q']\!]e \subseteq \mathcal{P}[\![Q]\!]e$ for some P', Q' with $e, \Gamma \vDash \{P'\}\,\mathbb{C}\,\{Q'\}$ by the induction hypothesis. Suppose that $e, \gamma \vDash \Gamma$ and that $(s, \sigma) \in \mathcal{P}[\![P]\!]e$. It follows that $(s, \sigma) \in \mathcal{P}[\![P']\!]e$ also, and since $e, \Gamma \vDash \{P'\}\,\mathbb{C}\,\{Q'\}$ we know that for all $d_1 \in \mathcal{D}$, $s_0 \in S$, $\sigma_0, \sigma_1 \in \Sigma$ and $\bar{x} \subseteq \mathrm{X}$ with $d_1 \simeq_S (\bar{x})(s_0 + s)$ and $\sigma_1 = \sigma_0 \uplus \sigma$ that $\mathbb{C}, \gamma, d_1, \sigma_1 \not\Downarrow \lightning$. Moreover, $\mathbb{C}, \gamma, d_1, \sigma_1 \Downarrow d_2, \sigma_2$ for some $d_2 \in \mathcal{D}$ and $\sigma_2 \in \Sigma$ where $d_2 \simeq_S (\bar{x})(s_0 + s')$, $\sigma_2 = \sigma_0 \uplus \sigma'$ and $(s', \sigma') \in \mathcal{P}[\![Q']\!]e$. Since $\mathcal{P}[\![Q']\!]e \subseteq \mathcal{P}[\![Q]\!]e$ it follows that $(s', \sigma') \in \mathcal{P}[\![Q]\!]e$, as required.

Disj case:

Fix $e \in \text{Env}$. In this case $P = \bigvee_{i \in I} P_i$ and $Q = \bigvee_{i \in I} Q_i$ for some P_i, Q_i with $e, \Gamma \vDash \{P_i\}\,\mathbb{C}\,\{Q_i\}$ for each $i \in I$, by the inductive hypothesis. Suppose that $e, \gamma \vDash \Gamma$ and that $(s, \sigma) \in \mathcal{P}[\![P]\!]e$. It follows that $(s, \sigma) \in \mathcal{P}[\![P_j]\!]e$ for some $j \in I$.

Since $e, \Gamma \vDash \{P_j\}\,\mathbb{C}\,\{Q_j\}$ we know that for all $d_1 \in \mathcal{D}$, $s_0 \in S$, $\sigma_0, \sigma_1 \in \Sigma$ and $\bar{x} \subseteq \mathrm{X}$ with $d_1 \simeq_S (\bar{x})(s_0 + s)$ and $\sigma_1 = \sigma_0 \uplus \sigma$ that $\mathbb{C}, \gamma, d_1, \sigma_1 \not\Downarrow \lightning$. Moreover, $\mathbb{C}, \gamma, d_1, \sigma_1 \Downarrow d_2, \sigma_2$ for some $d_2 \in \mathcal{D}$ and $\sigma_2 \in \Sigma$ where $d_2 \simeq_S (\bar{x})(s_0 + s')$, $\sigma_2 = \sigma_0 \uplus \sigma'$ and $(s', \sigma') \in \mathcal{P}[\![Q_j]\!]e$. Since $\mathcal{P}[\![Q_j]\!]e \subseteq \mathcal{P}[\![Q]\!]e$ it follows that $(s', \sigma') \in \mathcal{P}[\![Q]\!]e$, as required.

Exsts case:

Fix $e \in \text{Env}$. In this case $P = \exists v.\, P'$ and $Q = \exists v.\, Q'$ for some P', Q' with $e[v \mapsto u], \Gamma \vDash \{P'\}\,\mathbb{C}\,\{Q'\}$ for some $u \in \text{Val}$ by the induction hypothesis. Suppose that $e, \gamma \vDash \Gamma$ and that $(s, \sigma) \in \mathcal{P}[\![\exists v.\, P']\!]e$. It follows that $(s, \sigma) \in \mathcal{P}[\![P']\!]e[v \mapsto u]$ for some $u \in \text{Val}$.

Since $e[v \mapsto u], \Gamma \vDash \{P'\}\,\mathbb{C}\,\{Q'\}$ we know that for all $d_1 \in \mathcal{D}$, $s_0 \in S$, $\sigma_0, \sigma_1 \in \Sigma$ and $\bar{x} \subseteq \mathrm{X}$ with $d_1 \simeq_S (\bar{x})(s_0 + s)$ and $\sigma_1 = \sigma_0 \uplus \sigma$ that $\mathbb{C}, \gamma, d_1, \sigma_1 \not\Downarrow \lightning$. Moreover, $\mathbb{C}, \gamma, d_1, \sigma_1 \Downarrow d_2, \sigma_2$ for some $d_2 \in \mathcal{D}$ and $\sigma_2 \in \Sigma$ where $d_2 \simeq_S (\bar{x})(s_0 + s')$, $\sigma_2 = \sigma_0 \uplus \sigma'$ and $(s', \sigma') \in \mathcal{P}[\![Q']\!]e[v \mapsto u]$. It follows that $(s', \sigma') \in \mathcal{P}[\![\exists v.\, Q]\!]e$, as required.

Fresh case:

Fix $e \in \text{Env}$. In this case $P = \text{И}\alpha.\, P'$ and $Q = \text{И}\alpha.\, Q'$ for some P', Q', α with $e[\alpha \mapsto x], \Gamma \vDash \{P'\}\,\mathbb{C}\,\{Q'\}$ for some fresh $x \in \mathrm{X}$ by the induction hypothesis. Suppose that $e, \gamma \vDash \Gamma$ and that $(s, \sigma) \in \mathcal{P}[\![\text{И}\alpha.\, P']\!]e$. It follows that $(s, \sigma) \in \mathcal{P}[\![P']\!]e[\alpha \mapsto x]$ for some fresh $x \in \mathrm{X}$.

Since $e[\alpha \mapsto x], \Gamma \vDash \{P'\}\,\mathbb{C}\,\{Q'\}$ we know that for all $d_1 \in \mathcal{D}$, $s_0 \in S$, $\sigma_0, \sigma_1 \in \Sigma$ and $\bar{x} \subseteq \mathrm{X}$ with $d_1 \simeq_S (\bar{x})(s_0 + s)$ and $\sigma_1 = \sigma_0 \uplus \sigma$ that $\mathbb{C}, \gamma, d_1, \sigma_1 \not\Downarrow \lightning$. Moreover, $\mathbb{C}, \gamma, d_1, \sigma_1 \Downarrow d_2, \sigma_2$ for some $d_2 \in \mathcal{D}$ and $\sigma_2 \in \Sigma$ where $d_2 \simeq_S (\bar{x})(s_0{+}s')$, $\sigma_2 = \sigma_0 \uplus \sigma'$ and $(s', \sigma') \in \mathcal{P}[\![Q']\!]e[\alpha \mapsto x]$. Since x was chosen to be fresh, we know that $x \# e, s$, and so $(s', \sigma') \in \mathcal{P}[\![\text{И}\alpha.\, Q]\!]e$, as required.

Skip case:

Fix $e \in \textsc{Env}$. In this case $\mathbb{C} = \texttt{skip}$ and $P = \mathsf{emp} = Q$. Suppose that $e, \gamma \vDash \Gamma$, $d_1 \in \mathcal{D}$, $s_0 \in S$, $\sigma_0 \in \Sigma$ and $\bar{x} \subseteq \mathrm{X}$ such that $d_1 \simeq_S (\bar{x})(s_0 + \mathsf{emp})$. The operational semantics states that $\texttt{skip}, \gamma, d_1, \sigma_0 \Downarrow d_1, \sigma_0$ and since $P = Q$ the result follows trivially.

Seq case:

Fix $e \in \textsc{Env}$. In this case $\mathbb{C} = \mathbb{C}_1 \ ; \ \mathbb{C}_2$ for some $\mathbb{C}_1, \mathbb{C}_2 \in \mathcal{L}_{\mathrm{CMD}}$ where $e, \Gamma \vDash \{P\}\,\mathbb{C}_1\,\{R\}$ and $e, \Gamma \vDash \{R\}\,\mathbb{C}_2\,\{Q\}$ for some R, by the inductive hypothesis. Suppose that $e, \gamma \vDash \Gamma$, and that $(s, \sigma) \in \mathcal{P}[\![P]\!]e$. Also suppose that $\mathbb{C}, \gamma, d_1, \sigma_1 \Downarrow o$ for some $d_1 \in \mathcal{D}$, $\sigma_1 \in \Sigma$ and $o \in \textsc{Out}$. The operational semantics requires that $\mathbb{C}_1, \gamma, d_1, \sigma_1 \Downarrow d_2, \sigma_2$ and $\mathbb{C}_2, \gamma, d_2, \sigma_2 \Downarrow o$ for some $d_2 \in \mathcal{D}$ and $\sigma_2 \in \Sigma$.

Since $e, \Gamma \vDash \{P\}\,\mathbb{C}_1\,\{R\}$ we know that for all $d_1 \in \mathcal{D}$, $s_0 \in S$, $\sigma_0, \sigma_1 \in \Sigma$ and $\bar{x} \subseteq \mathrm{X}$ with $d_1 \simeq_S (\bar{x})(s_0 + s)$ and $\sigma_1 = \sigma_0 \uplus \sigma$ that $\mathbb{C}, \gamma, d_1, \sigma_1 \not\Downarrow \lightning$. Moreover, $\mathbb{C}, \gamma, d_1, \sigma_1 \Downarrow d_2, \sigma_2$ for some $d_2 \in \mathcal{D}$ and $\sigma_2 \in \Sigma$ where $d_2 \simeq_S (\bar{x})(s_0{+}s')$, $\sigma_2 = \sigma_0 \uplus \sigma'$ and $(s', \sigma') \in \mathcal{P}[\![R]\!]e$.

Then, since $e, \Gamma \vDash \{R\}\,\mathbb{C}_2\,\{Q\}$ we also know that for all $d_2 \in \mathcal{D}$, $s_0 \in S$, $\sigma_0, \sigma_2 \in \Sigma$ and $\bar{x} \subseteq \mathrm{X}$ with $d_2 \simeq_S (\bar{x})(s_0 + s')$ and $\sigma_2 = \sigma_0 \uplus \sigma$ that $\mathbb{C}, \gamma, d_2, \sigma_2 \not\Downarrow \lightning$. Moreover, $\mathbb{C}, \gamma, d_2, \sigma_2 \Downarrow d_3, \sigma_3$ for some $d_3 \in \mathcal{D}$ and $\sigma_3 \in \Sigma$ where $d_3 \simeq_S (\bar{x})(s_0 + s'')$, $\sigma_3 = \sigma_0 \uplus \sigma''$ and $(s'', \sigma'') \in \mathcal{P}[\![Q]\!]e$, as required.

If case:

Fix $e \in \textsc{Env}$. In this case $\mathbb{C} = \texttt{if } B \texttt{ then } \mathbb{C}_1 \texttt{ else } \mathbb{C}_2$ for some $B \in \mathrm{B}\textsc{Expr}$, $\mathbb{C}_1, \mathbb{C}_2 \in \mathcal{L}_{\mathrm{CMD}}$ and $\mathcal{P}[\![P]\!]e \subseteq \mathsf{bsafe}(B)$ where $e, \Gamma \vDash \{P \wedge \mathcal{P}[\![B]\!]\}\,\mathbb{C}_1\,\{Q\}$ and $e, \Gamma \vDash \{P \wedge \neg\mathcal{P}[\![B]\!]\}\,\mathbb{C}_2\,\{Q\}$ by the induction hypothesis. Suppose that $e, \gamma \vDash \Gamma$ and that $(s, \sigma) \in \mathcal{P}[\![P]\!]e$. Since $\mathcal{P}[\![P]\!]e \subseteq \mathsf{bsafe}(B)$ we know that $\mathcal{B}[\![B]\!]\sigma$ is defined. Suppose that $\mathbb{C}, \gamma, d_1, \sigma_1 \Downarrow o$ for some $d_1 \in \mathcal{D}$, $\sigma_1 \in \Sigma$ and $o \in \textsc{Out}$.

If $\mathcal{B}[\![B]\!]\sigma_1 = \mathsf{true}$ then the operational semantics requires that $\mathbb{C}_1, \gamma, d_1, \sigma_1 \Downarrow o$. Since $e, \Gamma \vDash \{P \wedge \mathcal{P}[\![B]\!]\}\,\mathbb{C}_1\,\{Q\}$ we know that for all $d_1 \in \mathcal{D}$, $s_0 \in S$, $\sigma_0, \sigma_1 \in \Sigma$ and $\bar{x} \subseteq \mathrm{X}$ with $d_1 \simeq_S (\bar{x})(s_0 + s)$ and $\sigma_1 = \sigma_0 \uplus \sigma$ that $\mathbb{C}_1, \gamma, d_1, \sigma_1 \not\Downarrow \lightning$. Moreover, $\mathbb{C}_1, \gamma, d_1, \sigma_1 \Downarrow d_2, \sigma_2$ for some $d_2 \in \mathcal{D}$ and $\sigma_2 \in \Sigma$ where $d_2 \simeq_S (\bar{x})(s_0{+}s')$, $\sigma_2 = \sigma_0 \uplus \sigma'$

and $(s', \sigma') \in \mathcal{P}[\![Q]\!]e$ as required.

If, instead, $\mathcal{B}[\![B]\!]\sigma_1 = \mathsf{false}$ then the operational semantics requires that $\mathbb{C}_2, \gamma, d_1, \sigma_1 \Downarrow o$. Since $e, \Gamma \vDash \{P \wedge \neg\mathcal{P}[\![B]\!]\}\, \mathbb{C}_2\, \{Q\}$ we know that for all $d_1 \in \mathcal{D}$, $s_0 \in S$, $\sigma_0, \sigma_1 \in \Sigma$ and $\bar{x} \subseteq \mathrm{X}$ with $d_1 \simeq_S (\bar{x})(s_0 + s)$ and $\sigma_1 = \sigma_0 \uplus \sigma$ that $\mathbb{C}_2, \gamma, d_1, \sigma_1 \not\Downarrow \lightning$. Moreover, $\mathbb{C}_2, \gamma, d_1, \sigma_1 \Downarrow d_2, \sigma_2$ for some $d_2 \in \mathcal{D}$ and $\sigma_2 \in \Sigma$ where $d_2 \simeq_S (\bar{x})(s_0 + s')$, $\sigma_2 = \sigma_0 \uplus \sigma'$ and $(s', \sigma') \in \mathcal{P}[\![Q]\!]e$ as required.

While case:

Fix $e \in \mathrm{Env}$. In this case $\mathbb{C} = \texttt{while } B \texttt{ do } \mathbb{C}'$ for some $B \in \mathrm{BExpr}$, $\mathbb{C}' \in \mathcal{L}_{\mathrm{Cmd}}$, $Q = P \wedge \neg\mathcal{P}[\![B]\!]$ and $\mathcal{P}[\![P]\!]e \subseteq \mathsf{bsafe}(B)$ where $e, \Gamma \vDash \{P \wedge \mathcal{P}[\![B]\!]\}\, \mathbb{C}'\, \{P\}$ by the inductive hypothesis. Suppose that $e, \gamma \vDash \Gamma$ and that $(s, \sigma) \in \mathcal{P}[\![P]\!]e$. Since $\mathcal{P}[\![P]\!]e \subseteq \mathsf{bsafe}(B)$ we know that $\mathcal{B}[\![B]\!]\sigma$ is defined.

Suppose that $\mathbb{C}, \gamma, d_1, \sigma_1 \Downarrow o$ for some $d_1 \in \mathcal{D}$, $\sigma_1 \in \Sigma$ and $o \in \mathrm{Out}$. We need to establish that $o \neq \lightning$ and $o = d_2, \sigma_2$ with $d_2 \simeq_S (\bar{x})(s_0 + s')$ and $\sigma_2 = \sigma_0 \uplus \sigma'$ for some $d_2 \in \mathcal{D}$, $\sigma_0, \sigma_2 \in \Sigma$, $\bar{x} \subseteq \mathrm{X}$, $s_0 \in S$ and where $(s', \sigma') \in \mathcal{P}[\![Q]\!]e$. We proceed by induction on the structure of derivation of the operational semantics.

If $\mathcal{B}[\![B]\!]\sigma_1 = \mathsf{true}$ then the operational semantics requires that $\mathbb{C}' ; \mathbb{C}, \gamma, d_1, \sigma_1 \Downarrow o$. Since $e, \Gamma \vDash \{P \wedge \mathcal{P}[\![B]\!]\}\, \mathbb{C}'\, \{P\}$ and $(s, \sigma) \in \mathcal{P}[\![P \wedge \mathcal{P}[\![B]\!]]\!]e$ we know that for all $d_1 \in \mathcal{D}$, $s_0 \in S$, $\sigma_0, \sigma_1 \in \Sigma$ and $\bar{x} \subseteq \mathrm{X}$ with $d_1 \simeq_S (\bar{x})(s_0 + s)$ and $\sigma_1 = \sigma_0 \uplus \sigma$ that $\mathbb{C}', \gamma, d_1, \sigma_1 \not\Downarrow \lightning$. Moreover, $\mathbb{C}', \gamma, d_1, \sigma_1 \Downarrow d_1', \sigma_1'$ for some $d_1' \in \mathcal{D}$ and $\sigma_1' \in \Sigma$ where $d_1' \simeq_P (\bar{x})(s_0 + s'')$, $\sigma_1' = \sigma_0 \uplus \sigma''$ and $(s'', \sigma'') \in \mathcal{P}[\![P]\!]e$. Applying the inductive hypothesis for this inner induction, we can conclude that $o \neq \lightning$ and $(s', \sigma') \in \mathcal{P}[\![Q]\!]e$, as required.

If, instead, $\mathcal{B}[\![B]\!]\sigma_1 = \mathsf{false}$ then the operational semantics requires that $o = (d_1, \sigma_1)$ and it follows that $(s, \sigma) \in \mathcal{P}[\![(P \wedge \neg\mathcal{P}[\![B]\!])]\!]e = \mathcal{P}[\![Q]\!]e$, as required.

Assgn case:

Fix $e \in \mathrm{Env}$. In this case $\mathbb{C} = \texttt{x} := E$ for some $\texttt{x} \in \mathrm{Var}$, $E \in \mathrm{Expr}$, $P = \texttt{x} \Rrightarrow v * \sigma$ and $\mathcal{P}[\![\texttt{x} \Rrightarrow v * \sigma]\!]e \subseteq \mathsf{vsafe}(E)$ for some $v \in \mathrm{Val}$ and $\sigma \in \Sigma$ and $Q = \texttt{x} \Rrightarrow \mathcal{E}[\![E]\!]\sigma[\texttt{x} \mapsto v] * \sigma$. Suppose that $e, \gamma \vDash \Gamma$ and $(\mathsf{emp}, \sigma[\texttt{x} \mapsto v]) \in \mathcal{P}[\![P]\!]e$. By the definition of vsafe there is some $v' \in \mathrm{Val}$ such that $\mathcal{E}[\![E]\!]\sigma[\texttt{x} \mapsto v] = v'$.

Now suppose that $d_1 \in \mathcal{D}$, $s_0 \in S$, $\sigma_0, \sigma_1 \in \Sigma$ and $\bar{x} \subseteq \mathrm{X}$ with $d_1 \simeq_S (\bar{x})(s_0 + \mathsf{emp})$ and $\sigma_1 = \sigma_0 \uplus \sigma[\texttt{x} \mapsto v]$. The operational semantics states that $\texttt{x} := E, \gamma, d_1, \sigma_1 \Downarrow d_1, \sigma_1[\texttt{x} \mapsto v']$ and hence we have $(\mathsf{emp}, \sigma[\texttt{x} \mapsto v']) \in \mathcal{P}[\![Q]\!]e$, as required.

Local case:

Fix $e \in \text{Env}$. In this case $\mathbb{C} = \texttt{local}\ x\ \texttt{in}\ \mathbb{C}'$ for some $x \in \text{Var}$, $\mathbb{C}' \in \mathcal{L}_{\text{CMD}}$ and $\mathcal{P}[\![P]\!]e \cap \mathsf{vsafe}(\texttt{x}) \equiv \emptyset$ with $e, \Gamma \vDash \{\texttt{x} \Rightarrow - * P\}\,\mathbb{C}'\,\{\texttt{x} \Rightarrow - * Q\}$ by the inductive hypothesis. Suppose that $e, \gamma \vDash \Gamma$ and $(s, \sigma) \in \mathcal{P}[\![P]\!]e$. By the definition of vsafe, and since $\mathcal{P}[\![P]\!]e \cap \mathsf{vsafe}(\texttt{x}) \equiv \emptyset$, we know that $(s, \sigma[\texttt{x} \mapsto v]) \in \mathcal{P}[\![\texttt{x} \Rightarrow - * P]\!]e$ for every $v \in \text{Val}$ and $\texttt{x} \notin dom(\sigma)$.

Since $e, \Gamma \vDash \{\texttt{x} \Rightarrow - * P\}\,\mathbb{C}'\,\{\texttt{x} \Rightarrow - * Q\}$ we know that for all $d_1 \in \mathcal{D}$, $s_0 \in S$, $\sigma_0, \sigma_1 \in \Sigma$, and $\bar{x} \subseteq \text{X}$ with $d_1 \simeq_S (\bar{x})(s_0 + s)$ and $\sigma_1 = \sigma_0 \uplus \sigma[\texttt{x} \mapsto v]$ that $\mathbb{C}', \gamma, d_1, \sigma_1 \not\Downarrow \lightning$. Moreover, $\mathbb{C}', \gamma, d_1, \sigma_1 \Downarrow d_2, \sigma_2$ for some $d_2 \in \mathcal{D}$ and $\sigma_2 \in \Sigma$ where $d_2 \simeq_S (\bar{x})(s_0 + s')$, $\sigma_2 = \sigma_0 \uplus \sigma'$, $x \notin dom(\sigma')$ and $(s', \sigma'[\texttt{x} \mapsto w]) \in \mathcal{P}[\![\texttt{x} \Rightarrow - * Q]\!]e$ for some $v, w \in \text{Val}$. It follows that $(s', \sigma') \in \mathcal{P}[\![Q]\!]e$, as required.

PDef case:

Fix $e \in \text{Env}$. In this case $\mathbb{C} = \texttt{procs}\ \overrightarrow{\texttt{r}_1} := \texttt{f}_1(\overrightarrow{\texttt{x}_1})\{\mathbb{C}_1\}, ..., \overrightarrow{\texttt{r}_k} := \texttt{f}_k(\overrightarrow{\texttt{x}_k})\{\mathbb{C}_k\}\ \texttt{in}\ \mathbb{C}'$, Γ' makes no reference to any $\texttt{f}_i$, and, for some Γ that refers only to the $\texttt{f}_i$ procedures, $e, \Gamma', \Gamma \vDash \{P\}\,\mathbb{C}'\,\{Q\}$ and

$$\forall(\texttt{f}_i : \mathsf{P}_i \rightarrowtail \mathsf{Q}_i) \in \Gamma.\, e, \Gamma', \Gamma \vDash \begin{array}{c} \left\{\ \exists\overrightarrow{v_i}.\, \mathsf{P}_i(\overrightarrow{v_i}) * \overrightarrow{\texttt{x}_i} \Rightarrow \overrightarrow{v_i} * \overrightarrow{\texttt{r}_i} \Rightarrow -\ \right\} \\ \mathbb{C}_i \\ \left\{\ \exists\overrightarrow{w_i}.\, \mathsf{Q}_i(\overrightarrow{w_i}) * \overrightarrow{\texttt{x}_i} \Rightarrow - * \overrightarrow{\texttt{r}_i} \Rightarrow \overrightarrow{w_i}\ \right\} \end{array}$$

by the inductive hypothesis. Suppose that $e, \gamma \vDash \Gamma'$, $(s, \sigma) \in \mathcal{P}[\![P]\!]e$ and $\mathbb{C}, \gamma, d_1, \sigma_1 \Downarrow o$ for some $d_1 \in \mathcal{D}$, $\sigma_1 \in \Sigma$ and $o \in \text{Out}$. The operational semantics requires that

$$\mathbb{C}', [\texttt{f}_1 \mapsto (\overrightarrow{\texttt{x}_1}, \mathbb{C}_1, \overrightarrow{\texttt{r}_1}), ..., \texttt{f}_k \mapsto (\overrightarrow{\texttt{x}_k}, \mathbb{C}_k, \overrightarrow{\texttt{r}_k})] : \gamma, d_1, \sigma_1 \Downarrow o.$$

By the semantic triples for the procedure bodies, and the fact that $e, \gamma \vDash \Gamma'$, it must be the case that $e, [\texttt{f}_1 \mapsto (\overrightarrow{\texttt{x}_1}, \mathbb{C}_1, \overrightarrow{\texttt{r}_1}), ..., \texttt{f}_k \mapsto (\overrightarrow{\texttt{x}_k}, \mathbb{C}_k, \overrightarrow{\texttt{r}_k})] : \gamma \vDash \Gamma', \Gamma$. Since $e, \Gamma', \Gamma \vDash \{P\}\,\mathbb{C}'\,\{Q\}$ we know that for all $d_1 \in \mathcal{D}$, $s_0 \in S$, $\sigma_0, \sigma_1 \in \Sigma$ and $\bar{x} \subseteq \text{X}$ with $d_1 \simeq_S (\bar{x})(s_0 + s)$ and $\sigma_1 = \sigma_0 \uplus \sigma$ that

$$\mathbb{C}', [\texttt{f}_1 \mapsto (\overrightarrow{\texttt{x}_1}, \mathbb{C}_1, \overrightarrow{\texttt{r}_1}), ..., \texttt{f}_k \mapsto (\overrightarrow{\texttt{x}_k}, \mathbb{C}_k, \overrightarrow{\texttt{r}_k})] : \gamma, d_1, \sigma_1 \not\Downarrow \lightning.$$

Moreover,

$$\mathbb{C}'[\texttt{f}_1 \mapsto (\overrightarrow{\texttt{x}_1}, \mathbb{C}_1, \overrightarrow{\texttt{r}_1}), ..., \texttt{f}_k \mapsto (\overrightarrow{\texttt{x}_k}, \mathbb{C}_k, \overrightarrow{\texttt{r}_k})] : \gamma, d_1, \sigma_1 \Downarrow d_2, \sigma_2$$

for some $d_2 \in \mathcal{D}$ and $\sigma_2 \in \Sigma$ where $d_2 \simeq_S (\bar{x})(s_0 + s')$, $\sigma_2 = \sigma_0 \uplus \sigma'$ and $(s', \sigma') \in \mathcal{P}[\![Q]\!]e$, as required.

PCall case:

Fix $e \in \textsc{Env}$. In this case $\mathbb{C} = \texttt{call } \overrightarrow{\texttt{r}} := \texttt{f}(\overrightarrow{E})$ for some $(\texttt{f} : \mathsf{P} \rightarrowtail \mathsf{Q}) \in \Gamma$ and where

$$P = \{\mathsf{P}(\mathcal{E}[\![\overrightarrow{E}]\!]\sigma''[\overrightarrow{\texttt{r}} \mapsto \overrightarrow{v}]) * \overrightarrow{\texttt{r}} \Rightarrow \overrightarrow{v} * \sigma''\}$$
$$Q = \{\mathsf{Q}(\overrightarrow{w}) * \overrightarrow{\texttt{r}} \Rightarrow \overrightarrow{w} * \sigma''\}$$
$$\mathcal{P}[\![\overrightarrow{\texttt{r}} \Rightarrow \overrightarrow{v} * \sigma'']\!]e \subseteq \mathsf{vsafe}(\overrightarrow{E})$$

Suppose $e, \gamma \vDash \Gamma$, $(s, \sigma) \in \mathcal{P}[\![P]\!]e$ and $\mathbb{C}, \gamma, d_1, \sigma_1 \Downarrow o$ for some $d_1 \in \mathcal{D}$, $\sigma_1 \in \Sigma$ and $o \in \textsc{Out}$. By definition of $e, \gamma \vDash \Gamma$ it must be that, for some $\overrightarrow{\texttt{x}}, \overrightarrow{\texttt{y}} \in \textsc{Var}^*$, $\mathbb{C}' \in \mathcal{L}_{\textsc{Cmd}}$ and $\gamma' \in \textsc{PDef}^*$ with $((\overrightarrow{\texttt{x}}, \mathbb{C}', \overrightarrow{\texttt{y}}), \gamma') = \mathsf{lookup}(\texttt{f}, \gamma)$

$$e, \gamma' \vDash \begin{array}{c} \left\{\ \exists \overrightarrow{u}.\, \mathsf{P}(\overrightarrow{u}) * \overrightarrow{\texttt{x}} \Rightarrow \overrightarrow{u} * \overrightarrow{\texttt{y}} \Rightarrow -\ \right\} \\ \mathbb{C}' \\ \left\{\ \exists \overrightarrow{w}.\, \mathsf{Q}(\overrightarrow{w}) * \overrightarrow{\texttt{x}} \Rightarrow - * \overrightarrow{\texttt{y}} \Rightarrow \overrightarrow{w}\ \right\} \end{array}.$$

We now rule out the faulting cases of the `call` statement from Figure 4.2. Since $\mathsf{lookup}(\texttt{f}, \gamma)$ is defined and has the correct type (enforced by the types of P and Q) the first two faulting cases do not apply. By the vsafe condition, it follows that $\mathcal{E}[\![\overrightarrow{E}]\!]\sigma''[\overrightarrow{\texttt{r}} \mapsto \overrightarrow{v}] = \overrightarrow{u}$ is defined, and so the third faulting case does not apply either. If the fourth faulting case applied, then for some $\overrightarrow{w}$

$$\mathbb{C}', \gamma', d_1, \sigma_1[\overrightarrow{\texttt{y}} \mapsto \overrightarrow{w}][\overrightarrow{\texttt{x}} \mapsto \overrightarrow{u}] \Downarrow \lightning.$$

However, this would violate the precondition that `f` has a valid specification in the procedure definition environment, and so the fourth faulting case does not apply. The fifth and final faulting case is ruled out by the fact that P is only satisfied by a state where the return variables $\overrightarrow{\texttt{r}}$ are present in the variable store.

This leaves just the successful case, which requires that $o = (d_2, \sigma_2)$ for some $d_2 \in \mathcal{D}$ and $\sigma_2 \in \Sigma$ where $\sigma_2 = \sigma[\overrightarrow{\texttt{r}} \mapsto \sigma'(\overrightarrow{\texttt{y}})]$ for some σ' such that

$$\mathbb{C}', \gamma', d_2, \emptyset[\overrightarrow{\texttt{y}} \mapsto \overrightarrow{w}][\overrightarrow{\texttt{x}} \mapsto \overrightarrow{u}] \Downarrow d_2, \sigma'.$$

By our assumption that `f` has a valid specification in the procedure definition environment, it must be that $d_2 = (\bar{x})(s_0 + s')$ for some $\bar{x} \subseteq \mathrm{X}$, $s_0, s' \in S$ with $s' \in \mathsf{Q}(\sigma'(\overrightarrow{\texttt{y}}))$. It follows that $(s', \sigma[\overrightarrow{\texttt{r}} \mapsto \sigma'(\overrightarrow{\texttt{y}})]) \in \mathcal{P}[\![Q]\!]e$, as required.

PWeak case:

Fix $e \in \textsc{Env}$. In this case $\Gamma = \Gamma_1, \Gamma_2$ for some Γ_1, Γ_2 with $e, \Gamma_1 \vDash \{P\}\, \mathbb{C}\, \{Q\}$ by the inductive hypothesis. Suppose that $e, \gamma \vDash \Gamma$, then we also know that $e, \gamma \vDash \Gamma_1$ and so $e, \gamma \vDash \{P\}\, \mathbb{C}\, \{Q\}$, as required. $\square$

5 Fine-grained Reasoning for Program Modules

We have introduced a framework that provides fine-grained abstract reasoning for programs. Our programming language is parameterised by the choice of basic commands and our program state is parameterised by the choice of data structure that these commands manipulate. The reason for this parameterisation is to be able to apply our reasoning to different levels of abstraction.

We now consider a number of different abstractions such as trees, lists and heaps. We show how our reasoning framework can be applied to these different data models and how we use our framework to reason about client-level programs. In each case we construct an abstract module that can be plugged into our general reasoning framework.

5.1 Fine-grained Abstract Modules

A fine-grained abstract module is a collection of operations on some fine-grained abstract state model. For example, a tree module typically provides operations for traversing the tree structure, and adding, removing and moving nodes or subtrees; a list module typically provides operations for adding, removing and querying list elements; and similarly a heap module typically provides operations that allocate and dispose blocks of heap cells, and that fetch and mutate values stored in heap cells.

The programming language introduced in Chapter 4 can be instantiated for such abstract modules by the choice of basic commands $\textsc{Cmd}$. Our reasoning framework can be similarly instantiated for such abstract modules by the choice of the segment algebra $\mathcal{S}(\mathcal{M})$ and the axiomatisation of the basic commands $\mathrm{Ax}[\![(\cdot)]\!]$. Together, these three parameters constitute the notion of a fine-grained abstract module in our formalism.

Definition 5.1 (Fine-grained Abstract Module)**.** A *fine-grained abstract module* $\mathbb{A} = (\textsc{Cmd}_{\mathbb{A}}, \mathcal{S}(\mathcal{M}_{\mathbb{A}}), \textsc{Ax}[\![(\cdot)]\!]_{\mathbb{A}})$ consists of:

- ◇ a set of basic commands $\textsc{Cmd}_{\mathbb{A}}$;
- ◇ a segment algebra $\mathcal{S}(\mathcal{M}_{\mathbb{A}}) = (S_{\mathbb{A}}, \mathsf{emp}_{\mathbb{A}}, \leftharpoondown_{\mathbb{A}}, fa_{\mathbb{A}}, fh_{\mathbb{A}}, +_{\mathbb{A}}, \mathsf{comp}_{\mathbb{A}})$; and
- ◇ an axiomatisation for the basic commands

$$\textsc{Ax}[\![(\cdot)]\!]_{\mathbb{A}} : \textsc{Cmd}_{\mathbb{A}} \to \mathcal{P}(\textsc{Pred}_{\mathbb{A}} \times \textsc{Pred}_{\mathbb{A}}),$$

  where $\textsc{Pred}_{\mathbb{A}}$ is the set of predicates that are evaluated to sets of program states in $\mathcal{P}(S_{\mathbb{A}} \times \Sigma)$.

Recall that variable stores $\sigma \in \Sigma$ are finite partial functions $\sigma : \textsc{Var} \rightharpoonup_{\mathsf{fin}} \textsc{Val}$. We have deliberately left the definition of the value set $\textsc{Val}$ open ended so that it can be tailored to different abstract modules. We will mention any assumptions about the value set in the definition of each of our fine-grained abstract modules.

Notation: We denote the language determined by the abstract module $\mathbb{A}$ as $\mathcal{L}_{\mathbb{A}}$. We denote the axiomatic semantic judgement determined by the abstract module $\mathbb{A}$ as $\vdash_{\mathbb{A}}$. When the abstract module $\mathbb{A}$ can be inferred from context, the subscript $_{\mathbb{A}}$ may be dropped.

The concept of an abstract module was originally introduced in work on abstraction and refinement for local reasoning [26]. The main difference here is that we choose to base our fine-grained abstract modules on segment algebras rather than on context algebras. We have not made the basic commands of the module any more fine-grained, in most cases they are the same modules as were introduced before. It is our specifications and the resulting reasoning system that are fine-grained. By using segment algebras we are able to give small axioms for all of our module basic commands and we are also to derive small specifications for arbitrary programs that use these commands.

We now give a number of examples of fine-grained abstract modules, including a tree module (the original motivation for context logic [14]) and a heap module (the basis of separation logic [43][63]). We also show that our approach is scalable to more complex examples by considering a featherweight DOM module.

5.2 Fine-grained Tree Module

Trees have been the most common example of abstract reasoning to date, so it should be no surprise that the first abstract module we consider is one for manipulating tree structures $\mathbb{T} = (\textsc{Cmd}_{\mathbb{T}}, \mathcal{S}(\mathcal{M}_{\mathrm{T}}), \mathrm{Ax}[\![(\cdot)]\!]_{\mathbb{T}})$. Its commands consist of node-relative traversal, node creation, subtree deletion and tree move (append). The tree model consists of *uniquely-labelled* trees, where each label may only occur once in any given tree, context or segment. This ensures that nodes in a tree are uniquely addressable by their labels. It is therefore assumed that the set of tree labels, $\textsc{Id}$, is contained with in the value set, $\textsc{Val}$; that is, $\textsc{Id} \subseteq \textsc{Val}$.

We also need a constant value null, the *null reference*, to indicate the absence of such a reference. We require that $\mathsf{null} \notin \textsc{Id}$, so that it cannot be confused with a valid node reference, and that $\mathsf{null} \in \textsc{Val}$, so that it may be stored in variables. The set $\textsc{Id}_{\mathsf{null}} \overset{\mathrm{def}}{=} \textsc{Id} \cup \{\mathsf{null}\}$ consists of all valid node references and the null reference.

Definition 5.2 (Tree Update Commands)**.** The set of *tree update commands* $\textsc{Cmd}_{\mathbb{T}}$ is defined as:

$$
\begin{array}{rll}
\textsc{Cmd}_{\mathbb{T}} ::= & \mathtt{n} := \mathtt{getUp}(E) & \textit{get parent} \\
& \mathtt{n} := \mathtt{getLeft}(E) & \textit{get left sibling} \\
& \mathtt{n} := \mathtt{getRight}(E) & \textit{get right sibling} \\
& \mathtt{n} := \mathtt{getFirst}(E) & \textit{get first child} \\
& \mathtt{n} := \mathtt{getLast}(E) & \textit{get last child} \\
& \mathtt{newNodeAfter}(E) & \textit{node creation} \\
& \mathtt{deleteTree}(E) & \textit{subtree deletion} \\
& \mathtt{appendChild}(E, E') & \textit{tree move}
\end{array}
$$

where $\mathtt{n} \in \textsc{Var}$ ranges over program variables and $E \in \textsc{Expr}$ ranges over value expressions.

The intuitive meaning of these commands, which will be realised by their axiomatic semantics, is as follows:

- ◇ `getUp`(E), `getLeft`(E) and `getRight`(E) retrieve, respectively, the identifier of the immediate parent, left sibling and right sibling (if any) of the node identified by E. Require that E identifies a node that exists or they fault;

- ◇ `getFirst`(E) and `getLast`(E) retrieve, respectively, the identifiers of the first and last children (if any) of the node identified by E. Require that E identifies a node that exists or they fault;

- ◇ `newNodeAfter`$(\texttt{E})$ creates a new node with a fresh identifier and no children, which is inserted into the tree as the right sibling of the node identified by $\texttt{E}$. Requires that $\texttt{E}$ identifies a node that exists or is faults;

- ◇ `deleteTree`$(\texttt{E})$ deletes the subtree rooted at the node identified by $\texttt{E}$. Requires that $\texttt{E}$ identifies a node that exists or is faults; and

- ◇ `appendChild`$(\texttt{E}, \texttt{E}')$ removes the subtree rooted at the node identified by $\texttt{E}'$ and reinserts it into the tree as the last child of the node identified by $\texttt{E}$. Requires that $\texttt{E}$ and $\texttt{E}'$ identify nodes that exist, and that the node identified by $\texttt{E}'$ is not an ancestor of the node identified by $\texttt{E}$, or it faults.

We have already seen the tree segment algebra $\mathcal{S}(\mathcal{M}_{\mathrm{T}})$ in Example 3.44. In this model the nodes that are assigned in the tree are the resources available to the program. Node traversal and movement may only be performed on tree nodes that are available to the program; node creation makes new tree nodes available; and subtree deletion makes available tree nodes unavailable and clears their contents.

Definition 5.3 (Tree Axiomatisation)**.** The *tree axiomatisation*

$$\mathrm{Ax}[\![(\cdot)]\!]_{\mathbb{T}} : \mathrm{C{\small MD}}_{\mathbb{T}} \rightarrow \mathcal{P}(\mathrm{P{\small RED}}_{\mathrm{T}} \times \mathrm{P{\small RED}}_{\mathrm{T}})$$

is given in Figure 5.1 and Figure 5.2. Rather than using the form $\mathrm{Ax}[\![\varphi]\!]_{\mathbb{T}} = (P, Q)$, the axioms are given in the more traditional form of $\{P\}\ \varphi\ \{Q\}$.

Notation: Recall from § 4.2.2 that our predicates are parameterised by a multi-holed context algebra and its context formulae. In particular, for the tree module, we use $\mathsf{tree}(P_T)$ to describe a complete tree (a tree context that has no context holes). We also lift the rooted tree shorthand $\lceil ct \rceil$, from Definition 3.11, to predicates, writing $\lceil P_T \rceil$ for $\mathsf{H}\alpha.\,(\alpha{\leftarrow}P_T)$.

Most of our axioms should be unsurprising, although many now have smaller specifications than was possible with context logic. In particular, we now have a genuine small axiom for the `appendChild` command. The precondition of `appendChild`, $\alpha{\leftarrow}n[\gamma] * \beta{\leftarrow}m[\mathsf{tree}(ct)] * \sigma \wedge \mathcal{E}[\![\texttt{E}]\!]\sigma = n \wedge \mathcal{E}[\![\texttt{E}']\!]\sigma = m$, does not refer to any extra context, but describes just the node n and subtree at m being affected by the command and the variables need to evaluate the command's parameters.

$$\left\{\ \alpha{\leftarrow}m[\beta \otimes w[\delta] \otimes \gamma] * \mathtt{n} \Rrightarrow n * \sigma \wedge \mathcal{E}[\![E]\!]\sigma[\mathtt{n} \mapsto n] = w\ \right\}$$
$$\mathtt{n} := \mathtt{getUp}(E)$$
$$\left\{\ \alpha{\leftarrow}m[\beta \otimes w[\delta] \otimes \gamma] * \mathtt{n} \Rrightarrow m * \sigma\ \right\}$$

$$\left\{\ \lceil w[\beta] \rceil * \mathtt{n} \Rrightarrow n * \sigma \wedge \mathcal{E}[\![E]\!]\sigma[\mathtt{n} \mapsto n] = w\ \right\}$$
$$\mathtt{n} := \mathtt{getUp}(E)$$
$$\left\{\ \lceil w[\beta] \rceil * \mathtt{n} \Rrightarrow \mathsf{null} * \sigma\ \right\}$$

$$\left\{\ \alpha{\leftarrow}m[\beta] \otimes w[\gamma] * \mathtt{n} \Rrightarrow n * \sigma \wedge \mathcal{E}[\![E]\!]\sigma[\mathtt{n} \mapsto n] = w\ \right\}$$
$$\mathtt{n} := \mathtt{getLeft}(E)$$
$$\left\{\ \alpha{\leftarrow}m[\beta] \otimes w[\gamma] * \mathtt{n} \Rrightarrow m * \sigma\ \right\}$$

$$\left\{\ \alpha{\leftarrow}m[w[\beta] \otimes \gamma] * \mathtt{n} \Rrightarrow n * \sigma \wedge \mathcal{E}[\![E]\!]\sigma[\mathtt{n} \mapsto n] = w\ \right\}$$
$$\mathtt{n} := \mathtt{getLeft}(E)$$
$$\left\{\ \alpha{\leftarrow}m[w[\beta] \otimes \gamma] * \mathtt{n} \Rrightarrow \mathsf{null} * \sigma\ \right\}$$

$$\left\{\ \alpha{\leftarrow}w[\beta] \otimes m[\gamma] * \mathtt{n} \Rrightarrow n * \sigma \wedge \mathcal{E}[\![E]\!]\sigma[\mathtt{n} \mapsto n] = w\ \right\}$$
$$\mathtt{n} := \mathtt{getRight}(E)$$
$$\left\{\ \alpha{\leftarrow}w[\beta] \otimes m[\gamma] * \mathtt{n} \Rrightarrow m * \sigma\ \right\}$$

$$\left\{\ \alpha{\leftarrow}m[\beta \otimes w[\gamma]] * \mathtt{n} \Rrightarrow n * \sigma \wedge \mathcal{E}[\![E]\!]\sigma[\mathtt{n} \mapsto n] = w\ \right\}$$
$$\mathtt{n} := \mathtt{getRight}(E)$$
$$\left\{\ \alpha{\leftarrow}m[\beta \otimes w[\gamma]] * \mathtt{n} \Rrightarrow \mathsf{null} * \sigma\ \right\}$$

$$\left\{\ \alpha{\leftarrow}w[m[\beta] \otimes \gamma] * \mathtt{n} \Rrightarrow n * \sigma \wedge \mathcal{E}[\![E]\!]\sigma[\mathtt{n} \mapsto n] = w\ \right\}$$
$$\mathtt{n} := \mathtt{getFirst}(E)$$
$$\left\{\ \alpha{\leftarrow}w[m[\beta] \otimes \gamma] * \mathtt{n} \Rrightarrow m * \sigma\ \right\}$$

$$\left\{\ \alpha{\leftarrow}w[\varnothing] * \mathtt{n} \Rrightarrow n * \sigma \wedge \mathcal{E}[\![E]\!]\sigma[\mathtt{n} \mapsto n] = w\ \right\}$$
$$\mathtt{n} := \mathtt{getFirst}(E)$$
$$\left\{\ \alpha{\leftarrow}w[\varnothing] * \mathtt{n} \Rrightarrow \mathsf{null} * \sigma\ \right\}$$

$$\left\{\ \alpha{\leftarrow}w[\beta \otimes m[\gamma]] * \mathtt{n} \Rrightarrow n * \sigma \wedge \mathcal{E}[\![E]\!]\sigma[\mathtt{n} \mapsto n] = w\ \right\}$$
$$\mathtt{n} := \mathtt{getLast}(E)$$
$$\left\{\ \alpha{\leftarrow}w[\beta \otimes m[\gamma]] * \mathtt{n} \Rrightarrow m * \sigma\ \right\}$$

$$\left\{\ \alpha{\leftarrow}w[\varnothing] * \mathtt{n} \Rrightarrow n * \sigma \wedge \mathcal{E}[\![E]\!]\sigma[\mathtt{n} \mapsto n] = w\ \right\}$$
$$\mathtt{n} := \mathtt{getLast}(E)$$
$$\left\{\ \alpha{\leftarrow}w[\varnothing] * \mathtt{n} \Rrightarrow \mathsf{null} * \sigma\ \right\}$$

Figure 5.1: Small axioms for the tree module look-up commands.

$$\left\{\ \alpha\leftarrow w[\beta] * \sigma \wedge \mathcal{E}[\![E]\!]\sigma = w\ \right\}$$
$$\texttt{newNodeAfter}(E)$$
$$\left\{\ \exists m.\, \alpha\leftarrow w[\beta] \otimes m[\varnothing] * \sigma\ \right\}$$

$$\left\{\ \alpha\leftarrow w[\mathsf{tree}(ct)] * \sigma \wedge \mathcal{E}[\![E]\!]\sigma = w\ \right\}$$
$$\texttt{deleteTree}(E)$$
$$\left\{\ \alpha\leftarrow\varnothing * \sigma\ \right\}$$

$$\left\{\ \alpha\leftarrow n[\gamma] * \beta\leftarrow m[\mathsf{tree}(ct)] * \sigma \wedge \mathcal{E}[\![E]\!]\sigma = n \wedge \mathcal{E}[\![E']\!]\sigma = m\ \right\}$$
$$\texttt{appendChild}(E, E')$$
$$\left\{\ \alpha\leftarrow n[\gamma \otimes m[\mathsf{tree}(ct)]] * \beta\leftarrow\varnothing * \sigma\ \right\}$$

Figure 5.2: Small axioms for the tree module modification commands.

5.2.1 Tree Reasoning Examples

We now consider some example programs written using our tree module and show how to reason about these programs in our reasoning framework.

Example 5.4 (Double Tree Deletion)**.** The first example program we consider is the `delete2Trees` program discussed in chapter 2.

$$\begin{array}{rcl}\texttt{delete2Trees(n,m)} & ::= & \texttt{deleteTree(n)}\ ;\\ & & \texttt{deleteTree(m)}\end{array}$$

With our old context-based style of reasoning we were not able to compositionally build up a specification of the overall program from the specifications of the individual `deleteTree` commands. However, with our new reasoning framework we can build up the programs specification compositionally.

$$\left\{\ \alpha\leftarrow n[\mathsf{tree}(ct_1)] * \beta\leftarrow m[\mathsf{tree}(ct_2)] * \mathtt{n} \Rrightarrow n * \mathtt{m} \Rrightarrow m\ \right\}$$
$$\texttt{delete2Trees(n,m)}$$
$$\left\{\ \alpha\leftarrow\varnothing * \beta\leftarrow\varnothing * \mathtt{n} \Rrightarrow n * \mathtt{m} \Rrightarrow m\ \right\}$$

This is illustrated by the proof sketch given in Figure 5.3. The key step in the proof is the use of the separation frame rule to ignore the unused tree at each program step. The uses of the frame rule in the proof sketch are denoted by indentation. In the rest of our examples we will not be so explicit about the use of the frame rule, often directly applying the axioms of our commands to larger states. Notice that the precondition of the `delete2Trees` program is only valid for a program state where the variables `n` and `m` contain identifiers for nodes with completely disjoint subtrees.

$$\left\{\ \alpha{\leftarrow}n[\mathsf{tree}(ct_1)] * \beta{\leftarrow}m[\mathsf{tree}(ct_2)] * \mathtt{n} \Rightarrow n * \mathtt{m} \Rightarrow m\ \right\}$$

```
  { α←n[tree(ct₁)] ∗ n ⇒ n }
  deleteTree(n) ;
  { α←∅ ∗ n ⇒ n }
```

$$\left\{\ \alpha{\leftarrow}\varnothing * \beta{\leftarrow}m[\mathsf{tree}(ct_2)] * \mathtt{n} \Rightarrow n * \mathtt{m} \Rightarrow m\ \right\}$$

```
  { β←m[tree(ct₂)] ∗ m ⇒ m }
  deleteTree(m)
  { β←∅ ∗ m ⇒ m }
```

$$\left\{\ \alpha{\leftarrow}\varnothing * \beta{\leftarrow}\varnothing * \mathtt{n} \Rightarrow n * \mathtt{m} \Rightarrow m\ \right\}$$

Figure 5.3: Proof sketch for the `delete2Trees` program.

Example 5.5 (Node manipulation). Our new reasoning framework allows us to be a great deal more local in our specifications than was possible before. This point is illustrated by programs that only manipulate a small number of nodes, rather than complete subtrees. Consider a program `getCousin` that returns the first child of a node's right sibling if it exists (or null if it does not).

```
n := getCousin(m)  ::=  n := getRight(m) ;
                        if n ≠ null then
                          n := getFirst(n)
                        else
                          skip
```

The specification for the case where the command does not return null can be given as follows:

$$\left\{\ \alpha{\leftarrow}a[\beta] \otimes b[c \otimes \gamma] * \mathtt{n} \Rightarrow v * \mathtt{m} \Rightarrow a\ \right\}$$
$$\mathtt{n} := \mathtt{getCousin}(\mathtt{m})$$
$$\left\{\ \alpha{\leftarrow}a[\beta] \otimes b[c \otimes \gamma] * \mathtt{n} \Rightarrow c * \mathtt{m} \Rightarrow a\ \right\}$$

Notice that this specification does not need to make any mention of the children of node a or of children besides the first of node b. The specification is constrained to just those nodes which are being accessed by the program. This specification can be built up from the definition of the program body as illustrated by the sketch proof in Figure 5.4. In the cases where node a has no right sibling or node b has no children, the command will instead return null. We could specify, and prove, these cases in a similar fashion as above.

```
{ $\alpha{\leftarrow}a[\beta] \otimes b[c \otimes \gamma] * \mathtt{n} \Rightarrow v * \mathtt{m} \Rightarrow a$ }
n := getRight(m) ;
{ $\alpha{\leftarrow}a[\beta] \otimes b[c \otimes \gamma] * \mathtt{n} \Rightarrow b * \mathtt{m} \Rightarrow a$ }
if n ≠ null then
   { $\alpha{\leftarrow}a[\beta] \otimes b[c \otimes \gamma] * \mathtt{n} \Rightarrow b * \mathtt{m} \Rightarrow a$ }
   n := getFirst(n)
   { $\alpha{\leftarrow}a[\beta] \otimes b[c \otimes \gamma] * \mathtt{n} \Rightarrow c * \mathtt{m} \Rightarrow a$ }
else
   { false }
   skip
   { $\alpha{\leftarrow}a[\beta] \otimes b[c \otimes \gamma] * \mathtt{n} \Rightarrow c * \mathtt{m} \Rightarrow a$ }
{ $\alpha{\leftarrow}a[\beta] \otimes b[c \otimes \gamma] * \mathtt{n} \Rightarrow c * \mathtt{m} \Rightarrow a$ }
```

Figure 5.4: Proof sketch for the success case of the `getCousin` program.

Example 5.6 (Swapping Children)**.** We can also specify more complex updates. Consider a program `childSwap` which takes two nodes in the tree and swaps their subtrees so long as they are disjoint (it will fault otherwise).

```
childSwap(n, m)  ::=  local x in
                        newNodeAfter(n) ;
                        x := getRight(n) ;
                        appendAll(n, x) ;
                        appendAll(m, n) ;
                        appendAll(x, m) ;
                        deleteTree(x)
```

This program uses a helper function `appendAll` which appends all of the children of its first target node to its second target node. Again these nodes must have disjoint subtrees or the program will fault.

```
appendAll(n, m)  ::=  local y in
                        y := getFirst(n) ;
                        while y ≠ null do
                          appendChild(m, y) ;
                          y := getFirst(n)
```

We could specify the `childSwap` program as follows:

$$\left\{ \alpha{\leftarrow}n[\mathrm{tree}(ct_1)] * \beta{\leftarrow}m[\mathrm{tree}(ct_2)] * \mathtt{n} \Rightarrow n * \mathtt{m} \Rightarrow m \right\}$$
$$\mathtt{childSwap(n,m)}$$
$$\left\{ \alpha{\leftarrow}n[\mathrm{tree}(ct_2)] * \beta{\leftarrow}m[\mathrm{tree}(ct_1)] * \mathtt{n} \Rightarrow n * \mathtt{m} \Rightarrow m \right\}$$

The proof sketch for this program is a little more complex than for those given above due to the use of while loops in the helper function. We must first provide a specification for the `appendAll` program.

$$\left\{\ \alpha{\leftarrow}n[\mathrm{tree}(ct_1)] * \beta{\leftarrow}m[\mathrm{tree}(ct_2)] * \mathtt{n} \Rrightarrow n * \mathtt{m} \Rrightarrow m\ \right\}$$
$$\mathtt{appendAll(n,m)}$$
$$\left\{\ \alpha{\leftarrow}n[\varnothing] * \beta{\leftarrow}m[\mathrm{tree}(ct_2) \otimes \mathrm{tree}(ct_1)] * \mathtt{n} \Rrightarrow n * \mathtt{m} \Rrightarrow m\ \right\}$$

The proof sketch in Figure 5.5 shows that this specification holds for the `appendAll` program. We need to construct a loop invariant for the while loop. This step is not as straightforward as the other reasoning steps and requires a bit of thought. In this case we choose the invaraint to be,

$$\begin{array}{l}\left(\begin{array}{l}\exists a, ct, ct', ct''.\, ct \otimes a[ct'] \otimes ct'' = ct_1 \wedge \\ \alpha{\leftarrow}n[\mathrm{tree}(a[ct'] \otimes ct'')] * \beta{\leftarrow}m[\mathrm{tree}(ct_2 \otimes ct)] * \mathtt{n} \Rrightarrow n * \mathtt{m} \Rrightarrow m * \mathtt{y} \Rrightarrow a\end{array}\right) \\ \vee \left(\ \alpha{\leftarrow}n[\varnothing] * \beta{\leftarrow}m[\mathrm{tree}(ct_2 \otimes ct_1)]) * \mathtt{n} \Rrightarrow n * \mathtt{m} \Rrightarrow m * \mathtt{y} \Rrightarrow \mathsf{null}\ \right)\end{array}$$

The first disjunct covers the case where the subtree beneath node n is not empty and there is still some work for the while loop to do. It also ensures that the trees $a[ct']$, ct'' and ct all combine to give the original subtree ct_1 that was beneath node n. The second disjunct covers the case where the subtree beneath node n is empty, and hence the whole subtree has been moved. The next test of the while loop condition will then return false. Note that when we enter the loop for the first time, either $ct_1 = \varnothing$ and we are in the second case or there is some choice of $a[ct']$ and $[ct'']$ with $ct = \varnothing$ that puts us in the first case.

We can now go on to prove the overall `childSwap` program making use of our derived specification for `appendAll`. The proof sketch can be found in Figure 5.6.

```
$\{\ \alpha{\leftarrow}n[\mathrm{tree}(ct_1)] * \beta{\leftarrow}m[\mathrm{tree}(ct_2)] * \mathtt{n} \Rightarrow n * \mathtt{m} \Rightarrow m\ \}$
local y in
  $\{\ \alpha{\leftarrow}n[\mathrm{tree}(ct_1)] * \beta{\leftarrow}m[\mathrm{tree}(ct_2)] * \mathtt{n} \Rightarrow n * \mathtt{m} \Rightarrow m * \mathtt{y} \Rightarrow -\ \}$
  $\mathtt{y} := \mathtt{getFirst}(\mathtt{n})$ ;
  $\left\{ \begin{array}{l} \left( \begin{array}{l} \exists a, ct, ct'.\, a[ct] \otimes ct' = ct_1 \wedge \\ \alpha{\leftarrow}n[\mathrm{tree}(a[ct] \otimes ct')] * \beta{\leftarrow}m[\mathrm{tree}(ct_2)] * \mathtt{n} \Rightarrow n * \mathtt{m} \Rightarrow m * \mathtt{y} \Rightarrow a \end{array} \right) \\ \vee \left(\ \alpha{\leftarrow}n[\varnothing] * \beta{\leftarrow}m[\mathrm{tree}(ct_2)]\right) * \mathtt{n} \Rightarrow n * \mathtt{m} \Rightarrow m * \mathtt{y} \Rightarrow \mathsf{null}\ ) \end{array} \right\}$
  $\left\{ \begin{array}{l} \left( \begin{array}{l} \exists a, ct, ct', ct''.\, ct \otimes a[ct'] \otimes ct'' = ct_1 \wedge \\ \alpha{\leftarrow}n[\mathrm{tree}(a[ct'] \otimes ct'')] * \beta{\leftarrow}m[\mathrm{tree}(ct_2 \otimes ct)] * \mathtt{n} \Rightarrow n * \mathtt{m} \Rightarrow m * \mathtt{y} \Rightarrow a \end{array} \right) \\ \vee \left(\ \alpha{\leftarrow}n[\varnothing] * \beta{\leftarrow}m[\mathrm{tree}(ct_2 \otimes ct_1)]\right) * \mathtt{n} \Rightarrow n * \mathtt{m} \Rightarrow m * \mathtt{y} \Rightarrow \mathsf{null}\ ) \end{array} \right\}$
  while $\mathtt{y} \neq \mathsf{null}$ do
    $\left\{ \begin{array}{l} \exists a, ct, ct', ct''.\, ct \otimes a[ct'] \otimes ct'' = ct_1 \wedge \\ \alpha{\leftarrow}n[\mathrm{tree}(a[ct'] \otimes ct'')] * \beta{\leftarrow}m[\mathrm{tree}(ct_2 \otimes ct)] * \mathtt{n} \Rightarrow n * \mathtt{m} \Rightarrow m * \mathtt{y} \Rightarrow a \end{array} \right\}$
    $\mathtt{appendChild}(\mathtt{m}, \mathtt{y})$ ;
    $\left\{ \begin{array}{l} \exists a, ct, ct', ct''.\, ct \otimes a[ct'] \otimes ct'' = ct_1 \wedge \\ \alpha{\leftarrow}n[\mathrm{tree}(ct'')] * \beta{\leftarrow}m[\mathrm{tree}(ct_2 \otimes ct \otimes a[ct'])] * \mathtt{n} \Rightarrow n * \mathtt{m} \Rightarrow m * \mathtt{y} \Rightarrow a \end{array} \right\}$
    $\mathtt{y} := \mathtt{getFirst}(\mathtt{n})$
    $\left\{ \begin{array}{l} \left( \begin{array}{l} \exists a, ct, ct', ct''.\, ct \otimes a[ct'] \otimes ct'' = ct_1 \wedge \\ \alpha{\leftarrow}n[\mathrm{tree}(a[ct'] \otimes ct'')] * \beta{\leftarrow}m[\mathrm{tree}(ct_2 \otimes ct)] \\ * \mathtt{n} \Rightarrow n * \mathtt{m} \Rightarrow m * \mathtt{y} \Rightarrow a \end{array} \right) \\ \vee \left(\ \alpha{\leftarrow}n[\varnothing] * \beta{\leftarrow}m[\mathrm{tree}(ct_2 \otimes ct_1)]\right) * \mathtt{n} \Rightarrow n * \mathtt{m} \Rightarrow m * \mathtt{y} \Rightarrow \mathsf{null}\ ) \end{array} \right\}$
  $\{\ \alpha{\leftarrow}n[\varnothing] * \beta{\leftarrow}m[\mathrm{tree}(ct_2 \otimes ct_1)]) * \mathtt{n} \Rightarrow n * \mathtt{m} \Rightarrow m * \mathtt{y} \Rightarrow \mathsf{null}\ \}$
  $\{\ \alpha{\leftarrow}n[\varnothing] * \beta{\leftarrow}m[\mathrm{tree}(ct_2) \otimes \mathrm{tree}(ct_1)] * \mathtt{n} \Rightarrow n * \mathtt{m} \Rightarrow m * \mathtt{y} \Rightarrow \mathsf{null}\ \}$
$\{\ \alpha{\leftarrow}n[\varnothing] * \beta{\leftarrow}m[\mathrm{tree}(ct_2) \otimes \mathrm{tree}(ct_1)] * \mathtt{n} \Rightarrow n * \mathtt{m} \Rightarrow m\ \}$
```

Figure 5.5: Proof sketch for the `appendAll` program.

```
$\{\ \alpha{\leftarrow}n[\mathrm{tree}(ct_1)] * \beta{\leftarrow}m[\mathrm{tree}(ct_2)] * \mathtt{n} \Rrightarrow n * \mathtt{m} \Rrightarrow m\ \}$
local x in
  $\{\ \alpha{\leftarrow}n[\mathrm{tree}(ct_1)] * \beta{\leftarrow}m[\mathrm{tree}(ct_2)] * \mathtt{n} \Rrightarrow n * \mathtt{m} \Rrightarrow m * \mathtt{x} \Rrightarrow -\ \}$
  newNodeAfter(n) ;
  $\{\ \exists a.\, \alpha{\leftarrow}n[\mathrm{tree}(ct_1)] \otimes a[\varnothing] * \beta{\leftarrow}m[\mathrm{tree}(ct_2)] * \mathtt{n} \Rrightarrow n * \mathtt{m} \Rrightarrow m * \mathtt{x} \Rrightarrow -\ \}$
  x := getRight(n) ;
  $\{\ \exists a.\, \alpha{\leftarrow}n[\mathrm{tree}(ct_1)] \otimes a[\varnothing] * \beta{\leftarrow}m[\mathrm{tree}(ct_2)] * \mathtt{n} \Rrightarrow n * \mathtt{m} \Rrightarrow m * \mathtt{x} \Rrightarrow a\ \}$
  appendAll(n, x) ;
  $\{\ \exists a.\, \alpha{\leftarrow}n[\varnothing] \otimes a[\mathrm{tree}(ct_1)] * \beta{\leftarrow}m[\mathrm{tree}(ct_2)] * \mathtt{n} \Rrightarrow n * \mathtt{m} \Rrightarrow m * \mathtt{x} \Rrightarrow a\ \}$
  appendAll(m, n) ;
  $\{\ \exists a.\, \alpha{\leftarrow}n[\mathrm{tree}(ct_2)] \otimes a[\mathrm{tree}(ct_1)] * \beta{\leftarrow}m[\varnothing] * \mathtt{n} \Rrightarrow n * \mathtt{m} \Rrightarrow m * \mathtt{x} \Rrightarrow a\ \}$
  appendAll(x, m) ;
  $\{\ \exists a.\, \alpha{\leftarrow}n[\mathrm{tree}(ct_2)] \otimes a[\varnothing] * \beta{\leftarrow}m[\mathrm{tree}(ct_1)] * \mathtt{n} \Rrightarrow n * \mathtt{m} \Rrightarrow m * \mathtt{x} \Rrightarrow a\ \}$
  deleteTree(x)
  $\{\ \exists a.\, \alpha{\leftarrow}n[\mathrm{tree}(ct_2)] * \beta{\leftarrow}m[\mathrm{tree}(ct_1)] * \mathtt{n} \Rrightarrow n * \mathtt{m} \Rrightarrow m * \mathtt{x} \Rrightarrow a\ \}$
$\{\ \alpha{\leftarrow}n[\mathrm{tree}(ct_2)] * \beta{\leftarrow}m[\mathrm{tree}(ct_1)] * \mathtt{n} \Rrightarrow n * \mathtt{m} \Rrightarrow m\ \}$
```

Figure 5.6: Proof sketch for the `childSwap` program.

5.3 Fine-grained List Module

Next we consider the list module $\mathbb{L} = (\textsc{Cmd}_{\mathbb{L}}, \mathcal{S}(\mathcal{M}_{\mathbb{L}}), \textsc{Ax}[\![(\cdot)]\!]_{\mathbb{L}})$ which is a somewhat more exotic example of an abstract module. The list module provides an addressable set of lists of unique elements which we call a *list-store*. Each list can be manipulated independently in a number of ways, new lists can be constructed and existing lists can be deleted. Later, in chapter 6, we will see that this module can be used as part of an implementation of the tree module considered above. In particular, we will store a tree node's children in a list from this module.

It is assumed that the set of list addresses, $\textsc{Adr}$, is contained within the value set, $\textsc{Val}$; that is, $\textsc{Adr} \subseteq \textsc{Val}$. We also need a constant value null, the *null reference*, for use in situations where a list address or list value does not occur, to indicate the absence of such a value. We require that $\mathsf{null} \notin \textsc{Adr}$, so that it cannot be confused with a valid list address, and that $\mathsf{null} \in \textsc{Val}$, so that it may be stored in variables. The set $\textsc{Adr}_{\mathsf{null}} \overset{\text{def}}{=} \textsc{Adr} \cup \{\mathsf{null}\}$ consists of all valid list addresses and the null reference.

Definition 5.7 (List Update Commands)**.** The set of *list update commands* $\textsc{Cmd}_{\mathbb{L}}$ is defined as:

$\textsc{Cmd}_{\mathbb{L}}$ ::=	$\mathtt{x} := E.\mathtt{getHead}()$	*get first value*
	$\mathtt{x} := E.\mathtt{getTail}()$	*get last value*
	$\mathtt{x} := E.\mathtt{getNext}(E')$	*get next value*
	$\mathtt{x} := E.\mathtt{getPrev}(E')$	*get previous value*
	$\mathtt{x} := E.\mathtt{pop}()$	*stack-style pop*
	$E.\mathtt{push}(E')$	*stack-style push*
	$E.\mathtt{remove}(E')$	*value removal*
	$E.\mathtt{insert}(E', E'')$	*value insertion*
	$\mathtt{x} := \mathtt{newList}()$	*list creation*
	$E.\mathtt{deleteList}()$	*list deletion*

where $\mathtt{x} \in \textsc{Var}$ ranges over program variables and $E, E', ... \in \textsc{Expr}$ range over value expressions.

The intuitive meaning of these commands, which will be realised by their axiomatic semantics, is as follows:

◇ $E.\mathtt{getHead}()$ and $E.\mathtt{getTail}()$ retrieve, respectively, the first and last elements (if any) of the list identified by E. Require that E identifies a list that exists or they fault;

◇ $E.\mathtt{getNext}(E')$ and $E.\mathtt{getPrev}(E')$ retrieve, respectively, the elements (if any) following and preceding the element E' in the list identified by E. Require that E identifies a list that exists and that E' identifies an element in the list identified by E or they fault;

◇ $E.\mathtt{pop}()$ retrieves and removes the first element of the list identified by E (if the list is empty it simply returns null). Requires that E identifies a list that exists or it faults;

◇ $E.\mathtt{push}(E')$ adds the element E' to the start of the list identified by E. Requires that E identifies a list that exists and that E' identifies an element that is not in the list identified by E or it faults;

◇ $E.\mathtt{remove}(E')$ removes the element E' from the list identified by E. Requires that E identifies a list that exists and that E' identifies an element in the list identified by E or it faults;

- ◇ `E.insert(E′,E″)` inserts the element `E″` immediately following `E′` in the list identified by `E`. Requires that `E` identifies a list that exists, that `E′` identifies an element in the list identified by `E`, and that `E″` identifies an element that is not in the list identified by `E` or it faults;
- ◇ `newList()` creates a new list, initially empty, and returns its address; and
- ◇ `E.deleteList()` deletes the list identified by `E`. Requires that `E` identifies a list that exists or it faults.

We require that elements occur at most once in any given list. Thus `getNext`, `getPrev` and `insert` are unambiguous and `push` and `insert` fault if they are used to attempt to insert elements that are already present in the list.

We gave a list segment algebra in Example 3.45. However, this segment algebra only described properties of a single list. List-stores are similar to heaps in the sense that they are finite maps from addresses to values, except that now the values have intrinsic structure: they are lists of unique elements. We introduce a list-store segment algebra that can model such structures.

List-Store Segment Algebra

Recall the multi-holed list context algebra $\mathcal{M}_L = (\mathrm{L}_{\mathrm{VAL},\mathrm{X}}, \mathrm{X}, fv_{\mathrm{L}}, \bullet)$ from Example 3.33, where $cl \in \mathrm{L}_{\mathrm{VAL},\mathrm{X}}$ were defined as:

$$cl \quad ::= \quad \varepsilon \mid x \mid u \mid cl : cl$$

with the restriction that $u \in \textsc{Val}$, each hole label $x \in \mathrm{X}$ occurs at most once in a list context cl and the assumption that $:$ is associative with identity ε.

These contexts can only be used to model a single list. To enable us to model multiple lists we add annotations for list addresses to our labels. That is, we work with labels $x_i \in \mathrm{X}^{\textsc{Adr}}$ where $x \in \mathrm{X}$ and $i \in \textsc{Adr}$. The intuition is that a hole x_i can only occur within the corresponding list i.

We use this modified label set to provide the multi-holed list context algebra $\mathcal{M}_{\mathbb{L}} = (\mathrm{L}_{\mathrm{VAL},\mathrm{X}^{\mathrm{ADR}}}, \mathrm{X}^{\textsc{Adr}}, fv_{\mathrm{L}}, \bullet)$. This multi-holed context algebra is identical to $\mathcal{M}_L$ except that it uses our annotated label set for its hole labels. We write $l, l', ...$ for list contexts with no contexts holes.

Informally, list-store segments consist of sets of labelled list contexts, where labels can either be some $x_i \in \mathrm{X}^{\textsc{Adr}}$, corresponding to some piece of the list at address i, or the special label 0_i, used to indicate that a list context is rooted. A rooted list context, as before, cannot be extended to the left or right. We write $\mathrm{X}_0^{\textsc{Adr}}$ for the set of labels $\mathrm{X}^{\textsc{Adr}}$ extended with the set of empty labels $\{0_i \mid i \in \textsc{Adr}\}$. We do not allow any 0_i to be used as a hole label.

The list-store segment algebra is $\mathcal{S}(\mathcal{M}_{\mathbb{L}}) = (\mathrm{S}_{\mathbb{L}}, \emptyset, \leftarrow, fa_{\mathbb{L}}, fh_{\mathbb{L}}, +, \mathsf{comp}_{\mathbb{L}})$ where,

- ◇ the set of list-store segments $\mathrm{S}_{\mathbb{L}}$, ranged over by $sls, sls_1, ...$, is defined inductively as:

$$sls \quad ::= \quad \emptyset \mid \{(x_i, cl)\} \mid sls \uplus sls$$

  with list contexts $cl \in \mathrm{L}_{\mathrm{VAL},\mathrm{X}^{\textsc{Adr}}}$, addresses $x_i \in \mathrm{X}_0^{\textsc{Adr}}$, the restriction that addresses and hole labels are unique in a segment sls, the restriction that each value occurs as most once in a particular list i and the restriction that for each $(x_i, cl) \in sls$, $y_j \notin fh_{\mathbb{L}}(cl)$ if $i \neq j$ or $y = x$. The disjoint union of list-store segments $\uplus$ is only defined when the segments have disjoint addresses and contexts. The operation is both associative and commutative.

- ◇ the context addressing function

$$\leftarrow\ : \mathrm{X}_0^{\textsc{Adr}} \times \mathrm{L}_{\mathrm{VAL},\mathrm{X}^{\textsc{Adr}}} \rightharpoonup \mathrm{S}_{\mathbb{L}}$$

  is defined as:

$$x_i{\leftarrow}cl \quad \stackrel{\text{def}}{=} \quad \begin{cases} \{(x_i, cl)\} & \text{if } x_i \notin fh_L(cl) \text{ and for all } y_j,\ y_j \notin fh_L(cl) \text{ if } i \neq j \\ \text{undefined} & \text{otherwise} \end{cases}$$

- ◇ the free addresses function

$$fa_{\mathbb{L}} : \mathrm{S}_{\mathbb{L}} \to \mathcal{P}_{\mathsf{fin}}(\mathrm{X}^{\textsc{Adr}})$$

  is defined by induction on the structure of list-store segments as:

$$\begin{aligned} fa_{\mathbb{L}}(\emptyset) &\stackrel{\text{def}}{=} \emptyset \\ fa_{\mathbb{L}}(\{(x_i, cl)\}) &\stackrel{\text{def}}{=} \begin{cases} \emptyset & \text{if } x = 0 \\ \{x_i\} & \text{otherwise} \end{cases} \\ fa_{\mathbb{L}}(sls_1 \uplus sls_2) &\stackrel{\text{def}}{=} fa_{\mathbb{L}}(sls_1) \cup fa_{\mathbb{L}}(sls_2) \end{aligned}$$

- ◇ the free holes function

$$fh_{\mathbb{L}} : \mathrm{S}_{\mathbb{L}} \to \mathcal{P}_{\mathsf{fin}}(\mathrm{X}^{\textsc{Adr}})$$

is defined by induction on the structure of list-store segments as:

$$\begin{array}{rcl} fh_{\mathbb{L}}(\emptyset) & \overset{\text{def}}{=} & \emptyset \\ fh_{\mathbb{L}}(\{(x_i, cl)\}) & \overset{\text{def}}{=} & fh_{\mathrm{L}}(cl) \\ fh_{\mathbb{L}}(sls_1 \uplus sls_2) & \overset{\text{def}}{=} & fh_{\mathbb{L}}(sls_1) \cup fh_{\mathbb{L}}(sls_2) \end{array}$$

◇ the segment combination operator

$$+ : \mathrm{S}_{\mathbb{L}} \times \mathrm{S}_{\mathbb{L}} \to \mathrm{S}_{\mathbb{L}}$$

is defined as:

$$sls_1 + sls_2 \overset{\text{def}}{=} \begin{cases} sls_1 \uplus sls_2 & \text{if } fa_{\mathbb{L}}(sls_1) \cap fa_{\mathbb{L}}(sls_2) = \emptyset \\ & \text{and } fh_{\mathbb{L}}(sls_1) \cap fh_{\mathbb{L}}(sls_2) = \emptyset \\ \text{undefined} & \text{otherwise} \end{cases}$$

◇ the segment compression operator

$$\mathsf{comp}_{\mathbb{L}} : \mathrm{X} \times \mathrm{S}_{\mathbb{L}} \to \mathrm{S}_{\mathbb{L}}$$

is defined as:

$$\mathsf{comp}_{\mathbb{L}}(x_i, sls) \overset{\text{def}}{=} \begin{cases} sls & \text{if } x_i \notin fa_{\mathbb{L}}(sls) \\ & \text{and } x_i \notin fh_{\mathbb{L}}(sls) \\ sls' + \{(z_i, cl \bullet_{x_i} cl')\} & \text{if } \exists sls', z_i, cl, cl'. \\ & sls = sls' + \{(z_i, cl), (x_i, cl')\} \\ & \text{and } x_i \in fh_{\mathrm{L}}(cl) \\ sls' + \{(0_i, cl)\} & \text{if } \exists sls', x_i, cl.\, sls = sls' + \{(x_i, cl)\} \\ & \text{and } x_i \notin fh_{\mathbb{L}}(sls') \\ \text{undefined} & \text{otherwise} \end{cases}$$

It is necessary to deal with complete lists in order to specify a number of the update and lookup commands on lists. For example, `getHead` returns the first item in a list. Given the partial list-store segment $(x_i, u_1 : u_2)$, it is not clear that u_1 is the first element of the list. Indeed, if the list-store segment also contains $(z_i, u_0 : x_i)$ then u_1 is certainly not the first element of list i. However, given the rooted list-store segment $(0_i, u_1 : u_2)$ it is certain that u_1 is the first element of list i.

We use disjointness, just as in heaps, to talk about properties of different lists.

$$\{(0_i, cl)\} \uplus \{(0_j, cl')\} \quad = \quad \{(0_i, cl), (0_j, cl')\}$$

Compression is then used to join together and break apart pieces of the same list.

$$\mathsf{comp}_{\mathbb{L}}(x_i, \{(z_i, u_1 : x_i), (x_i, u_2 : u_3)\}) \quad = \quad \{(z_i, u_1 : u_2 : u_3)\}$$

The segment model allows us to interleave these two types of composition so that we can describe arbitrary parts of the list-store structure.
Notation: We write $x_i{\leftarrow}cl$ for $\{(x_i, cl)\}$ and also write $i \Mapsto [\, cl \,]$ as shorthand for $\{(0_i, cl)\}$.

Definition 5.8 (List Axiomatisation)**.** The *list axiomatisation*

$$\textsc{Ax}[\![(\cdot)]\!]_{\mathbb{L}} : \textsc{Cmd}_{\mathbb{L}} \to \mathcal{P}(\textsc{Pred}_{\mathbb{L}} \times \textsc{Pred}_{\mathbb{L}})$$

is given in Figure 5.7 and Figure 5.8.

Notation: We lift the shorthand $i \Mapsto [\, cl \,]$ to predicates, writing $i \Mapsto [\, P_L \,]$ for $\mathsf{H}\alpha_i.\,(\alpha_i{\leftarrow}P_L)$.

The axioms for the basic commands of the list module describe just the state that is required or modified by the command. For example, the `getHead` command either needs to know which node is at the head of the list, or that the list is empty. In the first case it does not need any information about the rest of the list. Similarly, the `getNext` command only needs to know about the target list element and either the next element in the list or that the target node is at the end of the list. Slightly more complicated are the commands `insert` and `push`, which add nodes to a list. In order to be sure that the element to be added is not already in the list, the axioms needs to include the whole of the list in the precondition.

5.3.1 List Reasoning Examples

Reasoning about programs written in the list module is very similar to reasoning about programs written in the tree module. We shall cover one example here to illustrate the similarities in the reasoning.

Example 5.9 (List reversal)**.** One of the most common examples of list reasoning in the literature is that of list reversal. Here we consider a program that takes a list

$$\left\{\ i \Mapsto [\, u : \beta_i \,] * \mathtt{x} \Rightarrow v * \sigma \wedge \mathcal{E}\llbracket E \rrbracket \sigma[\mathtt{x} \mapsto v] = i\ \right\}$$
$$\mathtt{x} := E.\mathtt{getHead}()$$
$$\left\{\ i \Mapsto [\, u : \beta_i \,] * \mathtt{x} \Rightarrow u * \sigma\ \right\}$$

$$\left\{\ i \Mapsto [\, \varepsilon \,] * \mathtt{x} \Rightarrow v * \sigma \wedge \mathcal{E}\llbracket E \rrbracket \sigma[\mathtt{x} \mapsto v] = i\ \right\}$$
$$\mathtt{x} := E.\mathtt{getHead}()$$
$$\left\{\ i \Mapsto [\, \varepsilon \,] * \mathtt{v} \Rightarrow \mathsf{null} * \sigma\ \right\}$$

$$\left\{\ i \Mapsto [\, \beta_i : u \,] * \mathtt{x} \Rightarrow v * \sigma \wedge \mathcal{E}\llbracket E \rrbracket \sigma[\mathtt{x} \mapsto v] = i\ \right\}$$
$$\mathtt{x} := E.\mathtt{getTail}()$$
$$\left\{\ i \Mapsto [\, \beta_i : u \,] * \mathtt{x} \Rightarrow u * \sigma\ \right\}$$

$$\left\{\ i \Mapsto [\, \varepsilon \,] * \mathtt{x} \Rightarrow v * \sigma \wedge \mathcal{E}\llbracket E \rrbracket \sigma[\mathtt{x} \mapsto v] = i\ \right\}$$
$$\mathtt{x} := E.\mathtt{getTail}()$$
$$\left\{\ i \Mapsto [\, \varepsilon \,] * \mathtt{x} \Rightarrow \mathsf{null} * \sigma\ \right\}$$

$$\left\{\ \alpha_i \leftarrow (w : u) * \mathtt{x} \Rightarrow v * \sigma \wedge \mathcal{E}\llbracket E \rrbracket \sigma[\mathtt{x} \mapsto v] = i \wedge \mathcal{E}\llbracket E' \rrbracket \sigma[\mathtt{x} \mapsto v] = w\ \right\}$$
$$\mathtt{x} := E.\mathtt{getNext}(E')$$
$$\left\{\ \alpha_i \leftarrow (w : u) * \mathtt{x} \Rightarrow u * \sigma\ \right\}$$

$$\left\{\ i \Mapsto [\, \beta_i : w \,] * \mathtt{x} \Rightarrow v * \sigma \wedge \mathcal{E}\llbracket E \rrbracket \sigma[\mathtt{x} \mapsto v] = i \wedge \mathcal{E}\llbracket E' \rrbracket \sigma[\mathtt{x} \mapsto v] = w\ \right\}$$
$$\mathtt{x} := E.\mathtt{getNext}(E')$$
$$\left\{\ i \Mapsto [\, \beta_i : w \,] * \mathtt{x} \Rightarrow \mathsf{null} * \sigma\ \right\}$$

$$\left\{\ \alpha_i \leftarrow (u : w) * \mathtt{x} \Rightarrow v * \sigma \wedge \mathcal{E}\llbracket E \rrbracket \sigma[\mathtt{x} \mapsto v] = i \wedge \mathcal{E}\llbracket E' \rrbracket \sigma[\mathtt{x} \mapsto v] = w\ \right\}$$
$$\mathtt{x} := E.\mathtt{getPrev}(E')$$
$$\left\{\ \alpha_i \leftarrow (u : w) * \mathtt{x} \Rightarrow u * \sigma\ \right\}$$

$$\left\{\ i \Mapsto [\, w : \beta_i \,] * \mathtt{x} \Rightarrow v * \sigma \wedge \mathcal{E}\llbracket E \rrbracket \sigma[\mathtt{x} \mapsto v] = i \wedge \mathcal{E}\llbracket E' \rrbracket \sigma[\mathtt{x} \mapsto v] = w\ \right\}$$
$$\mathtt{x} := E.\mathtt{getPrev}(E')$$
$$\left\{\ i \Mapsto [\, w : \beta_i \,] * \mathtt{x} \Rightarrow \mathsf{null} * \sigma\ \right\}$$

Figure 5.7: Small axioms for the list module look-up commands.

$$\left\{\; i \Mapsto [\, u : \beta_i \,] * \mathtt{x} \Rightarrow v * \sigma \wedge \mathcal{E}[\![E]\!]\sigma[\mathtt{x} \mapsto v] = i \;\right\}$$
$$\mathtt{x} := E.\mathtt{pop}()$$
$$\left\{\; i \Mapsto [\, \beta_i \,] * \mathtt{x} \Rightarrow u * \sigma \;\right\}$$

$$\left\{\; i \Mapsto [\, \varepsilon \,] * \mathtt{x} \Rightarrow v * \sigma \wedge \mathcal{E}[\![E]\!]\sigma[\mathtt{x} \mapsto v] = i \;\right\}$$
$$\mathtt{x} := E.\mathtt{pop}()$$
$$\left\{\; i \Mapsto [\, \varepsilon \,] * \mathtt{x} \Rightarrow \mathsf{null} * \sigma \;\right\}$$

$$\left\{\; i \Mapsto [\, l \,] * \sigma \wedge v \notin l \wedge \mathcal{E}[\![E]\!]\sigma = i \wedge \mathcal{E}[\![E']\!]\sigma = v \;\right\}$$
$$E.\mathtt{push}(E')$$
$$\left\{\; i \Mapsto [\, v : l \,] * \sigma \;\right\}$$

$$\left\{\; \alpha_i \leftarrow v * \sigma \wedge \mathcal{E}[\![E]\!]\sigma = i \wedge \mathcal{E}[\![E']\!]\sigma = v \;\right\}$$
$$E.\mathtt{remove}(E')$$
$$\left\{\; \alpha_i \leftarrow \varepsilon * \sigma \;\right\}$$

$$\left\{\; i \Mapsto [\, l : v : l' \,] * \sigma \wedge (u \notin l + v + l') \wedge \mathcal{E}[\![E]\!]\sigma = i \wedge \mathcal{E}[\![E']\!]\sigma = v \wedge \mathcal{E}[\![E'']\!]\sigma = u \;\right\}$$
$$E.\mathtt{insert}(E', E'')$$
$$\left\{\; i \Mapsto [\, l : v : u : l' \,] * \sigma \;\right\}$$

$$\left\{\; \mathtt{x} \Rightarrow - \;\right\}$$
$$\mathtt{x} := \mathtt{newList}()$$
$$\left\{\; \exists i.\, i \Mapsto [\, \varepsilon \,] * \mathtt{x} \Rightarrow i \;\right\}$$

$$\left\{\; i \Mapsto [\, l \,] * \sigma \wedge \mathcal{E}[\![E]\!]\sigma = i \;\right\}$$
$$E.\mathtt{deleteList}()$$
$$\left\{\; \sigma \;\right\}$$

Figure 5.8: Small axioms for the list module modification commands.

and returns a new list which contains all of the contents of the old list in the reverse order.

```
x := i.listReverse()  ::=  local y in
                             x := newList() ;
                             y := i.pop() ;
                             while y ≠ null do
                               x.push(y) ;
                               y := i.pop()
                             i.deleteList()
```

We can specify this program as follows:

$$\left\{\ i \mapsto\!\!\!\!\Rightarrow [\,l\,] * \mathtt{x} \Rightarrow - * \mathtt{i} \Rightarrow i\ \right\}$$
$$\mathtt{x} := \mathtt{i.listReverse}()$$
$$\left\{\ \exists j.\, j \mapsto\!\!\!\!\Rightarrow [\,l^{\dagger}\,] * \mathtt{x} \Rightarrow j * \mathtt{i} \Rightarrow i\ \right\}$$

where we write $l^{\dagger}$ for the reflection of list l. For example, $(a : b : c)^{\dagger} = (c : b : a)$. This specification is derived from the specification of the body of the program as shown in Figure 5.9.

There are many other common programming patterns for list usage and using similar techniques we could also provide their specifications.

5.4 Fine-grained Heap Module

Those familiar with separation logic will probably be used to thinking about a heap module. Here we consider a fine-grained heap module, $\mathbb{H} = (\textsc{Cmd}_{\mathbb{H}}, \mathcal{S}(\mathcal{M}_{\mathrm{H}}), \mathrm{Ax}[\![(\cdot)]\!]_{\mathbb{H}})$ in our reasoning framework. Its commands consist of the usual heap allocation, disposal, mutation and lookup. Heaps are often thought of as finite partial functions from heap addresses ($\textsc{Adr}$) to values ($\textsc{Val}$). The address set is assumed to be the positive integers, i.e. $\textsc{Adr} = \mathbb{Z}^{+}$, which is contained within the value set, i.e. $\textsc{Adr} \subseteq \textsc{Val}$. This enables program variables and heap cells to hold pointers to other heap cells and arithmetic operations to be performed on heap addresses (pointer arithmetic). Again, we work with the address set $\textsc{Adr}_{\mathsf{null}} \stackrel{\text{def}}{=} \textsc{Adr} \cup \{\mathsf{null}\}$ consisting of all valid addresses plus the null reference.

```
$\{\ i \Mapsto [\,l\,] * \mathtt{x} \Rrightarrow - * \mathtt{i} \Rrightarrow i\ \}$
local y in
  $\{\ i \Mapsto [\,l\,] * \mathtt{x} \Rrightarrow - * \mathtt{i} \Rrightarrow i * \mathtt{y} \Rrightarrow -\ \}$
  x := newList() ;
  $\{\ \exists j.\, i \Mapsto [\,l\,] * j \Mapsto [\,\varepsilon\,] * \mathtt{x} \Rrightarrow j * \mathtt{i} \Rrightarrow i * \mathtt{y} \Rrightarrow -\ \}$
  y := i.pop() ;
  $\left\{\begin{array}{l} (\exists a, l', j.\, i \Mapsto [\,l'\,] * j \Mapsto [\,\varepsilon\,] * \mathtt{x} \Rrightarrow j * \mathtt{i} \Rrightarrow i * \mathtt{y} \Rrightarrow a \wedge a : l' = l) \\ \ \vee (\exists j.\, i \Mapsto [\,\varepsilon\,] * j \Mapsto [\,\varepsilon\,] * \mathtt{x} \Rrightarrow j * \mathtt{i} \Rrightarrow i * \mathtt{y} \Rrightarrow \mathsf{null}) \end{array}\right\}$
  $\left\{\begin{array}{l} \big(\exists a, l', l'', j.\, i \Mapsto [\,l'\,] * j \Mapsto [\,l''\,] * \mathtt{x} \Rrightarrow j * \mathtt{i} \Rrightarrow i * \mathtt{y} \Rrightarrow a \wedge l''^{\dagger} : a : l' = l\big) \\ \ \vee \big(\exists j.\, i \Mapsto [\,\varepsilon\,] * j \Mapsto [\,l^{\dagger}\,] * \mathtt{x} \Rrightarrow j * \mathtt{i} \Rrightarrow i * \mathtt{y} \Rrightarrow \mathsf{null}\big) \end{array}\right\}$
  while y $\neq$ null do
    $\{\ \exists a, l', l'', j.\, i \Mapsto [\,l'\,] * j \Mapsto [\,l''\,] * \mathtt{x} \Rrightarrow j * \mathtt{i} \Rrightarrow i * \mathtt{y} \Rrightarrow a * \sigma \wedge l''^{\dagger} : a : l' = l\ \}$
    x.push(y) ;
    $\{\ \exists a, l', l'', j.\, i \Mapsto [\,l'\,] * j \Mapsto [\,a + l''\,] * \mathtt{x} \Rrightarrow j * \mathtt{i} \Rrightarrow i * \mathtt{y} \Rrightarrow a \wedge l''^{\dagger} : a : l' = l\ \}$
    y := i.pop()
    $\left\{\begin{array}{l} \big(\exists a, l', l'', j.\, i \Mapsto [\,l'\,] * j \Mapsto [\,l''\,] * \mathtt{x} \Rrightarrow j * \mathtt{i} \Rrightarrow i * \mathtt{y} \Rrightarrow a \wedge l''^{\dagger} : a : l' = l\big) \\ \ \vee \big(\exists j.\, i \Mapsto [\,\varepsilon\,] * j \Mapsto [\,l^{\dagger}\,] * \mathtt{x} \Rrightarrow j * \mathtt{i} \Rrightarrow i * \mathtt{y} \Rrightarrow \mathsf{null}\big) \end{array}\right\}$
  $\{\ \exists j.\, i \Mapsto [\,\varepsilon\,] * j \Mapsto [\,l^{\dagger}\,] * \mathtt{x} \Rrightarrow j * \mathtt{i} \Rrightarrow i * \mathtt{y} \Rrightarrow \mathsf{null}\ \}$
  i.deleteList()
  $\{\ \exists j.\, j \Mapsto [\,l^{\dagger}\,] * \mathtt{x} \Rrightarrow j * \mathtt{i} \Rrightarrow i * \mathtt{y} \Rrightarrow \mathsf{null}\ \}$
$\{\ \exists j.\, j \Mapsto [\,l^{\dagger}\,] * \mathtt{x} \Rrightarrow j * \mathtt{i} \Rrightarrow i\ \}$
```

Figure 5.9: Proof sketch for the `listReverse` program.

Definition 5.10 (Heap Update Commands)**.** The set of *heap update commands* $\textsc{Cmd}_{\mathbb{H}}$ is defined as:

$$\begin{array}{rll} \textsc{Cmd}_{\mathbb{H}} ::= & \mathtt{x} := \mathtt{alloc}(E) & \textit{allocation} \\ & \mathtt{dispose}(E, E') & \textit{disposal} \\ & [E] := E' & \textit{mutation} \\ & \mathtt{x} := [E] & \textit{lookup} \end{array}$$

where $\mathtt{x} \in \textsc{Var}$ ranges over program variables and $E, E' \in \textsc{Expr}$ range over value expressions.

The intuitive meaning of these commands, which will be realised by their axiomatics semantics, is as follows:

- $\mathtt{x} := \mathtt{alloc}(E)$ allocates a contiguous block of cells in the heap of length E, returning the address of the first cell in $\mathtt{x}$. Requires that E evaluates to a positive integer or it faults;
- $\mathtt{dispose}(E, E')$ deallocates a contiguous block of cells in the heap at address E of length E'. Requires that all cells in the range E to $E + E'$ exist or it faults;
- $[E] := E'$ stores the value E' in the heap cell at address E. Requires that E identifies a cell that exists or it faults; and
- $\mathtt{x} := [E]$ loads the contents of the heap cell at address E into $\mathtt{x}$. Requires that E identifies a cell that exists or it faults.

We have already seen the heap segment algebra $\mathcal{S}(\mathcal{M}_{\mathrm{H}})$ in Example 3.46. In this model the cells that have values specified in the heap are the resources available to the program. Loads and stores can only be performed on heap cells that are available to the program; allocation makes new heap cells available; and deallocation makes available heap cells unavailable.

As discussed in Example 3.34, addresses and hole labels are used to track the parts of the heap when we break them apart. This gives us a way of logically identifying arbitrary portions of the heap. Due to the associativity and commutativity of disjoint heap union, heaps can be considered to have an arbitrary hole at their end. This uniformity of the structure is what allows separation logic to work without tracking labels. However, we choose to be more explicit with these labels so that all of our data structures are defined in a uniform style.

$$\{\ \mathtt{x} \Rightarrow v * \sigma \wedge w \geq 1 \wedge \mathcal{E}[\![E]\!]\sigma[\mathtt{x} \mapsto v] = w\ \}$$
$$\mathtt{x} := \mathtt{alloc}(E)$$
$$\{\ \exists y.\, \lceil y \mapsto - \star \ldots \star y + w \mapsto - \rceil * \mathtt{x} \Rightarrow y * \sigma\ \}$$

$$\{\ \alpha \leftarrow (w \mapsto - \star \ldots \star w + v \mapsto -) * \sigma \wedge \mathcal{E}[\![E]\!]\sigma = w \wedge \mathcal{E}[\![E']\!]\sigma = v\ \}$$
$$\mathtt{dispose}(E, E')$$
$$\{\ \alpha \leftarrow \mathsf{emp} * \sigma\ \}$$

$$\{\ \alpha \leftarrow (w \mapsto -) * \sigma \wedge \mathcal{E}[\![E]\!]\sigma = w \wedge \mathcal{E}[\![E']\!]\sigma = v\ \}$$
$$[E] := E'$$
$$\{\ \alpha \leftarrow (w \mapsto v) * \sigma\ \}$$

$$\{\ \alpha \leftarrow (w \mapsto y) * \mathtt{x} \Rightarrow v * \sigma \wedge \mathcal{E}[\![E]\!]\sigma[\mathtt{x} \mapsto v] = w\ \}$$
$$\mathtt{x} := [E]$$
$$\{\ \alpha \leftarrow (w \mapsto y) * \mathtt{x} \Rightarrow y * \sigma\ \}$$

Figure 5.10: Small axioms for the heap module.

Definition 5.11 (Heap Axiomatisation)**.** The *heap axiomatisation*

$$\mathrm{Ax}[\![(\cdot)]\!]_{\mathrm{H}} : \textsc{Cmd}_{\mathrm{H}} \to \mathcal{P}(\textsc{Pred}_{\mathrm{H}} \times \textsc{Pred}_{\mathrm{H}})$$

is given in Figure 5.10.

Notation: We lift the shorthand $\lceil ch \rceil$ to predicates, writing $\lceil P_H \rceil$ for $\mathsf{H}\alpha.\,(\alpha \leftarrow P_H)$.

Recall, from Example 3.46, that $\{(0, ch_1)\} + \{(0, ch_2)\} = \{(0, ch_1 \star ch_2)\}$. It follows that $\lceil P_H \rceil * \lceil Q_H \rceil = \lceil P_H \star Q_H \rceil$. Thus, when we work with root level heaps (heaps without abstract addresses), our segment logic reasoning resembles the corresponding separation logic reasoning.

From our segment based axioms, we can recover the axioms in the separation logic style by use of the revelation frame rule and the rule of consequence. As an example, consider the derivation for the heap assignment axiom given in Figure 5.11. We can recover the other separation logic style axioms from our segment logic axioms in a similar fashion.

Note that we could use a similar treatment as for heaps above to model the variable store as a segment algebra. This would allow us to reason about regions in the variable store. However, we have not found a need to think about the variable store in this way. The variable store is also the only component that is the same in each of our program modules, so we have chosen to work with a simplified model.

$$\dfrac{\dfrac{\begin{array}{l}\{\ \alpha{\leftarrow}(w \mapsto -) * \sigma \land \mathcal{E}\llbracket E\rrbracket\sigma = w \land \mathcal{E}\llbracket E'\rrbracket\sigma = v\ \}\\ [E] := E'\\ \{\ \alpha{\leftarrow}(w \mapsto v) * \sigma\ \}\end{array}}{\begin{array}{l}\{\ \mathsf{H}\alpha.\,(\alpha{\leftarrow}(w \mapsto -)) * \sigma \land \mathcal{E}\llbracket E\rrbracket\sigma = w \land \mathcal{E}\llbracket E'\rrbracket\sigma = v\ \}\\ [E] := E'\\ \{\ \mathsf{H}\alpha.\,(\alpha{\leftarrow}(w \mapsto v)) * \sigma\ \}\end{array}}\ \textsc{Rev Frame/Fresh}}{\begin{array}{l}\{\ \lceil w \mapsto - \rceil * \sigma \land \mathcal{E}\llbracket E\rrbracket\sigma = w \land \mathcal{E}\llbracket E'\rrbracket\sigma = v\ \}\\ [E] := E'\\ \{\ \lceil w \mapsto v \rceil * \sigma\ \}\end{array}}\ \textsc{Cons}$$

Figure 5.11: Deriving separation logic axioms from segment logic axioms.

5.4.1 Heap Reasoning Examples

Even though we have a more complex model, our heap reasoning still closely resembles that of separation logic. We give a couple of simple examples that illustrate this.

Notation: We make use of the standard binary cons cell notation $x \mapsto a{,}b$ which stands for $x \mapsto a * x + 1 \mapsto b$.

Example 5.12 (Simple Heap Update)**.** As a simple illustration of heap reasoning, consider the following heap update program that uses allocation and mutation to construct a two-element cyclic structure containing relative addresses:

$$\begin{array}{lcl}\texttt{smallList}(\mathtt{x},\mathtt{y}) & ::= & \mathtt{x} := \texttt{alloc}(2)\ ;\\ & & \mathtt{y} := \texttt{alloc}(2)\ ;\\ & & [\mathtt{x}+1] := \mathtt{y} - \mathtt{x}\ ;\\ & & [\mathtt{y}+1] := \mathtt{x} - \mathtt{y}\ ;\end{array}$$

The behaviour of the `smallList` program can be specified as follows:

$$\begin{array}{c}\left\{\ \mathtt{x} \Rrightarrow - * \mathtt{y} \Rrightarrow -\ \right\}\\ \texttt{smallList}(\mathtt{x},\mathtt{y})\\ \left\{\ \exists x, o.\ \lceil x \mapsto -{,}o \star (x+o) \mapsto -{,}(-o) \rceil * \mathtt{x} \Rrightarrow x * \mathtt{y} \Rrightarrow (x+o)\ \right\}\end{array}$$

The proof sketch in Figure 5.12 shows that this specification does indeed hold for the `smallList` program.

$$
\begin{array}{l}
\{\ \mathtt{x} \Rrightarrow - * \mathtt{y} \Rrightarrow -\ \}\\
\mathtt{x} := \mathtt{alloc}(2)\ ;\\
\{\ \exists x.\, \lceil x \mapsto -,- \rceil * \mathtt{x} \Rrightarrow x * \mathtt{y} \Rrightarrow -\ \}\\
\mathtt{y} := \mathtt{alloc}(2)\ ;\\
\{\ \exists x, y.\, \lceil x \mapsto -,- \star y \mapsto -,- \rceil * \mathtt{x} \Rrightarrow x * \mathtt{y} \Rrightarrow y\ \}\\
[\mathtt{x}+1] := \mathtt{y} - \mathtt{x}\ ;\\
\{\ \exists x, y.\, \lceil x \mapsto -,(y-x) \star y \mapsto -,- \rceil * \mathtt{x} \Rrightarrow x * \mathtt{y} \Rrightarrow y\ \}\\
[\mathtt{y}+1] := \mathtt{x} - \mathtt{y}\ ;\\
\{\ \exists x, y.\, \lceil x \mapsto -,(y-x) \star y \mapsto -,(x-y) \rceil * \mathtt{x} \Rrightarrow x * \mathtt{y} \Rrightarrow y\ \}\\
\{\ \exists x, o.\, \lceil x \mapsto -,o \star (x+o) \mapsto -,(-o) \rceil * \mathtt{x} \Rrightarrow x * \mathtt{y} \Rrightarrow (x+o)\ \}
\end{array}
$$

Figure 5.12: Proof sketch for the `smallList` program.

Example 5.13 (Abstract Predicates)**.** Our fine-grained abstract reasoning system still permits the use of abstract predicates. Consider the following program for recursively deleting a singly-linked list:

```
delList(x)  ::=  local y in
                   if x ≠ null then
                     y := [x + 1] ;
                     dispose(x, 2) ;
                     delList(y)
                   else skip
```

We can specify the behaviour of this program in terms of an abstract list predicate $\mathsf{list}(i)$ which is defined as:

$$\mathsf{list}(i) \overset{\text{def}}{=} (i = \mathsf{null} \wedge \mathsf{emp}) \vee (\exists j.\, \lceil i \mapsto -,j \rceil * \mathsf{list}(j))$$

This abstract predicate describes a list of binary cons cells in memory, with arbitrary contents in their first cell. The specification for the `delList` program can then be given as:

$$
\begin{array}{c}
\{\ \mathsf{list}(i) * \mathtt{x} \Rrightarrow i\ \}\\
\mathtt{delList}(\mathtt{x})\\
\{\ \mathtt{x} \Rrightarrow i\ \}
\end{array}
$$

Assuming that this specification holds for the recursive call, Figure 5.13 shows that this specification holds for the whole program. Notice how the abstract predicate is unfolded by one step at each pass through the recursive call. The base case of the

```
$\{ \ \mathsf{list}(i) * \mathsf{x} \Rrightarrow i \ \}$
local y in
   $\{ \ \mathsf{list}(i) * \mathsf{x} \Rrightarrow i * \mathsf{y} \Rrightarrow - \ \}$
   $\{ \ (i = \mathsf{null} \wedge \mathsf{emp}) \vee (\exists j. \lceil i \mapsto -,j \rceil * \mathsf{list}(j)) * \mathsf{x} \Rrightarrow i * \mathsf{y} \Rrightarrow - \ \}$
   if x $\neq$ null then
      $\{ \ \exists j. \lceil i \mapsto -,j \rceil * \mathsf{list}(j) * \mathsf{x} \Rrightarrow i * \mathsf{y} \Rrightarrow - \ \}$
      $\mathsf{y} := [\mathsf{x}+1]$ ;
      $\{ \ \exists j. \lceil i \mapsto -,j \rceil * \mathsf{list}(j) * \mathsf{x} \Rrightarrow i * \mathsf{y} \Rrightarrow j \ \}$
      dispose(x, 2) ;
      $\{ \ \exists j. \mathsf{list}(j) * \mathsf{x} \Rrightarrow i * \mathsf{y} \Rrightarrow j \ \}$
      delList(y)
      $\{ \ \exists j. \mathsf{x} \Rrightarrow i * \mathsf{y} \Rrightarrow j \ \}$
      $\{ \ \mathsf{x} \Rrightarrow i * \mathsf{y} \Rrightarrow - \ \}$
   else
      $\{ \ i = \mathsf{null} \wedge \mathsf{emp} * \mathsf{x} \Rrightarrow i * \mathsf{y} \Rrightarrow - \ \}$
      skip
      $\{ \ i = \mathsf{null} \wedge \mathsf{emp} * \mathsf{x} \Rrightarrow i * \mathsf{y} \Rrightarrow - \ \}$
      $\{ \ \mathsf{x} \Rrightarrow i * \mathsf{y} \Rrightarrow - \ \}$
   $\{ \ \mathsf{x} \Rrightarrow i * \mathsf{y} \Rrightarrow - \ \}$
$\{ \ \mathsf{x} \Rrightarrow i \ \}$
```

Figure 5.13: Proof sketch for the `delList` program.

induction is covered by the else branch of the `if-then-else` statement. The other branch of the `if-then-else` statement covers the inductive step.

5.5 Fine-grained DOM Module

Probably the most notable use of abstract local reasoning to date has been the formal specification of the W3C Document Object Model (or DOM). In joint work with Gardner, Smith and Zarfaty [33][34], I helped to identify, and formally specify, a core subset of the DOM commands for manipulating the tree-like structure of DOM. However, as mentioned before, we were not able to provide a small axiom for the `appendChild` command.

Having developed a fine-grained abstract local reasoning system, it would seem pertinent to return to DOM which motivated this work in the first place. In our previous work on DOM we chose to focus on a minimal subset of the DOM Core Level 1 tree update commands. In his thesis [64] Smith extended our work to cover all of

DOM Core Level 1. In this section we provide an abstract module for featherweight DOM $\mathbb{D} = (\textsc{Cmd}_{\mathbb{D}}, \mathcal{S}(\mathcal{M}_{\mathbb{D}}), \textsc{Ax}[\![(\cdot)]\!]_{\mathbb{D}})$. This work could be extended to cover all of DOM Core Level 1 using similar techniques to those in Smith's thesis.

The data structure presented in the DOM specification [68] is significantly more complex than that of a simple tree structure. In our previous work on providing a formal specification for the DOM specification [34], we made the decision to focus on the basic XML tree structure of the DOM specification, with simple text content. Our abstract data structure consisted of trees, forests, groves and strings. Trees t corresponded to part of the Node interface. Forests f were lists of trees and corresponded to sub-collections of the NodeList interface. Complete forests $[f]_{fid}$ with identifier fid corresponded directly to the NodeList interface. Groves g were sets of rooted trees and corresponded to the object store in which Nodes exist. Strings *str* corresponded to the DOMString type from the DOM specification.

The DOM specification updates data in place. This means that we must be able to refer to subdata directly. Each node and child list must therefore have a unique identifier which can be directly referenced by programs through program variables. We assume we have a countable infinite set of identifiers $\textsc{Id}$ and a finite set $\textsc{Ch} = \{\mathsf{a}, \mathsf{b}, \mathsf{c}, ...\}$ of text characters with a distinguished character $\mathtt{\#}$. We assume that expressions are extended to include *string expressions* $\textsc{SExpr}$ ranged over by $\mathtt{S}$, $\mathtt{S}_1$, *etc*, and also that program variables are able to store strings. As with trees, we work with the identifier set $\textsc{Id}_{\mathsf{null}} \stackrel{\text{def}}{=} \textsc{Id} \cup \{\mathsf{null}\}$ consisting of all valid identifiers and the null reference.

The featherweight DOM module represents the essence of the Node view of the DOM API with a minimal and sufficient set of update commands. We present the library in an imperative fashion, abandoning object orientated notation, to simplify the presentation and reuse our existing fine-grained abstract reasoning framework. Each method of the Node interface is, therefore, specified as an imperative command over the working grove. For example, the method call $\mathtt{E}$.`appendChild`($\mathtt{E}'$) from the DOM specification becomes the command `appendChild`($\mathtt{E}, \mathtt{E}'$) in our module.

We are only interested in the read-only properties of DOM nodes, so we represent each of these with a `get` command. For example, the $\mathtt{E}$.`parentNode` attribute is represented by the `getParentNode`($\mathtt{E}$) command. If we wanted to consider attributes that were not read only these would be represented by a pair of `get` and `set` commands, with the `set` commands defined in a similar fashion.

We omit some of the Node interface attributes and methods either because they are not concerned with the tree or text structure, or because they are redundant and may be expressed as the composition of other commands. For example, `insertBefore` can be implemented in terms of iterated `appendChild`.

In the DOM Module the `removeChild`(E, E') command does not delete the tree at $\mathtt{E}'$, it just moves it to the root level of the grove. It is not possible to a delete a tree from the grove in Featherweight DOM. Instead, it is natural to think of programs written in Featherweight DOM as being garbage collected programs. This follows the DOM specification, which deliberately declines to specify any destructive memory management methods in order to leave open the question of whether memory should be managed manually or be garbage collected. This choice is one of several that ensure that the DOM specification remains implementation independent.

Finally, we observe that neither the Node or NodeList interface provides a means of creating new nodes in the grove. However, the Document interface provides does provide the commands `createElement` and `createTextNode` with this functionality. We do not want to consider the full complexity of the Document interface and Element nodes, so we choose to add two new commands, `createNode`(S) and `createTextNode`(S), to featherweight DOM. These allow us to create new nodes and new text nodes respectively.

Definition 5.14 (Featherweight DOM Update Commands)**.** The set of *featherweight DOM update commands* $\textsc{Cmd}_{\mathbb{D}}$ is defined as:

$\textsc{Cmd}_{\mathbb{D}}$ ::=	$\mathtt{x} :=$ `createNode`(S)	*new element node*
	$\mathtt{x} :=$ `getNodeName`(E)	*get node name*
	$\mathtt{x} :=$ `getParentNode`(E)	*get parent node*
	$\mathtt{x} :=$ `getChildNodes`(E)	*get children*
	$\mathtt{x} :=$ `item`(E, E')	*get forest element*
	`appendChild`(E, E')	*append tree*
	`removeChild`(E, E')	*remove tree*
	$\mathtt{x} :=$ `createTextNode`(S)	*new text node*
	$\mathtt{x} :=$ `substringData`(E, E', E'')	*get substring*
	`appendData`(E, S)	*append string*
	`deleteData`(E, E', E'')	*erase substring*

The command names are chosen to match those of the existing DOM specification [68]. The intuitive meaning of these commands, which will be realised by their axiomatic semantics, is as follows:

◇ $\mathtt{x} :=$ `createNode`(S) creates a new element node at the root level of the grove, with its nodeName set to S, fresh node identifier i and fresh forest identifier

j, and assigns this identifier i to the program variable `x`. Requires that S is a valid element name (S does not contain the `#` character).

- ◇ `x` := `getNodeName`(E) assigns to the program variable `x` the nodeName value of the node identified by E, or `#text` if E identifies a text node. Requires that E identifies a node that exists or it faults.

- ◇ `x` := `getParentNode`(E) assigns to the program variable `x` the identifier of the parent of the node identified by E, if it exists, and null otherwise. Requires that E identifies a node that exists or it faults.

- ◇ `x` := `getChildNodes`(E) assigns to the program variable `x` the identifier of the child forest of the node identified by E. Requires that E identifies a node that exists and is not a text node, or it faults.

- ◇ `x` := `item`(E, E') assigns to the program variable `x` the identifier of the $(E' + 1)$th node in the child list identified by E, setting it to null if $(E'+1)$ evaluates to an invalid index. Requires that `E` identifies a child list that exists or it faults.

- ◇ `appendChild`(E, E') moves the subtree at the node identified by E' to the end of the child list of the node identified by E. Requires that E identifies a node that exists and is not a text node, and that E' identifies a node that exists and is not an ancestor of the node identified by E, or it faults.

- ◇ `removeChild`(E, E') removes the subtree at the node identified by E' from the child list of the node identified by E and re-inserts it as a separate DOM tree at the grove level. Requires that E identifies a node that exists and E' identifies a node that is a child of the node identified by E, or it faults.

- ◇ `x` := `createTextNode`(S) creates a new text node at the root level of the grove, with fresh identifier i, which contains the string S, and assigns this identifier i to the program variable `x`. Requires that S is a valid string (does not contain any illegal characters) or it faults.

- ◇ `x` := `substringData`(E, E', E'') assigns to the program variable `x` the substring of length E'' from the string of the text node identified by E starting at the E'th character. If $E' + E''$ exceeds the string length, then all the characters from the E'th character to the string end are returned. Requires that E identifies a text node that exists, E' and E'' be non-negative integers and E' be at most the string length, or it faults.

◇ $\texttt{appendData}(\texttt{E}, \texttt{S})$ appends the string $\texttt{S}$ to the end of the string contained in the text node identified by $\texttt{E}$. Requires that $\texttt{E}$ identifies a text node that exists or it faults.

◇ $\texttt{deleteData}(\texttt{E}, \texttt{E}', \texttt{E}'')$ deletes the substring of the string of text node identified by $\texttt{E}$ starting at the $\texttt{E}'$th character with length $\texttt{E}''$. If $\texttt{E}' + \texttt{E}''$ exceeds the string length, then all the characters from the $\texttt{E}'$th character to the string end are deleted. Requires that $\texttt{E}$ identifies a text node that exists, $\texttt{E}'$ and $\texttt{E}''$ be non-negative integers and $\texttt{E}'$ be at most the string length, or it faults.

Multi-Holed DOM Tree Context Algebra

We now give the data structure for our abstract DOM module, starting with the definition of a multi-holed DOM tree context algebra and then using this to define a DOM segment algebra.

DOM makes use of strings in both node labels and the contents of text nodes. The set of strings S_{CH}, ranged over by $str, str_1, ...$, is defined inductively as:

$$str \quad ::= \quad \varepsilon \mid \texttt{c} \mid str : str$$

where ε is the empty string, characters $\texttt{c} \in \mathrm{CH}$ and string concatenation : is associative with identity ε.

We now provide the DOM tree context structure that we shall later use to build up our DOM segments. In our previous work on DOM we found it necessary to give a context structure for trees, forest and groves. However, if we treat groves as sets of rooted trees, then it is enough for us to work with a more traditional tree structure, similar to that encountered in §5.2

The multi-holed DOM tree context algebra is defined by $\mathcal{M}_{\mathbb{D}} = (\mathrm{D}_{\mathrm{ID},\mathrm{X}}, \mathrm{X}, fh_{\mathbb{D}}, \bullet)$ where,

◇ the set of multi-holed DOM tree contexts $\mathrm{D}_{\mathrm{ID},\mathrm{X}}$, ranged over by $cdt, cdt_1, ...$, is defined inductively as:

$$cdt \quad ::= \quad \varnothing \mid x \mid str_i[cdt]_j \mid \texttt{\#text}_i[str] \mid cdt \otimes cdt$$

with the restriction that hole labels $x \in \mathrm{X}$ and identifiers $i, j \in \mathrm{ID}$ occur at most once in a DOM tree context cdt, and the assumption that $\otimes$ is associative with identity $\varnothing$.

◇ the free holes function

$$fh_{\mathbb{D}} : \mathrm{D}_{\mathrm{ID},\mathrm{X}} \to \mathcal{P}_{\mathsf{fin}}(\mathrm{X})$$

is defined by induction on the structure of multi-holed DOM tree contexts as:

$$\begin{array}{rcl}
fh_{\mathbb{D}}(\varnothing) & \overset{\text{def}}{=} & \emptyset \\
fh_{\mathbb{D}}(x) & \overset{\text{def}}{=} & \{x\} \\
fh_{\mathbb{D}}(str_i[cdt]_j) & \overset{\text{def}}{=} & fh_{\mathbb{D}}(cdt) \\
fh_{\mathbb{D}}(\texttt{\#text}_i[str]) & \overset{\text{def}}{=} & \emptyset \\
fh_{\mathbb{D}}(cdt_1 \otimes cdt_2) & \overset{\text{def}}{=} & fh_{\mathbb{D}}(cdt_1) \cup fh_{\mathbb{D}}(cdt_2)
\end{array}$$

◇ the context composition operator

$$\bullet : \mathrm{X} \times \mathrm{D}_{\mathrm{ID},\mathrm{X}} \times \mathrm{D}_{\mathrm{ID},\mathrm{X}} \rightharpoonup \mathrm{D}_{\mathrm{ID},\mathrm{X}}$$

is defined by induction on the structure of multi-holed DOM tree contexts as:

$$\begin{array}{rcl}
\varnothing \bullet_x cdt & \overset{\text{def}}{=} & \text{undefined} \\
y \bullet_x cdt & \overset{\text{def}}{=} & \begin{cases} cdt & \text{if } y = x \\ \text{undefined} & \text{otherwise} \end{cases} \\
str_i[cdt']_j \bullet_x cdt & \overset{\text{def}}{=} & \begin{cases} str_i[cdt' \bullet_x cdt]_j & \text{if } x \in fh_{\mathbb{D}}(cdt') \\ \text{undefined} & \text{otherwise} \end{cases} \\
\texttt{\#text}_i[str] \bullet_x cdt & \overset{\text{def}}{=} & \text{undefined} \\
(cdt_1 \otimes cdt_2) \bullet_x cdt & \overset{\text{def}}{=} & \begin{cases} (cdt_1 \bullet_x cdt) \otimes cdt_2 & \text{if } x \in fh_{\mathbb{D}}(cdt_1) \\ cdt_1 \otimes (cdt_2 \bullet_x cdt) & \text{if } x \in fh_{\mathbb{D}}(cdt_2) \\ \text{undefined} & \text{otherwise} \end{cases}
\end{array}$$

DOM trees are built out of two different types of nodes: *element* nodes $str_i[cdt]_j$ and *text* nodes $\texttt{\#text}_i[str]$. Element nodes contain a list of their children while text nodes contain a single string. Both types of node have a unique identifier i which allows for direct access to that node. Element nodes additionally have a child list identifier j which allows direct access to their children. In particular, the DOM specification provides a command called `getChildNodes` which returns a pointer to a node's child list. Both types of node have a node label. In the case of text nodes this is always the string `#text` which is prefixed with the distinguished character `#`. The node label of an element node is a string which must node include the `#` character. In the full DOM specification there are other special node labels which are also prefixed with the `#` character.

Notation: We use a shorthand for strings writing `abc` for `a : b : c`. We write $\textsc{EltNames} \subset \mathrm{S}_{\mathrm{CH}}$ to denote the set of strings without `#`.

Example 5.15 (Featherweight DOM Data Structure)**.** As an example of our data structure, consider the following XML structure:

```
⟨student⟩
  ⟨name⟩Joe Bloggs⟨/name⟩
  ⟨year⟩2007⟨/year⟩
  ⟨course⟩Computing⟨/course⟩
⟨/student⟩
```

If this XML is passed into DOM then, for some choice of identifiers, we would have the DOM tree:

$$\begin{array}{l} \texttt{student}_{i_1}[\\ \quad \texttt{name}_{i_2}[\texttt{\#text}_{i_5}[\texttt{Joe Bloggs}]]_{j_2} \\ \quad \otimes\, \texttt{year}_{i_3}[\texttt{\#text}_{i_6}[\texttt{2007}]]_{j_3} \\ \quad \otimes\, \texttt{course}_{i_4}[\texttt{\#text}_{i_7}[\texttt{Computing}]]_{j_4} \\ ]_{j_1} \end{array}$$

DOM Segment Algebra

We base our DOM segments on the multi-holed DOM tree context algebra $\mathcal{M}_{\mathbb{D}}$ given above. Informally, DOM segments can be thought of as sets of labelled DOM tree contexts, where labels can either be some $x \in \mathrm{X}$, or the special empty label 0, which indicates that a DOM tree context is rooted at the grove level. A grove-rooted DOM tree context is required to be a single tree node with no parent node. We write X_0 for the set of labels X extended with the special empty label 0. Note that 0 can not occur as a hole label, but it may occur as the address of more than one DOM tree context. The DOM data structure allows for there to be an unordered collection (or bag) of DOM trees rooted at the grove level.

The DOM segment algebra is defined by $\mathcal{S}(\mathcal{M}_{\mathbb{D}}) = (\mathrm{S}_{\mathbb{D}}, \emptyset, \leftarrow, fa_{\mathbb{D}}, fh_{\mathbb{D}}, +, \mathsf{comp}_{\mathbb{D}})$ where,

◇ the set of DOM segments $\mathrm{S}_{\mathbb{D}}$, ranged over by $sd, sd_1, ...$, is defined inductively as:

$$sd \ ::= \ \emptyset \mid \{(x, cdt)\} \mid sd \uplus sd$$

with DOM tree contexts $cdt \in \mathrm{D}_{\mathrm{ID},\mathrm{X}}$ and address labels $x \in \mathrm{X}_0$, the restriction that non-zero addresses, hole labels and identifiers are unique across the set

sd and the restriction that for each $\{(x, cdt)\} \in sd$, $x \notin fv_{\mathbb{D}}(cdt)$ and if $x = 0$, then there exist cdt', str, i and j such that $cdt = str_i[cdt']_j$; that is, grove-rooted trees have a single root node. The disjoint union of DOM segments $\uplus$ is only defined when the segments have disjoint addresses and contexts. The operation is both associative and commutative.

Notation: In a similar style to tree segments, we write $\lceil cdt \rceil$ as shorthand for $\{(0, cdt)\}$.

◇ the context addressing function

$$\leftarrow \, : \mathrm{X}_0 \times \mathrm{D}_{\mathrm{I_D},\mathrm{X}} \rightharpoonup \mathrm{S}_{\mathbb{D}}$$

is defined as:

$$x \leftarrow cdt \overset{\text{def}}{=} \begin{cases} \{(x, cdt)\} & \text{if } x \notin fh_{\mathbb{D}}(cdt) \\ \text{undefined} & \text{otherwise} \end{cases}$$

◇ the free addresses function

$$fa_{\mathbb{D}} : \mathrm{S}_{\mathbb{D}} \to \mathcal{P}_{\mathsf{fin}}(\mathrm{X})$$

is defined by induction on the structure of DOM segments as:

$$\begin{aligned} fa_{\mathbb{D}}(\emptyset) &\overset{\text{def}}{=} \emptyset \\ fa_{\mathbb{D}}(\{(x, cdt)\}) &\overset{\text{def}}{=} \begin{cases} \emptyset & \text{if } x = 0 \\ \{x\} & \text{otherwise} \end{cases} \\ fa_{\mathbb{D}}(sd_1 \uplus sd_2) &\overset{\text{def}}{=} fa_{\mathbb{D}}(sd_1) \cup fa_{\mathbb{D}}(sd_2) \end{aligned}$$

◇ the free holes function

$$fh_{\mathbb{D}} : \mathrm{S}_{\mathbb{D}} \to \mathcal{P}_{\mathsf{fin}}(\mathrm{X})$$

is defined by induction on the structure of DOM segments as:

$$\begin{aligned} fh_{\mathbb{D}}(\emptyset) &\overset{\text{def}}{=} \emptyset \\ fh_{\mathbb{D}}(\{(x, cdt)\}) &\overset{\text{def}}{=} fh_{\mathbb{D}}(cdt) \\ fh_{\mathbb{D}}(sd_1 \uplus sd_2) &\overset{\text{def}}{=} fh_{\mathbb{D}}(sd_1) \cup fh_{\mathbb{D}}(sd_2) \end{aligned}$$

◇ the segment combination operator

$$+ : \mathrm{S}_{\mathbb{D}} \times \mathrm{S}_{\mathbb{D}} \rightharpoonup \mathrm{S}_{\mathbb{D}}$$

is defined as:

$$sd_1 + sd_2 \overset{\text{def}}{=} \begin{cases} sd_1 \uplus sd_2 & \text{if } fa_{\mathbb{D}}(sd_1) \cap fa_{\mathbb{D}}(sd_2) = \emptyset \\ & \text{and } fh_{\mathbb{D}}(sd_1) \cap fh_{\mathbb{D}}(sd_2) = \emptyset \\ \text{undefined} & \text{otherwise} \end{cases}$$

◇ the segment compression operator

$$\mathsf{comp}_{\mathbb{D}} : \mathrm{X} \times \mathrm{S}_{\mathbb{D}} \rightharpoonup \mathrm{S}_{\mathbb{D}}$$

is defined as:

$$\mathsf{comp}_{\mathbb{D}}(x, sd) \overset{\text{def}}{=} \begin{cases} sd & \text{if } x \notin fa_{\mathbb{D}}(sd) \text{ and } x \notin fh_{\mathbb{D}}(sd) \\ sd' + \{(z, cdt \bullet_x cdt')\} & \text{if } \exists sd', z, cdt, cdt'. \\ & sd = sd' + \{(z, cdt), (x, cdt')\} \\ & \text{and } x \in fh_{\mathbb{D}}(cdt) \\ sd' + \{(0, cdt)\} & \text{if } \exists sd', cdt.\, sd = sd' + \{(x, cdt)\} \\ & \text{and } x \notin fh_{\mathbb{D}}(sd') \\ \text{undefined} & \text{otherwise} \end{cases}$$

The DOM segment algebra $\mathcal{S}(\mathcal{M}_{\mathbb{D}})$ is quite similar to the tree segment algebra $\mathcal{S}(\mathcal{M}_{\mathrm{T}})$ from Example 3.44. However, we shall see that the grove-rooted trees play a more significant role in our specification of the fine-grained DOM module than the rooted trees did in the specification of the fine-grained tree module in §5.2. There are several featherweight DOM commands that directly manipulate the grove level of the DOM data structure.

Definition 5.16 (Featherweight DOM Axiomatisation)**.** The *featherweight DOM axiomatisation*

$$\mathrm{Ax}[\![(\cdot)]\!]_{\mathbb{D}} : \textsc{Cmd}_{\mathbb{D}} \to \mathcal{P}(\textsc{Pred}_{\mathbb{D}} \times \textsc{Pred}_{\mathbb{D}})$$

is given in Figure 5.14, Figure 5.15 and Figure 5.16.

Notation: We denote the set of DOM tree formulae as P_D. The DOM tree formulae are identical to the tree formulae P_T with the obvious addition of node names and forest identifiers to the structure. We lift the rooted tree shorthand $\lceil cdt \rceil$ to predicates, writing $\lceil P_D \rceil$ for $\mathsf{H}\alpha.\,(\alpha \leftarrow P_D)$. We write $|N|$ for the length of list N and similarly $|str|$ for the length of string str.

We choose to split our axioms into three separate sets, one for describing the behaviour of commands on element nodes, one for describing the behaviour of commands on text nodes and one for describing the behaviour of commands on text. Splitting up our axioms in this way leads to a larger axiom set, but simpler individual axioms. Unlike our previous work on DOM, we now have genuine small axioms for all of our basic commands, including `appendChild`.

The specification of the `item` command makes use of a predicate $\mathsf{Ls}(N)$ which describes a one-layer list of nodes N. This predicate allows us to capture the minimal amount of resource required in order to locally describe the behaviour of the command. The $\mathsf{Ls}(N)$ predicate is defined inductively in terms of N as follows:

$$\begin{array}{rcl} \mathsf{Ls}([\,]) & = & \varnothing \\ \mathsf{Ls}((str, \alpha, i, j) : N) & = & str_i[\alpha]_j \otimes \mathsf{Ls}(N) \\ \mathsf{Ls}((\texttt{\#text}, i, str) : N) & = & \texttt{\#text}_i[str] \otimes \mathsf{Ls}(N) \end{array}$$

It is interesting to note the use of the $\mathsf{tree}(P_D)$ predicate in our axioms. Recall that this predicate is used to indicate a subtree that is complete (contains no context holes). In our fine-grained tree module axiomatisation (§5.2) our tree deletion and tree move commands required a complete tree in their precondition. Similarly, in our featherweight DOM axiomatisation the `appendChild` command requires that we move a complete subtree in order to avoid introducing a loop in the data structure. However, notice that the `removeChild` command does not require a complete tree in its precondition, even though it is moving the whole subtree. Recall that the `removeChild` command does not delete a subtree, but instead just moves it to the top level of the grove. Thus, there is no chance of breaking the structure or creating a loop within the data structure, so we do not need to rule out these possibilities in the precondition. Knowing that the root of the subtree has moved is enough to infer that the rest of the tree has also moved with it, since the root node has the same context hole beneath it both before and after the execution of the command.

5.5.1 DOM Reasoning Examples

Even though the featherweight DOM model is more complex than the other models we have seen so far, reasoning about programs written in this module is still relatively simple. We consider a few examples here: one where we implement a simple DOM Core Level 1 command using featherweight DOM; one showing a more complex DOM Core Level 1 command implementation; and one where we show how to apply our reasoning techniques to proving schema invariants.

$$\left\{\ \mathtt{x} \Rrightarrow i * \sigma \land \mathcal{E}\llbracket \mathtt{S} \rrbracket \sigma[\mathtt{x} \mapsto i] = \mathit{str} \land \mathit{str} \in \textsc{EltNames}\ \right\}$$
$$\mathtt{x} := \mathtt{createNode}(\mathtt{S})$$
$$\left\{\ \exists i, j.\ \lceil \mathit{str}_i[\varnothing]_j \rceil * \mathtt{x} \Rrightarrow i * \sigma\ \right\}$$

$$\left\{\ \alpha \leftarrow \mathit{str}_i[\beta]_j * \mathtt{x} \Rrightarrow v * \sigma \land \mathcal{E}\llbracket \mathtt{E} \rrbracket \sigma[\mathtt{x} \mapsto v] = i\ \right\}$$
$$\mathtt{x} := \mathtt{getNodeName}(\mathtt{E})$$
$$\left\{\ \alpha \leftarrow \mathit{str}_i[\beta]_j * \mathtt{x} \Rrightarrow \mathit{str} * \sigma\ \right\}$$

$$\left\{\ \alpha \leftarrow \mathit{str}'_j[\beta \otimes \mathit{str}_i[\gamma]_{i2} \otimes \delta]_k * \mathtt{x} \Rrightarrow v * \sigma \land \mathcal{E}\llbracket \mathtt{E} \rrbracket \sigma[\mathtt{x} \mapsto v] = i\ \right\}$$
$$\mathtt{x} := \mathtt{getParentNode}(\mathtt{E})$$
$$\left\{\ \alpha \leftarrow \mathit{str}'_j[\beta \otimes \mathit{str}_i[\gamma]_{i2} \otimes \delta]_k * \mathtt{x} \Rrightarrow j * \sigma\ \right\}$$

$$\left\{\ \lceil \mathit{str}_i[\alpha]_j \rceil * \mathtt{x} \Rrightarrow v * \sigma \land \mathcal{E}\llbracket \mathtt{E} \rrbracket \sigma[\mathtt{x} \mapsto v] = i\ \right\}$$
$$\mathtt{x} := \mathtt{getParentNode}(\mathtt{E})$$
$$\left\{\ \lceil \mathit{str}_i[\alpha]_j \rceil * \mathtt{x} \Rrightarrow \mathsf{null} * \sigma\ \right\}$$

$$\left\{\ \alpha \leftarrow \mathit{str}_i[\beta]_j * \mathtt{x} \Rrightarrow v * \sigma \land \mathcal{E}\llbracket \mathtt{E} \rrbracket \sigma[\mathtt{x} \mapsto v] = i\ \right\}$$
$$\mathtt{x} := \mathtt{getChildNodes}(\mathtt{E})$$
$$\left\{\ \alpha \leftarrow \mathit{str}_i[\beta]_j * \mathtt{x} \Rrightarrow j * \sigma\ \right\}$$

$$\left\{ \begin{array}{l} \alpha \leftarrow \mathit{str}_i[\mathsf{Ls}(N) \otimes \mathit{str}'_k[\gamma]_{k2} \otimes \delta]_j * \mathtt{x} \Rrightarrow v * \sigma \\ \land\, \mathcal{E}\llbracket \mathtt{E} \rrbracket \sigma[\mathtt{x} \mapsto v] = j \land \mathcal{E}\llbracket \mathtt{E}' \rrbracket \sigma[\mathtt{x} \mapsto v] = |N| \end{array} \right\}$$
$$\mathtt{x} := \mathtt{item}(\mathtt{E}, \mathtt{E}')$$
$$\left\{\ \alpha \leftarrow \mathit{str}_i[\mathsf{Ls}(N) \otimes \mathit{str}'_k[\gamma]_{k2} \otimes \delta]_j * \mathtt{x} \Rrightarrow k * \sigma\ \right\}$$

$$\left\{ \begin{array}{l} \alpha \leftarrow \mathit{str}_i[\mathsf{Ls}(N)]_j * \mathtt{x} \Rrightarrow v * \sigma \land \mathcal{E}\llbracket \mathtt{E} \rrbracket \sigma[\mathtt{x} \mapsto v] = j \\ \land\, \mathcal{E}\llbracket \mathtt{E}' \rrbracket \sigma[\mathtt{x} \mapsto v] = i \land (i < 0 \lor i \geq |N|) \end{array} \right\}$$
$$\mathtt{x} := \mathtt{item}(\mathtt{E}, \mathtt{E}')$$
$$\left\{\ \alpha \leftarrow \mathit{str}_i[\mathsf{Ls}(N)]_j * \mathtt{x} \Rrightarrow \mathsf{null} * \sigma\ \right\}$$

$$\left\{\ \alpha \leftarrow \mathit{str}_i[\gamma]_j * \beta \leftarrow \mathit{str}'_k[\mathsf{tree}(\mathit{cdt})]_{k_2} * \sigma \land \mathcal{E}\llbracket \mathtt{E} \rrbracket \sigma = i \land \mathcal{E}\llbracket \mathtt{E} \rrbracket \sigma = k\ \right\}$$
$$\mathtt{appendChild}(\mathtt{E}, \mathtt{E}')$$
$$\left\{\ \alpha \leftarrow \mathit{str}_i[\gamma \otimes \mathit{str}'_k[\mathsf{tree}(\mathit{cdt})]_{k2}]_j * \beta \leftarrow \varnothing * \sigma\ \right\}$$

$$\left\{\ \alpha \leftarrow \mathit{str}_i[\beta \otimes \mathit{str}'_k[\gamma]_{k2} \otimes \delta]_j * \sigma \land \mathcal{E}\llbracket \mathtt{E} \rrbracket \sigma = i \land \mathcal{E}\llbracket \mathtt{E}' \rrbracket \sigma = k\ \right\}$$
$$\mathtt{removeChild}(\mathtt{E}, \mathtt{E}')$$
$$\left\{\ \alpha \leftarrow \mathit{str}_i[\beta \otimes \delta]_j * \lceil \mathit{str}'_k[\gamma]_{k2} \rceil * \sigma\ \right\}$$

Figure 5.14: Featherweight DOM axioms for element node manipulation.

$$\left\{\ \alpha \leftarrow \texttt{\#text}_i[str] * \texttt{x} \Rrightarrow v * \sigma \land \mathcal{E}[\![E]\!]\sigma[\texttt{x} \mapsto v] = i\ \right\}$$
$$\texttt{x} := \texttt{getNodeName}(E)$$
$$\left\{\ \alpha \leftarrow \texttt{\#text}_i[str] * \texttt{x} \Rrightarrow \texttt{\#text} * \sigma\ \right\}$$

$$\left\{\ \alpha \leftarrow str'_j[\beta \otimes \texttt{\#text}_i[str] \otimes \gamma]_k * \texttt{x} \Rrightarrow v * \sigma \land \mathcal{E}[\![E]\!]\sigma[\texttt{x} \mapsto v] = i\ \right\}$$
$$\texttt{x} := \texttt{getParentNode}(E)$$
$$\left\{\ \alpha \leftarrow str'_j[\beta \otimes \texttt{\#text}_i[str] \otimes \gamma]_k * \texttt{x} \Rrightarrow j * \sigma\ \right\}$$

$$\left\{\ \lceil \texttt{\#text}_i[str] \rceil * \texttt{x} \Rrightarrow v * \sigma \land \mathcal{E}[\![E]\!]\sigma[\texttt{x} \mapsto v] = i\ \right\}$$
$$\texttt{x} := \texttt{getParentNode}(E)$$
$$\left\{\ \lceil \texttt{\#text}_i[str] \rceil * \texttt{x} \Rrightarrow \mathsf{null} * \sigma\ \right\}$$

$$\left\{\begin{array}{l} \alpha \leftarrow str_i \left[\ \mathsf{Ls}(N) \otimes \texttt{\#text}_k[str'] \otimes \delta\ \right]_j * \texttt{x} \Rrightarrow v * \sigma \\ \land\, \mathcal{E}[\![E]\!]\sigma[\texttt{x} \mapsto v] = j \land \mathcal{E}[\![E']\!]\sigma[\texttt{x} \mapsto v] = |N| \end{array}\right\}$$
$$\texttt{x} := \texttt{item}(E, E')$$
$$\left\{\ \alpha \leftarrow str_i[\mathsf{Ls}(N) \otimes \texttt{\#text}_k[str'] \otimes \delta]_j * \texttt{x} \Rrightarrow k * \sigma\ \right\}$$

$$\left\{\ \alpha \leftarrow str_i[\gamma]_j * \beta \leftarrow \texttt{\#text}_k[str'] * \sigma \land \mathcal{E}[\![E]\!]\sigma = i \land \mathcal{E}[\![E]\!]\sigma = k\ \right\}$$
$$\texttt{appendChild}(E, E')$$
$$\left\{\ \alpha \leftarrow str_i[\gamma \otimes \texttt{\#text}_k[str']]_j * \beta \leftarrow \varnothing * \sigma\ \right\}$$

$$\left\{\ \alpha \leftarrow str_i[\beta \otimes \texttt{\#text}_k[str'] \otimes \delta]_j * \sigma \land \mathcal{E}[\![E]\!]\sigma = i \land \mathcal{E}[\![E']\!]\sigma = k\ \right\}$$
$$\texttt{removeChild}(E, E')$$
$$\left\{\ \alpha \leftarrow str_i[\beta \otimes \delta]_j * \lceil \texttt{\#text}_k[str'] \rceil * \sigma\ \right\}$$

$$\left\{\ \texttt{x} \Rrightarrow i * \sigma \land \mathcal{E}[\![S]\!]\sigma[\texttt{x} \mapsto i] = str\ \right\}$$
$$\texttt{x} := \texttt{createTextNode}(S)$$
$$\left\{\ \exists i.\ \lceil \texttt{\#text}_i[str] \rceil * \texttt{x} \Rrightarrow i * \sigma\ \right\}$$

Figure 5.15: Featherweight DOM axioms for text node manipulation.

$$\left\{\begin{array}{l} \alpha{\leftarrow}\texttt{\#text}_i[str_1 : str : str_2] * \texttt{x} \Rightarrow v * \sigma \wedge \mathcal{E}[\![E]\!]\sigma[\texttt{x} \mapsto v] = i \\ \wedge\, \mathcal{E}[\![E']\!]\sigma[\texttt{x} \mapsto v] = |str_1| \wedge \mathcal{E}[\![E'']\!]\sigma[\texttt{x} \mapsto v] = |str| \end{array}\right\}$$
$$\texttt{x} := \texttt{substringData}(E, E', E'')$$
$$\left\{\ \alpha{\leftarrow}\texttt{\#text}_i[str_1 : str : str_2] * \texttt{x} \Rightarrow str * \sigma'\ \right\}$$

$$\left\{\begin{array}{l} \alpha{\leftarrow}\texttt{\#text}_i[str_1 : str] * \texttt{x} \Rightarrow s * \sigma \wedge \mathcal{E}[\![E]\!]\sigma[\texttt{x} \mapsto s] = i \\ \wedge\, \mathcal{E}[\![E']\!]\sigma[\texttt{x} \mapsto s] = |str_1| \wedge \mathcal{E}[\![E'']\!]\sigma[\texttt{x} \mapsto s] > |str| \end{array}\right\}$$
$$\texttt{x} := \texttt{substringData}(E, E', E'')$$
$$\left\{\ \alpha{\leftarrow}\texttt{\#text}_i[str_1 : str] * \texttt{x} \Rightarrow str * \sigma'\ \right\}$$

$$\left\{\ \alpha{\leftarrow}\texttt{\#text}_i[str] * \sigma \wedge \mathcal{E}[\![E]\!]\sigma = i \wedge \mathcal{E}[\![S]\!]\sigma = str'\ \right\}$$
$$\texttt{appendData}(E, S)$$
$$\left\{\ \alpha{\leftarrow}\texttt{\#text}_i[str : str'] * \sigma\ \right\}$$

$$\left\{\ \alpha{\leftarrow}\texttt{\#text}_i[str_1 : str : str_2] * \sigma \wedge \mathcal{E}[\![E]\!]\sigma = i \wedge \mathcal{E}[\![E']\!]\sigma = |str_1| \wedge \mathcal{E}[\![E'']\!]\sigma = |str|\ \right\}$$
$$\texttt{deleteData}(E, E', E'')$$
$$\left\{\ \alpha{\leftarrow}\texttt{\#text}_i[str_1 : str_2] * \sigma\ \right\}$$

$$\left\{\ \alpha{\leftarrow}\texttt{\#text}_i[str_1 : str] * \sigma \wedge \mathcal{E}[\![E]\!]\sigma = i \wedge \mathcal{E}[\![E']\!]\sigma = |str_1| \wedge \mathcal{E}[\![E'']\!]\sigma > |str|\ \right\}$$
$$\texttt{deleteData}(E, E', E'')$$
$$\left\{\ \alpha{\leftarrow}\texttt{\#text}_i[str_1] * \sigma\ \right\}$$

Figure 5.16: Featherweight DOM axioms for text manipulation.

Example 5.17 (Implementing DOM Core Level 1)**.** Featherweight DOM provides a minimal subset of the DOM Core Level 1 commands, but there are several basic update commands which we have chosen not to provide as basic module commands. We can implement each of these extra commands with our featherweight DOM module. As an example of this consider the `getFirstChild` command which returns the first child of some node, or null if the node has no children.

$$\begin{array}{lcl} \mathtt{x} := \mathtt{getFirstChild(i)} & ::= & \mathtt{x} := \mathtt{getChildNodes(i)}\ ; \\ & & \mathtt{x} := \mathtt{item}(\mathtt{x}, 0) \end{array}$$

We can derive a specification for this program as follows:

$$\begin{array}{l} \left\{\ \alpha{\leftarrow}str_i[str'_k[\beta] \otimes \gamma]_j * \mathtt{i} \Rightarrow i * \mathtt{x} \Rightarrow -\ \right\} \\ \mathtt{x} := \mathtt{getChildNodes(i)}\ ; \\ \left\{\ \alpha{\leftarrow}str_i[str'_k[\beta] \otimes \gamma]_j * \mathtt{i} \Rightarrow i * \mathtt{x} \Rightarrow j\ \right\} \\ \mathtt{x} := \mathtt{item}(\mathtt{x}, 0) \\ \left\{\ \alpha{\leftarrow}str_i[str'_k[\beta] \otimes \gamma]_j * \mathtt{i} \Rightarrow i * \mathtt{x} \Rightarrow k\ \right\} \end{array}$$

The two remaining cases: where the first child is a text node; or the node has no children, are both similar to this case. In our original work on DOM [33][34] we show how to implement other DOM Core Level 1 commands. We can apply similar techniques in our fine-grained reasoning system.

Example 5.18 (Basic Commands vs. Implementable Commands)**.** Using analogous techniques to those in Smith's thesis [64] we could implement all of the commands of DOM Core Level 1. However, in doing so we do not always produce the most elegant specifications for those commands we choose not to take as basic commands. As an example of this, consider the `insertBefore` command, which behaves in a similar way to the `appendChild` command.

- ◇ `insertBefore`(E, E', E'') moves the subtree at the node identified by E' to be the left sibling of the node identified by E'' which is a child of the node identified by E. If E'' evaluates to null then the subtree is instead moved to be the last child of the node identified by E (This is the behaviour of append). Requires that E identifies a node that exists and is not a text node, and that E' identifies a node that exists and is not an ancestor of the node identified by E , or it faults. If E'' does not evaluate to null then it also requires that E'' identifies a node that is a child of the node identified by E.

For the rest of this example, we assume that the expression parameters E, E' and E''

have been evaluated and their values stored in the variables n, m and r respectively. One way to implement the insertBefore command in our featherweight DOM module is with the following program:

```
insertBefore(n, m, r) ::= local c, x, y in
                            appendChild(n, m) ;
                            if r = null then
                              skip
                            else
                              c := 0 ;
                              x := getChildNodes(n) ;
                              y := item(x, c) ;
                              while y ≠ r do
                                c := c + 1 ;
                                y := item(x, c) ;
                              while y ≠ m do
                                appendChild(n, y) ;
                                y := item(x, c) ;
```

This program first moves m to the end of the list of n's children. The first while-loop then scans through the children of n looking for r. The second while-loop appends the children from r up to (but not including) m to the end of the list of n's children. The effect of this is to move m to the left of r in the list of n's children. Note that the second while-loop does not need to increment the counter c. When we append the cth node to the end of the list, the c + 1th node drops down the list one space and becomes the cth node in the list for the next loop iteration.

From this program we can derive the following specification for insertBefore in the case where the reference node identifier r is not null:

$$\left\{ \begin{array}{l} \alpha\leftarrow str_n[\mathsf{Ls}(N) \otimes str'_r[\mathsf{tree}(cdt_1)]_j \otimes \mathsf{tree}(cdt_2)]_i * \beta\leftarrow str''_m[\mathsf{tree}(cdt)]_k \\ * \,\mathtt{n} \Rightarrow n * \mathtt{m} \Rightarrow m * \mathtt{r} \Rightarrow r \end{array} \right\}$$

$$\mathtt{insertBefore(n, m, r)}$$

$$\left\{ \begin{array}{l} \alpha\leftarrow str_n[\mathsf{Ls}(N) \otimes str''_m[\mathsf{tree}(cdt)]_k \otimes str'_r[\mathsf{tree}(cdt_1)]_j \otimes \mathsf{tree}(cdt_2)]_i * \beta\leftarrow\varnothing \\ * \,\mathtt{n} \Rightarrow n * \mathtt{m} \Rightarrow m * \mathtt{r} \Rightarrow r \end{array} \right\}$$

We omit the specification for the case where r = null as insertBefore simply behaves as appendChild in this case. Note that the specification needs to include the one-layer list Ls(N) in order to be able to apply the axiom for item. Similarly, the specification needs to include the entire subtree beneath (cdt_1) and after (cdt_2) the node `r` in order to be able to apply the axiom for `appendChild`.

The proof sketch for this program, showing that the above specification holds, is given in Figure 5.17[1]. Notice that the derived specification is not the smallest specification that we could give for this behaviour of the command. If we instead chose to take `insertBefore` as one of our basic commands we could provide its axiom for the case where $\mathtt{r} \neq \mathsf{null}$ as:

$$\left\{ \begin{array}{l} \alpha{\leftarrow} str_n[\gamma \otimes str'_r[\delta]_j \otimes \zeta]_i * \beta{\leftarrow} str''_m[\mathsf{tree}(cdt)]_k \\ * \mathtt{n} \Rightarrow n * \mathtt{m} \Rightarrow m * \mathtt{r} \Rightarrow r \end{array} \right\}$$

$$\mathtt{insertBefore(n, m, r)}$$

$$\left\{ \begin{array}{l} \alpha{\leftarrow} str_n[\gamma \otimes str''_m[\mathsf{tree}(cdt)]_k \otimes str'_r[\delta]_j \otimes \zeta]_i * \beta{\leftarrow}\varnothing \\ * \mathtt{n} \Rightarrow n * \mathtt{m} \Rightarrow m * \mathtt{r} \Rightarrow r \end{array} \right\}$$

In our derived axiom we needed to include the list of nodes $\mathsf{Ls}(N)$ that proceed the reference node r, the complete tree cdt_1 beneath r and the complete tree cdt_2 that follows node r. Specifying the command directly, we need only mention the state required for the update to proceed without faulting. Unsurprisingly, the choice of basic commands affects the specifications that we can derive for programs of our modules. The choice of basic commands in the featherweight DOM module is sufficient to allow us to describe a wide range XML update programs.

Example 5.19 (Schema Invariants)**.** So far, all of our reasoning examples have concentrated on capturing the precise updates to the program state during a program's execution. However, it is sometimes desirable to prove a particular property about a program rather than proving the whole specification. One example of this is proving that programs satisfy XML schema invariants. As an example, we consider writing a program to update an XML document which satisfies the following XML schema:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
 elementFormDefault="qualified">
<xs:element name="studentDB">
  <xs:element name="student" minOccurs="0" maxOccurs="unbounded">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="name" type="string"/>
```

[1]We condense much of the proof sketch to concentrate on a few key steps. The full proof is moderately involved, due to the need to provide loop invariants for the while loops, but this is orthogonal to our discussion here.

```
$\left\{ \begin{array}{l} \alpha \leftarrow str_n[\mathsf{Ls}(N) \otimes str'_r[\mathsf{tree}(cdt_1)]_j \otimes \mathsf{tree}(cdt_2)]_i * \beta \leftarrow str''_m[\mathsf{tree}(cdt)]_k \\ * \mathtt{n} \Rightarrow n * \mathtt{m} \Rightarrow m * \mathtt{r} \Rightarrow r \end{array} \right\}$
local c, x, y in
  $\left\{ \begin{array}{l} \alpha \leftarrow str_n[\mathsf{Ls}(N) \otimes str'_r[\mathsf{tree}(cdt_1)]_j \otimes \mathsf{tree}(cdt_2)]_i * \beta \leftarrow str''_m[\mathsf{tree}(cdt)]_k \\ * \mathtt{n} \Rightarrow n * \mathtt{m} \Rightarrow m * \mathtt{r} \Rightarrow r * \mathtt{c} \Rightarrow - * \mathtt{x} \Rightarrow - * \mathtt{y} \Rightarrow - \end{array} \right\}$
  appendChild(n, m) ;
  $\left\{ \begin{array}{l} \alpha \leftarrow str_n[\mathsf{Ls}(N) \otimes str'_r[\mathsf{tree}(cdt_1)]_j \otimes \mathsf{tree}(cdt_2) \otimes str''_m[\mathsf{tree}(cdt)]_k]_i * \beta \leftarrow \varnothing \\ * \mathtt{n} \Rightarrow n * \mathtt{m} \Rightarrow m * \mathtt{r} \Rightarrow r * \mathtt{c} \Rightarrow - * \mathtt{x} \Rightarrow - * \mathtt{y} \Rightarrow - \end{array} \right\}$
  if r = null then
    skip
    $\{ \mathsf{false} \}$
  else
    c := 0 ;
    x := getChildNodes(n) ;
    y := item(x, c) ;
    $\left\{ \begin{array}{l} \alpha \leftarrow str_n[\mathsf{Ls}(N) \otimes str'_r[\mathsf{tree}(cdt_1)]_j \otimes \mathsf{tree}(cdt_2) \otimes str''_m[\mathsf{tree}(cdt)]_k]_i * \beta \leftarrow \varnothing \\ * \mathtt{n} \Rightarrow n * \mathtt{m} \Rightarrow m * \mathtt{r} \Rightarrow r * \mathtt{c} \Rightarrow 0 * \mathtt{x} \Rightarrow i * \mathtt{y} \Rightarrow - \end{array} \right\}$
    while y ≠ r do
      c := c + 1 ;
      y := item(x, c) ;
    $\left\{ \begin{array}{l} \alpha \leftarrow str_n[\mathsf{Ls}(N) \otimes str'_r[\mathsf{tree}(cdt_1)]_j \otimes \mathsf{tree}(cdt_2) \otimes str''_m[\mathsf{tree}(cdt)]_k]_i * \beta \leftarrow \varnothing \\ * \mathtt{n} \Rightarrow n * \mathtt{m} \Rightarrow m * \mathtt{r} \Rightarrow r * \mathtt{c} \Rightarrow |N| * \mathtt{x} \Rightarrow i * \mathtt{y} \Rightarrow r \end{array} \right\}$
    while y ≠ m do
      appendChild(n, y) ;
      y := item(x, c) ;
    $\left\{ \begin{array}{l} \alpha \leftarrow str_n[\mathsf{Ls}(N) \otimes str''_m[\mathsf{tree}(cdt)]_k \otimes str'_r[\mathsf{tree}(cdt_1)]_j \otimes \mathsf{tree}(cdt_2)]_i * \beta \leftarrow \varnothing \\ * \mathtt{n} \Rightarrow n * \mathtt{m} \Rightarrow m * \mathtt{r} \Rightarrow r * \mathtt{c} \Rightarrow |N| * \mathtt{x} \Rightarrow i * \mathtt{y} \Rightarrow m \end{array} \right\}$
$\left\{ \begin{array}{l} \alpha \leftarrow str_n[\mathsf{Ls}(N) \otimes str''_m[\mathsf{tree}(cdt)]_k \otimes str'_r[\mathsf{tree}(cdt_1)]_j \otimes \mathsf{tree}(cdt_2)]_i * \beta \leftarrow \varnothing \\ * \mathtt{n} \Rightarrow n * \mathtt{m} \Rightarrow m * \mathtt{r} \Rightarrow r \end{array} \right\}$
```

Figure 5.17: Proof sketch for the `insertBefore` program.

```
          <xs:element name="year" type="string"/>
          <xs:element name="course" type="string" />
        </xs:sequence>
      </xs:complexType>
    </xs:element>
</xs:element>
```

This schema describes a document that stores information about students enrolled at a particular university. The schema asserts that the root node of the document should be a `studentDB` node, whose children should be zero or more `student` nodes. Each `student` node should contain one `name` node, one `year` node and one `course` node. Each of these third level nodes should contain data of type string; that is, data in a text node. Note that we can use the DOM node identifiers as unique student ids.

In order to specify such an XML schema we need to provide some additional derived formulae, specific to DOM trees, given as follows:

$$\begin{aligned}
str_i[cdt] &\overset{\text{def}}{=} \exists j.\, str_i[cdt]_j \\
str[cdt]_j &\overset{\text{def}}{=} \exists i.\, str_i[cdt]_j \\
str[cdt] &\overset{\text{def}}{=} \exists i, j.\, str_i[cdt]_j \\
\Diamond_{\otimes} P &\overset{\text{def}}{=} \mathsf{true} \otimes P \otimes \mathsf{true} \\
\Box_{\otimes} P &\overset{\text{def}}{=} \neg \Diamond_{\otimes} \neg P
\end{aligned}$$

The first three formulae allow us to drop node and node-list identifiers when they are not important. The penultimate formula describes the property that P holds somewhere at this level of the tree. The last formula describes the property that P holds everywhere at this level of the tree.

We can now specify this XML schema with a DOM segment formula SDB:

$$SDB \overset{\text{def}}{=} \lceil \texttt{studentDB}[\mathsf{students}] \rceil$$

where

$$\begin{aligned}
\mathsf{students} &\overset{\text{def}}{=} \Box_{\otimes}(\exists str, cdt.\, str[cdt]) \Rightarrow \texttt{student}[\mathsf{name} \otimes \mathsf{year} \otimes \mathsf{course}] \\
\mathsf{name} &\overset{\text{def}}{=} \exists str.\, \texttt{name}[\texttt{\#text}[str]] \\
\mathsf{year} &\overset{\text{def}}{=} \exists str.\, \texttt{year}[\texttt{\#text}[str]] \\
\mathsf{course} &\overset{\text{def}}{=} \exists str.\, \texttt{course}[\texttt{\#text}[str]]
\end{aligned}$$

The SDB assertion describes a node `studentDB` all of whose children must be `student` nodes containing `name`, `year` and `course` data.

Now consider a featherweight DOM program which updates the `studentDB` document when a specified student `sid` changes course or leaves the university. We assume that if the student is leaving the university, then the input course to the program is null. Thus, the program checks if the course input is null and if it is it deletes the student record, and if it is not it updates that student's course appropriately.

```
courseChange(sid, crs)  ::=  local x, y in
                               x := getParentNode(sid) ;
                               if crs = null then
                                 removeChild(x, sid)
                               else
                                 y := createNode('course') ;
                                 x := createTextNode(crs) ;
                                 appendChild(y, x) ;
                                 x := getChildNodes(sid) ;
                                 x := item(x, 2) ;
                                 removeChild(sid, x) ;
                                 appendChild(sid, y)
```

In order for this program to run without faulting we need to know that the variable `sid` refers to a `student` node that is stored in the document. This safety property can be captured by the formula $S(i)$ assuming that `sid` maps to identifier i in the variable store.

$$S(i) \stackrel{\text{def}}{=} \mathsf{H}\alpha, \beta. \, (\alpha \leftarrow \mathtt{student}_i[\beta] * \mathsf{true})$$

Notice that the use of true in the formula allows for there to be any other program state present. We can now prove that the `courseChange` program maintains the schema formula SDB provided that this safety formula also holds. That is, we can prove:

$$\left\{ \; SDB \wedge S(i) * \mathsf{sid} \Rightarrow i * \mathsf{crs} \Rightarrow c \; \right\}$$
$$\mathtt{courseChange}(\mathtt{sid}, \mathtt{crs})$$
$$\left\{ \; SDB * \mathsf{true} * \mathsf{sid} \Rightarrow i * \mathsf{crs} \Rightarrow c \; \right\}$$

The proof of this specification is given in Figure 5.18. Recall that we treat featherweight DOM as a garbage collected language. We thus use true in the postcondition to refer to any uncollected garbage that was generated by the program. This garbage will no longer be used and can be safely ignored. Note that we could choose to more precisely characterise the garbage if doing so were of interest. In this case the garbage would either be a single student record, in the case where $\texttt{crs} = \mathsf{null}$, or a single course node, in the remaining case.

5.6 Combining Fine-grained Abstract Modules

It is often useful when programming to be able to make use of several modules. For example, in chapter 6 we show how to use a combination of the heap module $\mathbb{H}$ and the list module $\mathbb{L}$ to provide an implementation of the tree module $\mathbb{T}$.

Just as it is natural to combine segment algebras, as in Example 3.48, it is also natural to be able to combine the reasoning for multiple fine-grained abstract modules. The most intuitive approach is to take the union of the basic command sets of each module, whilst interpreting the basic commands over the product of their segment algebras. If the modules want to share any information, this must be done through the common variable store.

Definition 5.20 (Module Combination)**.** Given fine-grained abstract modules $\mathbb{A}_1 = (\textsc{Cmd}_{\mathbb{A}_1}, \mathcal{S}(\mathcal{M}_{\mathbb{A}_1}), \mathrm{Ax}[\![(\cdot)]\!]_{\mathbb{A}_1})$ and $\mathbb{A}_2 = (\textsc{Cmd}_{\mathbb{A}_2}, \mathcal{S}(\mathcal{M}_{\mathbb{A}_2}), \mathrm{Ax}[\![(\cdot)]\!]_{\mathbb{A}_2})$, their combination

$$\mathbb{A}_1 + \mathbb{A}_2 \overset{\text{def}}{=} (\textsc{Cmd}_{\mathbb{A}_1} \oplus \textsc{Cmd}_{\mathbb{A}_2}, \mathcal{S}(\mathcal{M}_{\mathbb{A}_1}) \times \mathcal{S}(\mathcal{M}_{\mathbb{A}_2}), \mathrm{Ax}[\![(\cdot)]\!]_{\mathbb{A}_1+\mathbb{A}_2})$$

is a fine-grained abstract module, where

- ◇ $\textsc{Cmd}_{\mathbb{A}_1} \oplus \textsc{Cmd}_{\mathbb{A}_2} \overset{\text{def}}{=} (\textsc{Cmd}_{\mathbb{A}_1} \times \{1\}) \cup (\textsc{Cmd}_{\mathbb{A}_2} \times \{2\})$ is the discriminated union of the command sets;
- ◇ $\mathcal{S}(\mathcal{M}_{\mathbb{A}_1}) \times \mathcal{S}(\mathcal{M}_{\mathbb{A}_2})$ is the product of the segment algebras; and
- ◇ $\mathrm{Ax}[\![(\cdot)]\!]_{\mathbb{A}_1+\mathbb{A}_2} : \textsc{Cmd}_{\mathbb{A}_1} \oplus \textsc{Cmd}_{\mathbb{A}_2} \to \mathcal{P}(\textsc{Pred}_{\mathbb{A}_1\times\mathbb{A}_2} \times \textsc{Pred}_{\mathbb{A}_1\times\mathbb{A}_2})$ is defined as

$$\begin{aligned}\mathrm{Ax}[\![(\varphi, 1)]\!]_{\mathbb{A}_1+\mathbb{A}_2} &\overset{\text{def}}{=} \{(\pi_1(P), \pi_1(Q)) \mid (P, Q) \in \mathrm{Ax}[\![\varphi]\!]_{\mathbb{A}_1}\} \\ \mathrm{Ax}[\![(\varphi, 2)]\!]_{\mathbb{A}_1+\mathbb{A}_2} &\overset{\text{def}}{=} \{(\pi_2(P), \pi_2(Q)) \mid (P, Q) \in \mathrm{Ax}[\![\varphi]\!]_{\mathbb{A}_2}\}\end{aligned}$$

  where
$$\begin{aligned}\mathcal{P}[\![\pi_1(P)]\!]e &\overset{\text{def}}{=} \{(s_1, \mathsf{emp}_2, \sigma) \mid (s_1, \sigma) \in \mathcal{P}[\![P]\!]e\} \\ \mathcal{P}[\![\pi_2(P)]\!]e &\overset{\text{def}}{=} \{(\mathsf{emp}_1, s_2, \sigma) \mid (s_2, \sigma) \in \mathcal{P}[\![P]\!]e\}\end{aligned}$$

```
$\{\ SDB \land S(i) * \mathtt{sid} \Rightarrow i * \mathsf{crs} \Rightarrow c\ \}$
$\left\{ \begin{array}{l} \lceil \mathsf{studentDB}[\mathsf{students} \otimes \mathsf{student}_i[\mathsf{name} \otimes \mathsf{year} \otimes \mathsf{course}] \otimes \mathsf{students}]\rceil \\ * \mathtt{sid} \Rightarrow i * \mathsf{crs} \Rightarrow c \end{array} \right\}$
local x, y in
  $\left\{ \begin{array}{l} \lceil \mathsf{studentDB}[\mathsf{students} \otimes \mathsf{student}_i[\mathsf{name} \otimes \mathsf{year} \otimes \mathsf{course}] \otimes \mathsf{students}]\rceil \\ * \mathtt{sid} \Rightarrow i * \mathsf{crs} \Rightarrow c * \mathsf{x} \Rightarrow - * \mathsf{y} \Rightarrow - \end{array} \right\}$
  x := getParentNode(sid) ;
  $\left\{ \begin{array}{l} \exists j.\ \lceil \mathsf{studentDB}_j[\mathsf{students} \otimes \mathsf{student}_i[\mathsf{name} \otimes \mathsf{year} \otimes \mathsf{course}] \otimes \mathsf{students}]\rceil \\ * \mathtt{sid} \Rightarrow i * \mathsf{crs} \Rightarrow c * \mathsf{x} \Rightarrow j * \mathsf{y} \Rightarrow - \end{array} \right\}$
  if crs = null then
    removeChild(x, sid)
    $\left\{ \begin{array}{l} \exists j.\ \lceil \mathsf{studentDB}_j[\mathsf{students} \otimes \mathsf{students}]\rceil * \lceil \mathsf{student}_i[\mathsf{name} \otimes \mathsf{year} \otimes \mathsf{course}]\rceil \\ * \mathtt{sid} \Rightarrow i * \mathsf{crs} \Rightarrow c * \mathsf{x} \Rightarrow j * \mathsf{y} \Rightarrow - \end{array} \right\}$
    $\{\ SDB * \mathsf{true} * \mathtt{sid} \Rightarrow i * \mathsf{crs} \Rightarrow c * \mathsf{x} \Rightarrow - * \mathsf{y} \Rightarrow -\ \}$
  else
    y := createNode('course') ;
    $\left\{ \begin{array}{l} \exists j,k.\ \lceil \mathsf{studentDB}_j[\mathsf{students} \otimes \mathsf{student}_i[\mathsf{name} \otimes \mathsf{year} \otimes \mathsf{course}] \otimes \mathsf{students}]\rceil \\ * \lceil \mathtt{course}_k[\varnothing]\rceil * \mathtt{sid} \Rightarrow i * \mathsf{crs} \Rightarrow c * \mathsf{x} \Rightarrow j * \mathsf{y} \Rightarrow k \end{array} \right\}$
    x := createTextNode(crs) ;
    $\left\{ \begin{array}{l} \exists j,k.\ \lceil \mathsf{studentDB}[\mathsf{students} \otimes \mathsf{student}_i[\mathsf{name} \otimes \mathsf{year} \otimes \mathsf{course}] \otimes \mathsf{students}]\rceil \\ * \lceil \mathtt{course}_k[\varnothing]\rceil * \lceil \#\mathtt{text}_j[c]\rceil * \mathtt{sid} \Rightarrow i * \mathsf{crs} \Rightarrow c * \mathsf{x} \Rightarrow j * \mathsf{y} \Rightarrow k \end{array} \right\}$
    appendChild(y, x) ;
    $\left\{ \begin{array}{l} \exists j,k.\ \lceil \mathsf{studentDB}[\mathsf{students} \otimes \mathsf{student}_i[\mathsf{name} \otimes \mathsf{year} \otimes \mathsf{course}] \otimes \mathsf{students}]\rceil \\ * \lceil \mathtt{course}_k[\#\mathtt{text}_j[c]]\rceil * \mathtt{sid} \Rightarrow i * \mathsf{crs} \Rightarrow c * \mathsf{x} \Rightarrow j * \mathsf{y} \Rightarrow k \end{array} \right\}$
    x := getChildNodes(sid) ;
    $\left\{ \begin{array}{l} \exists j,k.\ \lceil \mathsf{studentDB}[\mathsf{students} \otimes \mathsf{student}_i[\mathsf{name} \otimes \mathsf{year} \otimes \mathsf{course}]_j \otimes \mathsf{students}]\rceil \\ * \lceil \mathtt{course}_k[\#\mathtt{text}[c]]\rceil * \mathtt{sid} \Rightarrow i * \mathsf{crs} \Rightarrow c * \mathsf{x} \Rightarrow j * \mathsf{y} \Rightarrow k \end{array} \right\}$
    x := item(x, 2) ;
    $\left\{ \begin{array}{l} \exists j,k,c'. \\ \lceil \mathsf{studentDB}[\mathsf{students} \otimes \mathsf{student}_i[\mathsf{name} \otimes \mathsf{year} \otimes \mathtt{course}_j[\#\mathtt{text}[c']]] \otimes \mathsf{students}]\rceil \\ * \lceil \mathtt{course}_k[\#\mathtt{text}[c]]\rceil * \mathtt{sid} \Rightarrow i * \mathsf{crs} \Rightarrow c * \mathsf{x} \Rightarrow j * \mathsf{y} \Rightarrow k \end{array} \right\}$
    removeChild(sid, x) ;
    $\left\{ \begin{array}{l} \exists j,k,c'.\ \lceil \mathsf{studentDB}[\mathsf{students} \otimes \mathsf{student}_i[\mathsf{name} \otimes \mathsf{year}] \otimes \mathsf{students}]\rceil \\ * \lceil \mathtt{course}_k[\#\mathtt{text}[c]]\rceil * \lceil \mathtt{course}_j[\#\mathtt{text}[c']]\rceil * \mathtt{sid} \Rightarrow i * \mathsf{crs} \Rightarrow c * \mathsf{x} \Rightarrow j * \mathsf{y} \Rightarrow k \end{array} \right\}$
    appendChild(sid, y)
    $\left\{ \begin{array}{l} \exists j,k,c'. \\ \lceil \mathsf{studentDB}[\mathsf{students} \otimes \mathsf{student}_i[\mathsf{name} \otimes \mathsf{year} \otimes \mathtt{course}_k[\#\mathtt{text}[c]]] \otimes \mathsf{students}]\rceil \\ * \lceil \mathtt{course}_j[\#\mathtt{text}[c']]\rceil * \mathtt{sid} \Rightarrow i * \mathsf{crs} \Rightarrow c * \mathsf{x} \Rightarrow j * \mathsf{y} \Rightarrow k \end{array} \right\}$
    $\{\ SDB * \mathsf{true} * \mathtt{sid} \Rightarrow i * \mathsf{crs} \Rightarrow c * \mathsf{x} \Rightarrow - * \mathsf{y} \Rightarrow -\ \}$
  $\{\ SDB * \mathsf{true} * \mathtt{sid} \Rightarrow i * \mathsf{crs} \Rightarrow c * \mathsf{x} \Rightarrow - * \mathsf{y} \Rightarrow -\ \}$
$\{\ SDB * \mathsf{true} * \mathtt{sid} \Rightarrow i * \mathsf{crs} \Rightarrow c\ \}$
```

Figure 5.18: Schema preservation derivation.

Notation: When the command sets $\textsc{Cmd}_{\mathbb{A}_1}$ and $\textsc{Cmd}_{\mathbb{A}_2}$ are disjoint, we drop the tags when referring to the commands in the combined abstract module. When the tags are necessary, we indicate them with an appropriately placed subscript.

5.7 Remarks

We have shown how to provide fine-grained abstract modules for a range of different data structures. This demonstrates that the concept of a segment algebra is applicable to a wide range of data structures. Moreover, the use of segment algebras allows us to provide genuinely local specifications for the basic commands of our modules. This is a particular achievement for the append and remove commands in the tree and DOM modules for which we have been unable to provide small axioms in the past.

5.7.1 Locality

In his thesis [64], Smith discusses the issue of locality for certain module commands. As an example, consider DOM's `getParentNode` command. The footprint of this command is not uniform in size: if the target node is at the root level, then the footprint is just that node; if the target node is not at the root level, then the footprint is that node plus the node above it. Such behaviour is tricky to capture using context logic specifications.

In order to establish the soundness of his reasoning system, Smith had to refine the notion of locality to that of local with respect to some formula P. This allowed him to handle commands which had different behaviours at different levels of locality, such as `getParentNode`, by restricting the possible frames that could be applied to a state.

In our approach to soundness, as given in chapter 3, we interpret segment logic assertions in any possible extension to a complete data structure. By introducing the notion of a rooted segment, we can rule out any extensions that would try to add data above this segment. This allows us to provide disjoint specifications that capture the different behaviours of commands like `getParentNode`. Note that `getParentNode` has two axioms in each of Figure 5.14 and Figure 5.15. The first axiom captures that case where the target node has some parent, the second where it does not. Both cases are disjoint; that is, any given state can only satisfy one of the preconditions for `getParentNode`.

In general, our segment model is able to describe when data is complete (cannot be further extended) and so we have a more traditional interpretation of locality.

5.7.2 Copy Commands

In the modules considered above we have provided commands that analyse data structures, commands that create new data structures and commands that dispose of existing data structures. If one wanted to copy a data structure this would be possible using the analysis commands and the creation commands along with some careful looping/recursion. However, it would also be possible to extend the basic command sets with primitive copy commands. DOM, our motivating example of an abstract module, does not include any copy commands, which is why we have not considered them so far.

As an example of how to deal with copy commands, let us consider extending our tree module, from §5.2. In order to be able to specify such commands we require a notion of a *tree-shape*. A tree-shape stores the structure of a tree, but not the identifiers associated with that tree. For example, the tree $p[n[\varnothing] \otimes m[\varnothing]]$ has the shape $\circ[\circ[\varnothing] \otimes \circ[\varnothing]]$. More formally, we define tree-shapes $t_\circ \in \mathrm{T}_\circ$ inductively as:

$$t_\circ \ ::= \ \varnothing \mid \circ[t_\circ] \mid t_\circ \otimes t_\circ$$

where $\circ$ is a constant that represents a node and $\otimes$ is associative with identity $\varnothing$. Note that tree-shapes do not include context holes, so we only describe the shape of complete trees. We write $\langle ct \rangle$ for the shape of a tree context ct, with

$$\begin{aligned} \langle \varnothing \rangle &= \varnothing \\ \langle x \rangle &= \text{undefined} \\ \langle n[ct] \rangle &= \circ[\langle ct \rangle] \\ \langle ct_1 \otimes ct_2 \rangle &= \langle ct_1 \rangle \otimes \langle ct_2 \rangle \end{aligned}$$

We write $ct_1 \simeq ct_2$ when $\langle ct_1 \rangle = \langle ct_2 \rangle$. We then extend the variable store with tree-shape variables $\mathtt{t} \in \textsc{Var}_{\mathrm{T}_\circ}$ which allow us to store tree-shapes and we also extend our expressions to include tree-shape expressions $\mathtt{T}$, which have the form:

$$\mathtt{T} \ ::= \ \varnothing \mid \mathtt{t} \mid \circ[\mathtt{T}] \mid \mathtt{T} \otimes \mathtt{T}.$$

With these modifications to the variable store and expressions we can now extend the set of basic tree update commands with a tree copy command $\mathtt{t} := \mathtt{copyTree}(\mathtt{E})$

$$\left\{\begin{array}{l}\alpha{\leftarrow}n[\mathsf{tree}(ct)] * \mathtt{t} \Rrightarrow v * \sigma \\ \wedge\, \mathcal{E}[\![E]\!]\sigma[\mathtt{t} \Rrightarrow v] = n\end{array}\right\} \quad \mathtt{t} := \mathtt{copyTree}(E) \quad \left\{\begin{array}{l}\alpha{\leftarrow}n[\mathsf{tree}(ct)] \\ * \,\mathtt{t} \Rrightarrow \langle ct\rangle * \sigma\end{array}\right\}$$

$$\left\{\ \alpha{\leftarrow}n[\beta] * \sigma \wedge \mathcal{E}[\![E]\!]\sigma = n\ \right\}\ \mathtt{insertTreeAfter}(E, T)\ \left\{\ \alpha{\leftarrow}(n[\beta] \otimes T) * \sigma\ \right\}$$

Figure 5.19: Small axioms for the tree module commands involving tree-shapes

and a tree insertion command `insertTreeAfter`(E, T). The intuitive meaning of these commands, which will be realised by their axiomatic semantics, is as follows:

◇ `t` := `copyTree`(E) creates a copy of the shape of the subtree starting at the node identified by E and stores it in the program variable `t`. Requires that E identifies a node that exists or it faults;

◇ `insertTreeAfter`(E, T) creates a new tree, with a shape given by tree shape expression T, and inserts it into the working tree as the right sibling of the node identified by E. Requires that E identifies a node that exists or it faults.

We can then give the axioms for our two new commands as shown in Figure 5.19. We interpret the predicate $\circ[P]$ as $\exists n.\, n[P]$ allowing us to describe the shape of a tree in the program state for some arbitrary (but legal) choice of node identifiers in that tree. In particular, this allows us to interpret tree shape expressions T as assertions describing the shape of some complete tree.

5.7.3 Weakest Preconditions

The reasoning style presented above is very much focused on forwards reasoning. We start from some precondition and move through the program step by step until we arrive at a postcondition. This style has been used in the majority of separation logic tools to date.

Another common style of reasoning is backwards reasoning where you start with an arbitrary postcondition P and step backwards through the program using the weakest preconditions of each of the program steps to establish the most general precondition of a program. This style is commonly used to show completeness results, in particular completeness for straight-line code [71]

Our reasoning system, from chapter 4, is also able to provide weakest preconditions for our module commands. However, doing so requires the use of the separating conjunction adjoint $-\!\!*$ as well as the revelation adjoint $\oslash$, which have not shown up in the reasoning thus far. Both of these adjoints are used to express hypothetical properties about a program state. As an example let us consider the weakest precondition of the `deleteTree` command from the tree module given in §5.2.

$$\begin{array}{l}\left\{\ \exists n, ct.\, \mathsf{H}\alpha.\, (((\alpha{\leftarrow}\varnothing * \sigma) \mathrel{-\!\!*} (P \circledS \alpha)) * (\alpha{\leftarrow}n[\mathsf{tree}(ct)] * \sigma) \wedge \mathcal{E}[\![E]\!]\sigma = n)\ \right\} \\ \texttt{deleteTree}(E) \\ \left\{\ P\ \right\}\end{array}$$

We have briefly discussed a similar weakest precondition as one of our segment logic for trees formulae examples in chapter 3. The precondition here is given in our formal reasoning system, and is a little more complicated due to assertions about the variable store. When evaluated in an environment e, the precondition describes a set of states $\mathcal{P}[\![\mathsf{H}\alpha.\, ((\alpha{\leftarrow}\varnothing * \sigma \mathrel{-\!\!*} (P \circledS \alpha)) * \alpha{\leftarrow}n[\mathsf{tree}(ct)] * \sigma) \wedge \mathcal{E}[\![E]\!]\sigma = n]\!]e$ for some choice of n and ct. Each program state in this set is of the form $((x)(st_0), \sigma_0)$ for some fresh label x stored at α. The assertion furthermore states that, after uncompressing x, the state (st_0, σ_0) can be separated into two parts. The first part satisfies $(\alpha{\leftarrow}\varnothing * \sigma) \mathrel{-\!\!*} (P \circledS \alpha)$. This assertion describes a state that, when extended with an empty tree at address x and some variables σ, will satisfy P once x is compressed. The second part, which satisfies $\alpha{\leftarrow}n[\mathsf{tree}(ct)] * \sigma$, consists of a complete tree, with top node n at an address x, and some variables σ which are needed to evaluate the expression E.

Recall the small axiom for the `deleteTree`(E) command:

$$\begin{array}{c}\left\{\ \alpha{\leftarrow}n[\mathsf{tree}(ct)] * \sigma \wedge \mathcal{E}[\![E]\!]\sigma = n\ \right\} \\ \texttt{deleteTree}(E) \\ \left\{\ \alpha{\leftarrow}\varnothing * \sigma\ \right\}\end{array}$$

When the expression E evaluates to node identifier n the command removes the whole of the subtree with top node n from the working tree. Our frame rules tell us that running this command on a program state satisfying,

$$\exists n, ct.\, \mathsf{H}\alpha.\, (((\alpha{\leftarrow}\varnothing * \sigma) \mathrel{-\!\!*} (P \circledS \alpha)) * (\alpha{\leftarrow}n[\mathsf{tree}(ct)] * \sigma) \wedge \mathcal{E}[\![E]\!]\sigma = n)$$

will result in a program state satisfying,

$$\exists n, ct.\, \mathsf{H}\alpha.\, (((\alpha{\leftarrow}\varnothing * \sigma) \mathrel{-\!\!*} (P \circledS \alpha)) * (\alpha{\leftarrow}\varnothing * \sigma))$$

which is equivalent to P.

In his thesis [71] investigated the completeness of context logic. He showed that if you could derive the weakest preconditions for each of a module's basic commands

$\{\ \alpha{\leftarrow}n[\mathsf{tree}(ct)] * \sigma \wedge \mathcal{E}[\![E]\!]\sigma = n\ \}$
deleteTree(E)
$\{\ \alpha{\leftarrow}\varnothing_{\mathrm{T}} * \sigma\ \}$

Sep Frame

$\{\ \alpha{\leftarrow}n[\mathsf{tree}(ct)] * \sigma \wedge \mathcal{E}[\![E]\!]\sigma = n * (\alpha{\leftarrow}\varnothing_{\mathrm{T}} * \sigma \mathrel{-\!\!*} (P \oslash \alpha))\ \}$
deleteTree(E)
$\{\ \alpha{\leftarrow}\varnothing_{\mathrm{T}} * \sigma * (\alpha{\leftarrow}\varnothing_{\mathrm{T}} * \sigma \mathrel{-\!\!*} (P \oslash \alpha))\ \}$

Cons

$\{\ (\alpha{\leftarrow}\varnothing_{\mathrm{T}} * \sigma \mathrel{-\!\!*} (P \oslash \alpha)) * \alpha{\leftarrow}n[\mathsf{tree}(ct)] * \sigma \wedge \mathcal{E}[\![E]\!]\sigma = n\ \}$
deleteTree(E)
$\{\ P \oslash \alpha\ \}$

Rev Frame

$\{\ \alpha \circledR ((\alpha{\leftarrow}\varnothing_{\mathrm{T}} * \sigma \mathrel{-\!\!*} (P \oslash \alpha)) * \alpha{\leftarrow}n[\mathsf{tree}(ct)] * \sigma \wedge \mathcal{E}[\![E]\!]\sigma = n)\ \}$
deleteTree(E)
$\{\ \alpha \circledR (P \oslash \alpha)\ \}$

Exsts/Fresh

$\{\ \exists n, ct.\, \text{И}\alpha.\, \alpha \circledR ((\alpha{\leftarrow}\varnothing_{\mathrm{T}} * \sigma \mathrel{-\!\!*} (P \oslash \alpha)) * \alpha{\leftarrow}n[\mathsf{tree}(ct)] * \sigma \wedge \mathcal{E}[\![E]\!]\sigma = n)\ \}$
deleteTree(E)
$\{\ \exists n, ct.\, \text{И}\alpha.\, \alpha \circledR (P \oslash \alpha)\ \}$

Cons

$\{\ \exists n, ct.\, \mathsf{H}\alpha.\, \circledR ((\alpha{\leftarrow}\varnothing_{\mathrm{T}} * \sigma \mathrel{-\!\!*} (P \oslash \alpha)) * \alpha{\leftarrow}n[\mathsf{tree}(ct)] \wedge \mathcal{E}[\![E]\!]\sigma = n)\ \}$
deleteTree(E)
$\{\ P\ \}$

Figure 5.20: Derivation of the weakest precondition for deleteTree(E) from its small axiom.

from their small axioms, then his reasoning was complete for straight-line programs; that is, anything that is true for programs that contain no loops or recursion is provable in his reasoning framework. His result can be easily extended to segment logic and our fine-grained abstract reasoning framework.

We show, for the tree module from §5.2, how to derive the weakest precondition of deleteTree(E) from the small axiom of deleteTree(E). We have already discussed this informally above, but the proof in Figure 5.20 shows the derivation in detail. The separating frame rule is used to add on the program state which will collapse to satisfy P once the command has updated the existing program state. The consequence rule is used to rewrite the precondition and collapse the postcondition and the revelation frame rule is used to compress the tree segment at the label in variable α. Then the existential and freshness quantification rules are used to generalise the precondition. Finally, the consequence rule is used to further collapse the postcondition into the required form.

6 Abstraction and Refinement for Fine-grained Local Reasoning

Abstraction allows clients of a program module to reason about the module, without having to understand the specifics of how that module is implemented. This is essential for modular programming, as it allows for a program to be replaced by any other program that meets the same specification. However, it is not enough for our reasoning to be confined just to the abstract level. An important part of any abstraction technique is to be able to refine the abstraction to a specific implementation. Moreover, we must be able to show that this implementation satisfies the abstract specification that is being provided to the module's clients.

Traditional abstraction techniques [62][50] take a concrete program and produce an abstract specification for that program. Traditional refinement techniques [39][22] take an abstract specification and produce a correct implementation of that specification. Both approaches result in a program that correctly implements an abstract specification. Both are also well-established techniques in program verification, but have only been initially understood in the context of local reasoning.

Separation logic was extended with abstract predicates by Parkinson and Bierman [54] to enable program specifications to be abstracted. To the client, an abstract predicate is an opaque object that encapsulates some unknown representation of an abstract data-type. Abstract predicates inherit some of the benefits of disjointness and locality from separation logic. In particular, an operation on one abstract predicate does not affect other abstract predicates. However, it is not always possible for the client to take full advantage of the local behaviour that is provided by the abstraction itself. Consider a set module. An operation that removes some value, say 3, from the set is local at the abstract level. That is, it is independent of whether any other values are in the set. However, if we consider an implementation of the set as a sorted, singly-linked list in the heap starting at address h, then the operation of removing 3 from the set must traverse the list from h. The footprint of the operation must consist of the the part of the list from h up to, and including, the node with value 3. Using abstract predicates, the abstract footprint directly corresponds to the concrete footprint, so in this case would include all of the elements of the set less than or equal to 3. Consequently, abstract predicates cannot be used to present a *local* abstract specification for removing 3. *Concurrent abstract predicates* (also known as CAP) have recently been introduced by Dinsdale-Young, Dodds, Gardner, Parkinson and Vafeiadis [25] and do provide a method for capturing abstract level locality. We shall discuss how our work relates to CAP in chapter 7.

Filipović, O'Hearn, Torp-Smith and Yang have previously considered data refinement for local reasoning [31]. They studied modules built on top of the standard separation logic heap model. They observed that a client could violate a module's abstract boundary by dereferencing pointers into the modules internal state, and thereby break the refinement between abstract modules and their concrete implementations. In their motivating example, a simple memory allocator, a client could violate the concrete allocator's free list through memory pointers that had been deallocated. The abstract allocator, which maintains a set of free cells, is unaffected by such an access, so the refinement has been broken. The solution to this problem was to "blame the client" by introducing a modified operational semantics that treats such memory access violations as faulting executions. Using special simulation relations they were able to recover the soundness of data refinement. These techniques can be adapted to different data store models, however it is necessary for both the module and the client to use the same underlying heap model. We believe that the client should be able to work with whatever model they find most natural to them, and so we seek an alternative technique.

Our initial work on abstraction and refinement for local reasoning [26] applies data refinement to local reasoning to demonstrate that local reasoning is sound for module implementations. In contrast with [31] we worked with the axiomatic semantics of the language, rather than its operational semantics. We defined proof transformations which established that concrete implementations are correct with respect to abstract specifications. This approach avoided having to consider badly behaved client programs, as the proof system only makes guarantees about well behaved client programs. Moreover, the abstract and concrete levels in our refinements typically have different data store models, meaning that the concept of locality itself is different at each level. When we encountered a mismatch in the locality of the abstract and concrete levels we found a simple way to resolve the problem (we include some extra context in our proof transformations). Our work was based on context logic, but as we have already seen this causes problems with providing small axioms for certain types of commands. This means that our work does not scale to concurrent programs, nor can it be directly applied to segment logic.

$$\begin{tikzcd}
 & \mathbb{T} \arrow[d, swap, "\tau_3"] \arrow[ddd, bend left=60, "\tau_2"] \\
 & \mathbb{H}+\mathbb{L} \arrow[d, ""{name=A}] \\
\mathbb{L} \arrow[dr, swap, "\tau_1", ""{name=B}] & \mathbb{H}+\mathbb{H} \arrow[d, "\tau_4"] \\
 & \mathbb{H}
\arrow[from=B, to=A, dotted, bend left]
\end{tikzcd}$$

Figure 6.1: Module translations presented in chapter 6

We present the next step in our abstraction and refinement work, basing our reasoning on segment logic rather than context logic. Not only does this allow us to work with fine-grained abstract modules, as introduced in chapter 5, but it also allows us to handle locality mismatches in a more structured fashion. The simple locality fix from our previous work is not applicable to our new reasoning framework. Instead we must reason about the potential sharing that is taking place between segments that are disjoint at the abstract level, but possibly overlapping at the concrete level.

We consider how to provide implementations for a number of our modules introduced in chapter 5. The implementations we shall consider are illustrated in Figure 6.1. Our first refinement τ_1, described in detail in §6.2.2, provides an implementation of our fine-grained list module $\mathbb{L}$ in our heap module $\mathbb{H}$, where each list is represented by a standard singly-linked list of heap cells. We then provide two ways of refining our fine-grained tree module $\mathbb{T}$ into our heap module $\mathbb{H}$. The first of these τ_2, described in detail in §6.3.2, provides a direct implementation of abstract trees in the heap, where each tree node is represented by a contiguous block of heap cells. The second tree refinement uses our fine-grained list module $\mathbb{L}$ as an intermediate step in the refinement. We first provide an implementation of our fine-grained tree module $\mathbb{T}$ in terms of a combined heap and list module $\mathbb{H}+\mathbb{L}$. This refinement τ_3 is described in detail in §6.3.3. Since our approach is modular, this translation can be extended by the translation τ_1 to give a translation from the combined heap and list module $\mathbb{H}+\mathbb{L}$ to a paired heap module $\mathbb{H}+\mathbb{H}$. This is illustrated by the dotted arrow in Figure 6.1. Finally, in §6.3.4 we complete our refinement by showing that the paired heap module $\mathbb{H}+\mathbb{H}$ can be trivially implemented by the heap module $\mathbb{H}$.

In our setting we shall introduce two general techniques for verifying that module implementations are correct with respect to their abstract local specifications. These techniques rely on providing module translations which are either *locality-preserving* or *locality-breaking*.

Locality-preserving translations relate locality at the abstract level with locality at the implementation level. However, it is often the case that a command's implementation operates on more state than is included in the command's abstract footprint. For example, consider the `deleteTree` command from our fine-grained tree module $\mathbb{T}$. At the abstract level the command simply removes a subtree from the working tree. However, at the concrete level, the implementation of this command will additionally need to perform some pointer surgery on the surrounding state in order to maintain the tree structure. Our translation has to be able to include this additional state, called the *crust* in our previous work, in a way that still provides a *fiction of disjointness* at the abstract level.

Locality-breaking translations are more suited to cases where the locality at the abstract level does not correspond with the locality at the implementation level. For example, consider the set removal command discussed above. At the abstract level this works on a single value in the set. At the concrete level, where the set is represented by a singly-linked list, the implementation of this command could potentially traverse the whole list. We could still use the locality preserving technique in this case, but it will be harder to establish a correct fiction of disjointness. It seems more appropriate, in such cases, to prove soundness by establishing that the specifications of module commands are preserved in *any* wider program state. In this case we establish a *fiction of locality* at the abstract level.

The correctness of both approaches relies on the data refinement technique known as *forward simulation* (also called *L-simulation*) [22]. Simulations provide a way of relating abstract program states and program steps with concrete program states and program steps. In forward simulations one must show that the result of taking an abstract state, running some abstract code on it and refining the result is the same as taking the same initial abstract state, refining it first and running the concrete representation of that code on the result. We illustrate the desired behaviour in Figure 6.2. Assume we have some refinement relation α between abstract states A and concrete states C and a program $\mathbb{C}$ and its implementation $[\![\mathbb{C}]\!]$. The implementation $[\![\mathbb{C}]\!]$ soundly refines $\mathbb{C}$ if, running $\mathbb{C}$ on A_1 results in A_2 where $\alpha(A_2, C_2)$ holds *and* $\alpha(A_1, C_1)$ holds and the result of running $[\![\mathbb{C}]\!]$ on C_1 is C_2. Forwards simulation provides a compositional method for building up simulation results for whole programs from simulation results for individual commands.

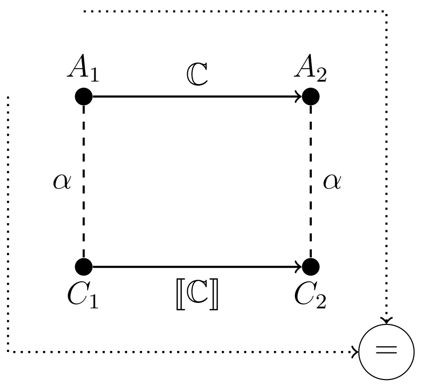

Figure 6.2: Forwards simulation

6.1 Fine-grained Module Translations

In chapter 5 we saw a number of fine-grained abstract modules for common data structures. We now show how to correctly implement one fine-grained abstract module in terms of another. In order to do this in a general way we introduce the concept of a *fine-grained module translation*.

Definition 6.1 (Fine-grained Module Translations). A *fine-grained module translation* $\tau : \mathbb{A} \to \mathbb{B}$ from fine-grained abstract module $\mathbb{A} = (\textsc{Cmd}_{\mathbb{A}}, \mathcal{S}(\mathcal{M}_{\mathbb{A}}), \mathrm{Ax}[\![(\cdot)]\!]_{\mathbb{A}})$ to fine-grained abstract module $\mathbb{B} = (\textsc{Cmd}_{\mathbb{B}}, \mathcal{S}(\mathcal{M}_{\mathbb{B}}), \mathrm{Ax}[\![(\cdot)]\!]_{\mathbb{B}})$ consists of:

- an *abstraction relation* $\alpha_\tau \subseteq \mathrm{S}_{\mathbb{B}} \times \mathrm{S}_{\mathbb{A}}$; and
- a *substitutive implementation function* $[\![(\cdot)]\!]_\tau : \mathcal{L}_{\mathbb{A}} \to \mathcal{L}_{\mathbb{B}}$ which uniformly substitutes each basic command of $\textsc{Cmd}_{\mathbb{A}}$ with a call to a procedure written in $\mathcal{L}_{\mathbb{B}}$.

We lift the abstraction relation $\alpha_\tau \subseteq \mathrm{S}_{\mathbb{B}} \times \mathrm{S}_{\mathbb{A}}$ to a *predicate translation* $[\![(\cdot)]\!]_\tau$: $\textsc{Pred}_{\mathbb{A}} \to \textsc{Pred}_{\mathbb{B}}$ such that:

$$\mathcal{P}[\![\, [\![P]\!]_\tau \,]\!]e \overset{\text{def}}{=} \{(s_{\mathbb{B}}, \sigma) \mid \text{there exists } s_{\mathbb{A}} \text{ s.t } (s_{\mathbb{A}}, \sigma) \in \mathcal{P}[\![P]\!]e \text{ and } s_{\mathbb{B}} \alpha_\tau s_{\mathbb{A}}\}$$

Additionally, we require that the predicate translation $[\![(\cdot)]\!]_\tau$ preserves disjunction and entailment. When the translation τ is implicit from context, the τ-subscripts on the abstraction relation, implementation function and predicate translation may be dropped.

In the context of a module translation $\tau : \mathbb{A} \to \mathbb{B}$, $\mathbb{A}$ is called the *abstract* or *high-level* module and $\mathbb{B}$ is called the *concrete* or *low-level* module. It is possible for a module to be abstract with respect to one translation and concrete with respect to another. It is also possible for a module to be both the abstract and concrete module with respect to a single translation.

Definition 6.2 (Sound Module Translation)**.** A fine-grained module translation $\tau : \mathbb{A} \to \mathbb{B}$ is said to be *sound* if for all $e \in \textsc{Env}$, $\Gamma \in \textsc{PSEnv}$, $P, Q \in \textsc{Pred}_{\mathbb{A}}$ and $\mathbb{C} \in \mathcal{L}_{\mathbb{A}}$,

$$e, \Gamma \vdash_{\mathbb{A}} \left\{ \ P \ \right\} \mathbb{C} \left\{ \ Q \ \right\} \quad \Longrightarrow \quad e, [\![\Gamma]\!]_\tau \vdash_{\mathbb{B}} \left\{ \ [\![P]\!]_\tau \ \right\} [\![\mathbb{C}]\!]_\tau \left\{ \ [\![Q]\!]_\tau \ \right\}.$$

where

$$[\![\Gamma]\!]_\tau \ = \ \{ f : [\![\mathsf{P}]\!]_\tau \rightarrowtail [\![\mathsf{Q}]\!]_\tau \,|\, (f : \mathsf{P} \rightarrowtail \mathsf{Q}) \in \Gamma \}$$

Intuitively, a sound module translation appears to be a reasonable correctness condition for a module implementation: everything that can be proved about the abstract module also holds for its implementation. There are, however, a few caveats.

Firstly, since we have elected to work with partial correctness, it is acceptable for an implementation to simply loop forever. If termination guarantees are required, they could either be made separately or a logic based on total correctness could be used. We have chosen to work with partial correctness for simplicity and on the basis that partial correctness is generally used in the separation logic and context logic literature [43][63][14].

Secondly, it is possible for the abstraction relation to lose information. For instance, if all predicates were unsatisfiable under translation then it would be possible to soundly implement every abstract command with `skip`. However, such an implementation is clearly useless. One way of mitigating this would be to consider a set of *initial predicates* that must be satisfiable under translation. A triple whose precondition is such an initial predicate is then meaningful under translation, since it cannot hold vacuously. A more stringent approach would be to require that the abstraction relation α_τ be surjective, and therefore every satisfiable predicate must be satisfiable under translation. However, we shall see that this condition is not met by all of the natural implementations we consider (§6.2.2 and §6.3.4 in particular).

We shall now look in detail at our two techniques for constructing sound module translations. We first discuss the locality-breaking technique as our results here are simpler than for the locality-preserving technique. Our results for the locality-breaking case are, in fact, very similar to those of our previous work in this area [26]. To establish our 'fiction of locality' we need to show that the implementation of each basic command from the abstract level satisfies the translation of its axioms under every possible frame. We have to make some adaptations to work with fine-grained modules, namely that we now model the data structure with a segment algebra, rather than a context algebra. However, our resulting translations will have much the same structure as before.

We will then turn our attention to the locality-preserving technique for constructing sound module translations. *Locality-preserving translations* closely preserve the structure of the fine-grained abstract module's segment algebra through the translation, which leads to an elegant inductive proof transformation from the abstract level to the concrete level. In particular, segment composition and segment compression at the abstract level correspond to segment composition and segment compression at the concrete level, and so the abstract frame rules (Frame and Rev Frame) are transformed to their corresponding concrete level counterparts. However, a great deal of care has to be taken to handle any locality mismatches between the abstract and concrete levels, and thus create the required fiction of disjointness. In particular, we have to find a way to reason about the state that is shared between the concrete representations of abstractly disjoint data structures.

6.1.1 Modularity

It is an important property of module translations that they be composable. Given module translations $\tau_1 : \mathbb{A}_1 \to \mathbb{A}_2$ and $\tau_2 : \mathbb{A}_2 \to \mathbb{A}_3$, we construct the module translation $\tau_2 \bullet \tau_1 : \mathbb{A}_1 \to \mathbb{A}_3$ in the natural fashion. If the module translations τ_1 and τ_2 are both sound, then so is their composition $\tau_2 \bullet \tau_1$. This allows us to construct module translations in a stepwise fashion.

A module translation $\tau : \mathbb{A}_1 \to \mathbb{A}_2$ can be naturally lifted to a module translation $\tau + \mathbb{B} : \mathbb{A}_1 + \mathbb{B} \to \mathbb{A}_2 + \mathbb{B}$ for any module $\mathbb{B}$. If τ is a sound module translation we might also expect $\tau + \mathbb{B}$ to be sound, but it is not obvious that this is the case. The techniques for constructing sound module translations that we introduce in this chapter do, however, admit such a lifting. This is because they transform proofs from module $\mathbb{A}_1$ to $\mathbb{A}_2$ in a fashion that preserves any additional module component. Thus, these techniques are modular, since transformations for independent modules can be combined in a soundness preserving fashion.

6.2 Locality-Breaking Translations

Sometimes the locality exhibited by a high-level module and the locality exhibited by its low-level implementation have no correspondence. We introduce *locality breaking translations* which have a low burden of proof for a sound module translation in

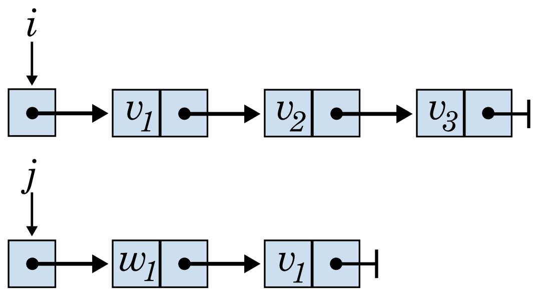

Figure 6.3: A representation of the list-store $i \Mapsto [\, v_1 : v_2 : v_3 \,] * j \Mapsto [\, w_1 : v_1 \,]$ as singly-linked lists in the heap.

such cases. We show that locality breaking translations give rise to sound module translations using similar theory as our previous work in this area [26]. We establish our 'fiction of locality' by showing that the implementation of the high-level basic commands satisfy the translation of their axioms under every possible frame.

We give a motivating example to illustrate this technique by providing a locality breaking translation $\tau_1 : \mathbb{L} \to \mathbb{H}$ from our list module $\mathbb{L}$ into our heap module $\mathbb{H}$. Recall that the fine-grained list module $\mathbb{L}$ provides an addressed set of lists of unique values. Later in this chapter we will see that this list module can be used as part of an implementation of our fine-grained tree module $\mathbb{T}$. In particular, such lists provide a good implementation of the child list of a tree node.

In our motivating example we implement each list from our abstract list module as a singly-linked list in the heap. An example of the list-store viewed in this way is shown in Figure 6.3. At the abstract level we were able to think of the operation of removing the value v_3 from the list at address i as requiring just the resource $i \Mapsto v_3$. However, in our linked list implementation the list at address i must be traversed from its head, all the way to the node containing the value v_3, in this case the whole list. We consider a direct approach to reasoning about the correctness of this implementation.

In order to prove the soundness of a module translation, it is necessary to demonstrate that there is a transformation from high-level proofs about programs that use the abstract module to low-level proofs of those programs which implement the module. Since our definition of predicate translations preserves disjunctions and entailments, as well as the variable store, the majority of our proof rules can be directly converted into their low-level counterparts. The three obvious exceptions to this are the separating frame rule Sep Frame, the revelation frame rule Rev Frame and the axiom rule Axiom. When we consider a module translation that breaks the the locality present at the abstract-level, we can restrict our proofs to those that only make use of the frame rules in a limited fashion. Intuitively this makes sense, as the purpose of the frame rules is just to factor out the parts of the program state that do not play a role in the program under consideration.

In segment logic we have two frame rules, whose interaction provides the ability to extend the program state. When thinking about locality-breaking translations it is enough to combine both of these rules into one frame rule and obtain a more standard view of the frame. The following SR Frame rule, where $\Pi \subseteq \mathrm{X}_{\mathbb{A}}$ is a set of abstract labels, captures the behaviour of both the Sep Frame and Rev Frame rules:

$$\text{SR Frame} : \quad \frac{e, \Gamma \vdash \left\{\ P\ \right\} \mathbb{C} \left\{\ Q\ \right\}}{e, \Gamma \vdash \left\{\ \Pi \circledR (P * R)\ \right\} \mathbb{C} \left\{\ \Pi \circledR (Q * R)\ \right\}}$$

Note that if we let $\Pi = \emptyset$ then we recover the separation frame rule (Sep Frame). Similarly, if we let $\Pi = \{\alpha\}$ and $R = \mathsf{emp}$ then we recover the revelation frame rule (Rev Frame).

It is commonly understood in the local reasoning community that it is possible to transform any proof into an equivalent proof in which the frame rule is only applied to basic statements (that is, basic commands and variable assignments) by factoring in the extra state earlier in the proof (that is, at the leaves of the proof). This intuition can be formalised by the following lemma:

Lemma 6.3 (Restricted-Frame Derivations)**.** Let $\mathbb{A}$ be a fine-grained abstract module. If there is a proof derivation of $e, \Gamma \vdash_{\mathbb{A}} \{P\}\, \mathbb{C}\, \{Q\}$ then there is also a derivation that only uses the SR frame rule in the following ways:

$$\dfrac{\dfrac{}{e, \Gamma \vdash_{\mathbb{A}} \{P\}\, \mathbb{C}\, \{Q\}}(\dagger)}{e, \Gamma \vdash_{\mathbb{A}} \{\Pi \circledR (P * R)\}\, \mathbb{C}\, \{\Pi \circledR (Q * R)\}}\text{SR Frame}$$

$$\dfrac{\dfrac{}{e, \Gamma \vdash_{\mathbb{A}} \{P\}\, \mathbb{C}\, \{Q\}}(\dagger)}{e, \Gamma \vdash_{\mathbb{A}} \{P * (\mathsf{emp} \times \sigma)\}\, \mathbb{C}\, \{Q * (\mathsf{emp} \times \sigma)\}}\text{SR Frame}$$

where (†) is either Axiom or Assgn. We prove this Lemma in §6.2.1.

Consider a module translation $\tau : \mathbb{A} \to \mathbb{B}$. Lemma 6.3 implies that it is only necessary to provide proofs of $e, [\![\gamma]\!]_\tau \vdash_{\mathbb{B}} \{[\![P]\!]_\tau\}\, [\![\mathbb{C}]\!]_\tau\, \{[\![Q]\!]_\tau\}$ when there is a proof of $e, \Gamma \vdash_{\mathbb{A}} \{P\}\, \mathbb{C}\, \{Q\}$ having the prescribed form. So long as there are proofs that the implementation of each command in $\mathrm{Cmd}_{\mathbb{A}}$ satisfies the translation of its axioms under every possible frame, the proof in $\mathbb{A}$ can be transformed into a proof in $\mathbb{B}$ by straightforward induction. In fact, we only need to consider singleton frames (that is, individual pairs $\bar{x} \subseteq \mathrm{X}_{\mathbb{A}}$ and $s_0 \in S_{\mathbb{A}}$) as we can treat any arbitrary frame as the disjunction of singleton frames and apply the Disj rule. We can further reduce our considerations to those singleton frames with no variable store component, since the variable store component can be added by the Sep Frame rule at the low-level. These considerations are formalised in the definition of a locality-breaking translation.

Definition 6.4 (Locality-Breaking Translations). A *locality-breaking translation* $\tau : \mathbb{A} \to \mathbb{B}$ is a module translation having the property that, for all $e \in \textsc{Env}$, $\Gamma \in \text{PS}\textsc{Env}$, $s \in S_{\mathbb{A}}$, $\bar{x} \subseteq \mathrm{X}_{\mathbb{A}}$, $\varphi \in \textsc{Cmd}_{\mathbb{A}}$ and $(P, Q) \in \textsc{Ax}[\![\varphi]\!]_{\mathbb{A}}$ there is a derivation of,

$$e, [\![\Gamma]\!]_\tau \vdash_{\mathbb{B}} \{[\![\Pi \circledR (P * R)]\!]_\tau\}\, [\![\varphi]\!]_\tau\, \{[\![\Pi \circledR (Q * R)]\!]_\tau\}$$

where $e(\Pi) = \bar{x}$ and $\mathcal{P}[\![R]\!]e = \{(s, \mathsf{emp})\}$.

Theorem 6.5 (Locality-Breaking Translation Soundness). A locality-breaking translation is a sound module translation.

A locality-breaking translation transforms proofs that use locality, in the form of the SR rule, at the abstract level into proofs that do not. To do so, we must directly prove that the abstract SR Frame rule is sound with respect to the implementation of each module operation. Hence we say that such a module translations provides a *fiction of locality.*

6.2.1 Soundness of Locality-Breaking Translations

Before we embark on the proof of our soundness theorem, we first give the proof of Lemma 6.3. The result is a special case of the more general result, that if there is a derivation of $e, \Gamma \vdash_{\mathbb{A}} \{P\}\, \mathbb{C}\, \{Q\}$ then there is a derivation of $e, F(\Gamma) \vdash_{\mathbb{A}} \{P\}\, \mathbb{C}\, \{Q\}$ with the required property, where

$$F(\Gamma) = \left\{ \mathtt{f} : \Pi \circledR (\mathsf{P} * R) \rightarrowtail \Pi \circledR (\mathsf{Q} * R) \;\middle|\; \begin{array}{l} R \in (\textsc{Env} \to \mathcal{P}(\mathrm{S}_{\mathbb{A}})), \\ \Pi \in (\textsc{Env} \to \mathcal{P}(\mathrm{X})) \\ \text{and } (\mathtt{f} : \mathsf{P} \rightarrowtail \mathsf{Q}) \in \Gamma \end{array} \right\}.$$

Note that $\Gamma \subseteq F(\Gamma)$ and $F(\Gamma) = F(F(\Gamma))$. Since procedure specifications are only relevant to the PDef and PCall rules, we omit them when considering the other rules. We also omit the logical environment in these cases, as this is unchanged by the proof transformation.

The proof of the generalised statement is by induction on the depth of the derivation. If the last rule applied in the derivation is anything other than the SR Frame rule or the PDef rule then it is simple to transform the derivation: simply apply the induction hypothesis to transform all of the premises and then apply the last rule using $F(\Gamma)$ in place of Γ. We now consider the two remaining cases, where the last rule applied is (i) SR Frame and (ii) PDef.

(i) Consider the case where the last rule of the derivation is SR:

$$\dfrac{\dfrac{\vdots}{\{P\}\,\mathbb{C}\,\{Q\}}(\star)}{\{\Pi\circledR(P*R)\}\,\mathbb{C}\,\{\Pi\circledR(Q*R)\}}\text{SR \textsc{Frame}}$$

Recall that the SR rule incorporates both the Frame and Rev Frame rules. By applying the disjunction rule, this can be reduced to the case of singleton frames R' where $\mathcal{P}\llbracket R'\rrbracket e = \{(s,\sigma)\}$, transforming the derivation as follows:

$$\dfrac{\text{for all } \mathcal{P}\llbracket R'\rrbracket e \subseteq \mathcal{P}\llbracket R\rrbracket e \quad \dfrac{\dfrac{\vdots}{\{P\}\,\mathbb{C}\,\{Q\}}(\star)}{\{\Pi\circledR(P*R')\}\,\mathbb{C}\,\{\Pi\circledR(Q*R')\}}\text{SR \textsc{Frame}}}{\{\Pi\circledR(P*R)\}\,\mathbb{C}\,\{\Pi\circledR(Q*R)\}}\textsc{Disj}$$

Now consider cases for $(\star)$, the last rule applied before SR Frame.

If the rule is Cons then, since $\mathcal{P}\llbracket P\rrbracket e \subseteq \mathcal{P}\llbracket P'\rrbracket e$ implies that $\mathcal{P}\llbracket \Pi\circledR(P*R')\rrbracket e \subseteq \mathcal{P}\llbracket \Pi\circledR(P'*R')\rrbracket e$, the application of the SR Frame rule can be moved earlier in the derivations, transforming it as follows:

$$\dfrac{\begin{array}{c}\mathcal{P}\llbracket \Pi\circledR(P*R')\rrbracket e \subseteq \mathcal{P}\llbracket \Pi\circledR(P'*R')\rrbracket e \\ \mathcal{P}\llbracket \Pi\circledR(Q'*R')\rrbracket e \subseteq \mathcal{P}\llbracket \Pi\circledR(Q*R')\rrbracket e\end{array} \quad \dfrac{\dfrac{\vdots}{\{P'\}\,\mathbb{C}\,\{Q'\}}}{\{\Pi\circledR(P'*R')\}\,\mathbb{C}\,\{\Pi\circledR(Q'*R')\}}\text{SR \textsc{Frame}}}{\{\Pi\circledR(P*R')\}\,\mathbb{C}\,\{\Pi\circledR(Q*R')\}}\textsc{Cons}$$

The application of the SR Frame rule can then be further pushed up the derivation tree by the inductive hypothesis.

If the rule is Disj then, since $*$ distributes over $\vee$, the derivation can be transformed as follows:

$$\dfrac{\text{for all } i\in I \quad \dfrac{\dfrac{\vdots}{\{P_i\}\,\mathbb{C}\,\{Q_i\}}}{\{\Pi\circledR(P_i*R')\}\,\mathbb{C}\,\{\Pi\circledR(Q_i*R')\}}\text{SR \textsc{Frame}}}{\{\Pi\circledR(\bigvee_{i\in I} P_i*R')\}\,\mathbb{C}\,\{\Pi\circledR(\bigvee_{i\in I} Q_i*R')\}}\textsc{Disj}$$

The application of the SR Frame rule can then be further pushed up the derivation tree by the inductive hypothesis.

If the rule is Local then it is possible that the frame R' includes a program variable with the same name as that being added by the `local` block. This means that the frame cannot in general be pushed into the `local` block. However, the frame can be split into its data structure and variable store components. That is $R' \Leftrightarrow R'_1 * R'_2$ where $\mathcal{P}[\![R'_1]\!]e = \{(s, \mathsf{emp})\}$ and $\mathcal{P}[\![R'_2]\!]e = \{(\mathsf{emp}, \sigma)\}$. Moreover, since the variable store cannot contain any addresses or hole labels we also know that $\Pi\circledR(P * R') \Leftrightarrow \Pi\circledR(P * R'_1) * R'_2$. The derivation can then be transformed as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\{P * \mathtt{x} \Rightarrow -\}\,\mathbb{C}'\,\{Q * \mathtt{x} \Rightarrow -\}}}{\{\Pi\circledR(P * R'_1 * \mathtt{x} \Rightarrow -)\}\,\mathbb{C}'\,\{\Pi\circledR(Q * R'_1 * \mathtt{x} \Rightarrow -)\}}\;\textsc{SR Frame}}{\{\Pi\circledR(P * R'_1) * \mathtt{x} \Rightarrow -\}\,\mathbb{C}'\,\{\Pi\circledR(Q * R'_1) * \mathtt{x} \Rightarrow -\}}\;\textsc{Cons}}{\{\Pi\circledR(P * R'_1)\}\,\mathtt{local\ x\ in}\ \mathbb{C}'\,\{\Pi\circledR(Q * R'_1)\}}\;\textsc{Local}}{\{\Pi\circledR(P * R'_1) * R'_2\}\,\mathtt{local\ x\ in}\ \mathbb{C}'\,\{\Pi\circledR(Q * R'_1) * R'_2\}}\;\textsc{SR Frame}}{\{\Pi\circledR(P * R')\}\,\mathtt{local\ x\ in}\ \mathbb{C}'\,\{\Pi\circledR(Q * R')\}}\;\textsc{Cons}$$

The side condition for the Local rule, that $\mathcal{P}[\![\Pi\circledR(P * R'_1\})]\!]e \cap \mathsf{vsafe}(\mathtt{x}) \equiv \emptyset$ follows from the original side condition that $\mathcal{P}[\![P]\!]e \cap \mathsf{vsafe}(\mathtt{x}) \equiv \emptyset$. The applications of the frame rule are now either of the variable only form, or can be further pushed up the derivation tree by the inductive hypothesis.

If the rule is PCall then it is again necessary to split the frame R' into its data structure and variable store components R'_1 and R'_2 as above. The PCall rule uses some $\mathtt{f} : \mathsf{P} \rightarrowtail \mathsf{Q} \in \Gamma$. By definition $\mathtt{f} : \Pi\circledR(\mathsf{P} * R'') \rightarrowtail \Pi\circledR(\mathsf{Q} * R'') \in F(\Gamma)$, for any R'' describing only some part of the data structure, in particular R'_1. The derivation can then be transformed as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathcal{P}[\![\overrightarrow{\mathtt{r}} \Rightarrow \overrightarrow{v} * \sigma']\!]e \subseteq \mathsf{vsafe}(\overrightarrow{E})}{e, F(\Gamma) \vdash_{\mathbb{A}} \begin{array}{c}\left\{\Pi\circledR\left(\mathsf{P}\left(\mathcal{E}[\![\overrightarrow{E}]\!]\sigma'[\overrightarrow{\mathtt{r}} \mapsto \overrightarrow{v}]\right) * R'_1\right) * (\overrightarrow{\mathtt{r}} \Rightarrow \overrightarrow{v} * \sigma')\right\}\\ \mathtt{call}\ \overrightarrow{\mathtt{r}} := \mathtt{f}\ (\overrightarrow{E})\\ \{\exists\overrightarrow{w}.\,\Pi\circledR\,(\mathsf{Q}\,(\overrightarrow{w}) * R'_1) * \overrightarrow{\mathtt{r}} \Rightarrow \overrightarrow{w} * \sigma')\}\end{array}}\;\textsc{PCall}}{e, F(\Gamma) \vdash_{\mathbb{A}} \begin{array}{c}\left\{\Pi\circledR\left(\left(\mathsf{P}\left(\mathcal{E}[\![\overrightarrow{E}]\!]\sigma'[\overrightarrow{\mathtt{r}} \mapsto \overrightarrow{v}]\right) * (\overrightarrow{\mathtt{r}} \Rightarrow \overrightarrow{v} * \sigma')\right) * R'_1\right)\right\}\\ \mathtt{call}\ \overrightarrow{\mathtt{r}} := \mathtt{f}\ (\overrightarrow{E})\\ \{\Pi\circledR\,(\exists\overrightarrow{w}.\,(\mathsf{Q}\,(\overrightarrow{w}) * \overrightarrow{\mathtt{r}} \Rightarrow \overrightarrow{w} * \sigma') * R'_1)\}\end{array}}\;\textsc{Cons}}{e, F(\Gamma) \vdash_{\mathbb{A}} \begin{array}{c}\left\{\Pi\circledR\left(\left(\mathsf{P}\left(\mathcal{E}[\![\overrightarrow{E}]\!]\sigma'[\overrightarrow{\mathtt{r}} \mapsto \overrightarrow{v}]\right) * (\overrightarrow{\mathtt{r}} \Rightarrow \overrightarrow{v} * \sigma')\right) * R'_1\right) * R'_2\right\}\\ \mathtt{call}\ \overrightarrow{\mathtt{r}} := \mathtt{f}\ (\overrightarrow{E})\\ \{\Pi\circledR\,(\exists\overrightarrow{w}.\,(\mathsf{Q}\,(\overrightarrow{w}) * \overrightarrow{\mathtt{r}} \Rightarrow \overrightarrow{w} * \sigma') * R'_1) * R'_2\}\end{array}}\;\textsc{SR Frame}}{e, F(\Gamma) \vdash_{\mathbb{A}} \begin{array}{c}\left\{\Pi\circledR\left(\left(\mathsf{P}\left(\mathcal{E}[\![\overrightarrow{E}]\!]\sigma'[\overrightarrow{\mathtt{r}} \mapsto \overrightarrow{v}]\right) * (\overrightarrow{\mathtt{r}} \Rightarrow \overrightarrow{v} * \sigma')\right) * R'\right)\right\}\\ \mathtt{call}\ \overrightarrow{\mathtt{r}} := \mathtt{f}\ (\overrightarrow{E})\\ \{\Pi\circledR\,(\exists\overrightarrow{w}.\,(\mathsf{Q}\,(\overrightarrow{w}) * \overrightarrow{\mathtt{r}} \Rightarrow \overrightarrow{w} * \sigma') * R')\}\end{array}}\;\textsc{Cons}$$

The application of the SR Frame rule is now in the variable only form.

The remaining cases for the rule applied at the penultimate step ($\star$) are straightforward.

(ii) Now consider the case where the last rule applied is the PDef rule:

$$\dfrac{\text{for all } (\mathtt{f}_i : \mathsf{P}_i \rightarrowtail \mathsf{Q}_i) \in \Gamma \quad \dfrac{\vdots}{e, \Gamma', \Gamma \vdash_{\mathbb{A}} \begin{array}{c}\{\exists \overrightarrow{v}.\, \mathsf{P}_i(\overrightarrow{v}) * \overrightarrow{\mathtt{x}_i} \Rightarrow \overrightarrow{v} * \overrightarrow{\mathtt{r}_i} \Rightarrow -\} \\ \mathbb{C}_i \\ \{\exists \overrightarrow{w}.\, \mathsf{Q}_i(\overrightarrow{w}) * \overrightarrow{\mathtt{x}_i} \Rightarrow - * \overrightarrow{\mathtt{r}_i} \Rightarrow \overrightarrow{w}\}\end{array}}}{(\ddagger)}$$

$$\dfrac{(\ddagger) \quad \dfrac{\vdots}{e, \Gamma', \Gamma \vdash_{\mathbb{A}} \{P\}\, \mathbb{C}'\, \{Q\}}}{e, \Gamma' \vdash_{\mathbb{A}} \{P\}\, \mathtt{procs}\ \overrightarrow{\mathtt{r}_1} := \mathtt{f}_1(\overrightarrow{\mathtt{x}_1})\{\mathbb{C}_1\}, ..., \overrightarrow{\mathtt{r}_k} := \mathtt{f}_k(\overrightarrow{\mathtt{x}_k})\{\mathbb{C}_k\}\ \mathtt{in}\ \mathbb{C}'\, \{Q\}}\ \text{PDef}$$

The derivations for the function bodies can be extended by applying the SR Frame rule, for all $R'' \in (\text{Env} \to \mathcal{P}(\mathrm{S}_{\mathbb{A}}))$, $\Pi \in (\text{Env} \to \mathcal{P}(\mathrm{X}))$ and $(\mathtt{f}_i : \mathsf{P}_i \rightarrowtail \mathsf{Q}_i) \in \Gamma$, to give:

$$\dfrac{\dfrac{\dfrac{\vdots}{e, \Gamma', \Gamma \vdash_{\mathbb{A}} \begin{array}{c}\{\exists \overrightarrow{v}.\, \mathsf{P}_i(\overrightarrow{v}) * \overrightarrow{\mathtt{x}_i} \Rightarrow \overrightarrow{v} * \overrightarrow{\mathtt{r}_i} \Rightarrow -\} \\ \mathbb{C}_i \\ \{\exists \overrightarrow{w}.\, \mathsf{Q}_i(\overrightarrow{w}) * \overrightarrow{\mathtt{x}_i} \Rightarrow - * \overrightarrow{\mathtt{r}_i} \Rightarrow \overrightarrow{w}\}\end{array}}}{e, \Gamma', \Gamma \vdash_{\mathbb{A}} \begin{array}{c}\{\Pi Ⓡ\, (\exists \overrightarrow{v}.\, (\mathsf{P}_i(\overrightarrow{v}) * \overrightarrow{\mathtt{x}_i} \Rightarrow \overrightarrow{v} * \overrightarrow{\mathtt{r}_i} \Rightarrow -) * (R'' \times \mathsf{emp}))\} \\ \mathbb{C}_i \\ \{\Pi Ⓡ\, (\exists \overrightarrow{w}.\, (\mathsf{Q}_i(\overrightarrow{w}) * \overrightarrow{\mathtt{x}_i} \Rightarrow - * \overrightarrow{\mathtt{r}_i} \Rightarrow \overrightarrow{w}) * (R'' \times \mathsf{emp}))\}\end{array}}\ \text{SR Frame}}{e, \Gamma', \Gamma \vdash_{\mathbb{A}} \begin{array}{c}\{\exists \overrightarrow{v}.\, \Pi Ⓡ(\mathsf{P}_i(\overrightarrow{v}) * R'') * \overrightarrow{\mathtt{x}_i} \Rightarrow \overrightarrow{v} * \overrightarrow{\mathtt{r}_i} \Rightarrow -\} \\ \mathbb{C}_i \\ \{\exists \overrightarrow{w}.\, \Pi Ⓡ(\mathsf{Q}_i(\overrightarrow{w}) * R'') * \overrightarrow{\mathtt{x}_i} \Rightarrow - * \overrightarrow{\mathtt{r}_i} \Rightarrow \overrightarrow{w}\}\end{array}}\ \text{Cons}$$

These derivations and the derivation of the premise $e, \Gamma', \Gamma \vdash_{\mathbb{A}} \{P\}\, \mathbb{C}'\, \{Q\}$ can be transformed by the inductive hypothesis so that they use the frame rule in the required fashion and work with the procedure specification environment $F(\Gamma', \Gamma) = F(\Gamma'), F(\Gamma)$. These derivations can then be recombined to give the required derivation as follows:

$$\dfrac{\text{for all } (\mathtt{f}_i : \mathsf{P}_i \rightarrowtail \mathsf{Q}_i) \in F(\Gamma) \quad \dfrac{\vdots}{e, F(\Gamma',\Gamma) \vdash_{\mathbb{A}} \begin{array}{c} \{\exists \overrightarrow{v}.\, \mathsf{P}_i(\overrightarrow{v}) * \overrightarrow{\mathtt{x}_i} \Rightarrow \overrightarrow{v} * \overrightarrow{\mathtt{r}_i} \Rightarrow -\} \\ \mathbb{C}_i \\ \{\exists \overrightarrow{w}.\, \mathsf{Q}_i(\overrightarrow{w}) * \overrightarrow{\mathtt{x}_i} \Rightarrow - * \overrightarrow{\mathtt{r}_i} \Rightarrow \overrightarrow{w}\}\end{array}}}{(\ddagger)}$$

$$\dfrac{(\ddagger) \quad \dfrac{\vdots}{e, F(\Gamma',\Gamma) \vdash_{\mathbb{A}} \{P\}\, \mathbb{C}' \,\{Q\}}}{e, F(\Gamma') \vdash_{\mathbb{A}} \{P\}\, \mathtt{procs}\ \overrightarrow{\mathtt{r}_1} := \mathtt{f}_1(\overrightarrow{\mathtt{x}_1})\{\mathbb{C}_1\}, ..., \overrightarrow{\mathtt{r}_k} := \mathtt{f}_k(\overrightarrow{\mathtt{x}_k})\{\mathbb{C}_k\}\ \mathtt{in}\ \mathbb{C}\,\{Q\}}\ \textsc{PDef}$$

The two further conditions on the PDef rule not included above, that the procedure specification environment Γ only specifies the procedures that are defined in the `procs` block under consideration and that these procedures must have different names to any that occur in the existing procedure specification environment Γ', hold for the transformed derivation because F preserves the names of the functions in the procedure specifications.

This concludes the proof of Lemma 6.3.

Let $\tau : \mathbb{A} \to \mathbb{B}$ be a locality-breaking translation. To show that τ is a sound module translation, it is necessary to establish that whenever there is a derivation of $e, \Gamma \vdash_{\mathbb{A}} \{P\}\, \mathbb{C}\, \{Q\}$ there is a derivation of $e, \llbracket\Gamma\rrbracket_\tau \vdash_{\mathbb{B}} \{\llbracket P\rrbracket_\tau\}\, \llbracket\mathbb{C}\rrbracket_\tau\, \{\llbracket Q\rrbracket_\tau\}$. First transform the high-level derivation into a restricted-frame derivation using Lemma 6.3. Then transform the resulting derivation into the required low-level derivation by replacing each subderivation of the form

$$\dfrac{\dfrac{}{e, \Gamma \vdash_{\mathbb{A}} \{P\}\, \varphi\, \{Q\}}\ \textsc{Axiom}}{e, \Gamma \vdash_{\mathbb{A}} \{\Pi\circledR(P * R)\}\, \varphi\, \{\Pi\circledR(Q * R)\}}\ \textsc{SR Frame}$$

with the derivation

$$\dfrac{\text{for all } \mathcal{P}\llbracket R'\rrbracket e \in \mathcal{P}\llbracket R\rrbracket e \quad \dfrac{\dfrac{\dfrac{(\star)}{e, \llbracket\Gamma\rrbracket_\tau \vdash_{\mathbb{B}} \{\llbracket\Pi\circledR(P * R_1')\rrbracket_\tau\}\, \llbracket\varphi\rrbracket_\tau\, \{\llbracket\Pi\circledR(Q * R_1')\rrbracket_\tau\}}}{e, \llbracket\Gamma\rrbracket_\tau \vdash_{\mathbb{B}} \{\llbracket\Pi\circledR(P * R_1') * R_2'\rrbracket_\tau\}\, \llbracket\varphi\rrbracket_\tau\, \{\llbracket\Pi\circledR(Q * R_1') * R_2'\rrbracket_\tau\}}\ \textsc{SR Frame}}{e, \llbracket\Gamma\rrbracket_\tau \vdash_{\mathbb{B}} \{\llbracket\Pi\circledR(P * R')\rrbracket_\tau\}\, \llbracket\varphi\rrbracket\, \{\llbracket\Pi\circledR(Q * R')\rrbracket_\tau\}}\ \textsc{Cons}}{e, \llbracket\Gamma\rrbracket_\tau \vdash_{\mathbb{B}} \{\llbracket\Pi\circledR(P * R)\rrbracket_\tau\}\, \llbracket\varphi\rrbracket_\tau\, \{\llbracket\Pi\circledR(Q * R)\rrbracket_\tau\}}\ \textsc{Disj}$$

where $(\star)$ stands for the framed derivation provided by the locality-breaking translation (Definition 6.4), and replacing all other rules with their low-level equivalents. In our replacement derivation $\mathcal{P}\llbracket R'\rrbracket e = \{(s, \sigma)\}$, $\mathcal{P}\llbracket R_1'\rrbracket e = \{(s, \mathsf{emp})\}$ and $\mathcal{P}\llbracket R_2'\rrbracket =$

$\{(\mathsf{emp}, \sigma)\}$. This means that $R' \Leftrightarrow R'_1 * R'_2$ and $\Pi Ⓡ (P * R') \Leftrightarrow \Pi Ⓡ (P * R'_1) * R'_2$ for all Π and P.

This completes the proof of Theorem 6.5.

Including the Conjunction Rule

If we wish to add the conjunction rule to the locality-breaking theory, we can add a case to the proof of Lemma 6.3 to deal with pushing the SR Frame rule over the Conj rule, in a similar fashion to the Disj case. The result requires that the segment algebra $\mathcal{S}(\mathcal{M}_{\mathbb{A}})$ be cancellative.

Definition 6.6 (Cancellativity).
A segment algebra $\mathcal{S}(\mathcal{M}) = (S, \mathsf{emp}, \leftarrow, fa, fh, +, \mathsf{comp})$ is cancellative if, for all $s_0, s_1, s_2 \in S$, $s_0 + s_1 = s_0 + s_2$ implies $s_1 = s_2$.

Cancellativity ensures that, in the case of singleton frames $\{(s, \sigma)\}$, we have $(\bigwedge_{i \in I} P_i) * \{(s, \sigma)\} \equiv \bigwedge_{i \in I}(P_i * \{(s, \sigma)\})$[1]. It is also necessary for the predicate translation $[\![(\cdot)]\!]_\tau$ to distribute over conjunction; that is, $[\![P \wedge Q]\!]_\tau \equiv [\![P]\!]_\tau \wedge [\![Q]\!]_\tau$. This is equivalent to the condition that the abstraction relation α is functional; that is, it defines a partial function from concrete states to abstract states.

6.2.2 Module Translation $\tau_1 : \mathbb{L} \to \mathbb{H}$

Our first module translation example is an implementation of the fine-grained list-store module $\mathbb{L}$ with singly linked lists in the heap module $\mathbb{H}$.

Notation: To simplify the presentation of the model part of our translations we define a number of structural and logic operations for sets. For an arbitrary set X

[1]Note that in general this property does not hold

and $p, q \in \mathcal{P}(X)$ we have:

$$\begin{aligned}
\mathsf{true} &\stackrel{\text{def}}{=} \mathcal{P}(X) \\
\mathsf{false} &\stackrel{\text{def}}{=} \emptyset \\
\exists v.\, p &\stackrel{\text{def}}{=} \{s \mid \text{there exists } u \in \textsc{Val}.\, s \in p[u/v]\} \\
\lceil p \rceil &\stackrel{\text{def}}{=} \{\lceil s \rceil \mid s \in p\} \\
(x)(p) &\stackrel{\text{def}}{=} \{(x)(s) \mid s \in p\} \\
p + q &\stackrel{\text{def}}{=} \{s_1 + s_2 \mid s_1 \in p \text{ and } s_2 \in q\} \\
p \star q &\stackrel{\text{def}}{=} \{s_1 \star s_2 \mid s_1 \in p \text{ and } s_2 \in q\} \\
p \wedge q &\stackrel{\text{def}}{=} p \cap q \\
p \vee q &\stackrel{\text{def}}{=} p \cup q \\
p \wedge (x = y) &\stackrel{\text{def}}{=} \begin{cases} p & \text{if } x = y \\ \emptyset & \text{otherwise} \end{cases}
\end{aligned}$$

Note that we only use the $\lceil . \rceil$, $+$ and $\star$ operations when they are well-defined on elements of the set X.

Definition 6.7 ($\tau_1 : \mathbb{L} \to \mathbb{H}$)**.** The module translation $\tau_1 : \mathbb{L} \to \mathbb{H}$ is constructed as follows:

- $\diamond$ the abstraction relation $\alpha_{\tau_1} \subseteq \mathrm{S_H} \times \mathrm{S}_{\mathbb{L}}$ is defined by,

$$\alpha_{\tau_1} \stackrel{\text{def}}{=} \{(sh, sls) \mid sh \in (\!| sls |\!)\}$$

where $(\!|(\cdot)|\!) : \mathrm{S}_{\mathbb{L}} \to \mathcal{P}(\mathrm{S_H})$ is defined by induction on the structure of list-store segments as:

$$\begin{aligned}
(\!|\emptyset|\!) &\stackrel{\text{def}}{=} \{\mathsf{emp}\} \\
(\!|\{(x_i, cl)\}|\!) &\stackrel{\text{def}}{=} \begin{cases} \exists y.\, \lceil \{i \mapsto y\} \star \langle\!\langle cl \rangle\!\rangle^{(y,\mathsf{null})} \rceil & \text{if } x_i = 0_i \text{ and } fh_{\mathrm{L}}(cl) = \emptyset \\ \mathsf{false} & \text{otherwise} \end{cases} \\
(\!|sls_1 \uplus sls_2|\!) &\stackrel{\text{def}}{=} (\!|sls_1|\!) +_{\mathrm{H}} (\!|sls_2|\!)
\end{aligned}$$

and where $\langle\!\langle(\cdot)\rangle\!\rangle^{(\cdot)} : \mathrm{L}_{\mathrm{VAL},\mathrm{X}^{\textsc{Adr}}} \times (\textsc{Adr}_{\mathsf{null}} \times \textsc{Adr}_{\mathsf{null}}) \to \mathcal{P}(\mathrm{H}_{\textsc{Adr},\mathrm{X}})$ is defined by induction on the structure of complete lists as:

$$\begin{aligned}
\langle\!\langle \varepsilon \rangle\!\rangle^{(x,y)} &\stackrel{\text{def}}{=} \{\mathsf{emp}\} \wedge (x = y) \\
\langle\!\langle z_i \rangle\!\rangle^{(x,y)} &\stackrel{\text{def}}{=} \mathsf{false} \\
\langle\!\langle v \rangle\!\rangle^{(x,y)} &\stackrel{\text{def}}{=} \{x \mapsto v{,}y\} \\
\langle\!\langle cl_1 : cl_2 \rangle\!\rangle^{(x,y)} &\stackrel{\text{def}}{=} \exists z.\, \langle\!\langle cl_1 \rangle\!\rangle^{(x,z)} \star \langle\!\langle cl_2 \rangle\!\rangle^{(z,y)}
\end{aligned}$$

◇ the *substitutive implementation function* is given by replacing each list module command with a call to the correspondingly named procedure given in Figure 6.4 and Figure 6.5, where

$$\begin{aligned}
E.\texttt{value} &\overset{\text{def}}{=} E \\
E.\texttt{next} &\overset{\text{def}}{=} E + 1 \\
\texttt{x} := \texttt{newNode}() &\overset{\text{def}}{=} \texttt{x} := \texttt{alloc}(2) \\
\texttt{x} := \texttt{newRoot}() &\overset{\text{def}}{=} \texttt{x} := \texttt{alloc}(1) \\
\texttt{disposeNode}(E) &\overset{\text{def}}{=} \texttt{dispose}(E, 2) \\
\texttt{disposeRoot}(E) &\overset{\text{def}}{=} \texttt{dispose}(E, 1).
\end{aligned}$$

Note that list commands of the form `E.command`(...) are implemented as procedures of the form `command`(`e`, ...), where the value of expression E is passed to the procedure in its first variable `e`.

Notice that this abstraction relation is not surjective, since incomplete and partial lists do not have corresponding heap relations (they are mapped to false). The intuition behind this is that incomplete and partial lists are just a useful tool to enable reasoning about complete lists. It is common for clients of our list module to work with just complete lists, in particular only complete lists can be created or deleted by our module commands. Of course, it is perfectly acceptable to use assertions and specifications that refer to incomplete or partial lists within client proofs. In fact, doing so allows us to provide more fine-grained specifications of list manipulating programs. The transformations of our proofs to their low-level versions will compete any partial lists by making use of Lemma 6.3.

Our choice not to represent incomplete or partial lists at the low-level makes it simpler to prove that τ_1 is a locality-breaking translation. It is only necessary to prove that the axioms hold under the translation for frames that complete all of the lists given in the precondition. In all other cases, the precondition will just translate to false, and so the low-level triple will hold trivially.

Theorem 6.8 (Soundess of τ_1)**.** The module translation τ_1 is a locality-breaking translation.

We do not cover every case of the proof here, but show details for two cases that illustrate the technique of proving the correctness of the axioms under the translation. We first give a proof of a simple case, showing that the implementation of the `deleteList` command satisfies its translated specification in any frame. We then give a proof of a more complex case, showing that the implementation of the

```
proc v := getHead(i){
  local x in
    x := [i] ;
    if x = null then
      v := x
    else
      v := [x.value]
}

proc v := getTail(i){
  local x, y in
    x := [i] ;
    if x = null then
      v := x
    else
      y := [x.next] ;
      while y ≠ null do
        x := y ;
        y := [x.next]
      v := [x.value]
}

proc v := getNext(i, w){
  local x in
    x := [i] ;
    v := [x.value] ;
    while v ≠ w do
      x := [x.next] ;
      v := [x.value]
    x := [x.next] ;
    if x = null then
      v := x
    else
      v := [x.value]
}

proc v := getPrev(i, w){
  local x, y in
    x := [i] ;
    v := [x.value] ;
    if v = w then
      v := null
    else
      while v ≠ w do
        y := x ;
        x := [y.next] ;
        v := [x.value]
      v := [y.value]
}

proc v := pop(i){
  local x, y in
    x := [i] ;
    if x = null then
      v := x
    else
      y := [x.next] ;
      [i] := y ;
      v := [x.value] ;
      disposeNode(x)
}

proc push(i, v){
  local x, y in
    x := newNode() ;
    y := [i] ;
    [x.value] := v ;
    [x.next] := y ;
    [i] := x
}
```

Figure 6.4: Procedures for the heap-based implementation of the list module.

```
proc remove(i, v){
  local u, x, y, z in
    x := [i] ;
    u := [x.value] ;
    y := [x.next] ;
    if u = v then
      [i] := y ;
      disposeNode(x)
    else
      u := [y.value] ;
      while u ≠ v do
        x := y ;
        y := [x.next] ;
        u := [y.value]
      z := [y.next] ;
      [x.next] := z ;
      disposeNode(y)
}

proc i := newList(){
  i := newRoot() ;
  [i] := null
}
```

```
proc insert(i, v, w){
  local u, x, y, z in
    x := [i] ;
    u := [x.value] ;
    while u ≠ v do
      x := [x.next] ;
      u := [x.value]
    y := [x.next] ;
    z := newNode() ;
    [z.value] := w ;
    [z.next] := y ;
    [x.next] := z
}

proc deleteList(i){
  local x, y in
    x := [i] ;
    while x ≠ null do
      y := x ;
      x := [y.next] ;
      disposeNode(y)
    disposeRoot(i)
}
```

Figure 6.5: Procedures for the heap-based implementation of the list module.

`getNext` command satisfies its translated specification in any frame. The implementations of the other basic commands can be shown to satisfy their translated specifications in a similar fashion. Our proofs are analogous to those found in our previous work in this area [27].

Implementation Correctness: `deleteList`

Recall the specification of the `deleteList` command from Figure 5.8.

$$\begin{gathered}\left\{\ i \Mapsto [\,l\,] * \sigma \wedge \mathcal{E}[\![E]\!]\sigma = i\ \right\}\\ E.\mathtt{deleteList}()\\ \left\{\ \sigma\ \right\}\end{gathered}$$

Fix arbitrary $e \in \textsc{Env}$, $i \in \textsc{Adr}$, $l \in \textsc{Val}^*$, $sls \in \mathrm{S}_{\mathbb{L}}$ and $\bar{x} \in \mathcal{P}(\mathrm{X}^{\textsc{Adr}})$ such that $\mathcal{P}[\![R]\!]e = \{(sls, \mathsf{emp})\}$ and $e(\Pi) = \bar{x}$. It is sufficient to show that the procedure body of `deleteList` (from Figure 6.5) meets the following specification:

$$\begin{gathered}\left\{\ [\![\,\mathsf{H}\Pi.\,(R * i \Mapsto [\,l\,] * \mathtt{i} \Rrightarrow i)\,]\!]_{\tau_1}\ \right\}\\ \mathtt{deleteList(i)}\\ \left\{\ [\![\,\mathsf{H}\Pi.\,(R * \mathtt{i} \Rrightarrow i)\,]\!]_{\tau_1}\ \right\}\end{gathered}$$

Now, either this specification holds trivially since the precondition is equivalent to false, or

$$\mathsf{H}\Pi.\,(R * i \Mapsto [\,l\,] * \mathtt{i} \Rrightarrow i) \quad\Leftrightarrow\quad R' * i \Mapsto [\,l\,] * \mathtt{i} \Rrightarrow i$$

with $\mathcal{P}[\![R']\!]e = \{(sls', \mathsf{emp})\}$ for some $sls' \in \mathrm{S}_{\mathbb{L}}$ where all of the lists in sls' are complete. The proof outline for this second case is given in Figure 6.6. In this case the list at i is already a complete list, with no context holes in it. So as long as we do not add any incomplete or partial lists to the list-store in the frame, the translation will be defined.

```
$\{\ [\![\, \mathsf{H}\Pi.\, (R * i \Mapsto [\,l\,] * \mathtt{i} \Rrightarrow i)\,]\!]_{\tau_1}\ \}$
$\{\ [\![\, R' * i \Mapsto [\,l\,] * \mathtt{i} \Rrightarrow i\,]\!]_{\tau_1}\ \}$
$\{\ [\![\, R'\,]\!]_{\tau_1} * \exists y.\, (\lceil i \mapsto y \star \langle\!\langle l \rangle\!\rangle^{(y,\mathsf{null})} \rceil * \mathtt{i} \Rrightarrow i)\ \}$
local x, y in
   $\{\ [\![\, R'\,]\!]_{\tau_1} * \exists y.\, (\lceil i \mapsto y \star \langle\!\langle l \rangle\!\rangle^{(y,\mathsf{null})} \rceil * \mathtt{i} \Rrightarrow i * \mathtt{x} \Rrightarrow - * \mathtt{y} \Rrightarrow -)\ \}$
   x := [i] ;
   $\{\ [\![\, R'\,]\!]_{\tau_1} * \exists y.\, (\lceil i \mapsto y \star \langle\!\langle l \rangle\!\rangle^{(y,\mathsf{null})} \rceil * \mathtt{i} \Rrightarrow i * \mathtt{x} \Rrightarrow y * \mathtt{y} \Rrightarrow -)\ \}$
   $\left\{ \begin{array}{l} [\![\, R'\,]\!]_{\tau_1} * \exists y, l, z.\, (\lceil i \mapsto - \star y \mapsto -,z \star \langle\!\langle l \rangle\!\rangle^{(z,\mathsf{null})} \rceil \times \mathtt{i} \Rrightarrow i * \mathtt{x} \Rrightarrow y * \mathtt{y} \Rrightarrow -) \\ \vee [\![\, R'\,]\!]_{\tau_1} * \exists y.\, (\lceil i \mapsto - \rceil * \mathtt{i} \Rrightarrow i * \mathtt{x} \Rrightarrow \mathsf{null} * \mathtt{y} \Rrightarrow -) \end{array} \right\}$
   while x $\neq$ null do
      $\{\ [\![\, R'\,]\!]_{\tau_1} * \exists y, l, z.\, (\lceil i \mapsto - \star y \mapsto -,z \star \langle\!\langle l \rangle\!\rangle^{(z,\mathsf{null})} \rceil * \mathtt{i} \Rrightarrow i * \mathtt{x} \Rrightarrow y * \mathtt{y} \Rrightarrow -)\ \}$
      y := x ;
      $\{\ [\![\, R'\,]\!]_{\tau_1} * \exists y, l, z.\, (\lceil i \mapsto - \star y \mapsto -,z \star \langle\!\langle l \rangle\!\rangle^{(z,\mathsf{null})} \rceil * \mathtt{i} \Rrightarrow i * \mathtt{x} \Rrightarrow y * \mathtt{y} \Rrightarrow y)\ \}$
      x := [y.next] ;
      $\{\ [\![\, R'\,]\!]_{\tau_1} * \exists y, l, z.\, (\lceil i \mapsto - \star y \mapsto -,z \star \langle\!\langle l \rangle\!\rangle^{(z,\mathsf{null})} \rceil * \mathtt{i} \Rrightarrow i * \mathtt{x} \Rrightarrow z * \mathtt{y} \Rrightarrow y)\ \}$
      disposeNode(y)
      $\{\ [\![\, R'\,]\!]_{\tau_1} * \exists y, l, z.\, (\lceil i \mapsto - \star \langle\!\langle l \rangle\!\rangle^{(z,\mathsf{null})} \rceil * \mathtt{i} \Rrightarrow i * \mathtt{x} \Rrightarrow z * \mathtt{y} \Rrightarrow y)\ \}$
      $\left\{ \begin{array}{l} [\![\, R'\,]\!]_{\tau_1} * \exists y, l, z.\, (\lceil i \mapsto - \star y \mapsto -,z \star \langle\!\langle l \rangle\!\rangle^{(z,\mathsf{null})} \rceil * \mathtt{i} \Rrightarrow i * \mathtt{x} \Rrightarrow y * \mathtt{y} \Rrightarrow -) \\ \vee [\![\, R'\,]\!]_{\tau_1} * \exists y.\, (\lceil i \mapsto - \rceil * \mathtt{i} \Rrightarrow i * \mathtt{x} \Rrightarrow \mathsf{null} * \mathtt{y} \Rrightarrow -) \end{array} \right\}$
   $\{\ [\![\, R'\,]\!]_{\tau_1} * \exists y.\, (\lceil i \mapsto - \rceil * \mathtt{i} \Rrightarrow i * \mathtt{x} \Rrightarrow \mathsf{null} * \mathtt{y} \Rrightarrow -)\ \}$
   disposeRoot(i)
   $\{\ [\![\, R'\,]\!]_{\tau_1} * \exists y.\, (\mathtt{i} \Rrightarrow i * \mathtt{x} \Rrightarrow \mathsf{null} * \mathtt{y} \Rrightarrow -)\ \}$
$\{\ [\![\, R'\,]\!]_{\tau_1} * \exists y.\, \mathtt{i} \Rrightarrow i\ \}$
$\{\ [\![\, R' * \mathtt{i} \Rrightarrow i\,]\!]_{\tau_1}\ \}$
$\{\ [\![\, \mathsf{H}\Pi.\, (R * \mathtt{i} \Rrightarrow i)\,]\!]\ \}$
```

Figure 6.6: Proof outline for the `deleteList` implementation in τ_1.

Implementation Correctness: `getNext`

Recall the specifications of the `getNext` command from Figure 5.7.

$$\left\{\ \alpha_i{\leftarrow}(w+u) * \mathtt{v} \Rightarrow v * \sigma \wedge \mathcal{E}[\![E]\!]\sigma[\mathtt{v} \mapsto v] = i \wedge \mathcal{E}[\![E']\!]\sigma[\mathtt{v} \mapsto v] = w\ \right\}$$
$$\mathtt{v} := E\texttt{.getNext}(E')$$
$$\left\{\ \alpha_i{\leftarrow}(w+u) * \mathtt{v} \Rightarrow u * \sigma\ \right\}$$

$$\left\{\ i \Mapsto [\,\beta_i + w\,] * \mathtt{v} \Rightarrow v * \sigma \wedge \mathcal{E}[\![E]\!]\sigma[\mathtt{v} \mapsto v] = i \wedge \mathcal{E}[\![E']\!]\sigma[\mathtt{v} \mapsto v] = w\ \right\}$$
$$\mathtt{v} := E\texttt{.getNext}(E')$$
$$\left\{\ i \Mapsto [\,\beta_i + w\,] * \mathtt{v} \Rightarrow \mathsf{null} * \sigma\ \right\}$$

Fix arbitrary $e \in \textsc{Env}$, $i \in \textsc{Adr}$, $w, u \in \textsc{Val}$, $sls \in \mathrm{S}_{\mathbb{L}}$ and $\bar{x} \in \mathcal{P}(\mathrm{X}^{\textsc{Adr}})$ such that $\mathcal{P}[\![R]\!]e = \{(sls, \mathsf{emp})\}$ and $e(\Pi) = \bar{x}$. It is sufficient to show that the procedure body of `getNext` (from Figure 6.4) meets the following specifications:

$$\left\{\ [\![\,\mathsf{H}\Pi.\,(R * (\alpha_i{\leftarrow}(w+u) * \mathtt{i} \Rightarrow i * \mathtt{w} \Rightarrow w * \mathtt{v} \Rightarrow -))\,]\!]\ \right\}$$
$$\mathtt{v} := \mathtt{getNext}(\mathtt{i}, \mathtt{w})$$
$$\left\{\ [\![\,\mathsf{H}\Pi.\,(R * (\alpha_i{\leftarrow}(w+u) * \mathtt{i} \Rightarrow i * \mathtt{w} \Rightarrow w * \mathtt{v} \Rightarrow u))\,]\!]\ \right\}$$

$$\left\{\ [\![\,\mathsf{H}\Pi.\,(R * (i \Mapsto [\,\beta_i + w\,] * \mathtt{i} \Rightarrow i * \mathtt{w} \Rightarrow w * \mathtt{v} \Rightarrow -))\,]\!]\ \right\}$$
$$\mathtt{v} := \mathtt{getNext}(\mathtt{i}, \mathtt{w})$$
$$\left\{\ [\![\,\mathsf{H}\Pi.\,(R * (i \Mapsto [\,\beta_i + w\,] * \mathtt{i} \Rightarrow i * \mathtt{w} \Rightarrow w * \mathtt{v} \Rightarrow \mathsf{null}))\,]\!]\ \right\}$$

Consider the first specification for `getNext`. Either this holds trivially, since the precondition is equivalent to false, or

$$\mathsf{H}\Pi.\,(R * (\alpha_i{\leftarrow}(w+u) * \mathtt{i} \Rightarrow i * \mathtt{w} \Rightarrow w * \mathtt{v} \Rightarrow -))$$
$$\Leftrightarrow$$
$$R' * (i \Mapsto [\,l_1 : w : u : l_2\,] * \mathtt{i} \Rightarrow i * \mathtt{w} \Rightarrow w * \mathtt{v} \Rightarrow -)$$

with $\mathcal{P}[\![R']\!]e = \{(sls', \mathsf{emp})\}$ for some $sls' \in \mathrm{S}_{\mathbb{L}}$ where all of the lists in sls' are complete and $l_1, l_2 \in \textsc{Val}^*$ with w and u not in either l_1 or l_2. The proof outline for this second case is given in Figure 6.8. In this case the list segment at i is partial, so we only consider the frames which at least complete the list i and do not add any further incomplete or partial lists to the list-store.

Now consider the second specification for `getNext`. Again, either this holds trivially, since the precondition is equivalent to false, or

$$\mathsf{H}\Pi.\,(R * (i \mapsto [\,\beta_i + w\,] * \mathtt{i} \Rightarrow i * \mathtt{w} \Rightarrow w * \mathtt{v} \Rightarrow -))$$
$$\Leftrightarrow$$
$$R' * (i \mapsto [\,l + w\,] * \mathtt{i} \Rightarrow i * \mathtt{w} \Rightarrow w * \mathtt{v} \Rightarrow -)$$

with $\mathcal{P}[\![R']\!]e = \{sls', \mathsf{emp}\}$ for some $sls' \in \mathrm{S}_{\mathbb{L}}$ where all of the lists in sls' are complete and $l \in \mathrm{VAL}^*$ with w not in l. The proof outline for this second case is given in Figure 6.9. In this case the list segment at i is incomplete: it contains a hole which the frame must fully fill for the translation to be defined. Thus, we only consider frames which at least complete the list i and do not add any further incomplete or partial lists to the list-store.

In both specification cases the `getNext` implementation performs the same search for the value w in the list. The proof outline for this common part is given in Figure 6.7.

6.2.3 Locality-Breaking Limitations

The implementation correctness proofs for the locality-breaking translation considered above are really rather simple. We can reason about the correctness of the implementation, with respect to a given axiom, for all frames in just a few steps. If the frame applied to the axiom's precondition does not complete the list i, or adds some other incomplete list, then the translation of the list-store results in false and our proof obligation is vacuous. The only cases we need to consider in more detail are those cases where the frame completes list i and possibly adds some other complete lists. In such cases, we can always use the separating frame rule at the low-level to hide away these other lists and focus on the implementation's effect on list i. We have seen that a single proof sketch can then be used to show that the translated axiom is satisfied by the implementation.

In general, it is not always the case that we can capture all of the frames that can be applied to our axioms in such an elegant way. As an example, consider providing a locality-breaking translation of our fine-grained abstract tree module $\mathbb{T}$. In a similar way to the translation above we could choose only to translate complete trees into their concrete representations. Regardless of the implementation chosen though, the $\mathtt{appendChild}(\mathtt{n}, \mathtt{m})$ command is going to pose us with a problem.

At the abstract-level we only need to think about the node `n` and the subtree at `m`. However, at the concrete-level (for the non vacuous cases) we have to consider the whole tree. There are three possible states that the tree could be extended to from

```
$\{\ \lceil i \mapsto x \star \langle\!\langle l : w \rangle\!\rangle^{(x,y)} \rceil * \mathtt{i} \Rrightarrow i * \mathtt{w} \Rrightarrow w * \mathtt{v} \Rrightarrow - * \mathtt{x} \Rrightarrow -\ \}$
x := [i] ;
$\{\ \lceil i \mapsto x \star \langle\!\langle l : w \rangle\!\rangle^{(x,y)} \rceil * \mathtt{i} \Rrightarrow i * \mathtt{w} \Rrightarrow w * \mathtt{v} \Rrightarrow - * \mathtt{x} \Rrightarrow x\ \}$
$\left\{\begin{array}{l} \left(\begin{array}{l} \exists v, l', z.\, l = v : l' \land \lceil i \mapsto x \star x \mapsto v{,}z \star \langle\!\langle l' : w \rangle\!\rangle^{(z,y)} \rceil \\ \lor\, l = \varepsilon \land \lceil i \mapsto x \star x \mapsto w{,}y \rceil \end{array}\right) \\ * \mathtt{i} \Rrightarrow i * \mathtt{w} \Rrightarrow w * \mathtt{v} \Rrightarrow - * \mathtt{x} \Rrightarrow x \end{array}\right\}$
v := [x.value] ;
$\left\{\begin{array}{l} \exists v, l_1, l_2, z, z'.\, l : w = l_1 : v : l_2 \land \lceil i \mapsto x \star \langle\!\langle l_1 \rangle\!\rangle^{(x,z)} \star z \mapsto v{,}z' \star \langle\!\langle l_2 \rangle\!\rangle^{(z',y)} \rceil \\ * \mathtt{i} \Rrightarrow i * \mathtt{w} \Rrightarrow w * \mathtt{v} \Rrightarrow v * \mathtt{x} \Rrightarrow z \end{array}\right\}$
while v $\neq$ w do
    $\left\{\begin{array}{l} \exists v, v', l_1, l_2, z, z', z''.\, l : w = l_1 : v : v' : l_2 \\ \land \lceil i \mapsto x \star \langle\!\langle l_1 \rangle\!\rangle^{(x,z)} \star z \mapsto v{,}z' \star z' \mapsto v'{,}z'' \star \langle\!\langle l_2 \rangle\!\rangle^{(z'',y)} \rceil \\ * \mathtt{i} \Rrightarrow i * \mathtt{w} \Rrightarrow w * \mathtt{v} \Rrightarrow v * \mathtt{x} \Rrightarrow z \end{array}\right\}$
    x := [x.next] ;
    $\left\{\begin{array}{l} \exists v, v', l_1, l_2, z, z', z''.\, l : w = l_1 : v : v' : l_2 \\ \land \lceil i \mapsto x \star \langle\!\langle l_1 \rangle\!\rangle^{(x,z)} \star z \mapsto v{,}z' \star z' \mapsto v'{,}z'' \star \langle\!\langle l_2 \rangle\!\rangle^{(z'',y)} \rceil \\ * \mathtt{i} \Rrightarrow i * \mathtt{w} \Rrightarrow w * \mathtt{v} \Rrightarrow v * \mathtt{x} \Rrightarrow z' \end{array}\right\}$
    v := [x.value]
    $\left\{\begin{array}{l} \exists v, v', l_1, l_2, z, z', z''.\, l : w = l_1 : v : v' : l_2 \\ \land \lceil i \mapsto x \star \langle\!\langle l_1 \rangle\!\rangle^{(x,z)} \star z \mapsto v{,}z' \star z' \mapsto v'{,}z'' \star \langle\!\langle l_2 \rangle\!\rangle^{(z'',y)} \rceil \\ * \mathtt{i} \Rrightarrow i * \mathtt{w} \Rrightarrow w * \mathtt{v} \Rrightarrow v' * \mathtt{x} \Rrightarrow z' \end{array}\right\}$
    $\left\{\begin{array}{l} \exists v, l_1, l_2, z, z'.\, l : w = l_1 : v : l_2 \land \lceil i \mapsto x \star \langle\!\langle l_1 \rangle\!\rangle^{(x,z)} \star z \mapsto v{,}z' \star \langle\!\langle l_2 \rangle\!\rangle^{(z',y)} \rceil \\ * \mathtt{i} \Rrightarrow i * \mathtt{w} \Rrightarrow w * \mathtt{v} \Rrightarrow v * \mathtt{x} \Rrightarrow z \end{array}\right\}$
$\{\ \exists z.\, \lceil i \mapsto x \star \langle\!\langle l \rangle\!\rangle^{(x,z)} \star z \mapsto w{,}y \rceil * \mathtt{i} \Rrightarrow i * \mathtt{w} \Rrightarrow w * \mathtt{v} \Rrightarrow w * \mathtt{x} \Rrightarrow z\ \}$
x := [x.next] ;
$\{\ \exists z.\, \lceil i \mapsto x \star \langle\!\langle l \rangle\!\rangle^{(x,z)} \star z \mapsto w{,}y \rceil * \mathtt{i} \Rrightarrow i * \mathtt{w} \Rrightarrow w * \mathtt{v} \Rrightarrow w * \mathtt{x} \Rrightarrow y\ \}$
$\{\ \lceil i \mapsto x \star \langle\!\langle l : w \rangle\!\rangle^{(x,y)} \rceil * \mathtt{i} \Rrightarrow i * \mathtt{w} \Rrightarrow w * \mathtt{v} \Rrightarrow w * \mathtt{x} \Rrightarrow y\ \}$
```

Figure 6.7: Proof outline for the search part of the `getNext` implementation in τ_1 (common part).

```
$\{\ \llbracket \mathsf{H}\Pi.\,(R * (\alpha_i{\leftarrow}(w+u) * \mathtt{i} \Rrightarrow i * \mathtt{w} \Rrightarrow w * \mathtt{x} \Rrightarrow -))\,\rrbracket_{\tau_1}\ \}$
$\left\{\begin{array}{l} \llbracket R' \rrbracket_{\tau_1} * \exists x,y,z.\,\lceil i \mapsto x \star \langle\!\langle l_1 : w\rangle\!\rangle^{(x,y)} \star y \mapsto u,z \star \langle\!\langle l_2\rangle\!\rangle^{(z,\mathsf{null})}\rceil \\ * \,\mathtt{i} \Rrightarrow i * \mathtt{w} \Rrightarrow w * \mathtt{v} \Rrightarrow - \end{array}\right\}$
  $\{\ \exists x,y,z.\,\lceil i \mapsto x \star \langle\!\langle l_1 : w\rangle\!\rangle^{(x,y)}\rceil * \lceil y \mapsto u,z\rceil * \mathtt{i} \Rrightarrow i * \mathtt{w} \Rrightarrow w * \mathtt{v} \Rrightarrow -\ \}$
  local x in
    $\{\ \exists x,y,z.\,\lceil i \mapsto x \star \langle\!\langle l_1 : w\rangle\!\rangle^{(x,y)}\rceil * \lceil y \mapsto u,z\rceil * \mathtt{i} \Rrightarrow i * \mathtt{w} \Rrightarrow w * \mathtt{v} \Rrightarrow - * \mathtt{x} \Rrightarrow -\ \}$
                      (see Figure 6.7)
    $\{\ \exists x,y,z.\,\lceil i \mapsto x \star \langle\!\langle l_1 : w\rangle\!\rangle^{(x,y)}\rceil * \lceil y \mapsto u,z\rceil * \mathtt{i} \Rrightarrow i * \mathtt{w} \Rrightarrow w * \mathtt{v} \Rrightarrow w * \mathtt{x} \Rrightarrow y\ \}$
    if x = null then
      v := x
    else
      v := [x.value]
    $\{\ \exists x,y,z.\,\lceil i \mapsto x \star \langle\!\langle l_1 : w\rangle\!\rangle^{(x,y)}\rceil * \lceil y \mapsto u,z\rceil * \mathtt{i} \Rrightarrow i * \mathtt{w} \Rrightarrow w * \mathtt{v} \Rrightarrow u * \mathtt{x} \Rrightarrow y\ \}$
  $\{\ \exists x,y,z.\,\lceil i \mapsto x \star \langle\!\langle l_1 : w\rangle\!\rangle^{(x,y)}\rceil * \lceil y \mapsto u,z\rceil * \mathtt{i} \Rrightarrow i * \mathtt{w} \Rrightarrow w * \mathtt{v} \Rrightarrow u\ \}$
$\left\{\begin{array}{l} \llbracket R' \rrbracket_{\tau_1} * \exists x,y,z.\,\lceil i \mapsto x \star \langle\!\langle l_1 : w\rangle\!\rangle^{(x,y)} \star y \mapsto u,z \star \langle\!\langle l_2\rangle\!\rangle^{(z,\mathsf{null})}\rceil \\ * \,\mathtt{i} \Rrightarrow i * \mathtt{w} \Rrightarrow w * \mathtt{v} \Rrightarrow v \end{array}\right\}$
$\{\ \llbracket \mathsf{H}\Pi.\,(R * (\alpha_i{\leftarrow}(w+u) * \mathtt{i} \Rrightarrow i * \mathtt{w} \Rrightarrow w * \mathtt{v} \Rrightarrow u))\,\rrbracket_{\tau_1}\ \}$
```

Figure 6.8: Proof outline for the `getNext` implementation in τ_1 (success case).

```
$\{\ \llbracket \mathsf{H}\Pi.\,(R * (i \Mapsto [\,\beta_i + w\,] * \mathtt{i} \Rrightarrow i * \mathtt{w} \Rrightarrow w * \mathtt{v} \Rrightarrow -))\,\rrbracket_{\tau_1}\ \}$
$\{\ \llbracket R' \rrbracket_{\tau_1} * \exists x.\,\lceil i \mapsto x \star \langle\!\langle l : w\rangle\!\rangle^{(x,\mathsf{null})}\rceil * \mathtt{i} \Rrightarrow i * \mathtt{w} \Rrightarrow w * \mathtt{v} \Rrightarrow -\ \}$
  $\{\ \lceil i \mapsto x \star \langle\!\langle l : w\rangle\!\rangle^{(x,\mathsf{null})}\rceil * \mathtt{i} \Rrightarrow i * \mathtt{w} \Rrightarrow w * \mathtt{v} \Rrightarrow -\ \}$
  local x in
    $\{\ \lceil i \mapsto x \star \langle\!\langle l : w\rangle\!\rangle^{(x,\mathsf{null})}\rceil * \mathtt{i} \Rrightarrow i * \mathtt{w} \Rrightarrow w * \mathtt{v} \Rrightarrow - * \mathtt{x} \Rrightarrow -\ \}$
                      (see Figure 6.7)
    $\{\ \lceil i \mapsto x \star \langle\!\langle l : w\rangle\!\rangle^{(x,\mathsf{null})}\rceil * \mathtt{i} \Rrightarrow i * \mathtt{w} \Rrightarrow w * \mathtt{v} \Rrightarrow w * \mathtt{x} \Rrightarrow \mathsf{null}\ \}$
    if x = null then
      v := x
    else
      v := [x.value]
    $\{\ \lceil i \mapsto x \star \langle\!\langle l : w\rangle\!\rangle^{(x,\mathsf{null})}\rceil * \mathtt{i} \Rrightarrow i * \mathtt{w} \Rrightarrow w * \mathtt{v} \Rrightarrow \mathsf{null} * \mathtt{x} \Rrightarrow \mathsf{null}\ \}$
  $\{\ \lceil i \mapsto x \star \langle\!\langle l : w\rangle\!\rangle^{(x,\mathsf{null})}\rceil * \mathtt{i} \Rrightarrow i * \mathtt{w} \Rrightarrow w * \mathtt{v} \Rrightarrow \mathsf{null}\ \}$
$\{\ \llbracket R' \rrbracket_{\tau_1} * \exists x.\,\lceil i \mapsto x \star \langle\!\langle l : w\rangle\!\rangle^{(x,\mathsf{null})}\rceil * \mathtt{i} \Rrightarrow i * \mathtt{w} \Rrightarrow w * \mathtt{v} \Rrightarrow \mathsf{null}\ \}$
$\{\ \llbracket \mathsf{H}\Pi.\,(R * (i \Mapsto [\,\beta_i + w\,] * \mathtt{i} \Rrightarrow i * \mathtt{w} \Rrightarrow w * \mathtt{v} \Rrightarrow \mathsf{null}))\,\rrbracket_{\tau_1}\ \}$
```

Figure 6.9: Proof outline for the `getNext` implementation in τ_1 (failure case).

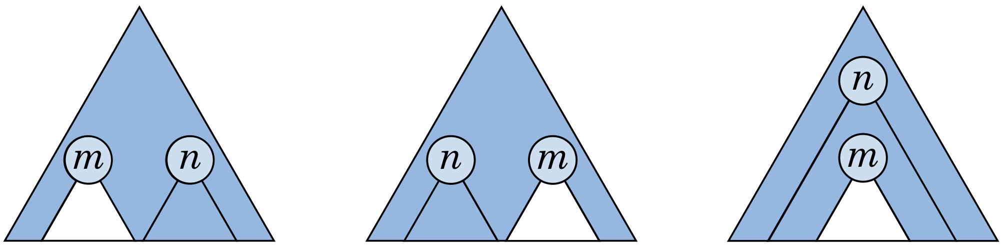

Figure 6.10: The 3 patterns of frame for `appendChild`.

the precondition of the axiom for the `appendChild` command. Either the subtree at `m` is somewhere to the left of `n`, the subtree at `m` is somewhere to the right of `n`, or the subtree at `m` is somewhere beneath `n`. The three states are illustrated in Figure 6.10.

The behaviour of an implementation of the `appendChild` command will be subtly different on each of these states. For example, the nodes may be visited in different orders or the pointer surgery required to maintain the tree may have to interact in different ways. To prove that an implementation of the `appendChild` command is correct with respect to some abstract specification we will have no choice but to check the implementation in each of these cases. This means we will need to provide three proof sketches for each axiom for the same piece of code.

Ideally, we want to be able to prove the correctness of every piece of code using just one proof sketch per axiom. We want to be able to reason about the correctness of such implementations without having to think about all of the possible frames they could be operating within. In effect, we want to mirror the locality of our high-level reasoning system in our low-level implementations. This desire leads us on to the technique of providing locality-preserving translations.

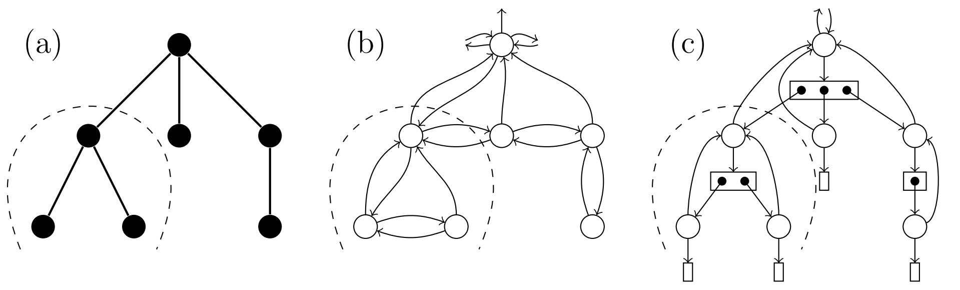

Figure 6.11: An abstract tree from $\mathbb{T}$ (a), and representations of the tree in $\mathbb{H}$ (b), and in $\mathbb{H} \times \mathbb{L}$ (c).

6.3 Locality-Preserving Translations

Sometimes, there is a close correspondence between the locality exhibited by a high-level module and the locality exhibited by its low-level implementation. In this section, we expand on this intuition and formalise the concept of a *locality-preserving translation*. We show that locality-preserving translations give rise to sound module translations and then consider several examples. In §6.3.2 we give a locality-preserving translation $\tau_2 : \mathbb{T} \to \mathbb{H}$ which uses the heap module $\mathbb{H}$ to implement the tree module $\mathbb{T}$. Similarly, in §6.3.3 we give another locality-preserving translation $\tau_3 : \mathbb{T} \to \mathbb{H} + \mathbb{L}$ which uses a combination of the heap module $\mathbb{H}$ and list module $\mathbb{L}$ to implement the tree module $\mathbb{T}$. Finally, in §6.3.4 we give a locality-preserving translation $\tau_4 : \mathbb{H} + \mathbb{H} \to \mathbb{H}$ which implements a double heap with a single heap. It is worth mentioning that whilst this last translation is sound, it is not, however, surjective.

First, we explain what it means for there to be a close correspondence between locality at the high-level and locality at the low-level. Figure 6.11 depicts a typical tree from the module $\mathbb{T}$ (a), together with possible representations of that tree in the heap module $\mathbb{H}$ (b), and in the combined heap-and-list module $\mathbb{H} \times \mathbb{L}$ (c). In Figure 6.11(b), each tree node is represented by a memory block of four pointer fields (depicted by a circle with outgoing arrows) which record the addresses of the memory blocks representing the left sibling, parent, first child and right sibling of the node. When there is no such node (for example, when a node has no children) the pointer field holds the null value (depicted by the absence of an arrow). In Figure 6.11(c), each tree node is represented by a memory block of two pointer fields which record the addresses of the parent node and the child list of the node. This child list (depicted by a box with dots for each value in the list) is a list of pointers to the node's children.

These examples exhibit simple inductive transformations from the abstract data structure to its concrete representations. It should be possible to lift these inductive transformations to the segment level, and thus give simple inductive transformations from high-level proofs to low-level proofs. In particular, it should be possible to transform high-level segments into a low-level segments. Such a transformation is said to preserve locality.

We wish to transform high-level proofs about an abstract program into corresponding low-level proofs about its implementation. We aim do this by simply replacing the high-level predicates with their low-level representations. However, we must take care in how we represent segments at the low-level. At the abstract level segments are agnostic to the segments that are placed beside them, around them, or within their holes, but the concrete representations of these segments need to know some information about these additional segments, and vice versa. In particular we need to know if the pointers contained within each segment's representation link into the representation of other segments. For example, consider the structures in Figure 6.11. Breaking apart the subtree denoted by the dashed line at the abstract-level (a) is simple. However, the same separation at the concrete-level (b and c) requires us to track the pointers that cross the breaking point (the arrows passing over the dashed line). An update to one of these concrete subtrees may have an effect on the values stored in these pointers.

We track such pointers by translating each abstract label $x \in \mathrm{X}_{\mathbb{A}}$ to an *interface* $I \in \mathcal{I}$ which records this 'knowledge' that segments have about each other. Specifically, the possible representations of an abstract segment s is given by the set of concrete segments $(\!|s|\!)^{\eta}$, where η is a function that maps each address and hole label in s to its interface. There are two types of interface corresponding to the two positions a label can occur in. An abstract address maps to an *address-interface* that contains the information needed to compress the addressed segment with another. An abstract hole label maps to a *hole-interface* that contains the information needed to fill the hole with the contents of another segment. Notice that these two concepts are closely related. In particular, the address-interface of one segment will be the same as a hole-interface of another segment when the abstract address and hole label are the same.

Most of the proof rules should transform simply to their low-level counterparts. However, the separation frame rule Sep Frame, revelation frame rule Rev Frame and axiom rule Axiom each require some care. Consider the operation of disposing the subtree indicated by the dashed line in Figure 6.11. At the abstract level, it is clear that the resource required to run the command is just the subtree that is to

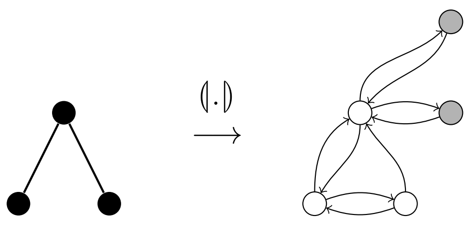

Figure 6.12: Crust inclusive translation.

be deleted. This is reflected in the axiom for `deleteTree`:

$$\left\{\ \alpha{\leftarrow}w[\mathrm{tree}(ct)] * \sigma \wedge \mathcal{E}[\![E]\!]\sigma = w\ \right\}\ \ \texttt{deleteTree}(E)\ \ \left\{\ \alpha{\leftarrow}\varnothing * \sigma\ \right\}$$

However, in both implementations something more than this representation of the subtree is required: for the heap implementation in Figure 6.11 (b), the pointers into the deleted subtree must be updated; for the heap-and-list implementation in Figure 6.11 (c), the pointers to the subtree's top-level nodes must be removed from their parent's child list. In both cases, the low-level footprint of `deleteTree` is larger than its intuitive concrete representation. The axiom for `deleteTree` cannot, therefore, be translated by just replacing the high-level predicates with their intuitive concrete representations.

This mismatch between the abstract footprint and the concrete footprint corresponds to a mismatch between the locality of the abstract and concrete modules. In order to be able to provide a locality preserving translation, we need to find some way to repair our translation from the abstract module. We do this by modifying the translation to include some *crust* which corresponds to the extra resource that is required to reason about the command implementations. This introduces a 'fiction of disjointness', as segments that were disjoint at the abstract-level may now overlap at the concrete-level. Figure 6.12 shows how we extend the heap representation of the subtree segment, indicated by the dashed line in Figure 6.11, to include this extra crust. The shaded nodes are the extra nodes that need to be included in the translation. However, we have to take care when introducing the crust to our translation. Considering the same example tree as before, Figure 6.13 shows us how the crusts of the heap representation of the first and last subtrees overlap at the concrete level. If we are not careful with this overlap then our translation may result in an undefined representation. We need to take great care to ensure that such state overlaps are correctly shared by our concrete segments. In general, we

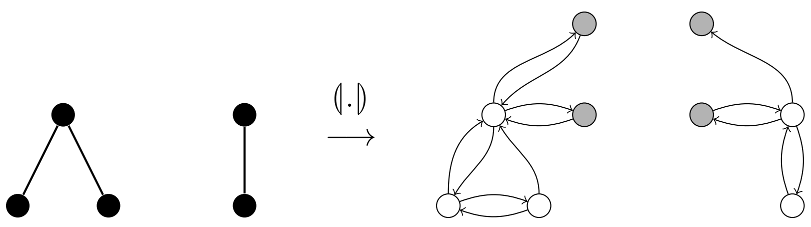

Figure 6.13: Translation with overlapping crusts.

will appeal to a permissions model to ensure that such shared data is maintained in a consistent view. Unfortunately, we will see that this model has to be defined on a case by case basis. For some translations a simple permissions model will suffice, but for others developing such a model can be rather involved.

Once we have modified our translation from the abstract data structure to the concrete data structure, we have to check that we can still translate abstract-level frames into concrete-level frames. It is essential that the concrete representation preserves certain properties of the abstract segment structure. In particular, if the abstract segment structure represents a disjoint splitting of the segment into two subsegments, its concrete representation should also be able to be split into the representations of these subsegments. This means that our translation must preserve the separating conjunction $*$. Loosely,

$$[\![P * Q]\!]_\tau \;\; \equiv \;\; [\![P]\!]_\tau * [\![Q]\!]_\tau.$$

Similarly, we must also ensure that when the abstract segment structure represents a compressed segment, its concrete representation should also be compressed. This means that our translation must also preserve the revelation operator Ⓡ. However, recall that we translate abstract labels to concrete interfaces, so a compression of one label at the abstract level may correspond to a compression of a set of labels (in the concrete interface) at the concrete level. The property we wish to show is loosely,

$$[\![\alpha \text{Ⓡ} P]\!]_\tau \;\; \equiv \;\; [\![\alpha]\!]_\tau \text{Ⓡ} [\![P]\!]_\tau.$$

With these two properties we can be sure that the separation frame rule and revelation frame rule are preserved across our translation. To deal with the Axiom rule we simply need to prove that the implementations of the basic commands satisfy the low-level representations of their specifications. The *axiom correctness* *property* must, therefore, ensure that the low-level specifications still hold under our translation. Loosely, we need to check that

$$\left\{ \ [\![P]\!]_\tau \ \right\} [\![\varphi]\!] \left\{ \ [\![Q]\!]_\tau \ \right\}$$

holds for every $(P, Q) \in \mathrm{Ax}[\![\varphi]\!]$. We shall see that the remaining inference rules can be more straightforwardly translated from the abstract-level to the concrete-level, despite the introduction of crusts and interfaces to our translation.

Having fleshed out the intuition behind locality-preserving translations, we now introduce their formal definition. We first define the concept of *pre-locality-preserving translations* and then restrict locality-preserving translations to being those that exhibit the required properties discussed above. We then prove a general result that locality-preserving translations are sound module translations.

Definition 6.9 (Pre-Locality-Preserving Translation). A *pre-locality preserving translation* $\tau : \mathbb{A} \to \mathbb{B}$ consists of:

- ◇ a set of interfaces $\mathcal{I}$ consisting of concrete-level identifiers, addresses and labels.
- ◇ an interface function $\eta : \mathrm{X}_\mathbb{A} \rightharpoonup_{\mathsf{fin}} \mathcal{I}$ which maps abstract identifiers to their concrete interfaces.
- ◇ a segment representation function $(\!|(\cdot)|\!)^{(\cdot)} : \mathrm{S}_\mathbb{A} \times (\mathrm{X}_\mathbb{A} \rightharpoonup_{\mathsf{fin}} \mathcal{I}) \to \mathcal{P}(\mathrm{S}_\mathbb{B})$;
- ◇ a substitutive implementation function $[\![(\cdot)]\!]_\tau : \mathcal{L}_\mathbb{A} \to \mathcal{L}_\mathbb{B}$.

We construct a module translation from a pre-locality-preserving translation by defining the abstraction relation in terms of the segment representation function. For any given η, the abstraction relation α_τ is defined to be

$$\alpha_\tau \quad \overset{\text{def}}{=} \quad \{(s_\mathbb{B}, s_\mathbb{A}) \mid s_\mathbb{B} \in (\!|s_\mathbb{A}|\!)^\eta\}.$$

There is often a natural choice of η, but in general any choice is permissible. We lift our abstraction relation to a predicate translation in the standard way. We also define a label predicate translation $[\![\alpha]\!]_\tau$ such that,

$$\mathcal{P}[\![\ [\![\alpha]\!]_\tau\ ]\!]e = \{y \mid e(\alpha) = x \text{ and } y \in \mathrm{X}_\mathbb{B} \text{ and } y \in \eta(x)\}.$$

This translation allows us to relate abstract label predicates with their corresponding sets of concrete label predicates.

A pre-locality preserving translation does not necessarily provide a sound module translation. However, if a pre-locality preserving translation satisfies the following three properties then it is a locality preserving translation which is a sound module translation. To simplify our presentation we work with the same operations on sets as introduced in §6.2.2.

Property 1 (Combination Preservation)**.** Segment combination is preserved by the segment representation function. That is, for all $s_1, s_2 \in S_{\mathbb{A}}$ and $\eta \in (\mathrm{X}_{\mathbb{A}} \rightharpoonup_{\text{fin}} \mathcal{I})$,

$$(\!| s_1 +_{\mathbb{A}} s_2 |\!)^{\eta} \quad \equiv \quad (\!| s_1 |\!)^{\eta} +_{\mathbb{B}} (\!| s_2 |\!)^{\eta}$$

Property 2 (Compression Preservation)**.** Segment compression is preserved by the segment representation function. That is, for all $x \in \mathrm{X}_{\mathbb{A}}$, $s \in S_{\mathbb{A}}$ and $\eta \in (\mathrm{X}_{\mathbb{A}} \rightharpoonup_{\text{fin}} \mathcal{I})$, there exists $I \in \mathcal{I}$ and $\bar{x} \in \mathcal{P}(\mathrm{X}_{\mathbb{B}})$ with $\bar{y} = \mathsf{labs}(I)$ such that,

$$(\!| (x)(s)_{\mathbb{A}} |\!)^{\eta} \quad \equiv \quad (\bar{y})((\!| s |\!)^{\eta[x \mapsto I]})_{\mathbb{B}}$$

where $\mathsf{labs}(I) = \{y \mid y \in I \text{ and } y \in \mathrm{X}_{\mathbb{B}}\}$ is the set of labels from $\mathrm{X}_{\mathbb{B}}$ that occur in interface $I \in \mathcal{I}$.

Property 3 (Axiom Correctness)**.** For all $e \in \textsc{Env}$, $\Gamma \in \text{PS}\textsc{Env}$, $\varphi \in \textsc{Cmd}_{\mathbb{A}}$, $(P, Q) \in \mathrm{Ax}[\![\varphi]\!]_{\mathbb{A}}$ and $\eta \in (\mathrm{X}_{\mathbb{A}} \rightharpoonup_{\text{fin}} \mathcal{I})$,

$$e, [\![\Gamma]\!]_{\tau} \vdash_{\mathbb{B}} \left\{ \; [\![P]\!]_{\tau} \; \right\} \; [\![\varphi]\!]_{\tau} \; \left\{ \; [\![Q]\!]_{\tau} \; \right\}$$

where $[\![P]\!]_{\tau}$ is the lifting of the abstraction relation α_{τ} to predicates, as defined in §6.1, and

$$[\![\Gamma]\!]_{\tau} \quad = \quad \{ \, f : [\![\mathsf{P}]\!]_{\tau} \rightarrowtail [\![\mathsf{Q}]\!]_{\tau} \mid (f : \mathsf{P} \rightarrowtail \mathsf{Q}) \in \Gamma \, \}$$

Definition 6.10 (Locality-Preserving Translation)**.** A *locality preserving translation* is a pre-locality-preserving translation that satisfies Properties 1, 2 and 3.

Theorem 6.11 (Locality-Preserving Translation Soundness)**.** A locality preserving translation is a sound module translation.

6.3.1 Soundness of Locality-Preserving Translations

Proposition 6.12 (Sound Transformation)**.** For all $e \in \textsc{Env}$, $\Gamma \in \text{PS}\textsc{Env}$, $P, Q \in \textsc{Pred}_{\mathbb{A}}$, $\mathbb{C} \in \mathcal{L}_{\mathbb{A}}$ and $\eta \in (\mathrm{X}_{\mathbb{A}} \rightharpoonup_{\text{fin}} \mathcal{I})$,

$$e, \Gamma \vdash_{\mathbb{A}} \left\{ \; P \; \right\} \; \mathbb{C} \; \left\{ \; Q \; \right\} \quad \Longrightarrow \quad e, [\![\Gamma]\!]_{\tau} \vdash_{\mathbb{B}} \left\{ \; [\![P]\!]_{\tau} \; \right\} \; [\![\mathbb{C}]\!]_{\tau} \; \left\{ \; [\![Q]\!]_{\tau} \; \right\}$$

Before we embark on the proof of Proposition 6.12 we first state, and prove, two auxiliary lemmas.

Lemma 6.13 (Separation Preservation)**.** The separating conjunction $*$ is preserved by the predicate representation function. That is, for all $P, Q \in \textsc{Pred}_{\mathbb{A}}$ and $\eta \in (\mathrm{X}_{\mathbb{A}} \rightharpoonup_{\text{fin}} \mathcal{I})$

$$[\![P * Q]\!]_\tau \;\Leftrightarrow\; [\![P]\!]_\tau * [\![Q]\!]_\tau$$

Proof. It is sufficient to show for all $e \in \textsc{Env}$ under the conditions above, that

$$\mathcal{P}[\![\, [\![P * Q]\!]_\tau \,]\!]e \;\equiv\; \mathcal{P}[\![\, [\![P]\!]_\tau * [\![Q]\!]_\tau \,]\!]e.$$

Fix arbitrary $P, Q \in \textsc{Pred}_{\mathbb{A}}$, $e \in \textsc{Env}$ and $\eta \in (\mathrm{X}_{\mathbb{A}} \rightharpoonup_{\text{fin}} \mathcal{I})$.

$$\begin{array}{rcl}
\mathcal{P}[\![\, [\![P * Q]\!]_\tau \,]\!]e & \equiv & \{(s_{\mathbb{B}}, \sigma) \mid (s_{\mathbb{A}}, \sigma) \in \mathcal{P}[\![P * Q]\!]e \text{ and } s_{\mathbb{B}} \alpha_\tau s_{\mathbb{A}}\} \\
& \equiv & \{(s_{\mathbb{B}}, \sigma) \mid (s_{\mathbb{A}}, \sigma) \in \mathcal{P}[\![P * Q]\!]e \text{ and } s_{\mathbb{B}} \in (\!| s_{\mathbb{A}} |\!)^\eta\} \\
& \equiv & \left\{ (s_{\mathbb{B}}, \sigma_1 * \sigma_2) \;\middle|\; \begin{array}{l} (s'_{\mathbb{A}}, \sigma_1) \in \mathcal{P}[\![P]\!]e \text{ and } (s''_{\mathbb{A}}, \sigma_2) \in \mathcal{P}[\![Q]\!]e \\ \text{and } s_{\mathbb{B}} \in (\!| s'_{\mathbb{A}} + s''_{\mathbb{A}} |\!)^\eta \end{array} \right\} \\
\text{(Property 1)} & \equiv & \left\{ (s_{\mathbb{B}}, \sigma_1 * \sigma_2) \;\middle|\; \begin{array}{l} (s'_{\mathbb{A}}, \sigma_1) \in \mathcal{P}[\![P]\!]e \text{ and } (s''_{\mathbb{A}}, \sigma_2) \in \mathcal{P}[\![Q]\!]e \\ \text{and } s_{\mathbb{B}} \in (\!| s'_{\mathbb{A}} |\!)^\eta + (\!| s''_{\mathbb{A}} |\!)^\eta \end{array} \right\} \\
& \equiv & \left\{ (s'_{\mathbb{B}} + s''_{\mathbb{B}}, \sigma_1 * \sigma_2) \;\middle|\; \begin{array}{l} (s'_{\mathbb{A}}, \sigma_1) \in \mathcal{P}[\![P]\!]e \text{ and } (s''_{\mathbb{A}}, \sigma_2) \in \mathcal{P}[\![Q]\!]e \\ \text{and } s'_{\mathbb{B}} \in (\!| s'_{\mathbb{A}} |\!)^\eta \text{ and } s''_{\mathbb{B}} \in (\!| s''_{\mathbb{A}} |\!)^\eta \end{array} \right\} \\
& \equiv & \left\{ (s'_{\mathbb{B}} + s''_{\mathbb{B}}, \sigma_1 * \sigma_2) \;\middle|\; \begin{array}{l} (s'_{\mathbb{A}}, \sigma_1) \in \mathcal{P}[\![P]\!]e \text{ and } (s''_{\mathbb{A}}, \sigma_2) \in \mathcal{P}[\![Q]\!]e \\ \text{and } s'_{\mathbb{B}} \alpha_\tau s'_{\mathbb{A}} \text{ and } s''_{\mathbb{B}} \alpha_\tau s''_{\mathbb{A}} \end{array} \right\} \\
& \equiv & \mathcal{P}[\![\, [\![P]\!]_\tau * [\![Q]\!]_\tau \,]\!]e
\end{array}$$

□

Lemma 6.14 (Revelation Preservation)**.** The revelation operator Ⓡ is preserved by the predicate representation function. That is, for all $\alpha \in (\textsc{Env} \to \mathrm{X})$, $P \in \textsc{Pred}_{\mathbb{A}}$ and $\eta \in (\mathrm{X}_{\mathbb{A}} \rightharpoonup_{\text{fin}} \mathcal{I})$,

$$[\![\alpha Ⓡ P]\!]_\tau \;\Leftrightarrow\; [\![\alpha]\!]_\tau Ⓡ [\![P]\!]_\tau$$

Proof. It is sufficient to show for all $e \in \textsc{Env}$ under the conditions above, that

$$\mathcal{P}[\![\, [\![\alpha Ⓡ P]\!]_\tau \,]\!]e \;\equiv\; \mathcal{P}[\![\, [\![\alpha]\!]_\tau Ⓡ [\![P]\!]_\tau \,]\!]e$$

Fix arbitrary $P \in \textsc{Pred}_{\mathbb{A}}$, $e \in \textsc{Env}$, $\eta \in (\mathrm{X} \rightharpoonup_{\text{fin}} \mathcal{I})$. Assume that $e(\alpha) = x$ for some $x \in \mathrm{X}_{\mathbb{A}}$ and there exists $I \in \mathcal{I}$ and $\bar{x} \in \mathcal{P}(\mathrm{X}_{\mathbb{B}})$ with $\mathsf{labs}(I) = \bar{x}$.

$$
\begin{array}{rcl}
\mathcal{P}\llbracket\, \llbracket \alpha Ⓡ P \rrbracket_{\tau} \,\rrbracket e & \equiv & \{(s_{\mathbb{B}}, \sigma) \mid (s_{\mathbb{A}}, \sigma) \in \mathcal{P}\llbracket \alpha Ⓡ P \rrbracket e \text{ and } s_{\mathbb{B}} \alpha_{\tau} s_{\mathbb{A}}\} \\
& \equiv & \{(s_{\mathbb{B}}, \sigma) \mid (s_{\mathbb{A}}, \sigma) \in \mathcal{P}\llbracket \alpha Ⓡ P \rrbracket e \text{ and } s_{\mathbb{B}} \in (\!| s_{\mathbb{A}} |\!)^{\eta}\} \\
& \equiv & \{(s_{\mathbb{B}}, \sigma) \mid e(\alpha) = x \text{ and } (s'_{\mathbb{A}}, \sigma) \in \mathcal{P}\llbracket P \rrbracket e \text{ and } s_{\mathbb{B}} \in (\!| (x)(s'_{\mathbb{A}}) |\!)^{\eta}\} \\
\text{(Property 2)} & \equiv & \left\{ (s_{\mathbb{B}}, \sigma) \;\middle|\; \begin{array}{l} e(\alpha) = x \text{ and } (s'_{\mathbb{A}}, \sigma) \in \mathcal{P}\llbracket P \rrbracket e \\ \text{and } s_{\mathbb{B}} \in (\bar{x})((\!| s'_{\mathbb{A}} |\!)^{\eta[x \mapsto I]}) \end{array} \right\} \\
& \equiv & \left\{ ((\bar{x})(s'_{\mathbb{B}}), \sigma) \;\middle|\; \begin{array}{l} e(\alpha) = x \text{ and } (s'_{\mathbb{A}}, \sigma) \in \mathcal{P}\llbracket P \rrbracket e \\ \text{and } s'_{\mathbb{B}} \in (\!| s'_{\mathbb{A}} |\!)^{\eta[x \mapsto I]} \end{array} \right\} \\
& \equiv & \left\{ ((\bar{x})(s'_{\mathbb{B}}), \sigma) \;\middle|\; \begin{array}{l} \llbracket \alpha \rrbracket_{\tau} e = \bar{x} \text{ and } (s'_{\mathbb{A}}, \sigma) \in \mathcal{P}\llbracket P \rrbracket e \\ \text{and } s'_{\mathbb{B}} \alpha_{\tau} s'_{\mathbb{A}} \end{array} \right\} \\
& \equiv & \mathcal{P}\llbracket\, \llbracket \alpha \rrbracket_{\tau} Ⓡ \llbracket P \rrbracket_{\tau} \,\rrbracket e
\end{array}
$$

□

Locality-Preserving Translation Soundness

The proof of Proposition 6.12 inductively transforms a proof in module $\mathbb{A}$ into a proof in module $\mathbb{B}$.

Proof. The proof is by induction on the structure of the proof of $e, \Gamma \vdash_{\mathbb{A}} \{P\}\, \mathbb{C}\, \{Q\}$, in each case we consider the last rule applied in the proof. We assume, as the inductive hypothesis, that the translated premises of each rule have proofs in $\mathbb{B}$. We show how to derive a proof of the translated conclusions from these translated premises. We omit the logical environment and procedure specification environment from our proofs when they plays no part in the derivation. Note that since our translations do not affect the variable store, the $\mathsf{vsafe}(E)$, $\mathsf{bsafe}(B)$ and $\mathcal{P}\llbracket B \rrbracket$ predicates are also not affected by the translations. We make use of this in several of the proof cases.

Axiom case:
This follows immediately from the Axiom Correctness Property (Property 3).

Sep Frame case:

$$
\dfrac{\dfrac{\left\{\ \llbracket P \rrbracket_{\tau}\ \right\}\ \llbracket \mathbb{C} \rrbracket_{\tau}\ \left\{\ \llbracket Q \rrbracket_{\tau}\ \right\}}{\left\{\ \llbracket P \rrbracket_{\tau} * \llbracket R \rrbracket_{\tau}\ \right\}\ \llbracket \mathbb{C} \rrbracket_{\tau}\ \left\{\ \llbracket Q \rrbracket_{\tau} * \llbracket R \rrbracket_{\tau}\ \right\}}\ \textsc{Sep Frame}}{\left\{\ \llbracket P * R \rrbracket_{\tau}\ \right\}\ \llbracket \mathbb{C} \rrbracket_{\tau}\ \left\{\ \llbracket Q * R \rrbracket_{\tau}\ \right\}}\ \text{Lemma 6.13}
$$

Rev Frame case:

$$\dfrac{\dfrac{\left\{\ [\![P]\!]_\tau\ \right\}\ [\![\mathbb{C}]\!]_\tau\ \left\{\ [\![Q]\!]_\tau\ \right\}}{\left\{\ [\![\alpha]\!]_\tau \circledR [\![P]\!]_\tau\ \right\}\ [\![\mathbb{C}]\!]_\tau\ \left\{\ [\![\alpha]\!]_\tau \circledR [\![Q]\!]_\tau\ \right\}}\ \textsc{Rev Frame}^*}{\left\{\ [\![\alpha \circledR P]\!]_\tau\ \right\}\ [\![\mathbb{C}]\!]_\tau\ \left\{\ [\![\alpha \circledR Q]\!]_\tau\ \right\}}\ \text{Lemma 6.14}$$

Note that the revelation frame rule (Rev Frame) may need to be used zero, one or many times depending on the evaluation of $[\![\alpha]\!]_\tau$.

Cons case:

$$\dfrac{\dfrac{\mathcal{P}[\![P]\!]e \subseteq \mathcal{P}[\![P']\!]e}{\mathcal{P}[\![[\![P]\!]_\tau]\!]e \subseteq \mathcal{P}[\![[\![P']\!]_\tau]\!]e}\quad \left\{\ [\![P']\!]_\tau\ \right\}\ [\![\mathbb{C}]\!]_\tau\ \left\{\ [\![Q']\!]_\tau\ \right\}\quad \dfrac{\mathcal{P}[\![Q']\!]e \subseteq \mathcal{P}[\![Q]\!]e}{\mathcal{P}[\![[\![Q']\!]_\tau]\!]e \subseteq \mathcal{P}[\![[\![Q]\!]_\tau]\!]e}}{\left\{\ [\![P]\!]_\tau\ \right\}\ [\![\mathbb{C}]\!]_\tau\ \left\{\ [\![Q]\!]_\tau\ \right\}}\ \textsc{Cons}$$

Disj case:

$$\dfrac{\dfrac{\text{for all } i \in I.\left\{\ [\![P_i]\!]_\tau\ \right\}\ [\![\mathbb{C}]\!]_\tau\ \left\{\ [\![Q_i]\!]_\tau\ \right\}}{\left\{\ \bigvee_{i\in I}[\![P_i]\!]_\tau\ \right\}\ [\![\mathbb{C}]\!]_\tau\ \left\{\ \bigvee_{i\in I}[\![Q_i]\!]_\tau\ \right\}}\ \textsc{Disj}}{\left\{\ [\![\bigvee_{i\in I} P_i]\!]_\tau\ \right\}\ [\![\mathbb{C}]\!]_\tau\ \left\{\ [\![\bigvee_{i\in I} Q_i]\!]_\tau\ \right\}}$$

Exsts case:

$$\dfrac{\dfrac{\left\{\ [\![P]\!]_\tau\ \right\}\ [\![\mathbb{C}]\!]_\tau\ \left\{\ [\![Q]\!]_\tau\ \right\}}{\left\{\ \exists v.\,[\![P]\!]_\tau\ \right\}\ [\![\mathbb{C}]\!]_\tau\ \left\{\ \exists v.\,[\![Q]\!]_\tau\ \right\}}\ \textsc{Exsts}}{\left\{\ [\![\exists v.\,P]\!]_\tau\ \right\}\ [\![\mathbb{C}]\!]_\tau\ \left\{\ [\![\exists v.\,Q]\!]_\tau\ \right\}}$$

Fresh case:

$$\dfrac{\dfrac{\left\{\ [\![P]\!]_\tau\ \right\}\ [\![\mathbb{C}]\!]_\tau\ \left\{\ [\![Q]\!]_\tau\ \right\}}{\left\{\ \text{И}[\![\alpha]\!]_\tau.\,[\![P]\!]_\tau\ \right\}\ [\![\mathbb{C}]\!]_\tau\ \left\{\ \text{И}[\![\alpha]\!]_\tau.\,[\![Q]\!]_\tau\ \right\}}\ \textsc{Fresh}^*}{\left\{\ [\![\text{И}\alpha.\,P]\!]_\tau\ \right\}\ [\![\mathbb{C}]\!]_\tau\ \left\{\ [\![\text{И}\alpha.\,Q]\!]_\tau\ \right\}}$$

Note that the freshness quantification rule (Fresh) may need to be used zero, one or many times depending on the evaluation of $[\![\alpha]\!]_\tau$.

Skip case:

$$\begin{array}{cl}
\hline \left\{\ \mathsf{emp}_{\mathbb{B}}\ \right\}\ \mathtt{skip}\ \left\{\ \mathsf{emp}_{\mathbb{B}}\ \right\} & \textsc{Skip} \\
\hline \left\{\ \mathsf{emp}_{\mathbb{B}} * [\![\mathsf{emp}_{\mathbb{A}}]\!]_\tau\ \right\}\ \mathtt{skip}\ \left\{\ \mathsf{emp}_{\mathbb{B}} * [\![\mathsf{emp}_{\mathbb{A}}]\!]_\tau\ \right\} & \textsc{Sep Frame} \\
\hline \left\{\ [\![\mathsf{emp}_{\mathbb{A}}]\!]_\tau\ \right\}\ \mathtt{skip}\ \left\{\ [\![\mathsf{emp}_{\mathbb{A}}]\!]_\tau\ \right\} & \textsc{Cons} \\
\hline\hline \left\{\ [\![\mathsf{emp}_{\mathbb{A}}]\!]_\tau\ \right\}\ [\![\mathtt{skip}]\!]_\tau\ \left\{\ [\![\mathsf{emp}_{\mathbb{A}}]\!]_\tau\ \right\} &
\end{array}$$

Seq case:

$$\begin{array}{cl}
\left\{\ [\![P]\!]_\tau\ \right\}\ [\![\mathbb{C}_1]\!]_\tau\ \left\{\ [\![Q]\!]_\tau\ \right\} \quad \left\{\ [\![Q]\!]_\tau\ \right\}\ [\![\mathbb{C}_2]\!]_\tau\ \left\{\ [\![R]\!]_\tau\ \right\} & \\
\hline \left\{\ [\![P]\!]_\tau\ \right\}\ [\![\mathbb{C}_1]\!]_\tau\ ;\ [\![\mathbb{C}_2]\!]_\tau\ \left\{\ [\![R]\!]_\tau\ \right\} & \textsc{Seq} \\
\hline\hline \left\{\ [\![P]\!]_\tau\ \right\}\ [\![\mathbb{C}_1\ ;\ \mathbb{C}_2]\!]_\tau\ \left\{\ [\![R]\!]_\tau\ \right\} &
\end{array}$$

If case:

$$\begin{array}{cl}
\dfrac{\mathcal{P}[\![P]\!]e \subseteq \mathsf{bsafe}(B)}{\mathcal{P}[\![[\![P]\!]_\tau]\!]e \subseteq \mathsf{bsafe}(B)} \quad
\begin{array}{c}
\left\{\ [\![P \wedge \mathcal{P}[\![B]\!]]\!]_\tau\ \right\}\ [\![\mathbb{C}_1]\!]_\tau\ \left\{\ [\![Q]\!]_\tau\ \right\} \\
\left\{\ [\![P \wedge \neg\mathcal{P}[\![B]\!]]\!]_\tau\ \right\}\ [\![\mathbb{C}_2]\!]_\tau\ \left\{\ [\![Q]\!]_\tau\ \right\} \\
\hline\hline
\left\{\ [\![P]\!]_\tau \wedge \mathcal{P}[\![B]\!]\ \right\}\ [\![\mathbb{C}_1]\!]_\tau\ \left\{\ [\![Q]\!]_\tau\ \right\} \\
\left\{\ [\![P]\!]_\tau \wedge \neg\mathcal{P}[\![B]\!]\ \right\}\ [\![\mathbb{C}_2]\!]_\tau\ \left\{\ [\![Q]\!]\tau\ \right\}
\end{array} & \\
\hline \left\{\ [\![P]\!]_\tau\ \right\}\ \mathtt{if}\ B\ \mathtt{then}\ [\![\mathbb{C}_1]\!]_\tau\ \mathtt{else}\ [\![\mathbb{C}_2]\!]_\tau\ \left\{\ [\![Q]\!]_\tau\ \right\} & \textsc{If} \\
\hline\hline \left\{\ [\![P]\!]_\tau\ \right\}\ [\![\mathtt{if}\ B\ \mathtt{then}\ \mathbb{C}_1\ \mathtt{else}\ \mathbb{C}_2]\!]_\tau\ \left\{\ [\![Q]\!]_\tau\ \right\} &
\end{array}$$

While case:

$$\begin{array}{cl}
\dfrac{\mathcal{P}[\![P]\!]e \subseteq \mathsf{bsafe}(B)}{\mathcal{P}[\![[\![P]\!]_\tau]\!]e \subseteq \mathsf{bsafe}(B)} \quad
\begin{array}{c}
\left\{\ [\![P \wedge \mathcal{P}[\![B]\!]]\!]_\tau\ \right\}\ [\![\mathbb{C}]\!]_\tau\ \left\{\ [\![P]\!]_\tau\ \right\} \\
\hline\hline
\left\{\ [\![P]\!]_\tau \wedge \mathcal{P}[\![B]\!]\ \right\}\ [\![\mathbb{C}]\!]_\tau\ \left\{\ [\![P]\!]_\tau\ \right\}
\end{array} & \\
\hline \left\{\ [\![P]\!]_\tau\ \right\}\ \mathtt{while}\ B\ \mathtt{do}\ [\![\mathbb{C}]\!]_\tau\ \left\{\ [\![P]\!]_\tau \wedge \neg\mathcal{P}[\![B]\!]\ \right\} & \textsc{While} \\
\hline\hline \left\{\ [\![P]\!]_\tau\ \right\}\ [\![\mathtt{while}\ B\ \mathtt{do}\ \mathbb{C}]\!]_\tau\ \left\{\ [\![P \wedge \neg\mathcal{P}[\![B]\!]]\!]_\tau\ \right\} &
\end{array}$$

Assgn case:

$$\begin{array}{c}
\mathcal{P}[\![\mathtt{x} \Rrightarrow v * \sigma]\!]e \subseteq \mathsf{vsafe}(E) \\ \hline
\left\{\ \mathtt{x} \Rrightarrow v * \sigma\ \right\}\ \mathtt{x} := E\ \left\{\ \mathtt{x} \Rrightarrow \mathcal{E}[\![E]\!]\sigma[\mathtt{x} \mapsto v] * \sigma\ \right\} \quad \textsc{Assgn} \\ \hline
\left\{\ [\![\mathsf{emp}_{\mathbb{A}}]\!]_{\tau} * \mathtt{x} \Rrightarrow v * \sigma\ \right\}\ \mathtt{x} := E\ \left\{\ [\![\mathsf{emp}_{\mathbb{A}}]\!]_{\tau} * \mathtt{x} \Rrightarrow \mathcal{E}[\![E]\!]\sigma[\mathtt{x} \mapsto v] * \sigma\ \right\} \quad \textsc{Sep Frame} \\ \hline\hline
\left\{\ [\![\mathsf{emp}_{\mathbb{A}} * \mathtt{x} \Rrightarrow v * \sigma]\!]_{\tau}\ \right\}\ [\![\mathtt{x} := E]\!]_{\tau}\ \left\{\ [\![\mathsf{emp}_{\mathbb{A}} * \mathtt{x} \Rrightarrow \mathcal{E}[\![E]\!]\sigma[\mathtt{x} \mapsto v] * \sigma]\!]_{\tau}\ \right\} \\ \hline\hline
\left\{\ [\![\mathtt{x} \Rrightarrow v * \sigma]\!]_{\tau}\ \right\}\ [\![\mathtt{x} := E]\!]_{\tau}\ \left\{\ [\![\mathtt{x} \Rrightarrow \mathcal{E}[\![E]\!]\sigma[\mathtt{x} \mapsto v] * \sigma]\!]_{\tau}\ \right\}
\end{array}$$

Local case:

$$\dfrac{\dfrac{\mathcal{P}[\![P]\!]e \cap \mathsf{vsafe}(\mathtt{x}) \equiv \emptyset}{\mathcal{P}[\![[\![P]\!]_{\tau}]\!]e \cap \mathsf{vsafe}(\mathtt{x}) \equiv \emptyset} \qquad \begin{array}{c} \left\{\ [\![\mathtt{x} \Rrightarrow - * P]\!]_{\tau}\ \right\}\ [\![\mathbb{C}]\!]_{\tau}\ \left\{\ [\![\mathtt{x} \Rrightarrow - * Q]\!]_{\tau}\ \right\} \\ \hline\hline \left\{\ \mathtt{x} \Rrightarrow - * [\![P]\!]_{\tau}\ \right\}\ [\![\mathbb{C}]\!]_{\tau}\ \left\{\ \mathtt{x} \Rrightarrow - * [\![Q]\!]_{\tau}\ \right\} \end{array}}{\begin{array}{c} \left\{\ [\![P]\!]_{\tau}\ \right\}\ \mathtt{local\ x\ in}\ [\![\mathbb{C}]\!]_{\tau}\ \left\{\ [\![Q]\!]_{\tau}\ \right\} \\ \hline\hline \left\{\ [\![P]\!]_{\tau}\ \right\}\ [\![\mathtt{local\ x\ in}\ \mathbb{C}]\!]_{\tau}\ \left\{\ [\![Q]\!]_{\tau}\ \right\} \end{array}} \quad \textsc{Local}$$

PDef case:

$$\begin{array}{c}
\text{for all } (\mathtt{f}_i : \mathsf{P}_i \rightarrowtail \mathsf{Q}_i) \in \Gamma.\ e, [\![\Gamma', \Gamma]\!]_{\tau} \vdash_{\mathbb{B}} \begin{array}{c} \left\{\ [\![\exists \overrightarrow{v}.\, \mathsf{P}_i(\overrightarrow{v}) * \overrightarrow{\mathtt{x}_i} \Rrightarrow \overrightarrow{v} * \overrightarrow{\mathtt{r}_i} \Rrightarrow -]\!]_{\tau}\ \right\} \\ [\![\mathbb{C}_i]\!]_{\tau} \\ \left\{\ [\![\exists \overrightarrow{w}.\, \mathsf{Q}_i(\overrightarrow{w}) * \overrightarrow{\mathtt{x}_i} \Rrightarrow - * \overrightarrow{\mathtt{r}_i} \Rrightarrow \overrightarrow{w}]\!]_{\tau}\ \right\} \end{array} \\ \hline\hline
\text{for all } (\mathtt{f}_i : \mathsf{P}_i \rightarrowtail \mathsf{Q}_i) \in [\![\Gamma]\!]_{\tau}.\ e, [\![\Gamma', \Gamma]\!]_{\tau} \vdash_{\mathbb{B}} \begin{array}{c} \left\{\ \exists \overrightarrow{v}.\, \mathsf{P}_i(\overrightarrow{v}) * \overrightarrow{\mathtt{x}_i} \Rrightarrow \overrightarrow{v} * \overrightarrow{\mathtt{r}_i} \Rrightarrow -\ \right\} \\ [\![\mathbb{C}_i]\!]_{\tau} \\ \left\{\ \exists \overrightarrow{w}.\, \mathsf{Q}_i(\overrightarrow{w}) * \overrightarrow{\mathtt{x}_i} \Rrightarrow - * \overrightarrow{\mathtt{r}_i} \Rrightarrow \overrightarrow{w}\ \right\} \end{array} \\ \hline
(\star)
\end{array}$$

$$\dfrac{(\star) \quad e, [\![\Gamma', \Gamma]\!]_{\tau} \vdash_{\mathbb{B}} \left\{\ [\![P]\!]_{\tau}\ \right\}\ [\![\mathbb{C}]\!]_{\tau}\ \left\{\ [\![Q]\!]_{\tau}\ \right\}}{\begin{array}{c} \left\{\ [\![P]\!]_{\tau}\ \right\} \\ e, [\![\Gamma']\!]_{\tau} \vdash_{\mathbb{B}}\ [\![\mathtt{procs}\ \overrightarrow{\mathtt{r}_1} := \mathtt{f}_1(\overrightarrow{\mathtt{x}_1})\{\mathbb{C}_1\}, \ldots, \overrightarrow{\mathtt{r}_k} := \mathtt{f}_k(\overrightarrow{\mathtt{x}_k})\{\mathbb{C}_k\}\ \mathtt{in}\ \mathbb{C}]\!]_{\tau} \\ \left\{\ [\![Q]\!]_{\tau}\ \right\} \end{array}} \quad \textsc{PDef}$$

There are two more premises of the PDef rule, which we have not covered in the above derivation. These are easily dispatched since $[\![(\cdot)]\!]_{\tau}$ preserves the procedure names in a procedure specification environment.

PCall case:

$$\dfrac{\dfrac{\mathcal{P}[\![\overrightarrow{\mathtt{r}} \Rightarrow \overrightarrow{v} * \sigma]\!]e \subseteq \mathsf{vsafe}(\overrightarrow{E})}{[\![\Gamma, (\mathtt{f} : \mathsf{P} \rightarrowtail \mathsf{Q})]\!]_\tau \vdash_{\mathbb{B}} \begin{array}{c}\left\{\ [\![\mathsf{P}\,(\mathcal{E}[\![E]\!]\sigma[\overrightarrow{\mathtt{r}} \mapsto \overrightarrow{v}])]\!]_\tau * \overrightarrow{\mathtt{r}} \Rightarrow \overrightarrow{v} * \sigma\ \right\}\\ [\![\mathtt{call}\ \overrightarrow{\mathtt{r}} := \mathtt{f}(\overrightarrow{E})]\!]_\tau \\ \left\{\ \exists \overrightarrow{w}.\, [\![\mathsf{Q}(\overrightarrow{w})]\!]_\tau * \overrightarrow{\mathtt{r}} \Rightarrow \overrightarrow{w} * \sigma\ \right\}\end{array}}\ \textsc{PCall}}{[\![\Gamma, (\mathtt{f} : \mathsf{P} \rightarrowtail \mathsf{Q})]\!]_\tau \vdash_{\mathbb{B}} \begin{array}{c}\left\{\ [\![\mathsf{P}\,(\mathcal{E}[\![E]\!]\sigma[\overrightarrow{\mathtt{r}} \mapsto \overrightarrow{v}]) * \overrightarrow{\mathtt{r}} \Rightarrow \overrightarrow{v} * \sigma]\!]_\tau\ \right\}\\ [\![\mathtt{call}\ \overrightarrow{\mathtt{r}} := \mathtt{f}(\overrightarrow{E})]\!]_\tau \\ \left\{\ [\![\exists \overrightarrow{w}.\, \mathsf{Q}(\overrightarrow{w}) * \overrightarrow{\mathtt{r}} \Rightarrow \overrightarrow{w} * \sigma]\!]_\tau\ \right\}\end{array}}$$

PWeak case:

$$\dfrac{\dfrac{[\![\Gamma]\!]_\tau \vdash_{\mathbb{B}} \left\{\ [\![P]\!]_\tau\ \right\}\ [\![\mathbb{C}]\!]_\tau\ \left\{\ [\![Q]\!]_\tau\ \right\}}{[\![\Gamma]\!]_\tau, [\![\Gamma']\!]_\tau \vdash_{\mathbb{B}} \left\{\ [\![P]\!]_\tau\ \right\}\ [\![\mathbb{C}]\!]_\tau\ \left\{\ [\![Q]\!]_\tau\ \right\}}\ \textsc{PWeak}}{[\![\Gamma, \Gamma']\!]_\tau \vdash_{\mathbb{B}} \left\{\ [\![P]\!]_\tau\ \right\}\ [\![\mathbb{C}]\!]_\tau\ \left\{\ [\![Q]\!]_\tau\ \right\}}$$

□

This completes the proof of Theorem 6.11.

Including the Conjunction Rule

If we wish to add the conjunction rule to the locality-preserving theory, we can add a case to the proof of Proposition 6.12. The conjunction rule can be dealt with in the same fashion as the disjunction rule, provided that $(\!|(\cdot)|\!)^{(\cdot)}$ distributes over conjunction. Together, the following two conditions are sufficient to establish this:

- ◇ for all $s, s' \in \mathrm{S}_{\mathbb{A}}$ with $s \neq s'$, and all $\eta \in (\mathrm{X}_{\mathbb{A}} \rightharpoonup_{\mathsf{fin}} \mathcal{I})$, $(\!|s|\!)^\eta \cap (\!|s'|\!)^\eta \equiv \emptyset$; and
- ◇ for all $\eta \in (\mathrm{X}_{\mathbb{A}} \rightharpoonup_{\mathsf{fin}} \mathcal{I})$ and $P \in \textsc{Pred}_{\mathbb{A}}$ the predicate $[\![P]\!]_\tau$ is precise.

It is not a coincidence that these conditions are similar to those that prevent a command from behaving angelically given in chapter 4. In both cases the conditions are constraining the predicate transformers corresponding to the abstraction relation or command to being *conjunctive*.

6.3.2 Module Translation $\tau_2 : \mathbb{T} \to \mathbb{H}$

We now present a locality-preserving translation τ_2 from the tree module $\mathbb{T}$ into the heap module $\mathbb{H}$. This translation represents each tree node n as a block of four cells

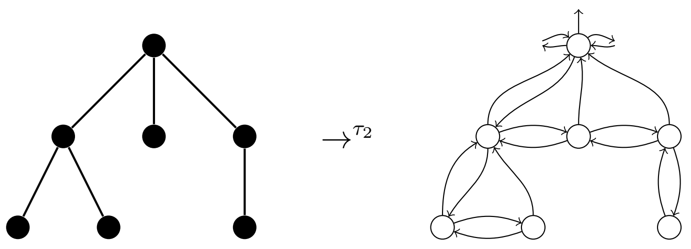

Figure 6.14: An abstract tree from $\mathbb{T}$ and its representation in $\mathbb{H}$.

in the heap, $n \mapsto l,u,d,r$, which contain pointers to the node's left sibling (l), parent (u), first child (d) and right sibling (r). This representation of the tree is illustrated in Figure 6.14.

An interface consists of the addresses of the tree's parent node and of the nodes immediately adjacent to the tree on the left and right, as well as the addresses of the left- and right-most nodes at the root level of the tree. These interfaces are represented in Figure 6.14 by the arrows into and out of the tree's root node.

Note that for the empty tree $\varnothing$, the addresses of the left- and right-most nodes at the root of the tree are not simply null, but rather the "address of the left-most node" should actually be the address of the node immediately adjacent to the tree on the right and the "address of the right-most node" should be the address of the node adjacent on the left. If we instead used null pointers, then $\varnothing$ would break up the continuous list of nodes. On the other hand, if the parent node, left sibling or right sibling do not exist, their addresses will be null.

In this translation from abstract tree segments into concrete heap segments, the abstract addresses and hole labels are mapped into some concrete state. It is possible for the interface function η to map multiple abstract labels to the same interface. This is particularly evident in the case of the segment, $x{\leftarrow}y$, which requires $\eta(x) = \eta(y)$. In such cases we need to ensure that the concrete state corresponding to the shared interface can be split and shared between the concrete heaps segments.

We manage such sharing by introducing the concept of partial heap cells $\check{x} \mapsto \check{v}$, $x \mapsto \check{v}$ and $\check{x} \mapsto v$ and using invariants to describe shared portions of state. Partial heap cells are used to weaken our knowledge about some piece of state and also to control a program's behaviour on that state. When we see a partial heap address $\check{x}$ or a partial value $\check{v}$ this is interpreted as only having partial knowledge about that cell or value. If we only have partial knowledge about a heap cell then we do not know if that cell is assigned or not. If we only have partial knowledge about a value then we do not know for sure what that value is.

We have to be careful when using values that were partial when they read. It is possible that such values may have been modified elsewhere in the program. We will see that under certain stability requirements such values may be safely used later in a program.

There are two ways in which we break up complete heap cells into partial pieces. The first of these breaks off a weak copy of the cell and the value it contains.

$$x \mapsto v \quad = \quad x \mapsto v \star (\check{x} \mapsto \check{v} \lor \check{x} = \mathsf{null})$$

Notice that the weak copy maintains none of the definite knowledge about the state of the cell or its value. The complete copy of the cell may be updated or even deleted elsewhere, so in the copy both x and v and annotated as partial $\check{-}$. If a program is to use the weak copy of the cell, it must take great care that it does not try to dereference cell x if it does not exist. Note that there is no limit to the number of times we can split off a weak copy of a heap cell.

The second way that we break up a complete heap cell is to split up the control over the cell and its contents.

$$x \mapsto v \quad = \quad x \mapsto \check{v} \star (\check{x} \mapsto v \lor \check{x} = \mathsf{null})$$

Here the first partial heap cell has enough state to allow the cell to be read, modified or deleted, but not to know about the actual value v. Similarly, the second partial heap cell has enough state to allow the cell to be read or its contents modified, but not enough to delete the cell. Again, when using the second partial heap cell a program must take care that it does not try to dereference the cell if it has been deleted. Note that we only ever allow one partial heap cell to keep the full knowledge about state of the cell or its value. That is,

$$\begin{aligned} x \mapsto \check{v} \star x \mapsto v &= \mathsf{false} \\ x \mapsto \check{v} \star x \mapsto \check{v} &= \mathsf{false} \\ x \mapsto v \star \check{x} \mapsto v &= \mathsf{false} \\ \check{x} \mapsto v \star \check{x} \mapsto v &= \mathsf{false} \end{aligned}$$

We shall see that we define the concrete interface (also called the crust) of our translation in terms of partial heap cells.

Notation: We write $x \mapsto l,u,d,r$ to mean $x \mapsto l \star x{+}1 \mapsto u \star x{+}2 \mapsto d \star x{+}3 \mapsto r$ and we write $(x \doteq y)$ to mean $\{\mathsf{emp}\} \land (x = y)$. We also drop module annotations when they can be inferred from context.

Definition 6.15 ($\tau_2 : \mathbb{T} \to \mathbb{H}$)**.** The pre-locality-preserving translation $\tau_2 : \mathbb{T} \to \mathbb{H}$ is constructed as follows:

- ◇ An interface $I = (i,j)(l,u,r) \in \mathcal{I}$ describes the address of the first node i and the last node j at the top level of the tree segment and the left node l, parent node u and right node r of the tree segment. Note that there are no addresses or hole labels in these interfaces, so $\mathsf{labs}(I) = \emptyset$ for all $I \in \mathcal{I}$.

- ◇ the segment representation function $(\!|(\cdot)|\!)^{(\cdot)} : \mathrm{S_T} \times (\mathrm{X} \rightharpoonup_{\mathrm{fin}} \mathcal{I}) \to \mathcal{P}(\mathrm{S_H})$ is defined by induction on the structure of tree segments as:

$$
\begin{aligned}
(\!|\emptyset|\!)^{\eta} &\overset{\text{def}}{=} \{\mathsf{emp}\} \\
(\!|\{(x,ct)\}|\!)^{\eta} &\overset{\text{def}}{=} \begin{cases} \exists i,j.\, \lceil \Cap^{(i,j)(\mathsf{null},\mathsf{null},\mathsf{null})} \star \langle\!\langle ct \rangle\!\rangle_{\eta}^{(i,j)(\mathsf{null},\mathsf{null},\mathsf{null})} \rceil & \text{if } x = 0 \\ \lceil \Cap^{(i,j)(\check{l},\check{u},\check{r})} \star \langle\!\langle ct \rangle\!\rangle_{\eta}^{(i,j)(\check{l},\check{u},\check{r})} \rceil \wedge \eta(x) = (i,j)(l,u,r) & \text{otherwise} \end{cases} \\
(\!|st_1 \uplus st_2|\!)^{\eta} &\overset{\text{def}}{=} (\!|st_1|\!)^{\eta} +_{\mathrm{H}} (\!|st_2|\!)^{\eta}
\end{aligned}
$$

where the upper crust predicate $\Cap^{(i,j)(\check{l},\check{u},\check{r})} \in \mathcal{P}(\mathrm{H}_{\mathrm{ADR,X}})$ is defined as:

$$
\begin{aligned}
\Cap^{(i,j)(\check{l},\check{u},\check{r})} \overset{\text{def}}{=}\; & (\{\check{l} \mapsto \check{-},\check{-},\check{-},i\} \vee (\check{l} = \mathsf{null} \wedge (\{\check{u} \mapsto \check{-},\check{-},i,\check{-}\} \vee \check{u} \doteq \mathsf{null}))) \\
& \star (\{\check{r} \mapsto j,\check{-},\check{-},\check{-}\} \vee \check{r} \doteq \mathsf{null})
\end{aligned}
$$

the context representation function $\langle\!\langle(\cdot)\rangle\!\rangle_{(\cdot)}^{(\cdot)} : \mathcal{C}_{\mathrm{T}} \times \mathcal{I} \times (\mathrm{X} \rightharpoonup_{\mathrm{fin}} \mathcal{I}) \to \mathcal{P}(\mathrm{H}_{\mathrm{ADR,X}})$ is defined by induction on the structure of multi-holed tree contexts as:

$$
\begin{aligned}
\langle\!\langle \varnothing \rangle\!\rangle_{\eta}^{(i,j)(l,u,r)} &\overset{\text{def}}{=} \{\mathsf{emp}\} \wedge (i = r) \wedge (j = l) \\
\langle\!\langle x \rangle\!\rangle_{\eta}^{(i,j)(l,u,r)} &\overset{\text{def}}{=} \Cup^{(\check{i},\check{j})(l,u,r)} \wedge (i = \check{i}) \wedge (j = \check{j}) \wedge \eta(x) = (i,j)(l,u,r) \\
\langle\!\langle n[ct] \rangle\!\rangle_{\eta}^{(i,j)(l,u,r)} &\overset{\text{def}}{=} \exists d,e.\, \{n \mapsto l,u,d,r\} \star \langle\!\langle ct \rangle\!\rangle_{\eta}^{(d,e)(\mathsf{null},n,\mathsf{null})} \wedge (i = n) \wedge (j = n) \\
\langle\!\langle ct_1 \otimes ct_2 \rangle\!\rangle_{\eta}^{(i,j)(l,u,r)} &\overset{\text{def}}{=} \exists p,q.\, \langle\!\langle ct_1 \rangle\!\rangle_{\eta}^{(i,p)(l,u,q)} \star \langle\!\langle ct_2 \rangle\!\rangle_{\eta}^{(q,j)(p,u,r)}
\end{aligned}
$$

and the lower crust predicate $\Cup^{(\check{i},\check{j})(l,u,r)} \in \mathcal{P}(\mathrm{H}_{\mathrm{ADR,X}})$ is defined as:

$$
\Cup^{(\check{i},\check{j})(l,u,r)} \overset{\text{def}}{=} \mathsf{ls}(\check{i},\check{j},l,u,r)
$$

where the ls predicate is defined as,

$$
\begin{aligned}
\mathsf{ls}(\check{i},\check{j},l,u,r) \overset{\text{def}}{=}\; & \{\mathsf{emp}\} \wedge (\check{i} = r) \wedge (\check{j} = l) \\
& \vee \exists k.\, \{\check{i} \mapsto l,u,\check{-},\check{k}\} \star \mathsf{ls}(\check{k},\check{j},\check{i},u,r)
\end{aligned}
$$

- ◇ the substitutive representation function is given by replacing each tree module command with a call to the correspondingly named procedure given in Figure 6.15 with,

$$\begin{aligned}
E.\texttt{left} &\overset{\text{def}}{=} E\\
E.\texttt{up} &\overset{\text{def}}{=} E+1\\
E.\texttt{down} &\overset{\text{def}}{=} E+2\\
E.\texttt{right} &\overset{\text{def}}{=} E+3\\
\texttt{n} := \texttt{newNode}() &\overset{\text{def}}{=} \texttt{n} := \texttt{alloc}(4)\\
\texttt{disposeNode}(E) &\overset{\text{def}}{=} \texttt{dispose}(E, 4).
\end{aligned}$$

The translation τ_2 is a crust inclusive translation in the terminology of our previous work [26]. Much of the translation is similar to the corresponding context based translation. The main difference is our treatment of the concrete interface, or crust.

The upper crust predicate $\Cap^{(i,j)(\check{l},\check{u},\check{r})}$ describes the concrete state that corresponds to an abstract address x with $\eta(x) = (i,j)(l,u,r)$. We illustrate this in Figure 6.16. The concrete address interface consists of the partial heap cells $\check{l}$, $\check{u}$ and $\check{r}$ which contain the definite pointers i and j appropriately. Any or all of these partial heap cells may in fact not exist, in which case the pointers in the tree are null. These partial heap cells correspond to the tree nodes that surround the tree at address x. Notice that we only have partial access to the pointers $\check{l}$, $\check{u}$ and $\check{r}$ in the tree, but full access to the pointers i and j in the concrete address interface. This means that a program run on this state can make modifications to the tree, but can only change the values of i and j in the concrete address interface. Thus, a program cannot delete nodes in the concrete address interface, or otherwise make modifications to the surrounding state.

The lower crust predicate $\Cup^{(\check{i},\check{j})(l,u,r)}$ describes the concrete state that corresponds to an abstract hole label x with $\eta(x) = (i,j)(l,u,r)$. We illustrate this in Figure 6.17. The concrete hole interface consists of a (potentially empty) list of partial heap cells from $\check{i}$ to $\check{j}$ which contains definite pointers to l, u and r in the appropriate places. These partial heap cells correspond to the top level of the tree that fills the context hole x. Notice that the pointers $\check{i}$ and $\check{j}$ into the hole are only partial, whilst the pointers l, u and r out of the list are complete. This means that a program run on this state can make modifications to the tree, but can only change the values of l, u and r in the list. Thus, a program cannot delete nodes in the concrete hole interface, or otherwise make modifications to the state within the context hole.

The upper and lower crusts for some label x consist of complimentary partial

```
proc n := getUp(m){
  n := [m.up]
}
proc n := getLeft(m){
  n := [m.left]
}
proc n := getFirst(m){
  n := [m.down]
}
proc n := getLast(m){
  local x in
    n := [m.down] ;
    if n ≠ null then
      x := [n.right] ;
      while x ≠ null do
        n := x ;
        x := [n.right]
}
proc appendChild(n, m){
  local x, y, z in
    x := [m.right] ;
    y := [m.left] ;
    z := [m.up] ;
    if x ≠ null then
      [x.left] := y
    if y ≠ null then
      [y.right] := x
    else
      if z ≠ null then
        [z.down] := x
    y := [n.down]
    if y = null then
      [n.down] := m
    else
      x := [y.right] ;
      while x ≠ null do
        y := x ;
        x := [y.right]
      [y.right] := m
    [m.left] := y ;
    [m.right] := null ;
    [m.up] := n
}
```

```
proc n := getRight(m){
  n := [m.right]
}

proc newNodeAfter(n){
  local x, y, z in
    y := [n.right] ;
    z := [n.up] ;
    x := newNode() ;
    [x.left] := n ;
    [x.up] := z ;
    [x.down] := null ;
    [x.right] := y ;
    [n.right] := x ;
    if y ≠ null then
      [y.left] := x
}

proc deleteTree(n){
  local x, y, z, w in
    x := [n.right] ;
    y := [n.left] ;
    z := [n.up] ;
    w := [n.down] ;
    call disposeForest(w) ;
    disposeNode(n) ;
    if x ≠ null then
      [x.left] := y ;
    if y ≠ null then
      [y.right] := x
    else
      if z ≠ null then
        [z.down] := x
}

proc disposeForest(n){
  local r, d in
    if n ≠ null then
      r := [n.right] ;
      call disposeForest (r) ;
      d := [n.down] ;
      call disposeForest (d) ;
      disposeNode(n)
}
```

Figure 6.15: Procedures for the heap-based implementation of the tree module.

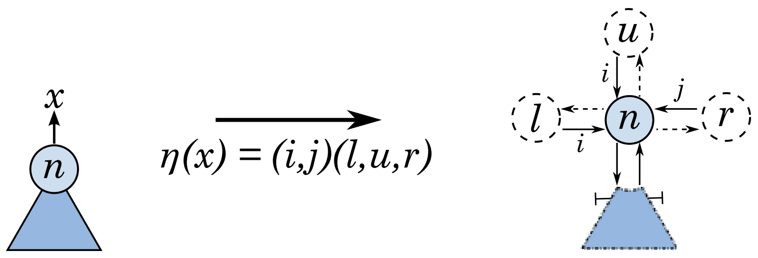

Figure 6.16: A translation in τ_2 which introduces some upper crust.

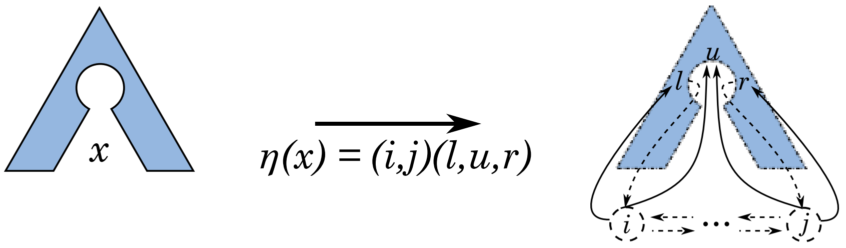

Figure 6.17: A translation in τ_2 which introduces some lower crust.

heap cells. This means that when you combine both of the crusts, you recover the complete heap cells associated with the concrete interface. We will see how this works in detail when we prove Lemma 6.18: crust inclusion.

Theorem 6.16 (Soundness of τ_2)**.** The pre-locality-preserving translation τ_2 is a locality-preserving translation.

Lemma 6.17 (Combination Preservation)**.** Segment combination is preserved by the segment representation function. That is, for all $st_1, st_2 \in S_\mathrm{T}$ and $\eta \in (\mathrm{X} \rightharpoonup_{\mathrm{fin}} \mathcal{I})$,

$$(\!| st_1 +_\mathrm{T} st_2 |\!)^\eta \quad \equiv \quad (\!| st_1 |\!)^\eta +_\mathrm{H} (\!| st_2 |\!)^\eta$$

Proof. This property follows from the definition of the segment representation function given in Definition 6.15. □

In order to prove the revelation preservation property for the translation τ_2 we require the crust inclusion lemma. This lemma states that given a context composition $ct \bullet_x ct'$ we can extract the concrete interface $\Cap^I$ corresponding to label x from the translation of $ct \bullet_x ct'$ plus its upper crust. This result relies on the use of partial heap cells to split the concrete interface corresponding to x into two pieces: one that is extracted as the upper crust of ct' and one that remains as the lower crust in the translation of ct.

Lemma 6.18 (Crust Inclusion)**.** For all $ct, ct' \in \mathrm{T}_{\mathrm{ID},\mathrm{X}}$, $I' \in \mathcal{I}$ and $\eta \in (\mathrm{X} \rightharpoonup_{\text{fin}} \mathcal{I})$, if $x \in fh_{\mathrm{T}}(ct)$ and $x \notin fh_{\mathrm{T}}(ct')$, then

$$\Cap^{I'} \star \langle\!\langle ct \bullet_x ct' \rangle\!\rangle_{\eta}^{I'} \quad \equiv \quad \exists I.\, \Cap^{I'} \star \langle\!\langle ct \rangle\!\rangle_{\eta[x \mapsto I]}^{I'} \star \Cap^{I} \star \langle\!\langle ct' \rangle\!\rangle_{\eta}^{I}$$

Proof. Proceed by induction on the structure of ct:

$ct = \varnothing$ case:

$x \notin fh_{\mathrm{T}}(\varnothing)$ which contradicts our assumption that $x \in fh_{\mathrm{T}}(ct)$, so this case holds vacuously.

$ct = y$ case:

If $y \neq x$ then $x \notin fh_{\mathrm{T}}(y)$ which contradicts our assumption that $x \in fh_{\mathrm{T}}(ct)$, so this case holds vacuously. If $y = x$ then let $I' = (i', j')(\check{l}', \check{u}', \check{r}')$ for some i', j', l', u' and r'. We can then show the following:

$$\begin{aligned}
\Cap^{I'} \star \langle\!\langle ct \bullet_x ct' \rangle\!\rangle_{\eta}^{I'} &\equiv \Cap^{I'} \star \langle\!\langle x \bullet_x ct' \rangle\!\rangle_{\eta}^{I'} \\
&\equiv \Cap^{I'} \star \langle\!\langle ct' \rangle\!\rangle_{\eta}^{I'} \\
&\equiv \Cap^{(i',j')(\check{l}',\check{u}',\check{r}')} \star \langle\!\langle ct' \rangle\!\rangle_{\eta}^{(i',j')(\check{l}',\check{u}',\check{r}')} \\
&\equiv (\{\check{l}' \mapsto \check{-},\check{-},\check{-},i'\} \vee (\check{l}' = \mathsf{null} \wedge (\{\check{u}' \mapsto \check{-},\check{-},i',\check{-}\} \vee \check{u}' \doteq \mathsf{null}))) \\
&\quad \star (\{\check{r}' \mapsto j',\check{-},\check{-},\check{-}\} \vee \check{r}' \doteq \mathsf{null}) \star \langle\!\langle ct' \rangle\!\rangle_{\eta}^{(i',j')(\check{l}',\check{u}',\check{r}')} \\
&\equiv (\{\check{l}' \mapsto \check{-},\check{-},\check{-},\check{i}'\} \vee (\check{l}' = \mathsf{null} \wedge (\{\check{u}' \mapsto \check{-},\check{-},\check{i}',\check{-}\} \vee \check{u}' \doteq \mathsf{null}))) \\
&\quad \star (\{\check{r}' \mapsto \check{j}',\check{-},\check{-},\check{-}\} \vee \check{r}' \doteq \mathsf{null}) \\
&\quad \star (\{\check{l}' \mapsto \check{-},\check{-},\check{-},i'\} \vee (\check{l}' = \mathsf{null} \wedge (\{\check{u}' \mapsto \check{-},\check{-},i',\check{-}\} \vee \check{u}' \doteq \mathsf{null}))) \\
&\quad \star (\{\check{r}' \mapsto j',\check{-},\check{-},\check{-}\} \vee \check{r}' \doteq \mathsf{null}) \star \langle\!\langle ct' \rangle\!\rangle_{\eta}^{(i',j')(\check{l}',\check{u}',\check{r}')} \\
&\equiv \exists i, j, l, u, r.\, (i = i') \wedge (j = j') \wedge (l = l') \wedge (u = u') \wedge (r = r') \\
&\quad \wedge (\{\check{l}' \mapsto \check{-},\check{-},\check{-},\check{i}'\} \vee (\check{l}' = \mathsf{null} \wedge (\{\check{u}' \mapsto \check{-},\check{-},\check{i}',\check{-}\} \vee \check{u}' \doteq \mathsf{null}))) \\
&\quad \star (\{\check{r}' \mapsto \check{j}',\check{-},\check{-},\check{-}\} \vee \check{r}' \doteq \mathsf{null}) \\
&\quad \star (\{\check{l} \mapsto \check{-},\check{-},\check{-},i\} \vee (\check{l} = \mathsf{null} \wedge (\{\check{u} \mapsto \check{-},\check{-},i,\check{-}\} \vee \check{u} \doteq \mathsf{null}))) \\
&\quad \star (\{\check{r} \mapsto j,\check{-},\check{-},\check{-}\} \vee \check{r} \doteq \mathsf{null}) \star \langle\!\langle ct' \rangle\!\rangle_{\eta}^{(i,j)(\check{l},\check{u},\check{r})} \\
&\equiv \exists i, j, l, u, r.\, (i = i') \wedge (j = j') \wedge (l = l') \wedge (u = u') \wedge (r = r') \\
&\quad \wedge (\{\check{l}' \mapsto \check{-},\check{-},\check{-},\check{i}'\} \vee (\check{l}' = \mathsf{null} \wedge (\{\check{u}' \mapsto \check{-},\check{-},\check{i}',\check{-}\} \vee \check{u}' \doteq \mathsf{null}))) \\
&\quad \star (\{\check{r}' \mapsto \check{j}',\check{-},\check{-},\check{-}\} \vee \check{r}' \doteq \mathsf{null}) \\
&\quad \star (\{\check{l} \mapsto \check{-},\check{-},\check{-},i\} \vee (\check{l} = \mathsf{null} \wedge (\{\check{u} \mapsto \check{-},\check{-},i,\check{-}\} \vee \check{u} \doteq \mathsf{null}))) \\
&\quad \star (\{\check{r} \mapsto j,\check{-},\check{-},\check{-}\} \vee \check{r} \doteq \mathsf{null}) \star \mathsf{ls}(\check{i}, \check{j}, \check{l}, \check{u}, \check{r}) * \langle\!\langle ct' \rangle\!\rangle_{\eta}^{(i,j)(\check{l},\check{u},\check{r})} \\
&\equiv \exists i, j, l, u, r.\, (i = i') \wedge (j = j') \wedge (l = l') \wedge (u = u') \wedge (r = r') \\
&\quad \wedge \Cap^{(i',j')(\check{l}',\check{u}',\check{r}')} \star \langle\!\langle x \rangle\!\rangle_{\eta[x \mapsto (i,j)(l,u,r)]}^{(i',j')(\check{l}',\check{u}',\check{r}')} \star \Cap^{(i,j)(\check{l},\check{u},\check{r})} \star \langle\!\langle ct' \rangle\!\rangle_{\eta}^{(i,j)(\check{l},\check{u},\check{r})} \\
&\equiv \exists I.\, (I = I') \wedge \Cap^{I'} \star \langle\!\langle x \rangle\!\rangle_{\eta[x \mapsto I]}^{I'} \star \Cap^{I} \star \langle\!\langle ct' \rangle\!\rangle_{\eta}^{I} \\
&\equiv \exists I.\, \Cap^{I'} \star \langle\!\langle ct \rangle\!\rangle_{\eta[x \mapsto I]}^{I'} \star \Cap^{I} \star \langle\!\langle ct' \rangle\!\rangle_{\eta}^{I}
\end{aligned}$$

$ct = n[ct'']$ case:

There are two cases to consider. If $x \notin fh_{\mathrm{T}}(ct'')$ then $x \notin fh_{\mathrm{T}}(n[ct''])$ which contradicts our assumption that $x \in fh_{\mathrm{T}}(ct)$, so this case holds vacuously. If $x \in fh_{\mathrm{T}}(ct'')$ then by the induction hypothesis,

$$\Cap^{I''} \star \langle\!\langle ct'' \bullet_x ct' \rangle\!\rangle^{I''}_{\eta} \quad\equiv\quad \exists I.\, \Cap^{I''} \star \langle\!\langle ct'' \rangle\!\rangle^{I''}_{\eta[x\mapsto I]} \star \Cap^{I} \star \langle\!\langle ct' \rangle\!\rangle^{I}_{\eta}$$

Let $I' = (i', j')(\check{l}', \check{u}', \check{r}')$ for some i', j', l', u' and r'. We can then show the following:

$$\begin{aligned}
\Cap^{I'} \star \langle\!\langle ct \bullet_x ct' \rangle\!\rangle^{I'}_{\eta} \quad &\equiv\quad \Cap^{I'} \star \langle\!\langle n[ct''] \bullet_x ct' \rangle\!\rangle^{I'}_{\eta} \\
&\equiv\quad \Cap^{I'} \star \langle\!\langle n[ct'' \bullet_x ct'] \rangle\!\rangle^{I'}_{\eta} \\
&\equiv\quad \Cap^{(i',j')(\check{l}',\check{u}',\check{r}')} \star \langle\!\langle n[ct'' \bullet_x ct'] \rangle\!\rangle^{(i',j')(\check{l}',\check{u}',\check{r}')}_{\eta} \\
&\equiv\quad \Cap^{(i',j')(\check{l}',\check{u}',\check{r}')} \star \exists i, j.\, \{n \mapsto \check{l}',\check{u}',i,\check{r}'\} \star \langle\!\langle ct'' \bullet_x ct' \rangle\!\rangle^{(i,j)(\mathsf{null},n,\mathsf{null})}_{\eta} \\
&\qquad \wedge (i' = n) \wedge (j' = n) \\
&\equiv\quad \Cap^{(i',j')(\check{l}',\check{u}',\check{r}')} \star \exists i, j.\, \{n \mapsto \check{l}',\check{u}',\check{i},\check{r}'\} \star \{\check{n} \mapsto \check{-},\check{-},i,\check{-}\} \\
&\qquad \star \langle\!\langle ct'' \bullet_x ct' \rangle\!\rangle^{(i,j)(\mathsf{null},\check{n},\mathsf{null})}_{\eta} \wedge (i' = n) \wedge (j' = n) \\
&\equiv\quad \Cap^{(i',j')(\check{l}',\check{u}',\check{r}')} \star \exists i, j, l, u, r.\, \{n \mapsto \check{l}',\check{u}',\check{i},\check{r}'\} \star \{\check{u} \mapsto \check{-},\check{-},i,\check{-}\} \\
&\qquad \star \langle\!\langle ct'' \bullet_x ct' \rangle\!\rangle^{(i,j)(\check{l},\check{u},\check{r})}_{\eta} \wedge (i' = n) \wedge (j' = n) \\
&\qquad \wedge (\check{l} = \mathsf{null}) \wedge (\check{u} = \check{n}) \wedge (\check{r} = \mathsf{null}) \\
&\equiv\quad \Cap^{(i',j')(\check{l}',\check{u}',\check{r}')} \star \exists i, j, l, u, r.\, \{n \mapsto \check{l}',\check{u}',\check{i},\check{r}'\} \star \Cap^{(i,j)(\check{l},\check{u},\check{r})} \\
&\qquad \star \langle\!\langle ct'' \bullet_x ct' \rangle\!\rangle^{(i,j)(\check{l},\check{u},\check{r})}_{\eta} \wedge (i' = n) \wedge (j' = n) \\
&\qquad \wedge (\check{l} = \mathsf{null}) \wedge (\check{u} = \check{n}) \wedge (\check{r} = \mathsf{null}) \\
(IH) \quad &\equiv\quad \Cap^{(i',j')(\check{l}',\check{u}',\check{r}')} \star \exists I, i, j, l, u, r.\, \{n \mapsto \check{l}',\check{u}',\check{i},\check{r}'\} \star \Cap^{(i,j)(\check{l},\check{u},\check{r})} \\
&\qquad \star \langle\!\langle ct'' \rangle\!\rangle^{(i,j)(\check{l},\check{u},\check{r})}_{\eta[x\mapsto I]} \star \Cap^{I} * \langle\!\langle ct' \rangle\!\rangle^{I}_{\eta} \wedge (i' = n) \wedge (j' = n) \\
&\qquad \wedge (\check{l} = \mathsf{null}) \wedge (\check{u} = \check{n}) \wedge (\check{r} = \mathsf{null}) \\
&\equiv\quad \Cap^{(i',j')(\check{l}',\check{u}',\check{r}')} \star \exists I, i, j, l, u, r.\, \{n \mapsto \check{l}',\check{u}',\check{i},\check{r}'\} \star \{\check{u} \mapsto \check{-},\check{-},i,\check{-}\} \\
&\qquad \star \langle\!\langle ct'' \rangle\!\rangle^{(i,j)(\check{l},\check{u},\check{r})}_{\eta[x\mapsto I]} \star \Cap^{I} \star \langle\!\langle ct' \rangle\!\rangle^{I}_{\eta} \wedge (i' = n) \wedge (j' = n) \\
&\qquad \wedge (\check{l} = \mathsf{null}) \wedge (\check{u} = \check{n}) \wedge (\check{r} = \mathsf{null}) \\
&\equiv\quad \Cap^{(i',j')(\check{l}',\check{u}',\check{r}')} \star \exists I, i, j.\, \{n \mapsto \check{l}',\check{u}',\check{i},\check{r}'\} \star \{\check{n} \mapsto \check{-},\check{-},i,\check{-}\} \\
&\qquad \star \langle\!\langle ct'' \rangle\!\rangle^{(i,j)(\mathsf{null},\check{n},\mathsf{null})}_{\eta[x\mapsto I]} \star \Cap^{I} \star \langle\!\langle ct' \rangle\!\rangle^{I}_{\eta} \wedge (i' = n) \wedge (j' = n) \\
&\equiv\quad \Cap^{(i',j')(\check{l}',\check{u}',\check{r}')} \star \exists I, i, j.\, \{n \mapsto \check{l}',\check{u}',i,\check{r}'\} \\
&\qquad \star \langle\!\langle ct'' \rangle\!\rangle^{(i,j)(\mathsf{null},n,\mathsf{null})}_{\eta[x\mapsto I]} \star \Cap^{I} \star \langle\!\langle ct' \rangle\!\rangle^{I}_{\eta} \wedge (i' = n) \wedge (j' = n) \\
&\equiv\quad \Cap^{(i',j')(\check{l}',\check{u}',\check{r}')} \star \exists I.\, \langle\!\langle n[ct''] \rangle\!\rangle^{(i',j')(\check{l}',\check{u}',\check{r}')}_{\eta[x\mapsto I]} \star \Cap^{I} \star \langle\!\langle ct' \rangle\!\rangle^{I}_{\eta} \\
&\equiv\quad \exists I.\, \Cap^{I'} \star \langle\!\langle n[ct''] \rangle\!\rangle^{I'}_{\eta[x\mapsto I]} \star \Cap^{I} \star \langle\!\langle ct' \rangle\!\rangle^{I}_{\eta} \\
&\equiv\quad \exists I.\, \Cap^{I'} \star \langle\!\langle ct \rangle\!\rangle^{I'}_{\eta[x\mapsto I]} \star \Cap^{I} \star \langle\!\langle ct' \rangle\!\rangle^{I}_{\eta}
\end{aligned}$$

$ct = ct_1 \otimes ct_2$ case:

There are four cases to consider. If $x \notin fh_{\mathrm{T}}(ct_1)$ and $x \notin fh_{\mathrm{T}}(ct_2)$ then $x \notin fh_{\mathrm{T}}(ct_1 \otimes ct_2)$ which contradicts our assumption that $x \in fh_{\mathrm{T}}(ct)$, so this case holds vacuously. If $x \in fh_{\mathrm{T}}(ct_1)$ and $x \in fh_{\mathrm{T}}(ct_2)$ then the tree context $ct_1 \otimes ct_2$ is not well formed and again this case holds vacuously. If $x \in fh_{\mathrm{T}}(ct_1)$ and $x \notin fh_{\mathrm{T}}(ct_2)$ then by the inductive hypothesis,

$$\Cap^{I''} \star \langle\!\langle (ct_1 \bullet_x ct') \rangle\!\rangle^{I''}_{\eta} \quad \equiv \quad \exists I.\, \Cap^{I''} \star \langle\!\langle ct_1 \rangle\!\rangle^{I''}_{\eta[x \mapsto I]} \star \Cap^{I} \star \langle\!\langle ct' \rangle\!\rangle^{I}_{\eta}$$

Let $I' = (i', j')(\check{l}', \check{u}', \check{r}')$ for some i', j', l', u' and r'. We can then show the following:

$$\begin{aligned}
\Cap^{I'} \star \langle\!\langle ct \bullet_x ct' \rangle\!\rangle^{I'}_{\eta} &\equiv \Cap^{I'} \star \langle\!\langle (ct_1 \otimes ct_2) \bullet_x ct' \rangle\!\rangle^{I'}_{\eta} \\
&\equiv \Cap^{I'} \star \langle\!\langle (ct_1 \bullet_x ct') \otimes ct_2 \rangle\!\rangle^{I'}_{\eta} \\
&\equiv \Cap^{(i',j')(\check{l}',\check{u}',\check{r}')} \star \langle\!\langle (ct_1 \bullet_x ct') \otimes ct_2 \rangle\!\rangle^{(i',j')(\check{l}',\check{u}',\check{r}')}_{\eta} \\
&\equiv \Cap^{(i',j')(\check{l}',\check{u}',\check{r}')} \star \exists a, b.\, \langle\!\langle (ct_1 \bullet_x ct') \rangle\!\rangle^{(i',a)(\check{l}',\check{u}',b)}_{\eta} \star \langle\!\langle ct_2 \rangle\!\rangle^{(b,j')(a,\check{u}',\check{r}')}_{\eta} \\
&\equiv (\{\check{l}' \mapsto \check{-},\check{-},\check{-},i'\} \vee (\check{l}' = \mathsf{null} \wedge (\{\check{u}' \mapsto \check{-},\check{-},i',\check{-}\} \vee \check{u}' \doteq \mathsf{null}))) \\
&\quad \star (\{\check{r}' \mapsto j',\check{-},\check{-},\check{-}\} \vee \check{r} \doteq \mathsf{null}) \\
&\quad \star \exists a, b.\, \langle\!\langle (ct_1 \bullet_x ct') \rangle\!\rangle^{(i',a)(\check{l}',\check{u}',b)}_{\eta} \star \langle\!\langle ct_2 \rangle\!\rangle^{(b,j')(a,\check{u}',\check{r}')}_{\eta} \\
&\equiv (\{\check{l}' \mapsto \check{-},\check{-},\check{-},i'\} \vee (\check{l}' = \mathsf{null} \wedge (\{\check{u}' \mapsto \check{-},\check{-},i',\check{-}\} \vee \check{u}' \doteq \mathsf{null}))) \\
&\quad \star (\{\check{r}' \mapsto j',\check{-},\check{-},\check{-}\} \vee \check{r} \doteq \mathsf{null}) \star (\{\check{b} \mapsto a,\check{-},\check{-},\check{-}\} \vee \check{b} \doteq \mathsf{null}) \\
&\quad \star \exists a, b.\, \langle\!\langle (ct_1 \bullet_x ct') \rangle\!\rangle^{(i',a)(\check{l}',\check{u}',\check{b})}_{\eta} \star \langle\!\langle ct_2 \rangle\!\rangle^{(b,j')(\check{a},\check{u}',\check{r}')}_{\eta} \\
&\equiv \exists a, b.\, (\{\check{r}' \mapsto j',\check{-},\check{-},\check{-}\} \vee \check{r} \doteq \mathsf{null}) \star \Cap^{(i',a)(\check{l}',\check{u}',\check{b})} \\
&\quad \star \langle\!\langle (ct_1 \bullet_x ct') \rangle\!\rangle^{(i',a)(\check{l}',\check{u}',\check{b})}_{\eta} \star \langle\!\langle ct_2 \rangle\!\rangle^{(b,j')(\check{a},\check{u}',\check{r}')}_{\eta} \\
(IH) &\equiv \exists I.\, (\{\check{r}' \mapsto j',\check{-},\check{-},\check{-}\} \vee \check{r} \doteq \mathsf{null}) \star \exists a, b.\, \Cap^{(i',a)(\check{l}',\check{u}',\check{b})} \\
&\quad \star \langle\!\langle ct_1 \rangle\!\rangle^{(i',a)(\check{l}',\check{u}',\check{b})}_{\eta[x \mapsto I]} \star \Cap^{I} \star \langle\!\langle ct' \rangle\!\rangle^{I}_{\eta} \star \langle\!\langle ct_2 \rangle\!\rangle^{(b,j')(\check{a},\check{u}',\check{r}')}_{\eta} \\
&\equiv \exists I.\, (\{\check{l}' \mapsto \check{-},\check{-},\check{-},i'\} \vee (\check{l}' = \mathsf{null} \wedge (\{\check{u}' \mapsto \check{-},\check{-},i',\check{-}\} \vee \check{u}' \doteq \mathsf{null}))) \\
&\quad \star (\{\check{r}' \mapsto j',\check{-},\check{-},\check{-}\} \vee \check{r} \doteq \mathsf{null}) \star \exists a, b.\, (\{\check{b} \mapsto a,\check{-},\check{-},\check{-}\} \vee \check{b} \doteq \mathsf{null}) \\
&\quad \star \langle\!\langle ct_1 \rangle\!\rangle^{(i',a)(\check{l}',\check{u}',\check{b})}_{\eta[x \mapsto I]} \star \Cap^{I} \star \langle\!\langle ct' \rangle\!\rangle^{I}_{\eta} \star \langle\!\langle ct_2 \rangle\!\rangle^{(b,j')(\check{a},\check{u}',\check{r}')}_{\eta} \\
&\equiv \exists I.\, (\{\check{l}' \mapsto \check{-},\check{-},\check{-},i'\} \vee (\check{l}' = \mathsf{null} \wedge (\{\check{u}' \mapsto \check{-},\check{-},i',\check{-}\} \vee \check{u}' \doteq \mathsf{null}))) \\
&\quad \star (\{\check{r}' \mapsto j',\check{-},\check{-},\check{-}\} \vee \check{r} \doteq \mathsf{null}) \\
&\quad \star \exists a, b.\, \langle\!\langle ct_1 \rangle\!\rangle^{(i',a)(\check{l}',\check{u}',b)}_{\eta[x \mapsto I]} \star \Cap^{I} \star \langle\!\langle ct' \rangle\!\rangle^{I}_{\eta} \star \langle\!\langle ct_2 \rangle\!\rangle^{(b,j')(a,\check{u}',\check{r}')}_{\eta} \\
&\equiv \exists I.\, (\{\check{l}' \mapsto \check{-},\check{-},\check{-},i'\} \vee (\check{l}' = \mathsf{null} \wedge (\{\check{u}' \mapsto \check{-},\check{-},i',\check{-}\} \vee \check{u}' \doteq \mathsf{null}))) \\
&\quad \star (\{\check{r}' \mapsto j',\check{-},\check{-},\check{-}\} \vee \check{r} \doteq \mathsf{null}) \\
&\quad \star \langle\!\langle ct_1 \otimes ct_2 \rangle\!\rangle^{(i',j')(\check{l}',\check{u}',\check{r}')}_{\eta[x \mapsto I]} \star \Cap^{I} \star \langle\!\langle ct' \rangle\!\rangle^{I}_{\eta} \\
&\equiv \exists I.\, \Cap^{(i',j')(\check{l}',\check{u}',\check{r}')} \star \langle\!\langle ct_1 \otimes ct_2 \rangle\!\rangle^{(i',j')(\check{l}',\check{u}',\check{r}')}_{\eta[x \mapsto I]} \star \Cap^{I} \star \langle\!\langle ct' \rangle\!\rangle^{I}_{\eta} \\
&\equiv \exists I.\, \Cap^{I'} \star \langle\!\langle ct_1 \otimes ct_2 \rangle\!\rangle^{I'}_{\eta[x \mapsto I]} \star \Cap^{I} \star \langle\!\langle ct' \rangle\!\rangle^{I}_{\eta} \\
&\equiv \exists I.\, \Cap^{I'} \star \langle\!\langle ct \rangle\!\rangle^{I'}_{\eta[x \mapsto I]} \star \Cap^{I} \star \langle\!\langle ct' \rangle\!\rangle^{I}_{\eta}
\end{aligned}$$

The final case for $x \notin fh_{\mathrm{T}}(ct_1)$ and $x \in fh_{\mathrm{T}}(ct_2)$ is analogous to the case given above. Note that (IH) denotes an application of the inductive hypothesis.

$\square$

Lemma 6.19 (Compression Preservation)**.** Segment compression is preserved by the segment representation function. That is, for all $x \in \mathrm{X}$, $st \in S_{\mathrm{T}}$ and $\eta \in (\mathrm{X} \rightharpoonup_{\text{fin}} \mathcal{I})$, there exists $I \in \mathcal{I}$ and $\bar{x} \in \mathcal{P}(\mathrm{X})$ with $\bar{x} = \mathsf{labs}(I)$ such that,

$$\llparenthesis (x)(st)_{\mathrm{T}} \rrparenthesis^{\eta} \;\equiv\; (\bar{x})(\llparenthesis st \rrparenthesis^{\eta[x \mapsto I]})_{\mathrm{H}}$$

Proof. Recall that in this translation $\mathsf{labs}(I) = \emptyset$ for all $I \in \mathcal{I}$. Thus, it is sufficient to show that,

$$\llparenthesis (x)(st)_{\mathrm{T}} \rrparenthesis^{\eta} \;\equiv\; \exists I.\, \llparenthesis st \rrparenthesis^{\eta[x \mapsto I]}$$

Case split on the occurrences of label x in segment st. There are four cases to consider:

(1) If $x \notin fa_{\mathrm{T}}(st)$ and $x \notin fh_{\mathrm{T}}(st)$, then $(x)(st) = st$. Any choice of I will suffice as it will never be referenced by the translation. We can then show the following:

$$\begin{aligned}
\llparenthesis (x)(st) \rrparenthesis^{\eta} \;&\equiv\; \llparenthesis st \rrparenthesis^{\eta} \\
&\equiv\; \exists I.\, \llparenthesis st \rrparenthesis^{\eta[x \mapsto I]}
\end{aligned}$$

(2) If $x \in fa_{\mathrm{T}}(st)$ and $x \notin fh_{\mathrm{T}}(st)$, then there exist some st', ct such that $st = st' \uplus \{(x, ct)\}$ where $x \notin fh_{\mathrm{T}}(st')$. Let $I = (i, j)(\mathsf{null}, \mathsf{null}, \mathsf{null})$ for some i and j. We can then show the following:

$$\begin{aligned}
\llparenthesis (x)(st) \rrparenthesis^{\eta} \;&\equiv\; \llparenthesis (x)(st' \uplus \{(x, ct)\}) \rrparenthesis^{\eta} \\
&\equiv\; \llparenthesis st' \uplus \{(0, ct)\} \rrparenthesis^{\eta} \\
&\equiv\; \llparenthesis st' \rrparenthesis^{\eta} + \llparenthesis \{(0, ct)\} \rrparenthesis^{\eta} \\
&\equiv\; \llparenthesis st' \rrparenthesis^{\eta} + \exists i, j.\, \lceil \Cap^{(i,j)(\mathsf{null},\mathsf{null},\mathsf{null})} \star \langle\!\langle ct \rangle\!\rangle_{\eta}^{(i,j)(\mathsf{null},\mathsf{null},\mathsf{null})} \rceil \\
&\equiv\; \llparenthesis st' \rrparenthesis^{\eta} + \exists i, j.\, \llparenthesis \{(x, ct)\} \rrparenthesis^{\eta[x \mapsto (i,j)(\mathsf{null},\mathsf{null},\mathsf{null})]} \\
&\equiv\; \llparenthesis st' \rrparenthesis^{\eta} + \exists I.\, \llparenthesis \{(x, ct)\} \rrparenthesis^{\eta[x \mapsto I]} \\
&\equiv\; \exists I.\, \llparenthesis st' \rrparenthesis^{\eta[x \mapsto I]} + \llparenthesis \{(x, ct)\} \rrparenthesis^{\eta[x \mapsto I]} \\
&\equiv\; \exists I.\, \llparenthesis st' \uplus \{(x, ct)\} \rrparenthesis^{\eta[x \mapsto I]} \\
&\equiv\; \exists I.\, \llparenthesis st \rrparenthesis^{\eta[x \mapsto I]}
\end{aligned}$$

(3) If $x \notin fa_{\mathrm{T}}(st)$ and $x \in fh_{\mathrm{T}}(st)$, then $(x)(st)$ is undefined so $\llparenthesis (x)(st) \rrparenthesis^{\eta} = \emptyset$. Let $I = (\mathsf{null}, \mathsf{null})(\mathsf{null}, \mathsf{null}, \mathsf{null})$, then $\llparenthesis st \rrparenthesis^{\eta[x \mapsto I]} = \emptyset$. If there are any nodes in the tree segment st, then for some node n we would have $n \mapsto - \wedge (n = \mathsf{null})$ which cannot be satisfied by any heap. If instead there are no nodes in the tree segment

st, then $(\!|st|\!)^{\eta[x\mapsto I]} = \lceil \mathsf{emp} \rceil = \emptyset$.

(4) If $x \in fa_{\mathrm{T}}(st)$ and $x \in fh_{\mathrm{T}}(st)$, then there exist some st', z, ct, ct' such that $st = st' \uplus \{(z, ct), (x, ct')\}$ where $x \notin fa_{\mathrm{T}}(st')$, $x \notin fh_{\mathrm{T}}(st')$ and $x \in fh_{\mathrm{T}}(ct)$. Tree segments do not contain cycles, so we can assume that $x \notin fh_{\mathrm{T}}(ct')$. Let $\eta(z) = I'$ for some $I' \in \mathcal{I}$. We can then show the following:

$$\begin{array}{rcl}
(\!|(x)(st)|\!)^{\eta} & \equiv & (\!|(x)(st' \uplus \{(z, ct), (x, ct')\})|\!)^{\eta} \\
 & \equiv & (\!|st' \uplus \{(z, ct \bullet_x ct')\}|\!)^{\eta} \\
 & \equiv & (\!|st'|\!)^{\eta} + (\!|\{(z, ct \bullet_x ct')\}|\!)^{\eta} \\
 & \equiv & (\!|st'|\!)^{\eta} + \lceil \Cap^{I'} \star \langle\!\langle ct \bullet_x ct' \rangle\!\rangle^{I'}_{\eta} \rceil \\
\text{(Lemma 6.18)} & \equiv & (\!|st'|\!)^{\eta} + \exists I.\, \lceil \Cap^{I'} \star \langle\!\langle ct \rangle\!\rangle^{I'}_{\eta[x\mapsto I]} \star \Cap^{I} \star \langle\!\langle ct' \rangle\!\rangle^{I}_{\eta} \rceil \\
 & \equiv & \exists I.\, (\!|st'|\!)^{\eta[x\mapsto I]} + \lceil \Cap^{I'} \star \langle\!\langle ct \rangle\!\rangle^{I'}_{\eta[x\mapsto I]} \star \Cap^{I} \star \langle\!\langle ct' \rangle\!\rangle^{I}_{\eta[x\mapsto I]} \rceil \\
 & \equiv & \exists I.\, (\!|st'|\!)^{\eta[x\mapsto I]} + \lceil \Cap^{I'} \star \langle\!\langle ct \rangle\!\rangle^{I'}_{\eta[x\mapsto I]} \rceil + \lceil \Cap^{I} \ast \langle\!\langle ct' \rangle\!\rangle^{I}_{\eta[x\mapsto I]} \rceil \\
 & \equiv & \exists I.\, (\!|st'|\!)^{\eta[x\mapsto I]} + (\!|\{(z, ct)\}|\!)^{\eta[x\mapsto I]} + (\!|\{(x, ct')\}|\!)^{\eta[x\mapsto I]} \\
 & \equiv & \exists I.\, (\!|st' \uplus \{(z, ct), (x, ct')\}|\!)^{\eta[x\mapsto I]} \\
 & \equiv & \exists I.\, (\!|st|\!)^{\eta[x\mapsto I]}
\end{array}$$

□

Lemma 6.20 (Axiom Correctness)**.** For all $e \in \textsc{Env}$, $\Gamma \in \textsc{PSEnv}$, $\varphi \in \textsc{Cmd}_{\mathbb{T}}$, $(P, Q) \in \mathrm{Ax}[\![\varphi]\!]_{\mathbb{T}}$ and $\eta \in (\mathrm{X}_{\mathbb{A}} \rightharpoonup_{\mathrm{fin}} \mathcal{I})$,

$$e, [\![\Gamma]\!]_{\tau_2} \vdash_{\mathbb{B}} \left\{ \ [\![P]\!]_{\tau_2} \ \right\} \ \ [\![\varphi]\!]_{\tau_2} \ \ \left\{ \ [\![Q]\!]_{\tau_2} \ \right\}$$

We do not give the proofs for all of the basic commands in the tree module, but give four examples that illustrate the techniques involved in the proofs. We first give a proof of a simple case, showing that the implementation of the `getUp` command satisfies its translated specification. We then move on to a series of increasingly more complex examples. We show that the `deleteTree` command satisfies its translated specification, which requires us to work with the upper crust of a segment. We then show that the `getLast` command satisfies its translated specification, which requires us to work with the lower crust of a segment. Finally, we show that the `appendChild` command satisfies its translated specification, which requires us to work with multiple segments and upper and lower crusts. The implementations of the other basic commands can be shown to satisfy their translated specifications in a similar fashion.

Axiom Correctness: `getUp`

Recall the specification of the `getUp` command from Figure 5.1.

$$\left\{\ \alpha{\leftarrow}m[\beta \otimes w[\delta] \otimes \gamma] * \mathtt{n} \Rightarrow n * \sigma \wedge \mathcal{E}[\![E]\!]\sigma[\mathtt{n} \mapsto n] = w\ \right\}$$
$$\mathtt{n} := \mathtt{getUp}(E)$$
$$\left\{\ \alpha{\leftarrow}m[\beta \otimes w[\delta] \otimes \gamma] * \mathtt{n} \Rightarrow m * \sigma\ \right\}$$

$$\left\{\ \lceil w[\beta] \rceil * \mathtt{n} \Rightarrow n * \sigma \wedge \mathcal{E}[\![E]\!]\sigma[\mathtt{n} \mapsto n] = w\ \right\}$$
$$\mathtt{n} := \mathtt{getUp}(E)$$
$$\left\{\ \lceil w[\beta] \rceil * \mathtt{n} \Rightarrow \mathsf{null} * \sigma\ \right\}$$

To prove that the first specification holds under our translation, suppose that $e(\alpha) = x$, $e(\beta) = y_1$, $e(\gamma) = y_2$ and $e(\delta) = z$ for some $x, y_1, y_2, z \in \mathrm{X}$. We can also assume that $\{x, y_1, y_2, z\} \subseteq dom(\eta)$, otherwise the translated precondition is equivalent to false, and that $\eta(x) = (i,j)(l,u,r)$, $\eta(y_1) = (i_1,j_1)(l_1,u_1,r_1)$, $\eta(y_2) = (i_2,j_2)(l_2,u_2,r_2)$ and $\eta(z) = (i',j')(l',u',r')$ for some choice of these interfaces. In Figure 6.18 we give a proof outline showing that the implementation of `getUp` (from Figure 6.15) satisfies the translation of its first specification.

To prove that the second specification holds under our translation, suppose that $e(\alpha) = x$ and $e(\beta) = y$ for some $x, y \in \mathrm{X}$. We can also assume that $\{x, y\} \subseteq dom(\eta)$, otherwise the precondition is equivalent to false, and that $\eta(x) = (i,j)(l,u,r)$ and $\eta(y) = (i',j')(l',u',r')$ form some choice of these interfaces. In Figure 6.19 we give a proof outline showing that the implementation of `getUp` (from Figure 6.15) satisfies the translation of its second specification.

The implementation considered in this example is very simple and does not need to access any of the extra state in either crust. However, the example illustrates how our translation converts a tree segment into a heap segment. In both proof outlines the indentation around the $\mathtt{n} := [\mathtt{m.up}]$ line is used to indicate the use of the separation frame rule. In further proofs we will not give so many explicit steps, but it is useful to see how our framework behaves in full on a simple example.

```
$\{\ \llbracket \alpha{\leftarrow}m[\beta \otimes w[\delta] \otimes \gamma] * \mathtt{n} \Rrightarrow - * \mathtt{m} \Rrightarrow w \rrbracket_{\tau_2}\ \}$
 proc n := getUp(m){
    $\{\ \lceil \Cap^{(i,j)(\check{l},\check{u},\check{r})} \star \langle\!\langle m[y_1 \otimes w[z] \otimes y_2] \rangle\!\rangle_{\eta}^{(i,j)(\check{l},\check{u},\check{r})} \rceil * \mathtt{n} \Rrightarrow - * \mathtt{m} \Rrightarrow w\ \}$
    $\left\{ \begin{array}{l} \left\lceil \begin{array}{l} \Cap^{(i,j)(\check{l},\check{u},\check{r})} \star m \mapsto \check{l},\check{u},\check{i}_1,\check{r} \star w \mapsto \check{j}_1,m,\check{i}',\check{i}_2 \\ \quad \star \Cup^{(\check{i}_1,\check{j}_1)(\mathsf{null},m,w)} \star \Cup^{(\check{i}',\check{j}')(\mathsf{null},w,\mathsf{null})} \star \Cup^{(\check{i}_2,\check{j}_2)(w,m,\mathsf{null})} \end{array} \right\rceil \\ * \mathtt{n} \Rrightarrow - * \mathtt{m} \Rrightarrow w \end{array} \right\}$
    $\left\{ \begin{array}{l} \left\lceil \begin{array}{l} \Cap^{(i,j)(\check{l},\check{u},\check{r})} \star m \mapsto \check{l},\check{u},\check{i}_1,\check{r} \star \Cup^{(\check{i}_1,\check{j}_1)(\mathsf{null},m,w)} \\ \quad \star \Cup^{(\check{i}',\check{j}')(\mathsf{null},w,\mathsf{null})} \star \Cup^{(\check{i}_2,\check{j}_2)(w,m,\mathsf{null})} \end{array} \right\rceil \\ * (\lceil w \mapsto \check{j}_1,m,\check{i}',\check{i}_2 \rceil * \mathtt{n} \Rrightarrow - * \mathtt{m} \Rrightarrow w) \end{array} \right\}$
      $\{\ \lceil w \mapsto \check{j}_1,m,\check{i}',\check{i}_2 \rceil * \mathtt{n} \Rrightarrow - * \mathtt{m} \Rrightarrow w\ \}$
      n := [m.up]
      $\{\ \lceil w \mapsto \check{j}_1,m,\check{i}',\check{i}_2 \rceil * \mathtt{n} \Rrightarrow m * \mathtt{m} \Rrightarrow w\ \}$
    $\left\{ \begin{array}{l} \left\lceil \begin{array}{l} \Cap^{(i,j)(\check{l},\check{u},\check{r})} \star m \mapsto \check{l},\check{u},\check{i}_1,\check{r} \star \Cup^{(\check{i}_1,\check{j}_1)(\mathsf{null},m,w)} \\ \quad \star \Cup^{(\check{i}',\check{j}')(\mathsf{null},w,\mathsf{null})} \star \Cup^{(\check{i}_2,\check{j}_2)(w,m,\mathsf{null})} \end{array} \right\rceil \\ * (\lceil w \mapsto \check{j}_1,m,\check{i}',\check{i}_2 \rceil * \mathtt{n} \Rrightarrow m * \mathtt{m} \Rrightarrow w) \end{array} \right\}$
    $\left\{ \begin{array}{l} \left\lceil \begin{array}{l} \Cap^{(i,j)(\check{l},\check{u},\check{r})} \star m \mapsto \check{l},\check{u},\check{i}_1,\check{r} \star w \mapsto \check{j}_1,m,\check{i}',\check{i}_2 \\ \quad \star \Cup^{(\check{i}_1,\check{j}_1)(\mathsf{null},m,w)} \star \Cup^{(\check{i}',\check{j}')(\mathsf{null},w,\mathsf{null})} \star \Cup^{(\check{i}_2,\check{j}_2)(w,m,\mathsf{null})} \end{array} \right\rceil \\ * \mathtt{n} \Rrightarrow m * \mathtt{m} \Rrightarrow w \end{array} \right\}$
    $\{\ \lceil \Cap^{(i,j)(\check{l},\check{u},\check{r})} \star \langle\!\langle m[y_1 \otimes w[z] \otimes y_2] \rangle\!\rangle_{\eta}^{(i,j)(\check{l},\check{u},\check{r})} \rceil * \mathtt{n} \Rrightarrow m * \mathtt{m} \Rrightarrow w\ \}$
 }
$\{\ \llbracket \alpha{\leftarrow}m[\beta \otimes w[\delta] \otimes \gamma] * \mathtt{n} \Rrightarrow m * \mathtt{m} \Rrightarrow w \rrbracket_{\tau_2}\ \}$
```

Figure 6.18: Proof outline for `getUp` implementation in τ_2 (success case).

```
$\left\{\ \llbracket \lceil w[\beta] \rceil * \mathtt{n} \Rightarrow - * \mathtt{m} \Rightarrow w \rrbracket_{\tau_2}\ \right\}$
 proc n := getUp(m){
    $\left\{\ \exists i, j.\ \lceil \Cap^{(i,j)(\mathsf{null},\mathsf{null},\mathsf{null})} \star \langle\!\langle w[y] \rangle\!\rangle_{\eta}^{(i,j)(\mathsf{null},\mathsf{null},\mathsf{null})} \rceil * \mathtt{n} \Rightarrow - * \mathtt{m} \Rightarrow w\ \right\}$
    $\left\{\begin{array}{l} \exists i, j.\ \lceil \Cap^{(i,j)(\mathsf{null},\mathsf{null},\mathsf{null})} \star w \mapsto \mathsf{null},\mathsf{null},\check{i}',\mathsf{null} \star \Cup_y^{(\check{i}',\check{j}')(\mathsf{null},w,\mathsf{null})} \rceil \\ * \mathtt{n} \Rightarrow - * \mathtt{m} \Rightarrow w \end{array}\right\}$
    $\left\{\begin{array}{l} \exists i, j.\ \lceil \Cap^{(i,j)(\mathsf{null},\mathsf{null},\mathsf{null})} \star \Cup_y^{(\check{i}',\check{j}')(\mathsf{null},w,\mathsf{null})} \rceil \\ * (\lceil w \mapsto \mathsf{null},\mathsf{null},\check{i}',\mathsf{null} \rceil * \mathtt{n} \Rightarrow - * \mathtt{m} \Rightarrow w) \end{array}\right\}$
       $\left\{\ \lceil w \mapsto \mathsf{null},\mathsf{null},\check{i}',\mathsf{null} \rceil * \mathtt{n} \Rightarrow - * \mathtt{m} \Rightarrow w\ \right\}$
       n := [m.up]
       $\left\{\ \lceil w \mapsto \mathsf{null},\mathsf{null},\check{i}',\mathsf{null} \rceil * \mathtt{n} \Rightarrow \mathsf{null} * \mathtt{m} \Rightarrow w\ \right\}$
    $\left\{\begin{array}{l} \exists i, j.\ \lceil \Cap^{(i,j)(\mathsf{null},\mathsf{null},\mathsf{null})} \star \Cup_y^{(\check{i}',\check{j}')(\mathsf{null},w,\mathsf{null})} \rceil \\ * (\lceil w \mapsto \mathsf{null},\mathsf{null},\check{i}',\mathsf{null} \rceil * \mathtt{n} \Rightarrow \mathsf{null} * \mathtt{m} \Rightarrow w) \end{array}\right\}$
    $\left\{\begin{array}{l} \exists i, j.\ \lceil \Cap^{(i,j)(\mathsf{null},\mathsf{null},\mathsf{null})} \star w \mapsto \mathsf{null},\mathsf{null},\check{i}',\mathsf{null} \star \Cup_y^{(\check{i}',\check{j}')(\mathsf{null},w,\mathsf{null})} \rceil \\ * \mathtt{n} \Rightarrow \mathsf{null} * \mathtt{m} \Rightarrow w \end{array}\right\}$
    $\left\{\ \exists i, j.\ \lceil \Cap^{(i,j)(\mathsf{null},\mathsf{null},\mathsf{null})} \star \langle\!\langle w[y] \rangle\!\rangle_{\eta}^{(i,j)(\mathsf{null},\mathsf{null},\mathsf{null})} \rceil * \mathtt{n} \Rightarrow \mathsf{null} * \mathtt{m} \Rightarrow w\ \right\}$
 }
$\left\{\ \llbracket \lceil w[\beta] \rceil * \mathtt{n} \Rightarrow \mathsf{null} * \mathtt{m} \Rightarrow w \rrbracket_{\tau_2}\ \right\}$
```

Figure 6.19: Proof outline for `getUp` implementation in τ_2 (null case).

Axiom Correctness: `deleteTree`

Recall the specification of the `deleteTree` command from Figure 5.2.

$$\left\{\ \alpha{\leftarrow}w[\mathsf{tree}(\mathit{ct})] * \sigma \land \mathcal{E}[\![E]\!]\sigma = w\ \right\}$$
$$\texttt{deleteTree}(\mathit{E})$$
$$\left\{\ \alpha{\leftarrow}\varnothing * \sigma\ \right\}$$

To prove that this specification holds under our translation, suppose that $e(\alpha) = x$ for some $x \in \mathrm{X}$. We can also assume that $x \in \mathit{dom}(\eta)$, otherwise the translated precondition is equivalent to false, and that $\eta(x) = (i, j)(l, u, r)$ for some choice of i, j, l, u and r. The predicate $\mathsf{tree}(\mathit{ct})$ tells us that the tree context ct has no context holes, so we let $e(\mathit{ct}) = t$ (recall that we use t to denote a tree context with no holes). In Figure 6.20 we give a proof outline showing that the implementation of `deleteTree` (from Figure 6.15) satisfies the translation of its specification.

The proof assumes that the helper function `disposeForest` can be specified as follows:

$$\left\{\ \lceil \langle\!\langle t \rangle\!\rangle_{\eta}^{(n,-)(-,-,\mathsf{null})} \rceil * \mathtt{n} \Rrightarrow n\ \right\}$$
$$\texttt{disposeForest}(\mathtt{n})$$
$$\left\{\ \mathtt{n} \Rrightarrow -\ \right\}$$

It is relatively simple to check that this specification holds, but the real point of interest in this example is the program's interaction with the upper crust $\Cap^{(i,j)(\check{l},\check{u},\check{r})}$. It is not enough for the `deleteTree` program just to delete the subtree at w. In order to preserve the structure of the tree the program also needs to update those pointers that were referencing this subtree. This means that the left sibling pointer of the node to the right of w needs to be updated, if the node exists, to point the the left sibling of w. Similarly, the right sibling of the node to the left of w needs to be updated, if the node exists, to point to the right sibling of w. If w has no left sibling then the first child pointer of the parent of w needs to be updated, if it exists, to point to the right sibling of w. Notice that all of these updates are occurring in the concrete address interface corresponding to abstract address x. In particular this means that these updates are occurring in partial heaps cells $\check{l}$, $\check{u}$ and $\check{r}$. It is important that the program check that these nodes exist before attempting to update their contents. Implicit in our reasoning is also the requirement that these partial heap cells do not change whilst the `deleteTree` program is running. That is, the partial heap cell $\check{l}$ read at the beginning of the program must be the same cell that is updated at the end of the program, and similarly for the other partial heap cells. This *stability* requirement is trivially satisfied since we are reasoning about sequential programs, so it is not possible for these partial heap cells to be modified whilst the `deleteTree` program is running. However, ensuring that such stability requirements hold in a concurrent setting would be significantly more taxing.

Axiom Correctness: `getLast`

Recall the specification of the `getLast` command from Figure 5.1.

$$\left\{\ \alpha{\leftarrow}w[\beta \otimes m[\gamma]] * \mathtt{n} \Rightarrow n * \sigma \wedge \mathcal{E}[\![E]\!]\sigma[\mathtt{n} \mapsto n] = w\ \right\}$$
$$\mathtt{n} := \mathtt{getLast}(E)$$
$$\left\{\ \alpha{\leftarrow}w[\beta \otimes m[\gamma]] * \mathtt{n} \Rightarrow m * \sigma\ \right\}$$

$$\left\{\ \alpha{\leftarrow}w[\varnothing] * \mathtt{n} \Rightarrow n * \sigma \wedge \mathcal{E}[\![E]\!]\sigma[\mathtt{n} \mapsto n] = w\ \right\}$$
$$\mathtt{n} := \mathtt{getLast}(E)$$
$$\left\{\ \alpha{\leftarrow}w[\varnothing] * \mathtt{n} \Rightarrow \mathsf{null} * \sigma\ \right\}$$

To prove that the first specification holds under our translation, suppose that $e(\alpha) = x$, $e(\beta) = y$ and $e(\gamma) = z$. We can also assume that $\{x, y, z\} \subseteq \mathit{dom}(\eta)$, otherwise the precondition is equivalent to false, and that $\eta(x) = (i, j)(l, u, r)$, $\eta(y) = (i', j')(l', u', r')$, $\eta(z) = (i'', j'')(l'', u'', r'')$ for some choice of these interfaces. In Figure 6.21 we give a proof outline showing that the implementation of `getLast` (from Figure 6.15) satisfies the translation of its first specification.

To prove that the second translation holds under our translation, suppose that $e(\alpha) = x$ for some $x \in \mathrm{X}$. We can also assume that $x \in \mathit{dom}(\eta)$, otherwise the precondition is equivalent to false, and that and that $\eta(x) = (i, j)(l, u, r)$ for some choice of this interface. In Figure 6.22 we give a proof outline showing that the implementation of `getLast` (from Figure 6.15) satisfies the translation of its second specification.

The proof of the first specification is the more complex case and requires interaction with the lower crust $\Cup^{(\check{i'},\check{j'})(\mathsf{null},w,m)}$. The first point of interest occurs at the line $\mathtt{n} := [\mathtt{m.down}]$ where we read the down pointer of node w. This down pointer is equal $\check{i'}$ which points into the lower crust. This will either be a pointer to some partial heap cell, or, if the lower crust is empty, it will be a pointer to m. In either case, we know that the subsequent test $\mathtt{n} \neq \mathsf{null}$ will certainly be true, so the program definitely enters the if branch. The code inside the if branch traverses a null terminated list to find the last node in that list. Thus, the program will step

```
$\{\ [\![\, \alpha {\leftarrow} w[\mathsf{tree}(ct)] * \mathtt{n} \Rrightarrow w \,]\!]_{\tau_2}\ \}$
proc deleteTree(n){
  $\{\ \lceil \Cap^{(i,j)(\check{l},\breve{u},\check{r})} \star \langle\!\langle w[t] \rangle\!\rangle_{\eta}^{(i,j)(\check{l},\breve{u},\check{r})} \rceil * \mathtt{n} \Rrightarrow w\ \}$
  local x, y, z, w in
    $\{\ \lceil \Cap^{(i,j)(\check{l},\breve{u},\check{r})} \star \langle\!\langle w[t] \rangle\!\rangle_{\eta}^{(i,j)(\check{l},\breve{u},\check{r})} \rceil * \mathtt{n} \Rrightarrow w * \mathtt{x} \Rrightarrow - * \mathtt{y} \Rrightarrow - * \mathtt{z} \Rrightarrow - * \mathtt{w} \Rrightarrow -\ \}$
    $\left\{ \begin{array}{l} \exists d,e.\, \lceil \Cap^{(i,j)(\check{l},\breve{u},\check{r})} \star w \mapsto \check{l},\breve{u},d,\check{r} \star \langle\!\langle t \rangle\!\rangle_{\eta}^{(d,e)(\mathsf{null},w,\mathsf{null})} \rceil \\ * \mathtt{n} \Rrightarrow w * \mathtt{x} \Rrightarrow - * \mathtt{y} \Rrightarrow - * \mathtt{z} \Rrightarrow - * \mathtt{w} \Rrightarrow - \end{array} \right\}$
    x := [n.right] ;  y := [n.left] ;  z := [n.up] ;  w := [n.down] ;
    $\left\{ \begin{array}{l} \exists d,e.\, \lceil \Cap^{(i,j)(\check{l},\breve{u},\check{r})} \star w \mapsto \check{l},\breve{u},d,\check{r} \star \langle\!\langle t \rangle\!\rangle_{\eta}^{(d,e)(\mathsf{null},w,\mathsf{null})} \rceil \\ * \mathtt{n} \Rrightarrow w * \mathtt{x} \Rrightarrow \check{r} * \mathtt{y} \Rrightarrow \check{l} * \mathtt{z} \Rrightarrow \breve{u} * \mathtt{w} \Rrightarrow d \end{array} \right\}$
    call disposeForest(w) ;
    $\{\ \lceil \Cap^{(i,j)(\check{l},\breve{u},\check{r})} \star w \mapsto \check{l},\breve{u},-,\check{r} \rceil * \mathtt{n} \Rrightarrow w * \mathtt{x} \Rrightarrow \check{r} * \mathtt{y} \Rrightarrow \check{l} * \mathtt{z} \Rrightarrow \breve{u} * \mathtt{w} \Rrightarrow -\ \}$
    disposeNode(n) ;
    $\{\ \lceil \Cap^{(i,j)(\check{l},\breve{u},\check{r})} \rceil * \mathtt{n} \Rrightarrow w * \mathtt{x} \Rrightarrow \check{r} * \mathtt{y} \Rrightarrow \check{l} * \mathtt{z} \Rrightarrow \breve{u} * \mathtt{w} \Rrightarrow -\ \}$
    $\left\{ \begin{array}{l} \left\lceil \begin{array}{l} (\check{l} \mapsto \check{-},\check{-},\check{-},i \vee (\check{l} = \mathsf{null} \wedge (\breve{u} \mapsto \check{-},\check{-},i,\check{-} \vee \breve{u} \doteq \mathsf{null}))) \\ \star (\check{r} \mapsto j,\check{-},\check{-},\check{-} \vee \check{r} \doteq \mathsf{null}) \end{array} \right\rceil \\ * \mathtt{n} \Rrightarrow w * \mathtt{x} \Rrightarrow \check{r} * \mathtt{y} \Rrightarrow \check{l} * \mathtt{z} \Rrightarrow \breve{u} * \mathtt{w} \Rrightarrow - \end{array} \right\}$
    if x $\neq$ null then
      [x.left] := y ;
    $\left\{ \begin{array}{l} \left\lceil \begin{array}{l} (\check{l} \mapsto \check{-},\check{-},\check{-},i \vee (\check{l} = \mathsf{null} \wedge (\breve{u} \mapsto \check{-},\check{-},i,\check{-} \vee \breve{u} \doteq \mathsf{null}))) \\ \star (\check{r} \mapsto \check{l},\check{-},\check{-},\check{-} \vee \check{r} \doteq \mathsf{null}) \end{array} \right\rceil \\ * \mathtt{n} \Rrightarrow w * \mathtt{x} \Rrightarrow \check{r} * \mathtt{y} \Rrightarrow \check{l} * \mathtt{z} \Rrightarrow \breve{u} * \mathtt{w} \Rrightarrow - \end{array} \right\}$
    if y $\neq$ null then
      [y.right] := x
    else
      if z $\neq$ null then
        [z.down] := x
    $\left\{ \begin{array}{l} \left\lceil \begin{array}{l} (\check{l} \mapsto \check{-},\check{-},\check{-},\check{r} \vee (\check{l} = \mathsf{null} \wedge (\breve{u} \mapsto \check{-},\check{-},\check{r},\check{-} \vee \breve{u} \doteq \mathsf{null}))) \\ \star (\check{r} \mapsto \check{l},\check{-},\check{-},\check{-} \vee \check{r} \doteq \mathsf{null}) \end{array} \right\rceil \\ * \mathtt{n} \Rrightarrow w * \mathtt{x} \Rrightarrow \check{r} * \mathtt{y} \Rrightarrow \check{l} * \mathtt{z} \Rrightarrow \breve{u} * \mathtt{w} \Rrightarrow - \end{array} \right\}$
    $\{\ \lceil \Cap^{(\check{r},\check{l})(\check{l},\breve{u},\check{r})} \rceil * \mathtt{n} \Rrightarrow w * \mathtt{x} \Rrightarrow \check{r} * \mathtt{y} \Rrightarrow \check{l} * \mathtt{z} \Rrightarrow \breve{u} * \mathtt{w} \Rrightarrow -\ \}$
    $\{\ \lceil \Cap^{(i,j)(\check{l},\breve{u},\check{r})} \star \langle\!\langle \varnothing \rangle\!\rangle_{\eta}^{(i,j)(\check{l},\breve{u},\check{r})} \rceil * \mathtt{n} \Rrightarrow w * \mathtt{x} \Rrightarrow \check{r} * \mathtt{y} \Rrightarrow \check{l} * \mathtt{z} \Rrightarrow \breve{u} * \mathtt{w} \Rrightarrow -\ \}$
  $\{\ \lceil \Cap^{(i,j)(\check{l},\breve{u},\check{r})} \star \langle\!\langle \varnothing \rangle\!\rangle_{\eta}^{(i,j)(\check{l},\breve{u},\check{r})} \rceil * \mathtt{n} \Rrightarrow w\ \}$
}
$\{\ [\![\, \alpha {\leftarrow} \varnothing * \mathtt{n} \Rrightarrow w \,]\!]_{\tau_2}\ \}$
```

Figure 6.20: Proof outline for `deleteTree` implementation in τ_2.

though the lower crust, not making any modifications to it, and end up setting `n` to the node m who's right pointer is null.

Axiom Correctness: `appendChild`

Recall the specification of the `appendChild` command from Figure 5.2.

$$\left\{\ \alpha{\leftarrow}n[\gamma] * \beta{\leftarrow}m[\text{tree}(ct)] * \sigma \wedge \mathcal{E}[\![E]\!]\sigma = n \wedge \mathcal{E}[\![E']\!]\sigma = m\ \right\}$$
$$\texttt{appendChild}(E, E')$$
$$\left\{\ \alpha{\leftarrow}n[\gamma \otimes m[\text{tree}(ct)]] * \beta{\leftarrow}\varnothing * \sigma\ \right\}$$

To prove that this specification holds under our translation, suppose that $e(\alpha) = x$, $e(\beta) = y$ and $e(\gamma) = z$ for some $x, y, z \in \mathrm{X}$. We can also assume that $\{x, y, z\} \subseteq dom(\eta)$, otherwise the translated precondition is equivalent to false, and that $\eta(x) = (i, j)(l, u, r)$, $\eta(y) = (i', j')(l', u', r')$, $\eta(z) = (i'', j'')(l'', u'', r'')$ for some choice of these interfaces. The predicate $\mathsf{tree}(ct)$ tells us that the tree context ct has no context holes, so we let $e(ct) = t$ (recall that we use t to denote a tree context with no holes). In Figures 6.23 and 6.24 we give the proof that the implementation of `appendChild` (from Figure 6.15) satisfies the translation of this specification.

The implementation of `appendChild` is the most complex implementation of a basic command in the translation τ_2. The proof of correctness for this implementation requires access to both the upper crust at address y and the lower crust at hole label z. Moreover, notice that the node m, and its subtree, are initially part of the tree segment at address β, but end up as part of the tree segment at address α. The fact that our translation takes tree segments to complete heaps allows for this resource transfer to occur at the concrete level. In fact, the changes to the concrete interfaces made by the program mean that the only way for the final heap segment to represent a tree segment is if this resource transfer has indeed taken place.

This concludes the proof of Theorem 6.16.

6.3.3 Module Translation $\tau_3 : \mathbb{T} \to \mathbb{H} + \mathbb{L}$

We now present a second locality-preserving translation τ_3 from the tree module $\mathbb{T}$ into the heap and list module $\mathbb{H} + \mathbb{L}$. This translation represents each tree node n as a block of two cells in the heap $n \mapsto p,i$, which contain pointers to the node's parent p and a list i that contains the node's children. This representation of the tree is illustrated in Figure 6.25.

```
$\{\ [\![\, \alpha{\leftarrow}w[\beta \otimes m[\gamma]] * \mathtt{n} \Rightarrow - * \mathtt{m} \Rightarrow w \,]\!]_{\tau_2}\ \}$
proc n := getLast(m){
  $\left\{\ \lceil \Cap^{(i,j)(\check{l},\check{u},\check{r})} \star \langle\!\langle w[y \otimes m[z]] \rangle\!\rangle_{\eta}^{(i,j)(\check{l},\check{u},\check{r})} \rceil * \mathtt{n} \Rightarrow - * \mathtt{m} \Rightarrow w\ \right\}$
  local x in
    $\left\{\ \lceil \Cap^{(i,j)(\check{l},\check{u},\check{r})} \star \langle\!\langle w[y \otimes m[z]] \rangle\!\rangle_{\eta}^{(i,j)(\check{l},\check{u},\check{r})} \rceil * \mathtt{n} \Rightarrow - * \mathtt{m} \Rightarrow w * \mathtt{x} \Rightarrow -\ \right\}$
    $\left\{ \begin{array}{l} \lceil \Cap^{(i,j)(\check{l},\check{u},\check{r})} \star w \mapsto \check{l},\check{u},\check{i}',\check{r} \star \Cup^{(\check{i}',\check{j}')(\mathsf{null},w,m)} \star m \mapsto \check{j}',w,\check{i}'',\mathsf{null} \star \Cup^{(\check{i}'',\check{j}''(\mathsf{null},m,\mathsf{null})} \rceil \\ * \mathtt{n} \Rightarrow - * \mathtt{m} \Rightarrow w * \mathtt{x} \Rightarrow - \end{array} \right\}$
    n := [m.down] ;
    $\left\{ \begin{array}{l} \lceil \Cap^{(i,j)(\check{l},\check{u},\check{r})} \star w \mapsto \check{l},\check{u},\check{i}',\check{r} \star \Cup^{(\check{i}',\check{j}')(\mathsf{null},w,m)} \star m \mapsto \check{j}',w,\check{i}'',\mathsf{null} \star \Cup^{(\check{i}'',\check{j}''(\mathsf{null},m,\mathsf{null})} \rceil \\ * \mathtt{n} \Rightarrow \check{i}' * \mathtt{m} \Rightarrow w * \mathtt{x} \Rightarrow - \end{array} \right\}$
    $\left\{ \begin{array}{l} \lceil \Cap^{(i,j)(\check{l},\check{u},\check{r})} \star w \mapsto \check{l},\check{u},\check{i}',\check{r} \star \mathsf{ls}(\check{i}', \check{j}', \mathsf{null}, w, m) \star m \mapsto \check{j}',w,\check{i}'',\mathsf{null} \star \Cup^{(\check{i}'',\check{j}''(\mathsf{null},m,\mathsf{null})} \rceil \\ * \mathtt{n} \Rightarrow \check{i}' * \mathtt{m} \Rightarrow w * \mathtt{x} \Rightarrow - \end{array} \right\}$
    if n $\neq$ null then
      $\left\{ \begin{array}{l} \lceil \Cap^{(i,j)(\check{l},\check{u},\check{r})} \star w \mapsto \check{l},\check{u},\check{i}',\check{r} \star \mathsf{ls}(\check{i}', \check{j}', \mathsf{null}, w, m) \star m \mapsto \check{j}',w,\check{i}'',\mathsf{null} \star \Cup^{(\check{i}'',\check{j}''(\mathsf{null},m,\mathsf{null})} \rceil \\ * \mathtt{n} \Rightarrow \check{i}' * \mathtt{m} \Rightarrow w * \mathtt{x} \Rightarrow - \end{array} \right\}$
      x := [n.right] ;
      $\left\{ \begin{array}{l} \exists k, k', k''. \left\lceil \begin{array}{l} \Cap^{(i,j)(\check{l},\check{u},\check{r})} \star w \mapsto \check{l},\check{u},\check{i}',\check{r} \star m \mapsto \check{j}',w,\check{i}'',\mathsf{null} \star \Cup^{(\check{i}'',\check{j}''(\mathsf{null},m,\mathsf{null})} \\ \star \mathsf{ls}(\check{i}', \check{k}, \mathsf{null}, w, \check{k}') \star \check{k}' \mapsto \check{k},w,-,\check{k}'' \star \mathsf{ls}(\check{k}'', \check{j}', \check{k}', w, m) \end{array} \right\rceil \\ * \mathtt{n} \Rightarrow \check{k}' * \mathtt{m} \Rightarrow w * \mathtt{x} \Rightarrow k'' \\ \vee \lceil \Cap^{(i,j)(\check{l},\check{u},\check{r})} \star w \mapsto \check{l},\check{u},\check{i}',\check{r} \star \mathsf{ls}(\check{i}', \check{j}', \mathsf{null}, w, m) \star m \mapsto \check{j}',w,\check{i}'',\mathsf{null} \rceil \\ * \mathtt{n} \Rightarrow m * \mathtt{m} \Rightarrow w * \mathtt{x} \Rightarrow \mathsf{null} \end{array} \right\}$
      while x $\neq$ null do
      $\left\{ \begin{array}{l} \exists k, k', k''. \left\lceil \begin{array}{l} \Cap^{(i,j)(\check{l},\check{u},\check{r})} \star w \mapsto \check{l},\check{u},\check{i}',\check{r} \star m \mapsto \check{j}',w,\check{i}'',\mathsf{null} \star \Cup^{(\check{i}'',\check{j}''(\mathsf{null},m,\mathsf{null})} \\ \star \mathsf{ls}(\check{i}', \check{k}, \mathsf{null}, w, \check{k}') \star \check{k}' \mapsto \check{k},w,-,\check{k}'' \star \mathsf{ls}(\check{k}'', \check{j}', \check{k}', w, m) \end{array} \right\rceil \\ * \mathtt{n} \Rightarrow \check{k}' * \mathtt{m} \Rightarrow w * \mathtt{x} \Rightarrow k'' \end{array} \right\}$
        n := x ;
        x := [n.right]
        $\left\{ \begin{array}{l} \exists k, k', k''. \left\lceil \begin{array}{l} \Cap^{(i,j)(\check{l},\check{u},\check{r})} \star w \mapsto \check{l},\check{u},\check{i}',\check{r} \star m \mapsto \check{j}',w,\check{i}'',\mathsf{null} \star \Cup^{(\check{i}'',\check{j}''(\mathsf{null},m,\mathsf{null})} \\ \star \mathsf{ls}(\check{i}', \check{k}, \mathsf{null}, w, \check{k}') \star \check{k}' \mapsto \check{k},w,-,\check{k}'' \star \mathsf{ls}(\check{k}'', \check{j}', \check{k}', w, m) \end{array} \right\rceil \\ * \mathtt{n} \Rightarrow \check{k}' * \mathtt{m} \Rightarrow w * \mathtt{x} \Rightarrow k'' \\ \vee \lceil \Cap^{(i,j)(\check{l},\check{u},\check{r})} \star w \mapsto \check{l},\check{u},\check{i}',\check{r} \star \mathsf{ls}(\check{i}', \check{j}', \mathsf{null}, w, m) \star m \mapsto \check{j}',w,\check{i}'',\mathsf{null} \rceil \\ * \mathtt{n} \Rightarrow m * \mathtt{m} \Rightarrow w * \mathtt{x} \Rightarrow \mathsf{null} \end{array} \right\}$
      $\left\{ \begin{array}{l} \lceil \Cap^{(i,j)(\check{l},\check{u},\check{r})} \star w \mapsto \check{l},\check{u},\check{i}',\check{r} \star \mathsf{ls}(\check{i}', \check{j}', \mathsf{null}, w, m) \star m \mapsto \check{j}',w,\check{i}'',\mathsf{null} \star \Cup^{(\check{i}'',\check{j}''(\mathsf{null},m,\mathsf{null})} \rceil \\ * \mathtt{n} \Rightarrow m * \mathtt{m} \Rightarrow w * \mathtt{x} \Rightarrow \mathsf{null} \end{array} \right\}$
    $\left\{\ \lceil \Cap^{(i,j)(\check{l},\check{u},\check{r})} \star \langle\!\langle w[y \otimes m[z]] \rangle\!\rangle_{\eta}^{(i,j)(\check{l},\check{u},\check{r})} \rceil * \mathtt{n} \Rightarrow m * \mathtt{m} \Rightarrow w * \mathtt{x} \Rightarrow \mathsf{null}\ \right\}$
  $\left\{\ \lceil \Cap^{(i,j)(\check{l},\check{u},\check{r})} \star \langle\!\langle w[y \otimes m[z]] \rangle\!\rangle_{\eta}^{(i,j)(\check{l},\check{u},\check{r})} \rceil * \mathtt{n} \Rightarrow m * \mathtt{m} \Rightarrow w\ \right\}$
}
$\{\ [\![\, \alpha{\leftarrow}w[\beta \otimes m[\gamma]] * \mathtt{n} \Rightarrow m * \mathtt{m} \Rightarrow w \,]\!]_{\tau_2}\ \}$
```

Figure 6.21: Proof outline for the `getLast` implementation in τ_2 (success case).

```
$\{ \; [\![ \alpha \leftarrow w[\varnothing] * \texttt{n} \Rightarrow - * \texttt{m} \Rightarrow w ]\!]_{\tau_2} \; \}$
proc n := getLast(m){
  $\left\{ \; \lceil \Cap^{(i,j)(\check{l},\check{u},\check{r})} \star \langle\!\langle w[\varnothing] \rangle\!\rangle_{\eta}^{(i,j)(\check{l},\check{u},\check{r})} \rceil * \texttt{n} \Rightarrow - * \texttt{m} \Rightarrow w \; \right\}$
  local x in
    $\left\{ \; \lceil \Cap^{(i,j)(\check{l},\check{u},\check{r})} \star \langle\!\langle w[\varnothing] \rangle\!\rangle_{\eta}^{(i,j)(\check{l},\check{u},\check{r})} \rceil * \texttt{n} \Rightarrow - * \texttt{m} \Rightarrow w * \texttt{x} \Rightarrow - \; \right\}$
    $\{ \; \lceil \Cap^{(i,j)(\check{l},\check{u},\check{r})} \star w \mapsto \check{l},\check{u},\mathsf{null},\check{r} \rceil * \texttt{n} \Rightarrow - * \texttt{m} \Rightarrow w * \texttt{x} \Rightarrow - \; \}$
    n := [m.down] ;
    $\{ \; \lceil \Cap^{(i,j)(\check{l},\check{u},\check{r})} \star w \mapsto \check{l},\check{u},\mathsf{null},\check{r} \rceil * \texttt{n} \Rightarrow \mathsf{null} * \texttt{m} \Rightarrow w * \texttt{x} \Rightarrow - \; \}$
    if n $\neq$ null then
      ⋮
    $\{ \; \lceil \Cap^{(i,j)(\check{l},\check{u},\check{r})} \star w \mapsto \check{l},\check{u},\mathsf{null},\check{r} \rceil * \texttt{n} \Rightarrow \mathsf{null} * \texttt{m} \Rightarrow w * \texttt{x} \Rightarrow - \; \}$
    $\left\{ \; \lceil \Cap^{(i,j)(\check{l},\check{u},\check{r})} \star \langle\!\langle w[\varnothing] \rangle\!\rangle_{\eta}^{(i,j)(\check{l},\check{u},\check{r})} \rceil * \texttt{n} \Rightarrow \mathsf{null} * \texttt{m} \Rightarrow w * \texttt{x} \Rightarrow - \; \right\}$
  $\left\{ \; \lceil \Cap^{(i,j)(\check{l},\check{u},\check{r})} \star \langle\!\langle w[\varnothing] \rangle\!\rangle_{\eta}^{(i,j)(\check{l},\check{u},\check{r})} \rceil * \texttt{n} \Rightarrow \mathsf{null} * \texttt{m} \Rightarrow w \; \right\}$
}
$\{ \; [\![ \alpha \leftarrow w[\varnothing] * \texttt{n} \Rightarrow \mathsf{null} * \texttt{m} \Rightarrow w ]\!]_{\tau_2} \; \}$
```

Figure 6.22: Proof outline for the `getLast` implementation in τ_2 (null case).

An interface consists of the address of the tree's parent node and the list of nodes at the root level of the tree. These interfaces are represented in Figure 6.25 by the arrows into and out of the trees root node.

Note that for the empty tree $\varnothing$, the list of nodes at the root of the tree must be the empty list ε. However, the implementation we are about to give assumes that every node in the tree, including root nodes, must have some parent (we will see that the `getLeft` and `getRight` command implementations first go to the parent and then use its child list to finds the appropriate sibling). We model this by introducing a 'dummy' node, called `top`, which acts as the parent node for the root nodes of our tree. The node `top` has no parent, but provides a constant reference to the list of root nodes of the tree. A program can only use the `top` node indirectly to access this node list. If a program looks up the parent of a root node, it will return null and not `top` (the implementation of `getUp` manages this behaviour).

As with our previous example we need to use the concept of partial heap cells to describe properties of shared portions of state. We also lift the concept of partial ownership to abstract lists, writing $\check{i} \mapsto [\check{l}]$ to be analogous to $\check{x} \mapsto \check{v}$.

Notation: We write $x \mapsto p,i$ to mean $x \mapsto p * x{+}1 \mapsto i$ and write $(x \doteq y)$ to mean

```
$\left\{\ \llbracket \alpha{\leftarrow}n[\gamma] \ast \beta{\leftarrow}m[\mathrm{tree}(ct)] \ast \mathtt{n} \Rrightarrow n \ast \mathtt{m} \Rrightarrow m \,\rrbracket_{\tau_2}\ \right\}$
proc appendChild(n, m){
  $\left\{\begin{array}{l} \lceil \Cap^{(i,j)(\check{l},\check{u},\check{r})} \star \langle\!\langle n[z] \rangle\!\rangle_\eta^{(i,j)(\check{l},\check{u},\check{r})} \star \Cap^{(i',j')(\check{l'},\check{u'},\check{r'})} \star \langle\!\langle m[t] \rangle\!\rangle_\eta^{(i',j')(\check{l'},\check{u'},\check{r'})} \rceil \\ \ast \mathtt{n} \Rrightarrow n \ast \mathtt{m} \Rrightarrow m \end{array}\right\}$
  local x, y, z in
  $\left\{\begin{array}{l} \lceil \Cap^{(i,j)(\check{l},\check{u},\check{r})} \star \langle\!\langle n[z] \rangle\!\rangle_\eta^{(i,j)(\check{l},\check{u},\check{r})} \star \Cap^{(i',j')(\check{l'},\check{u'},\check{r'})} \star \langle\!\langle m[t] \rangle\!\rangle_\eta^{(i',j')(\check{l'},\check{u'},\check{r'})} \rceil \\ \ast \mathtt{n} \Rrightarrow n \ast \mathtt{m} \Rrightarrow m \ast \mathtt{x} \Rrightarrow - \ast \mathtt{y} \Rrightarrow - \ast \mathtt{z} \Rrightarrow - \end{array}\right\}$
    $\left\{\begin{array}{l} \exists d,e.\, \lceil \Cap^{(i,j)(\check{l},\check{u},\check{r})} \star n \mapsto \check{l},\check{u},\check{i''},\check{r} \star \Cup_z^{(\check{i''},\check{j''})(\mathsf{null},n,\mathsf{null})} \rceil \\ \ast \lceil \Cap^{(i',j')(\check{l'},\check{u'},\check{r'})} \star m \mapsto \check{l'},\check{u'},d,\check{r'} \star \langle\!\langle t \rangle\!\rangle_\eta^{(d,e)(\mathsf{null},m,\mathsf{null})} \rceil \\ \ast \mathtt{n} \Rrightarrow n \ast \mathtt{m} \Rrightarrow m \ast \mathtt{x} \Rrightarrow - \ast \mathtt{y} \Rrightarrow - \ast \mathtt{z} \Rrightarrow - \end{array}\right\}$
    x := [m.right] ;   y := [m.left] ;   z := [m.up] ;
    $\left\{\begin{array}{l} \exists d,e.\, \lceil \Cap^{(i,j)(\check{l},\check{u},\check{r})} \star n \mapsto \check{l},\check{u},\check{i''},\check{r} \star \Cup_z^{(\check{i''},\check{j''})(\mathsf{null},n,\mathsf{null})} \rceil \\ \ast \lceil \Cap^{(i',j')(\check{l'},\check{u'},\check{r'})} \star m \mapsto \check{l'},\check{u'},d,\check{r'} \star \langle\!\langle t \rangle\!\rangle_\eta^{(d,e)(\mathsf{null},m,\mathsf{null})} \rceil \\ \ast \mathtt{n} \Rrightarrow n \ast \mathtt{m} \Rrightarrow m \ast \mathtt{x} \Rrightarrow \check{r'} \ast \mathtt{y} \Rrightarrow \check{l'} \ast \mathtt{z} \Rrightarrow \check{u'} \end{array}\right\}$
    $\left\{\begin{array}{l} \exists d,e.\, \lceil \Cap^{(i,j)(\check{l},\check{u},\check{r})} \star n \mapsto \check{l},\check{u},\check{i''},\check{r} \star \Cup_z^{(\check{i''},\check{j''})(\mathsf{null},n,\mathsf{null})} \rceil \\ \ast \left\lceil \begin{array}{l} (\check{l'} \mapsto \check{-},\check{-},\check{-},i' \vee (\check{l'} = \mathsf{null} \wedge (\check{u'} \mapsto \check{-},\check{-},i',\check{-} \vee \check{u'} \doteq \mathsf{null}))) \\ \star\, (\check{r'} \mapsto j',\check{-},\check{-},\check{-} \vee \check{r'} \doteq \mathsf{null}) \end{array} \right\rceil \\ \ast \lceil m \mapsto \check{l'},\check{u'},d,\check{r'} \star \langle\!\langle t \rangle\!\rangle_\eta^{(d,e)(\mathsf{null},m,\mathsf{null})} \rceil \\ \ast \mathtt{n} \Rrightarrow n \ast \mathtt{m} \Rrightarrow m \ast \mathtt{x} \Rrightarrow \check{r'} \ast \mathtt{y} \Rrightarrow \check{l'} \ast \mathtt{z} \Rrightarrow \check{u'} \end{array}\right\}$
    if x $\neq$ null then
      [x.left] := y
    $\left\{\begin{array}{l} \exists d,e.\, \lceil \Cap^{(i,j)(\check{l},\check{u},\check{r})} \star n \mapsto \check{l},\check{u},\check{i''},\check{r} \star \Cup_z^{(\check{i''},\check{j''})(\mathsf{null},n,\mathsf{null})} \rceil \\ \ast \left\lceil \begin{array}{l} (\check{l'} \mapsto \check{-},\check{-},\check{-},i' \vee (\check{l'} = \mathsf{null} \wedge (\check{u'} \mapsto \check{-},\check{-},i',\check{-} \vee \check{u'} \doteq \mathsf{null}))) \\ \star\, (\check{r'} \mapsto \check{l'},\check{-},\check{-},\check{-} \vee \check{r'} \doteq \mathsf{null}) \end{array} \right\rceil \\ \ast \lceil m \mapsto \check{l'},\check{u'},d,\check{r'} \star \langle\!\langle t \rangle\!\rangle_\eta^{(d,e)(\mathsf{null},m,\mathsf{null})} \rceil \\ \ast \mathtt{n} \Rrightarrow n \ast \mathtt{m} \Rrightarrow m \ast \mathtt{x} \Rrightarrow \check{r'} \ast \mathtt{y} \Rrightarrow \check{l'} \ast \mathtt{z} \Rrightarrow \check{u'} \end{array}\right\}$
    if y $\neq$ null then
      [y.right] := x
    else
      if z $\neq$ null then
        [z.down] := x
    $\left\{\begin{array}{l} \exists d,e.\, \lceil \Cap^{(i,j)(\check{l},\check{u},\check{r})} \star n \mapsto \check{l},\check{u},\check{i''},\check{r} \star \Cup_z^{(\check{i''},\check{j''})(\mathsf{null},n,\mathsf{null})} \rceil \\ \ast \left\lceil \begin{array}{l} (\check{l'} \mapsto \check{-},\check{-},\check{-},\check{r'} \vee (\check{l'} = \mathsf{null} \wedge (\check{u'} \mapsto \check{-},\check{-},\check{r'},\check{-} \vee \check{u'} \doteq \mathsf{null}))) \\ \star\, (\check{r'} \mapsto \check{l'},\check{-},\check{-},\check{-} \vee \check{r'} \doteq \mathsf{null}) \end{array} \right\rceil \\ \ast \lceil m \mapsto \check{l'},\check{u'},d,\check{r'} \star \langle\!\langle t \rangle\!\rangle_\eta^{(d,e)(\mathsf{null},m,\mathsf{null})} \rceil \\ \ast \mathtt{n} \Rrightarrow n \ast \mathtt{m} \Rrightarrow m \ast \mathtt{x} \Rrightarrow \check{r'} \ast \mathtt{y} \Rrightarrow \check{l'} \ast \mathtt{z} \Rrightarrow \check{u'} \end{array}\right\}$
    $\left\{\begin{array}{l} \exists d,e.\, \lceil \Cap^{(i,j)(\check{l},\check{u},\check{r})} \star n \mapsto \check{l},\check{u},\check{i''},\check{r} \star \Cup_z^{(\check{i''},\check{j''})(\mathsf{null},n,\mathsf{null})} \rceil \\ \ast \lceil \Cap^{(i',j')(\check{l'},\check{u'},\check{r'})} \star \langle\!\langle \varnothing \rangle\!\rangle_\eta^{(i',j')(\check{l'},\check{u'},\check{r'})} \rceil \ast \lceil m \mapsto \check{l'},\check{u'},d,\check{r'} \star \langle\!\langle t \rangle\!\rangle_\eta^{(d,e)(\mathsf{null},m,\mathsf{null})} \rceil \\ \ast \mathtt{n} \Rrightarrow n \ast \mathtt{m} \Rrightarrow m \ast \mathtt{x} \Rrightarrow \check{r'} \ast \mathtt{y} \Rrightarrow \check{l'} \ast \mathtt{z} \Rrightarrow \check{u'} \end{array}\right\}$
                                   ⋮
```

Figure 6.23: Proof outline for `appendChild` implementation in τ_2.

```
                    ⋮
  $\left\{\begin{array}{l} \exists d,e.\ \lceil \Cap^{(i,j)(\check{l},\check{u},\check{r})} \star n \mapsto \check{l},\check{u},\check{i}'',\check{r} \star \Cup_z^{(\check{i}'',\check{j}'')(\mathsf{null},n,\mathsf{null})}\rceil \\ * \lceil \Cap^{(i',j')(\check{l}',\check{u}',\check{r}')} \star \langle\!\langle\varnothing\rangle\!\rangle_\eta^{(i',j')(\check{l}',\check{u}',\check{r}')}\rceil * \lceil m \mapsto \check{l}',\check{u}',d,\check{r}' \star \langle\!\langle t\rangle\!\rangle_\eta^{(d,e)(\mathsf{null},m,\mathsf{null})}\rceil \\ * \mathtt{n} \Rrightarrow n * \mathtt{m} \Rrightarrow m * \mathtt{x} \Rrightarrow \check{r}' * \mathtt{y} \Rrightarrow \check{l}' * \mathtt{z} \Rrightarrow \check{u}' \end{array}\right\}$
  $\mathtt{y} := [\mathtt{n.down}]$
  $\left\{\begin{array}{l} \exists d,e.\ \lceil \Cap^{(i,j)(\check{l},\check{u},\check{r})} \star n \mapsto \check{l},\check{u},\check{i}'',\check{r} \star \Cup_z^{(\check{i}'',\check{j}'')(\mathsf{null},n,\mathsf{null})}\rceil \\ * \lceil \Cap^{(i',j')(\check{l}',\check{u}',\check{r}')} \star \langle\!\langle\varnothing\rangle\!\rangle_\eta^{(i',j')(\check{l}',\check{u}',\check{r}')}\rceil * \lceil m \mapsto \check{l}',\check{u}',d,\check{r}' \star \langle\!\langle t\rangle\!\rangle_\eta^{(d,e)(\mathsf{null},m,\mathsf{null})}\rceil \\ * \mathtt{n} \Rrightarrow n * \mathtt{m} \Rrightarrow m * \mathtt{x} \Rrightarrow \check{r}' * \mathtt{y} \Rrightarrow \check{i}'' * \mathtt{z} \Rrightarrow \check{u}' \end{array}\right\}$
  $\left\{\begin{array}{l} \exists d,e.\ \lceil \Cap^{(i,j)(\check{l},\check{u},\check{r})} \star n \mapsto \check{l},\check{u},\check{i}'',\check{r} \star \mathsf{ls}(\check{i}'',\check{j}'',\mathsf{null},n,\mathsf{null})\rceil \\ * \lceil \Cap^{(i',j')(\check{l}',\check{u}',\check{r}')} \star \langle\!\langle\varnothing\rangle\!\rangle_\eta^{(i',j')(\check{l}',\check{u}',\check{r}')}\rceil * \lceil m \mapsto \check{l}',\check{u}',d,\check{r}' \star \langle\!\langle t\rangle\!\rangle_\eta^{(d,e)(\mathsf{null},m,\mathsf{null})}\rceil \\ * \mathtt{n} \Rrightarrow n * \mathtt{m} \Rrightarrow m * \mathtt{x} \Rrightarrow \check{r}' * \mathtt{y} \Rrightarrow \check{i}'' * \mathtt{z} \Rrightarrow \check{u}' \end{array}\right\}$
  if $\mathtt{y} = \mathsf{null}$ then
    $[\mathtt{n.down}] := \mathtt{m}$
    $\left\{\begin{array}{l} \exists d,e.\ \lceil \Cap^{(i,j)(\check{l},\check{u},\check{r})} \star n \mapsto \check{l},\check{u},\check{i}'',\check{r} \star \mathsf{ls}(\check{i}'',\check{j}'',\mathsf{null},n,m)\rceil \\ * \lceil \Cap^{(i',j')(\check{l}',\check{u}',\check{r}')} \star \langle\!\langle\varnothing\rangle\!\rangle_\eta^{(i',j')(\check{l}',\check{u}',\check{r}')}\rceil * \lceil m \mapsto \check{l}',\check{u}',d,\check{r}' \star \langle\!\langle t\rangle\!\rangle_\eta^{(d,e)(\mathsf{null},m,\mathsf{null})}\rceil \\ * \mathtt{n} \Rrightarrow n * \mathtt{m} \Rrightarrow m * \mathtt{x} \Rrightarrow - * \mathtt{y} \Rrightarrow \check{j}'' * \mathtt{z} \Rrightarrow \check{u}' \end{array}\right\}$
  else
    $\mathtt{x} := [\mathtt{y.right}]$ ;
    while $\mathtt{x} \neq \mathsf{null}$ do
      $\mathtt{y} := \mathtt{x}$ ;   $\mathtt{x} := [\mathtt{y.right}]$
    $[\mathtt{y.right}] := \mathtt{m}$
    $\left\{\begin{array}{l} \exists d,e.\ \lceil \Cap^{(i,j)(\check{l},\check{u},\check{r})} \star n \mapsto \check{l},\check{u},\check{i}'',\check{r} \star \mathsf{ls}(\check{i}'',\check{j}'',\mathsf{null},n,m)\rceil \\ * \lceil \Cap^{(i',j')(\check{l}',\check{u}',\check{r}')} \star \langle\!\langle\varnothing\rangle\!\rangle_\eta^{(i',j')(\check{l}',\check{u}',\check{r}')}\rceil * \lceil m \mapsto \check{l}',\check{u}',d,\check{r}' \star \langle\!\langle t\rangle\!\rangle_\eta^{(d,e)(\mathsf{null},m,\mathsf{null})}\rceil \\ * \mathtt{n} \Rrightarrow n * \mathtt{m} \Rrightarrow m * \mathtt{x} \Rrightarrow - * \mathtt{y} \Rrightarrow \check{j}'' * \mathtt{z} \Rrightarrow \check{u}' \end{array}\right\}$
  $\left\{\begin{array}{l} \exists d,e.\ \lceil \Cap^{(i,j)(\check{l},\check{u},\check{r})} \star n \mapsto \check{l},\check{u},\check{i}'',\check{r} \star \Cup_z^{(\check{i}'',\check{j}'')(\mathsf{null},n,m)}\rceil \\ * \lceil \Cap^{(i',j')(\check{l}',\check{u}',\check{r}')} \star \langle\!\langle\varnothing\rangle\!\rangle_\eta^{(i',j')(\check{l}',\check{u}',\check{r}')}\rceil * \lceil m \mapsto \check{l}',\check{u}',d,\check{r}' \star \langle\!\langle t\rangle\!\rangle_\eta^{(d,e)(\mathsf{null},m,\mathsf{null})}\rceil \\ * \mathtt{n} \Rrightarrow n * \mathtt{m} \Rrightarrow m * \mathtt{x} \Rrightarrow - * \mathtt{y} \Rrightarrow \check{j}'' * \mathtt{z} \Rrightarrow \check{u}' \end{array}\right\}$
  $[\mathtt{m.left}] := \mathtt{y}$ ;   $[\mathtt{m.right}] := \mathsf{null}$ ;   $[\mathtt{m.up}] := \mathtt{n}$
  $\left\{\begin{array}{l} \exists d,e.\ \lceil \Cap^{(i,j)(\check{l},\check{u},\check{r})} \star n \mapsto \check{l},\check{u},\check{i}'',\check{r} \star \Cup_z^{(\check{i}'',\check{j}'')(\mathsf{null},n,m)}\rceil \\ * \lceil \Cap^{(i',j')(\check{l}',\check{u}',\check{r}')} \star \langle\!\langle\varnothing\rangle\!\rangle_\eta^{(i',j')(\check{l}',\check{u}',\check{r}')}\rceil * \lceil m \mapsto \check{j}'',n,d,\mathsf{null} \star \langle\!\langle t\rangle\!\rangle_\eta^{(d,e)(\mathsf{null},m,\mathsf{null})}\rceil \\ * \mathtt{n} \Rrightarrow n * \mathtt{m} \Rrightarrow m * \mathtt{x} \Rrightarrow - * \mathtt{y} \Rrightarrow \check{j}'' * \mathtt{z} \Rrightarrow \check{u}' \end{array}\right\}$
  $\left\{\begin{array}{l} \exists d,e. \left\lceil \begin{array}{l} \Cap^{(i,j)(\check{l},\check{u},\check{r})} \star n \mapsto \check{l},\check{u},\check{i}'',\check{r} \star \Cup_z^{(\check{i}'',\check{j}'')(\mathsf{null},n,m)} \\ \star\, m \mapsto \check{j}'',n,d,\mathsf{null} \star \langle\!\langle t\rangle\!\rangle_\eta^{(d,e)(\mathsf{null},m,\mathsf{null})} \end{array}\right\rceil \\ * \lceil \Cap^{(i',j')(\check{l}',\check{u}',\check{r}')} \star \langle\!\langle\varnothing\rangle\!\rangle_\eta^{(i',j')(\check{l}',\check{u}',\check{r}')}\rceil \\ * \mathtt{n} \Rrightarrow n * \mathtt{m} \Rrightarrow m * \mathtt{x} \Rrightarrow - * \mathtt{y} \Rrightarrow \check{j}'' * \mathtt{z} \Rrightarrow \check{u}' \end{array}\right\}$
 $\left\{\begin{array}{l} \lceil \Cap^{(i,j)(\check{l},\check{u},\check{r})} \star \langle\!\langle n[z \otimes m[t]]\rangle\!\rangle_\eta^{(i,j)(\check{l},\check{u},\check{r})} \star \Cap^{(i',j')(\check{l}',\check{u}',\check{r}')} \star \langle\!\langle\varnothing\rangle\!\rangle_\eta^{(i',j')(\check{l}',\check{u}',\check{r}')}\rceil \\ * \mathtt{n} \Rrightarrow n * \mathtt{m} \Rrightarrow m \end{array}\right\}$
}
$\{\ [\![\, \alpha \leftarrow n[\gamma \otimes m[\mathrm{tree}(ct)]] * \beta \leftarrow \varnothing * \mathtt{n} \Rrightarrow n * \mathtt{m} \Rrightarrow m \,]\!]_{\tau_2}\ \}$
```

Figure 6.24: Proof outline for `appendChild` implementation in τ_2 continued.

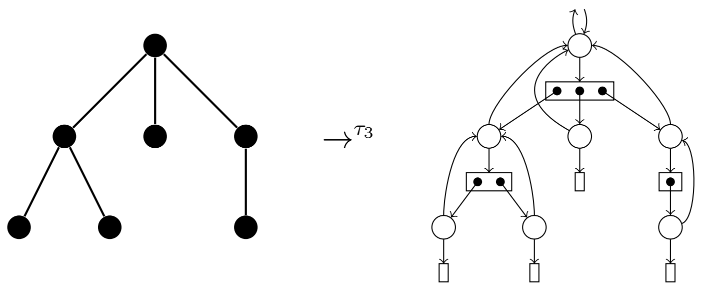

Figure 6.25: An abstract tree from $\mathbb{T}$ and its representation in $\mathbb{H}+\mathbb{L}$.

$\{\mathsf{emp}\} \wedge (x = y)$. We again drop module annotations when they can be inferred from context.

Definition 6.21 ($\tau_3 : \mathbb{T} \to \mathbb{H}+\mathbb{L}$)**.** The pre-locality preserving translation $\tau_3 : \mathbb{T} \to \mathbb{H}+\mathbb{L}$ is constructed as follows:

- ◇ an interface $I = (l, p) \in \mathcal{I}$ consists of a list of addresses l that describes the root level nodes of the tree and an address p that describes the parent node of the tree (possibly $\mathtt{top}$). Note that there are no addresses or hole labels in these interfaces, so $\mathsf{labs}(I) = \emptyset$ for all $I \in \mathcal{I}$
- ◇ the segment representation function $(\!|(\cdot)|\!)^{(\cdot)} : S_{\mathrm{T}} \times (\mathrm{X} \rightharpoonup_{\text{fin}} \mathcal{I}) \to S_{\mathrm{H}\times\mathbb{L}}$ is defined by induction on the structure of tree segments as:

$$\begin{array}{rcl}
\langle\!\langle \emptyset \rangle\!\rangle^{\eta} & \stackrel{\text{def}}{=} & \{\mathsf{emp} \times \mathsf{emp}\} \\
\langle\!\langle \{(x, ct)\} \rangle\!\rangle^{\eta} & \stackrel{\text{def}}{=} & \begin{cases} \exists l.\, \Cap^{(l,\check{\mathtt{top}})} +_{\mathbb{H}+\mathbb{L}} \langle\!\langle ct \rangle\!\rangle_{\eta}^{(l,\check{\mathtt{top}})} & \text{if } x = 0 \\ \Cap^{(l,\check{p})} +_{\mathbb{H}+\mathbb{L}} \langle\!\langle ct \rangle\!\rangle_{\eta}^{(l,\check{p})} \wedge \eta(x) = (l,p) & \text{otherwise} \end{cases} \\
\langle\!\langle st_1 \uplus st_2 \rangle\!\rangle^{\eta} & \stackrel{\text{def}}{=} & \langle\!\langle st_1 \rangle\!\rangle^{\eta} +_{\mathbb{H}+\mathbb{L}} \langle\!\langle st_2 \rangle\!\rangle^{\eta}
\end{array}$$

  where the upper crust formula $\Cap^{(l,\check{p})} \in S_{\mathrm{H}\times\mathbb{L}}$ is defined as,

$$\Cap^{(l,\check{p})} \stackrel{\text{def}}{=} \exists i, l_1, l_2.\, \{\lceil \check{p} \mapsto \check{-}, \check{i} \rceil \times \check{i} \Mapsto [\,\check{l_1} : l : \check{l_2}\,]\} + \sum_{v \in l_1 : l_2} \{\lceil \check{v} \mapsto \check{p}, \check{-} \rceil \times \mathsf{emp}\}$$

  the context representation function $\langle\!\langle (\cdot) \rangle\!\rangle_{(\cdot)}^{(\cdot)} : \mathcal{C}_{\mathrm{T}} \times \mathcal{I} \times (\mathrm{X} \rightharpoonup_{\text{fin}} \mathcal{I}) \to S_{\mathrm{H}\times\mathbb{L}}$ is defined by induction on the structure of multi-holed tree contexts as:

$$\begin{array}{rcl}
\langle\!\langle \varnothing \rangle\!\rangle_{\eta}^{(l,p)} & \stackrel{\text{def}}{=} & \{\mathsf{emp} \times \mathsf{emp}\} \wedge (l = \varepsilon) \\
\langle\!\langle x \rangle\!\rangle_{\eta}^{(l,p)} & \stackrel{\text{def}}{=} & \Cup^{(\check{l},p)} \wedge (l = \check{l}) \wedge (\eta(x) = (l,p)) \\
\langle\!\langle n[ct] \rangle\!\rangle_{\eta}^{(l,p)} & \stackrel{\text{def}}{=} & \exists i, l'.\, \{\lceil n \mapsto p, i \rceil \times i \Mapsto [\,l'\,]\} +_{\mathbb{H}+\mathbb{L}} \langle\!\langle ct \rangle\!\rangle_{\eta}^{(l',n)} \wedge (l = n) \\
\langle\!\langle ct_1 \otimes ct_2 \rangle\!\rangle_{\eta}^{(l,p)} & \stackrel{\text{def}}{=} & \exists l_1, l_2.\, \langle\!\langle ct_1 \rangle\!\rangle_{\eta}^{(l_1,p)} +_{\mathbb{H}+\mathbb{L}} \langle\!\langle ct_2 \rangle\!\rangle_{\eta}^{(l_2,p)} \wedge (l = l_1 : l_2);
\end{array}$$

and the lower crust formula $\Cup^{(\check{l},p)} \in S_{\mathrm{H}\times\mathbb{L}}$ is defined as,

$$\Cup^{\check{l},p} \overset{\text{def}}{=} \sum_{v\in l}\{\lceil \check{v} \mapsto p, \check{-}\rceil \times \mathsf{emp}\}$$

◇ the substitutive representation function is given by replacing each tree module command with a call to the correspondingly named procedure given in Figure 6.28, with,

$$\begin{aligned}
E.\texttt{parent} &\overset{\text{def}}{=} E \\
E.\texttt{children} &\overset{\text{def}}{=} E+1 \\
\texttt{n} := \texttt{newNode}() &\overset{\text{def}}{=} \texttt{n} := \texttt{alloc}(2) \\
\texttt{disposeNode}(E) &\overset{\text{def}}{=} \texttt{dispose}(E, 2).
\end{aligned}$$

Recall that elements of the set $\mathrm{S}_{\mathrm{H}\times\mathbb{L}}$ are of the form (sh, sls) where $sh \in \mathrm{S}_{\mathrm{H}}$ and $sls \in \mathrm{S}_{\mathbb{L}}$. The $+_{\mathbb{H}+\mathbb{L}}$ operation then is defined as:

$$(sh_1, sls_1) +_{\mathbb{H}+\mathbb{L}} (sh_2, sls_2) \overset{\text{def}}{=} (sh_1 +_{\mathbb{H}} sh_2, sls_1 +_{\mathbb{L}} sls_2)$$

The translation τ_3 is also a crust inclusive translation in the terminology of our previous work [26]. As before, this translation has a lot in common with our context based translation between the same modules. Again, the main difference is our treatment of the concrete interface, or crust.

The upper crust predicate $\Cap^{(l,\check{p})}$ describes the concrete state that corresponds to an abstract address x with $\eta(x) = (l, p)$. This is illustrated in Figure 6.26. The concrete address interface consists of a partial heap cell corresponding to the parent node $\check{p}$ of the root level of the tree, this may be the unique dummy node `top`. It also contains the partial list corresponding to the child list of $\check{p}$ and the weak partial heap cells for each node in this child list. Access to this list is required by several of our implementations, but in particular it is required by `newNodeAfter` which needs to insert a new node into this list. To be able to reason about inserting a value into a list we need to know that the value in question does not already occur in the list. The only way we can be sure this is the case in our reasoning is to know that the value we are trying to insert is a heap address, as are the addresses already in the list. By including the partial heap cells we can use the disjointness property of $\star$ to establish the the value we are inserting is not already in the list. Notice that the only pointers that we have full access to in the crust is the list of addresses l at the root of the tree. This means that a program run on this state can only modify

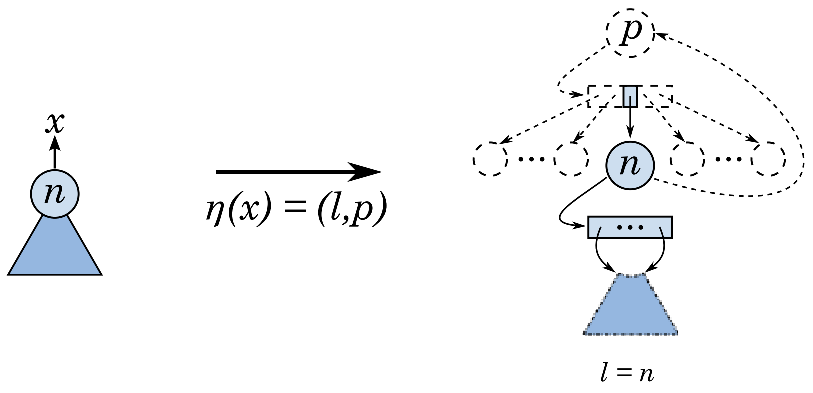

Figure 6.26: A translation in τ_3 which introduces some upper crust.

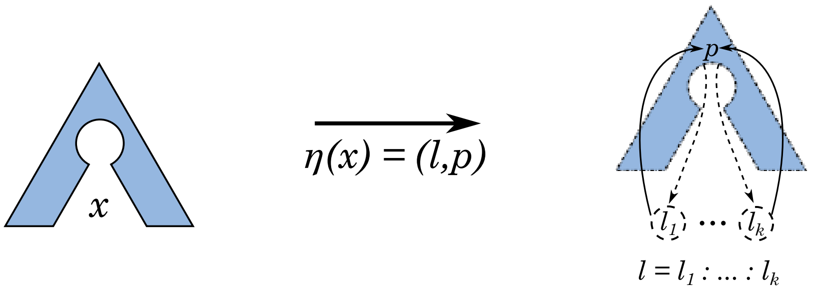

Figure 6.27: A translation in τ_3 which introduces some lower crust.

the crust by changing the values in this list. The program cannot make any other modifications to the surrounding state.

The translation fills in each context hole with a list of node addresses and extends the state with a lower crust. The lower crust predicate $\Cup^{\hat{l},p}$ describes the concrete state that corresponds to an abstract hole label x with $\eta(x) = (l, p)$. This is illustrated in Figure 6.27. The concrete hole interface consists of partial heap cells for each of the nodes that is referenced in the list l. Access to this list may be required by the implementation of `appendChild` which needs to insert a new node into such a list. As above, we can only reason about list insertion if we have access to the whole list and the heap cells stored in that list. Notice that the only pointers that we have full access to in the crust are the parent pointers to p. This means that a program run on this state can only modify the crust by changing the values of these pointers. The program cannot make any other modifications to the surrounding state.

In this translation we can again see that the upper and lower crusts for some label x consist of complimentary partial heaps cells/lists. When combined, we recover the complete heap cells and lists associated with the concrete interface. In order to prove the compression preservation property for this translation, we will need to show a crust inclusion result similar to that from our previous example.

```
proc n := getUp(m){
  n := [m.parent] ;
  if n = top then
    n := null
}

proc n := getLeft(m){
  local x, y in
    x := [m.parent] ;
    y := [x.children] ;
    n := y.getPrev(m)
}

proc n := getRight(m){
  local x, y in
    x := [m.parent] ;
    y := [x.children] ;
    n := y.getNext(m)
}

proc deleteTree(n){
  local x, y, z in
    x := [n.parent] ;
    y := [x.children] ;
    y.remove(n) ;
    y := [n.children] ;
    z := y.getHead() ;
    while z ≠ null do
      call deleteTree(z) ;
      z := y.getHead()
    deleteList(y) ;
    disposeNode(n)
}
```

```
proc n := getFirst(m){
  local x in
    x := [m.children] ;
    n := x.getHead()
}

proc n := getLast(m){
  local x in
    x := [m.children] ;
    n := x.getTail()
}

proc newNodeAfter(n){
  local x, y, z, w in
    x := [n.parent] ;
    z := [x.children] ;
    y := newNode() ;
    w := newList() ;
    [y.parent] := x ;
    [y.children] := w ;
    z.insert(n, y)
}

proc appendChild(n, m){
  local x, y in
    x := [m.parent] ;
    y := [x.children] ;
    y.remove(m) ;
    x := [n.children] ;
    y := x.getTail() ;
    x.insert(y, m)
}
```

Figure 6.28: Procedures for the heap and list-based implementation of the tree module.

Theorem 6.22 (Soundness of τ_3)**.** The pre-locality-preserving translation τ_3 is a locality-preserving translation.

Lemma 6.23 (Combination Preservation)**.** Segment combination is preserved by the segment representation function. That is, for all $st_1, st_2 \in S_{\mathbb{T}}$ and $\eta \in (\mathrm{X} \rightharpoonup_{\text{fin}} \mathcal{I})$,

$$\llparenthesis st_1 \uplus st_2 \rrparenthesis^{\eta} \quad = \quad \llparenthesis st_1 \rrparenthesis^{\eta} +_{\mathbb{H}+\mathbb{L}} \llparenthesis st_2 \rrparenthesis^{\eta}$$

Proof. This property follows from the definition of the segment representation function given in Definition 6.21. $\square$

In order to prove the revelation preservation property for the translation τ_3 we require the crust inclusion lemma. This lemma states that given a context composition $ct \bullet_x ct'$ we can extract the concrete interface $\Cap^{I}$ corresponding to label x from the translation of $ct \bullet_x ct'$ plus its upper crust. This result relies on the use of partial heap cells and lists to split the concrete interface corresponding to x into two pieces: one that is extracted as the upper crust of ct' and one that remains as the lower crust in the translation of ct.

Lemma 6.24 (Crust Inclusion)**.** For all $ct, ct' \in \mathrm{T}_{\mathrm{ID},\mathrm{X}}$, $I' \in \mathcal{I}$ and $\eta \in (\mathrm{X} \rightharpoonup_{\text{fin}} \mathcal{I})$, if $x \in fh_{\mathrm{T}}(ct)$ and $x \notin fh_{\mathrm{T}}(ct')$, then

$$\Cap^{I'} + \langle\!\langle ct \bullet_x ct' \rangle\!\rangle^{I'}_{\eta} \quad = \quad \exists I.\, \Cap^{I'} + \langle\!\langle ct \rangle\!\rangle^{I'}_{\eta[x \mapsto I]} + \Cap^{I} + \langle\!\langle ct' \rangle\!\rangle^{I}_{\eta}$$

Proof. Proceed by induction on the structure of ct.

$ct = \varnothing$ case:

$x \notin fh_{\mathrm{T}}(\varnothing)$ which contradicts our assumption that $x \in fh_{\mathrm{T}}(ct)$, so this case holds vacuously.

$ct = y$ case:

If $y \neq x$ then $x \notin fh_{\mathrm{T}}(y)$ which contradicts our assumption that $x \in fh_{\mathrm{T}}(ct)$, so this case holds vacuously. If $y = x$ then let $I' = (l', \check{p}')$ for some l' and p'. We can show the following:

$$
\begin{aligned}
\Cap^{I'} + \langle\!\langle ct \bullet_x ct' \rangle\!\rangle_{\eta}^{I'} &= \Cap_{z}^{I'} + \langle\!\langle x \bullet_x ct' \rangle\!\rangle_{\eta}^{I'} \\
&= \Cap^{I'} + \langle\!\langle ct' \rangle\!\rangle_{\eta}^{I'} \\
&= \Cap^{(l',\check{p'})} + \langle\!\langle ct' \rangle\!\rangle_{\eta}^{(l',\check{p'})} \\
&= \exists i, l_1, l_2.\, \{\lceil \check{p'} \mapsto \check{-}, \check{i} \rceil \times \check{i} \Mapsto [\, \check{l_1} : l' : \check{l_2} \,]\} + \langle\!\langle ct' \rangle\!\rangle_{\eta}^{(l',\check{p'})} \\
&\quad + \sum_{v \in l_1 : l_2} \{\lceil \check{v} \mapsto \check{p'}, \check{-} \rceil \times \mathsf{emp}\} \\
&= \exists i, l_1, l_2.\, \{\lceil \check{p'} \mapsto \check{-}, \check{i} \rceil \times \check{i} \Mapsto [\, \check{l_1} : \check{l'} : \check{l_2} \,]\} \\
&\quad + \{\lceil \check{p'} \mapsto \check{-}, \check{i} \rceil \times \check{i} \Mapsto [\, \check{l_1} : l' : \check{l_2} \,]\} + \langle\!\langle ct' \rangle\!\rangle_{\eta}^{(l',\check{p'})} \\
&\quad + \sum_{v \in l_1 : l_2} \{\lceil \check{v} \mapsto \check{p'}, \check{-} \rceil \times \mathsf{emp}\} + \sum_{v \in l_1 : l_2} \{\lceil \check{v} \mapsto \check{p'}, \check{-} \rceil \times \mathsf{emp}\} \\
&= \exists i, l_1, l_2, l, p.\, (l = \check{l'}) \wedge (p = \check{p'}) \wedge \{\lceil \check{p'} \mapsto \check{-}, \check{i} \rceil \times \check{i} \Mapsto [\, \check{l_1} : \check{l'} : \check{l_2} \,]\} \\
&\quad + \{\lceil \check{p} \mapsto \check{-}, \check{i} \rceil \times \check{i} \Mapsto [\, \check{l_1} : l : \check{l_2} \,]\} + \langle\!\langle ct' \rangle\!\rangle_{\eta}^{(l,\check{p})} \\
&\quad + \sum_{v \in l_1 : l_2} \{\lceil \check{v} \mapsto \check{p'}, \check{-} \rceil \times \mathsf{emp}\} + \sum_{v \in l_1 : l_2} \{\lceil \check{v} \mapsto \check{p}, \check{-} \rceil \times \mathsf{emp}\} \\
&= \exists l, p.\, \Cap^{(l',\check{p'})} + \langle\!\langle x \rangle\!\rangle_{\eta[x \mapsto (l,p)]}^{(l',\check{p'})} + \Cap^{(l,\check{p})} + \langle\!\langle ct' \rangle\!\rangle_{\eta}^{(l,\check{p})} \\
&= \exists I.\, \Cap^{I'} + \langle\!\langle ct \rangle\!\rangle_{\eta[x \mapsto I]}^{I'} + \Cap^{I} + \langle\!\langle ct' \rangle\!\rangle_{\eta}^{I}
\end{aligned}
$$

$ct = n[ct'']$ case:

There are two cases to consider. If $x \notin fh_{\mathrm{T}}(ct'')$ then $x \notin fh_{\mathrm{T}}(n[ct''])$ which contradicts our assumption that $x \in fh_{\mathrm{T}}(ct)$, so this case holds vacuously. If $x \in fh_{\mathrm{T}}(ct'')$ then by the induction hypothesis,

$$\Cap^{(l,\check{n})} + \langle\!\langle ct'' \bullet_x ct' \rangle\!\rangle_\eta^{(l,\check{n})} \quad = \quad \exists I.\, \Cap^{(l,\check{n})} + \langle\!\langle ct'' \rangle\!\rangle_{\eta[x\mapsto I]}^{(l,\check{n})} + \Cap^{I} + \langle\!\langle ct' \rangle\!\rangle_\eta^{I}$$

Let $I' = (l', \check{p}')$ for some l' and p'. We can then show the following:

$$\begin{array}{rcl}
\Cap^{I'} + \langle\!\langle ct \bullet_x ct' \rangle\!\rangle_\eta^{I'} & = & \Cap^{I'} + \langle\!\langle n[ct''] \bullet_x ct' \rangle\!\rangle_\eta^{I'} \\
& = & \Cap^{I'} + \langle\!\langle n[ct'' \bullet_x ct'] \rangle\!\rangle_\eta^{I'} \\
& = & \Cap^{(l',\check{p}')} + \langle\!\langle n[ct'' \bullet_x ct'] \rangle\!\rangle_\eta^{(l',\check{p}')} \\
& = & \Cap^{(l',\check{p}')} + \exists i, l.\, \{\lceil n \mapsto \check{p}', i \rceil \times i \Mapsto [\, l \,]\} \\
& & \quad + \langle\!\langle ct'' \bullet_x ct' \rangle\!\rangle_\eta^{(l,n)} \wedge (l' = n) \\
& = & \Cap^{(l',\check{p}')} + \exists i, l.\, \{\lceil n \mapsto \check{p}', i \rceil \times i \Mapsto [\, \check{l} \,]\} + \{\lceil \check{n} \mapsto \check{-}, \check{i} \rceil \times \check{i} \Mapsto [\, l \,]\} \\
& & \quad + \langle\!\langle ct'' \bullet_x ct' \rangle\!\rangle_\eta^{(l,\check{n})} \wedge (l' = n) \\
& = & \Cap^{(l',\check{p}')} + \exists i, l, l_1, l_2.\, \{\lceil n \mapsto \check{p}', i \rceil \times i \Mapsto [\, \check{l} \,]\} \wedge (l_1 = \varepsilon) \wedge (l_2 = \varepsilon) \\
& & \quad + \Cap^{(l,\check{n})} + \langle\!\langle ct'' \bullet_x ct' \rangle\!\rangle_\eta^{(l,\check{n})} \wedge (l' = n) \\
(IH) & = & \Cap^{(l',\check{p}')} + \exists i, l, l_1, l_2.\, \{\lceil n \mapsto \check{p}', i \rceil \times i \Mapsto [\, \check{l} \,]\} \wedge (l_1 = \varepsilon) \wedge (l_2 = \varepsilon) \\
& & \quad + \exists I.\, \Cap^{(l,\check{n})} + \langle\!\langle ct'' \rangle\!\rangle_{\eta[x\mapsto I]}^{(l,\check{n})} + \Cap^{I} + \langle\!\langle ct' \rangle\!\rangle_\eta^{I} \wedge (l' = n) \\
& = & \Cap^{(l',\check{p}')} + \exists i, l.\, \{\lceil n \mapsto \check{p}', i \rceil \times i \Mapsto [\, \check{l} \,]\} + \{\lceil \check{n} \mapsto \check{-}, \check{i} \rceil \times \check{i} \Mapsto [\, l \,]\} \\
& & \quad + \langle\!\langle ct'' \rangle\!\rangle_{\eta[x\mapsto I]}^{(l,\check{n})} + \Cap^{I} + \langle\!\langle ct' \rangle\!\rangle_\eta^{I} \wedge (l' = n) \\
& = & \Cap^{(l',\check{p}')} + \exists I.\, \langle\!\langle n[ct''] \rangle\!\rangle_{\eta[x\mapsto I]}^{(l',\check{p}')} + \Cap^{I} + \langle\!\langle ct' \rangle\!\rangle_\eta^{I} \\
& = & \exists I.\, \Cap^{I'} + \langle\!\langle ct \rangle\!\rangle_{\eta[x\mapsto I]}^{I'} + \Cap^{I} + \langle\!\langle ct' \rangle\!\rangle_\eta^{I}
\end{array}$$

$ct = ct_1 \otimes ct_2$ case:

There are four cases to consider. If $x \notin fh_{\mathrm{T}}(ct_1)$ and $x \notin fh_{\mathrm{T}}(ct_2)$ then $x \notin fh_{\mathrm{T}}(ct_1 \otimes ct_2)$ which contradicts our assumption that $x \in fh_{\mathrm{T}}(ct)$, so this case holds vacuously. If $x \in fh_{\mathrm{T}}(ct_1)$ and $x \in fh_{\mathrm{T}}(ct_2)$ then the tree context $ct_1 \otimes ct_2$ is not well formed and again this case holds vacuously. If $x \in fh_{\mathrm{T}}(ct_1)$ and $x \notin fh_{\mathrm{T}}(ct_2)$ then by the inductive hypothesis,

$$\Cap^{(l'_1,\check{p}')} + \langle\!\langle ct_1 \bullet_x ct' \rangle\!\rangle_\eta^{(l'_1,\check{p}')} \quad = \quad \exists I.\, \Cap^{(l'_1,\check{p}')} + \langle\!\langle ct_1 \rangle\!\rangle_{\eta[x\mapsto I]}^{(l'_1,\check{p}')} + \Cap^{I} + \langle\!\langle ct' \rangle\!\rangle_\eta^{I}$$

Let $I' = (l', \check{p}')$ for some l' and p'. We can then show the following:

$$
\begin{aligned}
\Cap^{I'} + \langle\!\langle ct \bullet_x ct' \rangle\!\rangle_\eta^{I'} &= \Cap^{I'} + \langle\!\langle (ct_1 \otimes ct_2) \bullet_x ct' \rangle\!\rangle_\eta^{I'} \\
&= \Cap^{I'} + \langle\!\langle (ct_1 \bullet_x ct') \otimes ct_2 \rangle\!\rangle_\eta^{I'} \\
&= \Cap^{(l',\check{p}')} + \langle\!\langle (ct_1 \bullet_x ct') \otimes ct_2 \rangle\!\rangle_\eta^{(l',\check{p}')} \\
&= \Cap^{(l',\check{p}')} + \exists l'_1, l'_2.\, \langle\!\langle (ct_1 \bullet_x ct') \rangle\!\rangle_\eta^{(l'_1,\check{p}')} + \langle\!\langle ct_2 \rangle\!\rangle_\eta^{(l'_2,\check{p}')} \wedge (l' = l'_1 : l'_2) \\
&= \exists i, l_1, l_2.\, \{\lceil \check{p} \mapsto \check{-}, \check{i} \rceil \times \check{i} \Mapsto [\, \check{l}_1 : l' : \check{l}_2 \,]\} \\
&\quad + \sum_{v \in l_1 : l_2} \{\lceil \check{v} \mapsto \check{p}, \check{-} \rceil \times \mathsf{emp}\} \\
&\quad + \exists l'_1, l'_2.\, \langle\!\langle (ct_1 \bullet_x ct') \rangle\!\rangle_\eta^{(l'_1,\check{p}')} + \langle\!\langle ct_2 \rangle\!\rangle_\eta^{(l'_2,\check{p}')} \wedge (l' = l'_1 : l'_2) \\
&= \exists i, l_1, l_2, l'_1, l'_2.\, \{\lceil \check{p} \mapsto \check{-}, \check{i} \rceil \times \check{i} \Mapsto [\, \check{l}_1 : l'_1 : l'_2 : \check{l}_2 \,]\} \\
&\quad + \sum_{v \in l_1 : l_2} \{\lceil \check{v} \mapsto \check{p}, \check{-} \rceil \times \mathsf{emp}\} \\
&\quad + \langle\!\langle (ct_1 \bullet_x ct') \rangle\!\rangle_\eta^{(l'_1,\check{p}')} + \langle\!\langle ct_2 \rangle\!\rangle_\eta^{(l'_2,\check{p}')} \wedge (l' = l'_1 : l'_2) \\
&= \exists i, l_1, l_2, l'_1, l'_2.\, \{\lceil \check{p} \mapsto \check{-}, \check{i} \rceil \times \check{i} \Mapsto [\, \check{l}_1 : \check{l}'_1 : l'_2 : \check{l}_2 \,]\} \\
&\quad + \{\lceil \check{p} \mapsto \check{-}, \check{i} \rceil \times \check{i} \Mapsto [\, \check{l}_1 : l'_1 : \check{l}'_2 : \check{l}_2 \,]\} \\
&\quad + \sum_{v \in l_1 : l_2} \{\lceil \check{v} \mapsto \check{p}, \check{-} \rceil \times \mathsf{emp}\} + \sum_{v \in l'_2} \{\lceil \check{v} \mapsto \check{p}, \check{-} \rceil \times \mathsf{emp}\} \\
&\quad + \langle\!\langle (ct_1 \bullet_x ct') \rangle\!\rangle_\eta^{(l'_1,\check{p}')} + \langle\!\langle ct_2 \rangle\!\rangle_\eta^{(l'_2,\check{p}')} \wedge (l' = l'_1 : l'_2) \\
&= \exists i, l_1, l_2, l'_1, l'_2.\, \{\lceil \check{p} \mapsto \check{-}, \check{i} \rceil \times \check{i} \Mapsto [\, \check{l}_1 : \check{l}'_1 : l'_2 : \check{l}_2 \,]\} \\
&\quad + \{\lceil \check{p} \mapsto \check{-}, \check{i} \rceil \times \check{i} \Mapsto [\, \check{l}_1 : l'_1 : \check{l}'_2 : \check{l}_2 \,]\} \\
&\quad + \sum_{v \in l_1 : l'_2 : l_2} \{\lceil \check{v} \mapsto \check{p}, \check{-} \rceil \times \mathsf{emp}\} \\
&\quad + \langle\!\langle (ct_1 \bullet_x ct') \rangle\!\rangle_\eta^{(l'_1,\check{p}')} + \langle\!\langle ct_2 \rangle\!\rangle_\eta^{(l'_2,\check{p}')} \wedge (l' = l'_1 : l'_2) \\
&= \exists i, l_1, l_2, l'_1, l'_2.\, \{\lceil \check{p} \mapsto \check{-}, \check{i} \rceil \times \check{i} \Mapsto [\, \check{l}_1 : \check{l}'_1 : l'_2 : \check{l}_2 \,]\} \\
&\quad + \Cap^{l'_1, p'} + \langle\!\langle (ct_1 \bullet_x ct') \rangle\!\rangle_\eta^{(l'_1,\check{p}')} + \langle\!\langle ct_2 \rangle\!\rangle_\eta^{(l'_2,\check{p}')} \wedge (l' = l'_1 : l'_2) \\
(IH) \quad &= \exists i, l_1, l_2, l'_1, l'_2.\, \{\lceil \check{p} \mapsto \check{-}, \check{i} \rceil \times \check{i} \Mapsto [\, \check{l}_1 : \check{l}'_1 : l'_2 : \check{l}_2 \,]\} \wedge (l' = l'_1 : l'_2) \\
&\quad + \exists I.\, \Cap^{(l'_1,\check{p}')} + \langle\!\langle ct_1 \rangle\!\rangle_{\eta[x \mapsto I]}^{(l'_1,\check{p}')} + \Cap^{I} + \langle\!\langle ct' \rangle\!\rangle_\eta^{I} + \langle\!\langle ct_2 \rangle\!\rangle_\eta^{(l'_2,\check{p}')} \\
&= \exists i, l_1, l_2, l'_1, l'_2.\, \{\lceil \check{p} \mapsto \check{-}, \check{i} \rceil \times \check{i} \Mapsto [\, \check{l}_1 : \check{l}'_1 : l'_2 : \check{l}_2 \,]\} \\
&\quad + \{\lceil \check{p} \mapsto \check{-}, \check{i} \rceil \times \check{i} \Mapsto [\, \check{l}_1 : l'_1 : \check{l}'_2 : \check{l}_2 \,]\} \wedge (l' = l'_1 : l'_2) \\
&\quad + \sum_{v \in l_1 : l'_2 : l_2} \{\lceil \check{v} \mapsto \check{p}, \check{-} \rceil \times \mathsf{emp}\} \\
&\quad + \exists I.\, \Cap^{(l'_1,\check{p}')} + \langle\!\langle ct_1 \rangle\!\rangle_{\eta[x \mapsto I]}^{(l'_1,\check{p}')} + \Cap^{I} + \langle\!\langle ct' \rangle\!\rangle_\eta^{I} + \langle\!\langle ct_2 \rangle\!\rangle_\eta^{(l'_2,\check{p}')} \\
&= \exists l'_1, l'_2.\, + \exists I.\, \Cap^{(l',\check{p}')} + \langle\!\langle ct_1 \rangle\!\rangle_{\eta[x \mapsto I]}^{(l'_1,\check{p}')} + \Cap^{I} + \langle\!\langle ct' \rangle\!\rangle_\eta^{I} \\
&\quad + \langle\!\langle ct_2 \rangle\!\rangle_\eta^{(l'_2,\check{p}')} \wedge (l' = l'_1 : l'_2) \\
&= \exists I.\, \Cap^{(l',\check{p}')} + \langle\!\langle ct_1 \otimes ct_2 \rangle\!\rangle_{\eta[x \mapsto I]}^{(l',\check{p}')} + \Cap^{I} + \langle\!\langle ct' \rangle\!\rangle_\eta^{I} \\
&= \exists I.\, \Cap^{I'} + \langle\!\langle ct \rangle\!\rangle_{\eta[x \mapsto I]}^{I'} + \Cap^{I} + \langle\!\langle ct' \rangle\!\rangle_{\eta[x \mapsto I]}^{I}
\end{aligned}
$$

The final case for $x \notin fh_{\mathrm{T}}(ct_1)$ and $x \in fh_{\mathrm{T}}(ct_2)$ is analogous to the case given above. Note that (IH) denotes an application of the inductive hypothesis.

$\square$

Lemma 6.25 (Compression Preservation)**.** Segment compression is preserved by the segment representation function. That is, for all $x \in \mathrm{X}$, $st \in S_{\mathrm{T}}$ and $\eta \in (\mathrm{X} \rightharpoonup_{\mathrm{fin}} \mathcal{I})$, there exists $I \in \mathcal{I}$ and $\bar{x} \in \mathcal{P}(\mathrm{X})$ with $\bar{x} = \mathsf{labs}(I)$ such that,

$$(\!|(x)(st)_{\mathrm{T}}|\!)^{\eta} \quad \equiv \quad (\bar{x})((\!|st|\!)^{\eta[x \mapsto I]})_{\mathbb{H}+\mathbb{L}}$$

Proof. Recall that in this translation $\mathsf{labs}(I) = \emptyset$ for all $I \in \mathcal{I}$. Thus it is sufficient to show that,

$$(\!|(x)(st)|\!)^{\eta} \equiv \exists I.\, (\!|st|\!)^{\eta[x \mapsto I]}$$

Case split on the occurrences of label x in segment st. There are four cases to consider:

(1) If $x \notin fa_{\mathrm{T}}(st)$ and $x \notin fh_{\mathrm{T}}(st)$, then $(x)(st) = st$. Any choice of I will suffice as it will never be referenced by the translation. We can then show the following:

$$\begin{array}{rcl} (\!|(x)(st)|\!)^{\eta} & = & (\!|st|\!)^{\eta} \\ & = & \exists I.\, (\!|st|\!)^{\eta[x \mapsto I]} \end{array}$$

(2) If $x \in fa_{\mathrm{T}}(st)$ and $x \notin fh_{\mathrm{T}}(st)$, then there exist some st', ct such that $st = st' \uplus \{(x, ct)\}$ where $x \notin fh_{\mathrm{T}}(st')$. Let $I = (l, \check{\mathtt{top}})$ for some l. We can then show the following:

$$\begin{array}{rcl} (\!|(x)(st)|\!)^{\eta} & = & (\!|(x)(st' \uplus \{(x, ct)\})|\!)^{\eta} \\ & = & (\!|st' \uplus \{(0, ct)\}|\!)^{\eta} \\ & = & (\!|st'|\!)^{\eta} + (\!|\{(0, ct)\}|\!)^{\eta} \\ & = & (\!|st'|\!)^{\eta} + \exists l.\, \Cap^{(l, \check{\mathtt{top}})} + \langle\!\langle ct \rangle\!\rangle_{\eta}^{(l, \check{\mathtt{top}})} \\ & = & (\!|st'|\!)^{\eta} + \exists l.\, (\!|\{(x, ct)\}|\!)^{\eta[x \mapsto (l, \check{\mathtt{top}})]} \\ & = & (\!|st'|\!)^{\eta} + \exists I.\, (\!|\{(x, ct)\}|\!)^{\eta[x \mapsto I]} \\ & = & \exists I.\, (\!|st'|\!)^{\eta[x \mapsto I]} + (\!|\{(x, ct)\}|\!)^{\eta[x \mapsto I]} \\ & = & \exists I.\, (\!|st' \uplus \{(x, ct)\}|\!)^{\eta[x \mapsto I]} \\ & = & \exists I.\, (\!|st|\!)^{\eta[x \mapsto I]} \end{array}$$

(3) If $x \notin fa_{\mathrm{T}}(st)$ and $x \in fh_{\mathrm{T}}(st)$, then $(x)(st)$ is undefined, so $(\!|(x)(st)|\!)^{\eta} = \emptyset$. Let $I = (\varepsilon, \mathsf{null})$, then $(\!|st|\!)^{\eta[x \mapsto I]} = \emptyset$ since every abstract tree node is required to have a parent in our translation.

(4) If $x \in fa_{\mathrm{T}}(st)$ and $x \in fh_{\mathrm{T}}(st)$, then there exist some st', z, ct, ct' such that $st = st' \uplus \{(z, ct), (x, ct')\}$ where $x \notin fa_{\mathrm{T}}(st')$, $x \notin fh_{\mathrm{T}}(st')$ and $x \in fh_{\mathrm{T}}(ct)$. If $x \in fh_{\mathrm{T}}(ct')$ then the tree segment is not well-formed, so assume that $x \notin fh_{\mathrm{T}}(ct')$. Let $\eta(z) = I'$ for some $I' \in \mathcal{I}$. We can then show the following:

$$\begin{array}{rcl}
(\!|(x)(st)|\!)^{\eta} & = & (\!|(x)(st' \uplus \{(z, ct), (x, ct')\})|\!)^{\eta} \\
 & = & (\!|st' \uplus \{(z, ct \bullet_x ct')\}|\!)^{\eta} \\
 & = & (\!|st'|\!)^{\eta} + (\!|\{(z, ct \bullet_x ct')\}|\!)^{\eta} \\
 & = & (\!|st'|\!)^{\eta} + ⋒^{I'} + \langle\!\langle ct \bullet_x ct' \rangle\!\rangle^{I'}_{\eta} \\
\text{(Lemma 6.24)} & = & (\!|st'|\!)^{\eta} + \exists I.\, ⋒^{I'} + \langle\!\langle ct \rangle\!\rangle^{I'}_{\eta[x \mapsto I]} + ⋒^{I} + \langle\!\langle ct' \rangle\!\rangle^{I}_{\eta} \\
 & = & \exists I.\, (\!|st'|\!)^{\eta[x \mapsto I]} + ⋒^{I'} + \langle\!\langle ct \rangle\!\rangle^{I'}_{\eta[x \mapsto I]} + ⋒^{I} + \langle\!\langle ct' \rangle\!\rangle^{I}_{\eta[x \mapsto I]} \\
 & = & \exists I.\, (\!|st'|\!)^{\eta[x \mapsto I]} + (\!|\{(z, ct)\}|\!)^{\eta[x \mapsto I]} + (\!|\{(x, ct')\}|\!)^{\eta[x \mapsto I]} \\
 & = & \exists I.\, (\!|st' \uplus \{(z, ct), (x, ct')\}|\!)^{\eta[x \mapsto I]} \\
 & = & \exists I.\, (\!|st|\!)^{\eta[x \mapsto I]}
\end{array}$$

□

Lemma 6.26 (Axiom Correctness)**.** For all $e \in \textsc{Env}$, $\Gamma \in \textsc{PSEnv}$, $\varphi \in \textsc{Cmd}_{\mathbb{T}}$, $(P, Q) \in \mathrm{Ax}[\![\varphi]\!]_{\mathbb{T}}$ and $\eta \in (\mathrm{X}_{\mathbb{A}} \rightharpoonup_{\mathrm{fin}} \mathcal{I})$,

$$e, [\![\Gamma]\!]_{\tau_3} \vdash_{\mathbb{B}} \left\{ \ [\![P]\!]_{\tau_3} \ \right\} \ [\![\varphi]\!]_{\tau_3} \ \left\{ \ [\![Q]\!]_{\tau_3} \ \right\}$$

As with the previous translation, we will not give proofs for all of the basic commands in the tree module, but instead give an example (`deleteTree`) that illustrates the techniques involved in the proofs.

Axiom Correctness: `deleteTree`

Recall the specification of the `deleteTree` command from Figure 5.2.

$$\begin{array}{c}
\left\{ \ \alpha \leftarrow w[\mathsf{tree}(ct)] * \sigma \wedge \mathcal{E}[\![E]\!]\sigma = w \ \right\} \\
\mathtt{deleteTree}(E) \\
\left\{ \ \alpha \leftarrow \varnothing * \sigma \ \right\}
\end{array}$$

To prove that this specification holds under our translation, suppose that $e(\alpha) = x$ for some $x \in \mathrm{X}$. We can also assume that $x \in dom(\eta)$, otherwise the translated precondition is equivalent to false, and that $\eta(x) = (l, p)$ for some choice of l and p. The predicate $\mathsf{tree}(ct)$ tells us that the tree context ct has no context holes, so we let $e(ct) = t$ (recall that we use t to denote a tree context with no holes). In

Figure 6.29 and Figure 6.30 we give a proof outline showing that the implementation of `deleteTree` (from Figure 6.28) satisfies the translation of its specification.

The proof assumes that the translated specification holds for the recursive calls to the `deleteTree` procedure. Note that this requires us to have an upper crust for the subtree $n[t']$ that is being deleted in each iteration of the while loop. We can extract this upper crust from the predicate $(\lceil w \mapsto \check{p}, j \rceil * j \Mapsto [\, n : l'' \,]) * \langle\!\langle t'' \rangle\!\rangle_{\eta}^{(l'', w)}$ in a similar fashion to that seen in our proof of the crust inclusion lemma.

This concludes the proof of Theorem 6.22.

6.3.4 Module Translation $\tau_4 : \mathbb{H} + \mathbb{H} \to \mathbb{H}$

The last example of a locality-preserving translation that we consider is the natural implementation of a pair of heap modules $\mathbb{H} + \mathbb{H}$ with a single heap $\mathbb{H}$ that treats the two heaps as disjoint portions of the same heap. Not only does this example complete our stepwise refinement of the tree module $\mathbb{T}$, but it also demonstrates an example that does not result in a surjective abstraction relation and yet is still a sound locality preserving translation. The abstraction relation is not surjective as different abstract heaps may map into the same concrete heap.

Notation: Recall that the set $\mathrm{S_{H \times H}} = \mathrm{S_H} \times \mathrm{S_H}$, so elements of $\mathrm{S_{H\times H}}$ are of the form $sh \times sh'$. Similarly $\mathrm{X_{H\times H}} = \mathrm{X} \times \mathrm{X}$, so elements of $\mathrm{X_{H\times H}}$ are of the form $x \times x'$. Further recall that the label set X is countably infinite. This means that we can split the label set such that $\mathrm{X} = \mathrm{X}_1 \uplus \mathrm{X}_2$ with X_1 and X_2 both being countably infinite. We denote elements of the respective subsets with appropriate placed subscripts.

Definition 6.27 ($\tau_4 : \mathbb{H} + \mathbb{H} \to \mathbb{H}$)**.** The pre-locality preserving translation $\tau_4 : \mathbb{H} + \mathbb{H} \to \mathbb{H}$ is constructed as follows:

- ◇ an interface $I = (x_1, x_2) \in \mathcal{I}$ describes a pair of labels $x_1 \in \mathrm{X}_1$ and $x_2 \in \mathrm{X}_2$. Note that $\mathsf{labs}(I) = \{x_1, x_2\}$.

- ◇ the interface function $\eta : \mathrm{X_{H\times H}} \rightharpoonup_{\mathrm{fin}} \mathcal{I}$ is defined as:

$$\eta(x \times y) \quad \stackrel{\mathrm{def}}{=} \quad (x_1, y_2)$$

- ◇ the segment representation function $(\!|(\cdot)|\!)^{(\cdot)} : \mathrm{S_{H\times H}} \times (\mathrm{X_{H\times H}} \rightharpoonup_{\mathrm{fin}} \mathcal{I}) \to \mathcal{P}(\mathrm{S_H})$ is defined as:

$$(\!| sh_1 \times sh_2 |\!)^{\eta} \quad \stackrel{\mathrm{def}}{=} \quad \langle\!\langle sh_1 \rangle\!\rangle^1 +_{\mathrm{H}} \langle\!\langle sh_2 \rangle\!\rangle^2$$

  where $\langle\!\langle sh \rangle\!\rangle^i = sh\,[\,[\![\bar{x}]\!]^i / \bar{x}\,]$ with the label set $\bar{x} = fa(sh) \cup fh(sh)$ being replaced by the translated label set $[\![\bar{x}]\!]^i = \{x_i \mid x \in \bar{x}\}$.

```
$\left\{\ [\![\, \alpha \leftarrow w[\mathsf{tree}(ct)] * \mathtt{n} \Rightarrow w \,]\!]_{\tau_3}\ \right\}$
proc deleteTree(n){
  $\left\{\ \Cap^{(l,\check{p})} * \langle\!\langle w[t] \rangle\!\rangle_{\eta}^{(l,\check{p})} * \mathtt{n} \Rightarrow w\ \right\}$
  local x, y, z in
    $\left\{\ \Cap^{(l,\check{p})} * \langle\!\langle w[t] \rangle\!\rangle_{\eta}^{(l,\check{p})} * \mathtt{n} \Rightarrow w * \mathtt{x} \Rightarrow - * \mathtt{y} \Rightarrow - * \mathtt{z} \Rightarrow -\ \right\}$
    $\left\{\ \exists i, j, l_1, l_2, l'. \,(\lceil \check{p} \mapsto \check{-}, \check{i} \rceil \times \check{i} \Mapsto [\, \check{l_1} : w : \check{l_2} \,]) * \circledast_{v \in l_1 : l_2} (\lceil \check{v} \mapsto \check{-}, \check{-} \rceil \times \mathsf{emp})$
      $* \,(\lceil w \mapsto \check{p}, j \rceil \times j \Mapsto [\, l' \,]) * \langle\!\langle t \rangle\!\rangle_{\eta}^{(l',w)} * \mathtt{n} \Rightarrow w * \mathtt{x} \Rightarrow - * \mathtt{y} \Rightarrow - * \mathtt{z} \Rightarrow -\ \right\}$
    x := [n.parent] ;
    y := [x.children] ;
    $\left\{\ \exists i, j, l_1, l_2, l'. \,(\lceil \check{p} \mapsto \check{-}, \check{i} \rceil \times \check{i} \Mapsto [\, \check{l_1} : w : \check{l_2} \,]) * \circledast_{v \in l_1 : l_2} (\lceil \check{v} \mapsto \check{-}, \check{-} \rceil \times \mathsf{emp})$
      $* \,(\lceil w \mapsto \check{p}, j \rceil \times j \Mapsto [\, l' \,]) * \langle\!\langle t \rangle\!\rangle_{\eta}^{(l',w)} * \mathtt{n} \Rightarrow w * \mathtt{x} \Rightarrow \check{p} * \mathtt{y} \Rightarrow \check{i} * \mathtt{z} \Rightarrow -\ \right\}$
    y.remove(n) ;
    $\left\{\ \exists i, j, l_1, l_2, l'. \,(\lceil \check{p} \mapsto \check{-}, \check{i} \rceil \times \check{i} \Mapsto [\, \check{l_1} : \check{l_2} \,]) * \circledast_{v \in l_1 : l_2} (\lceil \check{v} \mapsto \check{-}, \check{-} \rceil \times \mathsf{emp})$
      $* \,(\lceil w \mapsto \check{p}, j \rceil \times j \Mapsto [\, l' \,]) * \langle\!\langle t \rangle\!\rangle_{\eta}^{(l',w)} * \mathtt{n} \Rightarrow w * \mathtt{x} \Rightarrow \check{p} * \mathtt{y} \Rightarrow \check{i} * \mathtt{z} \Rightarrow -\ \right\}$
    y := [n.children] ;
    $\left\{\ \exists i, j, l_1, l_2, l'. \,(\lceil \check{p} \mapsto \check{-}, \check{i} \rceil \times \check{i} \Mapsto [\, \check{l_1} : \check{l_2} \,]) * \circledast_{v \in l_1 : l_2} (\lceil \check{v} \mapsto \check{-}, \check{-} \rceil \times \mathsf{emp})$
      $* \,(\lceil w \mapsto \check{p}, j \rceil \times j \Mapsto [\, l' \,]) * \langle\!\langle t \rangle\!\rangle_{\eta}^{(l',w)} * \mathtt{n} \Rightarrow w * \mathtt{x} \Rightarrow \check{p} * \mathtt{y} \Rightarrow j * \mathtt{z} \Rightarrow -\ \right\}$
    z := y.getHead() ;
    $\left\{\ \exists i, j, l_1, l_2. \,(\lceil \check{p} \mapsto \check{-}, \check{i} \rceil \times \check{i} \Mapsto [\, \check{l_1} : \check{l_2} \,]) * \circledast_{v \in l_1 : l_2} (\lceil \check{v} \mapsto \check{-}, \check{-} \rceil \times \mathsf{emp})$
      $* \,\exists l', l'', k, n, t', t''. \,(\lceil w \mapsto \check{p}, j \rceil \times j \Mapsto [\, n : l'' \,]) * (n \mapsto w, k \times k \Mapsto [\, l' \,])$
      $* \,\langle\!\langle t' \rangle\!\rangle_{\eta}^{(l',n)} * \langle\!\langle t'' \rangle\!\rangle_{\eta}^{(l'',w)} * \mathtt{n} \Rightarrow w * \mathtt{x} \Rightarrow \check{p} * \mathtt{y} \Rightarrow j * \mathtt{z} \Rightarrow n$
      $\vee\, \exists i, j, l_1, l_2. \,(\lceil \check{p} \mapsto \check{-}, \check{i} \rceil \times \check{i} \Mapsto [\, \check{l_1} : \check{l_2} \,]) * \circledast_{v \in l_1 : l_2} (\lceil \check{v} \mapsto \check{-}, \check{-} \rceil \times \mathsf{emp})$
      $* \,(\lceil w \mapsto \check{p}, j \rceil \times j \Mapsto [\, \varepsilon \,]) * \langle\!\langle \varnothing \rangle\!\rangle_{\eta}^{(\varepsilon,w)} * \mathtt{n} \Rightarrow w * \mathtt{x} \Rightarrow \check{p} * \mathtt{y} \Rightarrow j * \mathtt{z} \Rightarrow \mathsf{null}\ \right\}$
    while z $\neq$ null do
    $\left\{\ \exists i, j, l_1, l_2. \,(\lceil \check{p} \mapsto \check{-}, \check{i} \rceil \times \check{i} \Mapsto [\, \check{l_1} : \check{l_2} \,]) * \circledast_{v \in l_1 : l_2} (\lceil \check{v} \mapsto \check{-}, \check{-} \rceil \times \mathsf{emp})$
      $* \,\exists l', l'', k, n, t', t''. \,(\lceil w \mapsto \check{p}, j \rceil \times j \Mapsto [\, n : l'' \,]) * (n \mapsto w, k \times k \Mapsto [\, l' \,])$
      $* \,\langle\!\langle t' \rangle\!\rangle_{\eta}^{(l',n)} * \langle\!\langle t'' \rangle\!\rangle_{\eta}^{(l'',w)} * \mathtt{n} \Rightarrow w * \mathtt{x} \Rightarrow \check{p} * \mathtt{y} \Rightarrow j * \mathtt{z} \Rightarrow n\ \right\}$
      call deleteTree(z) ;
      z := y.getHead()
      $\left\{\ \exists i, j, l_1, l_2. \,(\lceil \check{p} \mapsto \check{-}, \check{i} \rceil \times \check{i} \Mapsto [\, \check{l_1} : \check{l_2} \,]) * \circledast_{v \in l_1 : l_2} (\lceil \check{v} \mapsto \check{-}, \check{-} \rceil \times \mathsf{emp})$
        $* \,\exists l', l'', k, n, t', t''. \,(\lceil w \mapsto \check{p}, j \rceil \times j \Mapsto [\, n : l'' \,]) * (n \mapsto w, k \times k \Mapsto [\, l' \,])$
        $* \,\langle\!\langle t' \rangle\!\rangle_{\eta}^{(l',n)} * \langle\!\langle t'' \rangle\!\rangle_{\eta}^{(l'',w)} * \mathtt{n} \Rightarrow w * \mathtt{x} \Rightarrow \check{p} * \mathtt{y} \Rightarrow j * \mathtt{z} \Rightarrow n$
        $\vee\, \exists i, j, l_1, l_2. \,(\lceil \check{p} \mapsto \check{-}, \check{i} \rceil \times \check{i} \Mapsto [\, \check{l_1} : \check{l_2} \,]) * \circledast_{v \in l_1 : l_2} (\lceil \check{v} \mapsto \check{-}, \check{-} \rceil \times \mathsf{emp})$
        $* \,(\lceil w \mapsto \check{p}, j \rceil \times j \Mapsto [\, \varepsilon \,]) * \langle\!\langle \varnothing \rangle\!\rangle_{\eta}^{(\varepsilon,w)} * \mathtt{n} \Rightarrow w * \mathtt{x} \Rightarrow \check{p} * \mathtt{y} \Rightarrow j * \mathtt{z} \Rightarrow \mathsf{null}\ \right\}$
    $\left\{\ \exists i, j, l_1, l_2. \,(\lceil \check{p} \mapsto \check{-}, \check{i} \rceil \times \check{i} \Mapsto [\, \check{l_1} : \check{l_2} \,]) * \circledast_{v \in l_1 : l_2} (\lceil \check{v} \mapsto \check{-}, \check{-} \rceil \times \mathsf{emp})$
      $* \,(\lceil w \mapsto \check{p}, j \rceil \times j \Mapsto [\, \varepsilon \,]) * \mathtt{n} \Rightarrow w * \mathtt{x} \Rightarrow \check{p} * \mathtt{y} \Rightarrow j * \mathtt{z} \Rightarrow -\ \right\}$
                                    ⋮
```

Figure 6.29: Proof outline for `deleteTree` implementation in τ_3.

$$\vdots$$

$$\left\{ \begin{array}{l} \exists i, j, l_1, l_2.\, (\lceil \check{p} \mapsto \check{-}, \check{i} \rceil \times \check{i} \Mapsto [\, \check{l_1} : \check{l_2} \,]) * \circledast_{v \in l_1 : l_2} (\lceil \check{v} \mapsto \check{-}, \check{-} \rceil \times \mathsf{emp}) \\ * \, (\lceil w \mapsto \check{p}, j \rceil \times j \Mapsto [\, \varepsilon \,]) * \mathtt{n} \Rightarrow w * \mathtt{x} \Rightarrow \check{p} * \mathtt{y} \Rightarrow j * \mathtt{z} \Rightarrow - \end{array} \right\}$$

$\mathtt{deleteList}(\mathtt{y})$;

$$\left\{ \begin{array}{l} \exists i, l_1, l_2.\, (\lceil \check{p} \mapsto \check{-}, \check{i} \rceil \times \check{i} \Mapsto [\, \check{l_1} : \check{l_2} \,]) * \circledast_{v \in l_1 : l_2} (\lceil \check{v} \mapsto \check{-}, \check{-} \rceil \times \mathsf{emp}) \\ * \, (\lceil w \mapsto \check{p}, j \rceil \times \mathsf{emp}) * \mathtt{n} \Rightarrow w * \mathtt{x} \Rightarrow \check{p} * \mathtt{y} \Rightarrow j * \mathtt{z} \Rightarrow - \end{array} \right\}$$

$\mathtt{disposeNode}(\mathtt{n})$

$$\left\{ \begin{array}{l} \exists i, l_1, l_2.\, (\lceil \check{p} \mapsto \check{-}, \check{i} \rceil \times \check{i} \Mapsto [\, \check{l_1} : \check{l_2} \,]) * \circledast_{v \in l_1 : l_2} (\lceil \check{v} \mapsto \check{-}, \check{-} \rceil \times \mathsf{emp}) \\ * \, (\mathsf{emp} \times \mathsf{emp}) * \mathtt{n} \Rightarrow w * \mathtt{x} \Rightarrow \check{p} * \mathtt{y} \Rightarrow j * \mathtt{z} \Rightarrow - \end{array} \right\}$$

$$\left\{ \begin{array}{l} \exists i, l_1, l_2.\, (\lceil \check{p} \mapsto \check{-}, \check{i} \rceil \times \check{i} \Mapsto [\, \check{l_1} : l : \check{l_2} \,]) * \circledast_{v \in l_1 : l_2} (\lceil \check{v} \mapsto \check{-}, \check{-} \rceil \times \mathsf{emp}) \\ * \, (\mathsf{emp} \times \mathsf{emp}) \wedge (l = \varepsilon) * \mathtt{n} \Rightarrow w \end{array} \right\}$$

$$\left\{ \; \Cap^{(l, \check{p})} * \langle\!\langle \varnothing \rangle\!\rangle_{\eta}^{(l, \check{p})} * \mathtt{n} \Rightarrow w \; \right\}$$

$\}$

$$\left\{ \; [\![\, \alpha \leftarrow \varnothing * \mathtt{n} \Rightarrow w \,]\!]_{\tau_3} \; \right\}$$

Figure 6.30: Proof outline for `deleteTree` implementation in τ_3 continued.

◇ the substitutive representation function is given by replacing the commands for both heaps with their detagged versions, for example,

$$[\![\mathtt{dispose}_1(E, E')]\!]_{\tau_4} \overset{\text{def}}{=} [\![\mathtt{dispose}_2(E, E')]\!]_{\tau_4} \overset{\text{def}}{=} \mathtt{dispose}(E, E')$$

The translation τ_4 simply merges together the heap pair into a single heap, tagging the labels in each heap so that they do not clash in the resulting heap. This simple translation does not need to include any extra crust as the abstract and concrete levels have the same notion of locality.

Theorem 6.28 (Soundness of τ_4)**.** The pre-locality-preserving translation τ_4 is a locality-preserving translation.

The proof of this theorem is significantly simpler than in our previous examples. The combination preservation property holds from the definition of $(\!| s |\!)^{\eta}$ and the properties of $+$. The compression preservation property can be shown by observing that:

$$\begin{aligned}
(\!|(x \times y)(sh \times sh')|\!)^\eta &\overset{\text{def}}{=} (\!|(x)(sh) \times (y)(sh')|\!)^\eta \\
&\overset{\text{def}}{=} \langle\!\langle (x)(sh) \rangle\!\rangle^1 + \langle\!\langle (y)(sh') \rangle\!\rangle^2 \\
&\overset{\text{def}}{=} (x_1)(\langle\!\langle sh \rangle\!\rangle^1) + (y_2)(\langle\!\langle sh' \rangle\!\rangle^2) \\
&\overset{\text{def}}{=} (x_1, y_2)(\langle\!\langle sh \rangle\!\rangle^1 + \langle\!\langle sh' \rangle\!\rangle^2) \\
&\overset{\text{def}}{=} (x_1, y_2)((\!|sh \times sh'|\!)^\eta)
\end{aligned}$$

Finally, the axiom correctness property holds because the axioms of $\mathbb{H} + \mathbb{H}$ are directly translated to those of $\mathbb{H}$ with some extra frame. Notice, however, that this translation does not satisfy the first of our properties for including the conjunction rule in our theory, since $(\!|\lceil 1 \mapsto 0 \rceil \times \lceil \mathsf{emp} \rceil|\!)^\eta = \{\lceil 1 \mapsto 0 \rceil\} = (\!|\lceil \mathsf{emp} \rceil \times \lceil 1 \mapsto 0 \rceil|\!)^\eta$.

6.4 Remarks

We have shown how to refine abstract modules in our fine-grained local reasoning framework. This provides an alternative justification for the soundness of fine-grained abstract local reasoning with segment algebras. As with previous work, we have identified two general approaches for proving the correctness of an implementation with respect to an abstract specification: locality-breaking and locality-preserving translations. Locality-breaking translations establish a ‘fiction of locality’ by justifying abstract locality, even though this locality is not matched by the implementation. Locality-preserving translations instead relate the abstract locality of a module with the low-level locality of its implementation. This is complicated by the fact that disjoint structures at the high-level are not necessarily still disjoint at the low-level. Locality-preserving translations thus establish a ‘fiction of disjointness’ at the abstract level.

Locality-Breaking vs. Locality-Preserving

Our choice of names may seem to imply that our reasoning techniques are applicable in distinct cases, but both techniques can in fact be used in all cases.

As an example, consider our implementation of the list module from §6.2.2. We proved that this implementation was correct by providing a locality-breaking translation, since some of the basic commands had large low-level footprints that could act over the whole linked-list. We could equally have chosen to identify elements of the abstract list with nodes in the concrete linked-list and treated the part of the list leading up to the node of interest as the concrete interface, or crust.

As another example, consider our implementation of the tree module from §6.3.2.

We proved that this implementation was correct by providing a locality-preserving translation, since all of the basic commands had low-level footprints that were similar in size to their abstract footprints. We could instead have chosen to only translate complete rooted trees and proved each of the basic command's axioms under all possible frames.

The main difference in our two approaches is the burden of the proof of a sound translation. If the concrete data structure is relatively simple and the frames can all be considered in one form, then the locality-breaking technique tends to offer an easier correctness proof. If instead the concrete data structure is very complex, it may introduce a significant increase in the number of cases that would need to be proven with the locality-breaking approach. In such cases it may be desirable to use the locality-preserving technique. However, the locality-preserving technique is definitely the more complex of the two, and it is often non-trivial to work what model of permissions is needed to establish the 'fiction of disjointness'. At present the generation of such permissions models is somewhat ad-hoc. In future it would be interesting to see if a general permissions model could be found to ease this part of the proof burden.

Abstract Predicates

Our module translation functions could be viewed as abstract predicates of the concrete module. That is $[\![P]\!]_\tau$ could be viewed as an abstract predicate parameterised by P. However, viewing the translation function as a completely abstract entity does not translate abstract local reasoning between modules. We could add axioms to our translations, such as $[\![P]\!]_\tau \vee [\![Q]\!]_\tau \Leftrightarrow [\![P \vee Q]\!]_\tau$, which would allow the low-level inference rules to implement their high-level counterparts. However, abstract predicates do not currently provide a mechanism for exporting meta-theorems, such as the soundness of our frame rules. This means there is no way to expose the fact that if $\{P\}\,\mathbb{C}\,\{Q\}$ then so does $\{P * R\}\,\mathbb{C}\,\{Q * R\}$. It would be interesting to see the results of including such a mechanism in the abstract predicate methodology.

Abstraction and Refinement for Concurrency

Extending our results to the concurrent setting is not a trivial matter. In particular, our locality-preserving technique relies on the stability of assertions made about the crust. In the sequential case, where there can be no interference from the environment, such stability is automatically assured. However, in the concurrent case, checking that these assertions are indeed stable will require significantly more work.

We will need to introduce some control mechanisms, such as locking or transactions, that will be able to ensure that threads only interact in desirable ways. By controlling access to the crusts of our translation, we should be able to establish the stability of assertions about them.

7 Towards Concurrency

So far we have concentrated on reasoning about sequential programs. In this chapter we turn our attention to reasoning about concurrent programs.

In a concurrent program there can be a number of processes running at the same time. Early concurrency was mostly limited to using separate machines to tackle problems that required an intensive amount of computational power. Nowadays, even the humble family desktop computer has multiple processors for running day to day tasks. Concurrent programs mainly mainly independently of one another, however they will occasionally need to interact. When this interaction is useful it is termed *communication*, but when the interaction leads to undesirable results it is instead termed *interference*. The challenge of concurrent programing is to write programs that make use of communication without causing interference. However, concurrent programming is hard and error prone. The main issue lies with the possible process interactions being non-deterministic. Standard testing methods will be able to spot errors in individual threads, but some errors may only show up if, say, three threads are trying to perform a certain combination of actions. It is easy to miss cases, especially if they rely on the interactions of hundreds of threads, and end up with buggy code. For this reason, in practice, a lot of concurrent programs are written without any process interaction. One of the main aims of the formal verification community is to provide programmers with the tools they need to be able to correctly create highly interactive concurrent programs that are bug free.

There are two main methods for communicating between concurrent processes: channels and shared memory. *Channel-based* systems interact by sending messages across channels and reading messages from these channels. *Shared memory* systems instead interact by reading from and writing to shared locations in memory. In terms of formal verification, channel based systems are often reasoned about using process calculi such as the Pi Calculus [48]. A lot of progress has been made in reasoning about channel-based concurrency and this system is now reasonably well understood.

By contrast, shared memory concurrency is very poorly understood and shared memory programming tends to be very error prone. For these reasons the local reasoning community has chosen to focus a lot of its recent efforts in the direction of shared memory concurrency. Our hope is that providing formal reasoning for such uses of concurrency will aid in the development of correct programs that make use of shared memory concurrency.

In chapter 2 we saw some existing work for reasoning about shared memory concurrency at low-levels of abstraction. We now investigate how to bring these ideas into our fine-grained abstract reasoning framework from chapter 4.

7.1 Concurrency Terminology

Before we start to extended our fine-grained abstract reasoning framework, we shall first explain the concurrency terminology we will be using.

A *thread* is a process in a shared memory system. Some systems have a fixed number of threads, while other systems are more dynamic and allow threads to be created at run time. Some languages manage threads in a nested way, allowing a thread to be split into sub-threads which are joined together once the have all terminated. Other languages allow a thread to be spawned at any time, executing them in parallel, possibly collecting their results at some later point. It is common for each thread to be given a unique identifier to distinguish it from other threads.

Threads are said to be *synchronised* if they agree on the order in which some events will happen. This agreement is reached by the threads communicating via primitive operations provided by the hardware (for example mutual exclusion locks, atomic reads/writes or CAS). *Blocking* synchronisation refers to a programming style that uses mutual exclusion locks to arrange inter-thread synchronisation. When a thread want to access a shared resource it atomically checks that the resource is not in use and updates the resource to say it is in use. If the resource is already in use, then the thread waits (blocks) until the resource becomes available. When a thread finishes with a resource it updates the resource to say it is no longer in use. This style of synchronisation actually reduces the parallelism (or potential concurrency) of a system, so a great deal of care has to be taken to ensure that only relevant parts of the shared structure are locked in this way. Additionally the use of locking can lead to a number of other issues, such as deadlock (where threads hold the locks that each other need access to and so neither can progress) or livelock (where a thread enters an infinite loop whilst holding the lock on some resource). However, despite all of this, the use of locking is still very common indeed. *Non-blocking* synchronisation refers to a programming style that always achieves progress, even if some threads of the system are descheduled or fail. Usually this is achieved through the use of atomic reads/writes or CAS.

An operation is said to be *atomic* if it is performed as a single unit of work that cannot be interfered with by other operations. An atomic operation effectively stops the entire system whilst it is carried out. A whole sequence of commands can also be made into an atomic operation, so that no other threads can interfere with any stage of a computation, however, such uses of atomicity drastically slow down a concurrent system.

CAS (compare and swap) is a very common non-blocking synchronisation operation. It takes three arguments: a memory address, an expected value, and a new value. The operation atomically reads the memory address and checks to see if it contains the expected value. If it does it updates the memory address with the new value, otherwise it does nothing.

A *race condition* occurs when two threads try to access the same shared memory location at the same time. If reads/writes are not atomic then it is possible that reading this memory location may result in an inconsistent value and writing to this memory location my result in a corrupted value. Even if reads/writes are atomic, we still do not know the order in which the operations are performed, so we cannot necessarily know the result of running such operations concurrently. In practice, the main difficulty with concurrent programming is trying to avoid such race conditions.

7.2 Concurrent Segment Logic

The development of segment logic has allowed us to enrich our abstract reasoning with the separating conjunction $*$ which elegantly captures the property of abstract disjointness. Following in the style of concurrent separation logic, as introduced in chapter 2, it should now be possible to reason about concurrent update programs in our reasoning framework. We shall enrich the programming language of our framework to include several concurrency constructs and extend our reasoning system to handle these extra constructs.

7.2.1 Disjoint Concurrency

Our first step is to look at simple concurrent programs that operate on entirely disjoint parts of the data structure. The design of these programs is intended to rule out the possibility of any race conditions occurring.

Definition 7.1 (Programming Language with Parallel Threads)**.** The programming language $\mathcal{L}_{\text{CMD}}$, from definition 4.1, is extended to the language $\mathcal{L}'_{\text{CMD}}$ by adding a parallel thread construct.

$$\mathbb{C} \ ::= \ \ldots \mid \mathbb{C} \mid\mid \mathbb{C}$$

At the abstract level the finest grain of operation available to us is that of the basic commands $\varphi \in$ CMD. We therefore choose to treat each of these basic commands as an atomic operation. With this in mind, we then treat the semantics of parallel threads in terms of the possible interleavings of the basic commands in each thread. Informally we can represent the behaviour of parallel threads in the structural operational semantics style, where the $\rightsquigarrow$ relation describes a single program step:

$$\frac{\mathbb{C}_1, \gamma, d, \sigma \rightsquigarrow \mathbb{C}_1', \gamma, d', \sigma'}{\mathbb{C}_1 \mid\mid \mathbb{C}_2, \gamma, d, \sigma \rightsquigarrow \mathbb{C}_1' \mid\mid \mathbb{C}_2, \gamma, d', \sigma'} \qquad \frac{\mathbb{C}_2, \gamma, d, \sigma \rightsquigarrow \mathbb{C}_2', \gamma, d', \sigma'}{\mathbb{C}_1 \mid\mid \mathbb{C}_2, \gamma, d, \sigma \rightsquigarrow \mathbb{C}_1 \mid\mid \mathbb{C}_2', \gamma, d', \sigma'}$$

$$\frac{\mathbb{C}_1, \gamma, d, \sigma \rightsquigarrow \texttt{skip}, \gamma, d', \sigma'}{\mathbb{C}_1 \mid\mid \mathbb{C}_2, \gamma, d, \sigma \rightsquigarrow \mathbb{C}_2 \gamma, d', \sigma'} \qquad \frac{\mathbb{C}_2, \gamma, d, \sigma \rightsquigarrow \texttt{skip}, \gamma, d', \sigma'}{\mathbb{C}_1 \mid\mid \mathbb{C}_2, \gamma, d, \sigma \rightsquigarrow \mathbb{C}_1 \gamma, d', \sigma'}$$

$$\frac{\mathbb{C}_1, \gamma, d, \sigma \rightsquigarrow \lightning}{\mathbb{C}_1 \mid\mid \mathbb{C}_2, \gamma, d, \sigma \rightsquigarrow \lightning} \qquad \frac{\mathbb{C}_2, \gamma, d, \sigma \rightsquigarrow \lightning}{\mathbb{C}_1 \mid\mid \mathbb{C}_2, \gamma, d, \sigma \rightsquigarrow \lightning}$$

In §7.3 we will give a formal treatment of the semantics of parallel composition in terms of traces. However, this description should be sufficient to gain an intuitive understanding of our concurrency model.

Even such a simple addition to our basic programming language allows us to express a range of concurrent programs. Here we consider a couple of illustrative examples: in our tree module we look at a program that access disjoint resources and in our heap module we look at a program that uses the divide and conquer style of programming. Both of these programs link back to our motivating examples for the development of segment logic at the end of chapter 2.

Example 7.2 (Simple Disjoint Concurrency)**.** In chapter 2 we discussed the program `delete2Trees`, which deleted two disjoint trees, and in chapter 5 we showed how to reason about this program with segment logic. With our parallel thread construct we can now write a program `deletePair` that executes the two tree deletions in parallel:

$$\texttt{deletePair}(\texttt{n}, \texttt{m}) ::= \ \texttt{deleteTree}(\texttt{n}) \ \Big\| \ \texttt{deleteTree}(\texttt{m})$$

Example 7.3 (Divide and Conquer Concurrency)**.** Disjoint concurrency is by far the easiest form of concurrency to reason about, and it is not without its uses. Many algorithms are designed with the 'divide and conquer' style in mind. You start with a single thread, and this thread divides up the data structure into disjoint parts and creates sub-threads which run in parallel in these disjoint structures. Such a programming style ensures race freedom, and can also provide a significant speed-up for some some operations. One good example of this is the `parDeleteTree` program, discussed at the end of chapter 2, which makes use of parallel threads to speed up the deletion of a binary tree stored at `n`.

```
parTreeDelete(n) ::= local l, r in
                       if n ≠ null then
                         l := n.left ;
                         r := n.right ;
                         parTreeDelete(l) || parTreeDelete(r)
                         dispose(n)
```

The initial part of the program sets up pointers to the left and right child of the parent node. Two threads are then spawned to handle the deletions of these two subtrees. Once both threads have completed the main program then deleted the parent node.

7.2.2 Reasoning About Disjoint Concurrency

We can reason about programs that make use of disjoint concurrency using much the same techniques as concurrent separation logic. The most important addition of segment logic to the abstract reasoning setting is the addition of the separating conjunction $*$. With this operator we can easily express properties about disjoint portions of program state. Just as in concurrent separation logic, we extend our notion of a local Hoare triple so that $e, \Gamma \vDash \{P\}\, \mathbb{C}\, \{Q\}$ also ensures race freedom of the program $\mathbb{C}$. We then add to our reasoning framework an inference rule for reasoning about the execution of parallel threads.

Definition 7.4 (Disjoint Concurrency Inference Rules)**.** The Hoare logic rules of our reasoning system, from definition 4.13, are extended to include the following inference rule for parallel composition:

$$\textsc{Par}: \quad \frac{e, \Gamma \vdash \{P_1\}\ \mathbb{C}_1\ \{Q_1\} \qquad e, \Gamma \vdash \{P_2\}\ \mathbb{C}_2\ \{Q_2\}}{e, \Gamma \vdash \{P_1 * P_2\}\ \mathbb{C}_1 \mid\mid \mathbb{C}_2\ \{Q_1 * Q_2\}}$$

Notice that due to our treatment of variables as resource, we do not need to provide a side-condition for the Par rule. Each resource can only be sent to one side of the parallel call, so a variable cannot be used by both threads $\mathbb{C}_1$ and $\mathbb{C}_2$. For example, consider the following program:

$$\mathtt{x} := \mathtt{x} + 1 \,\Big\|\, \mathtt{y} := \mathtt{x}$$

We can easily provide specifications for each of the threads,

$$\begin{array}{c|c}
\left\{\ \mathtt{x} \Rightarrow v\ \right\} & \left\{\ \mathtt{y} \Rightarrow - * \mathtt{x} \Rightarrow v\ \right\} \\
\mathtt{x} := \mathtt{x} + 1 & \mathtt{y} := \mathtt{x} \\
\left\{\ \mathtt{x} \Rightarrow v + 1\ \right\} & \left\{\ \mathtt{y} \Rightarrow v * \mathtt{x} \Rightarrow v\ \right\}
\end{array}$$

However, the preconditions of the two threads are not compatible when joined with separating conjunction ($\mathtt{x} \Rightarrow - * \mathtt{x} \Rightarrow - \implies \mathsf{false}$). We cannot provide a specification for the overall program because of the race that occurs for access to the variable `x`.

If the threads access completely separate sets of program variables, then the specification for the overall program can be derived as expected. For example, consider the following small example with its sketch proof:

$$\begin{array}{c}
\left\{\ \mathtt{x} \Rightarrow - * \mathtt{y} \Rightarrow -\ \right\} \\
\begin{array}{c|c}
\left\{\ \mathtt{x} \Rightarrow -\ \right\} & \left\{\ \mathtt{y} \Rightarrow -\ \right\} \\
\mathtt{x} := 5 & \mathtt{y} := 7 \\
\left\{\ \mathtt{x} \Rightarrow 5\ \right\} & \left\{\ \mathtt{y} \Rightarrow 7\ \right\}
\end{array} \\
\left\{\ \mathtt{x} \Rightarrow 5 * \mathtt{y} \Rightarrow 7\ \right\}
\end{array}$$

The overall precondition requires that `x` and `y` denote separate program variables. The rest of the reasoning then proceeds in a straightforward fashion. Disjoint access to other shared resources, such as heap cells or tree nodes, can be reasoned about in a similar fashion.

Recall our simple disjoint concurrency program from example 7.2 which takes the `delete2Trees` program from chapter 2 and runs the two tree deletions in parallel. In chapter 5 we were able to provide the following specification of the `delete2Trees` program:

$$\begin{array}{c}
\left\{\ \alpha{\leftarrow}n[\mathsf{tree}(ct_1)] * \beta{\leftarrow}m[\mathsf{tree}(ct_2)] * \mathtt{n} \Rightarrow n * \mathtt{m} \Rightarrow m\ \right\} \\
\mathtt{delete2Trees}(\mathtt{n}, \mathtt{m}) \\
\left\{\ \alpha{\leftarrow}\varnothing * \beta{\leftarrow}\varnothing * \mathtt{n} \Rightarrow n * \mathtt{m} \Rightarrow m\ \right\}
\end{array}$$

The precondition expresses the property that we have pointers `n` and `m` to two subtrees n and m which are completely disjoint. In the postcondition both of the trees have been disposed. This disjointness property is all that is required to be able to run the two tree deletions in parallel. So, the `deletePair` program has the same specification as the `delete2Trees` program. We can construct the proof outline that demonstrates this as follows:

$$
\begin{array}{c}
\left\{\ \alpha{\leftarrow}n[\mathsf{tree}(ct_1)] * \beta{\leftarrow}m[\mathsf{tree}(ct_2)] * \mathtt{n} \Rightarrow n * \mathtt{m} \Rightarrow m\ \right\} \\
\begin{array}{c||c}
\left\{\ \alpha{\leftarrow}n[\mathsf{tree}(ct_1)] * \mathtt{n} \Rightarrow n\ \right\} & \left\{\ \beta{\leftarrow}m[\mathsf{tree}(ct_2)] * \mathtt{m} \Rightarrow m\ \right\} \\
\mathtt{deleteTree(n)} & \mathtt{deleteTree(m)} \\
\left\{\ \alpha{\leftarrow}\varnothing * \mathtt{n} \Rightarrow n\ \right\} & \left\{\ \beta{\leftarrow}\varnothing * \mathtt{m} \Rightarrow m\ \right\}
\end{array} \\
\left\{\ \alpha{\leftarrow}\varnothing * \beta{\leftarrow}\varnothing * \mathtt{n} \Rightarrow n * \mathtt{m} \Rightarrow m\ \right\}
\end{array}
$$

The Par rule also provides enough extra technology for us to reason about our 'divide and conquer' program from example 7.3. In order to specify the program we need to provide an abstract predicate that describes a binary tree in the heap. We define the binTree predicate as follows:

$$
\begin{array}{rcl}
\mathsf{binTree}(n) & \overset{\text{def}}{=} & \mathsf{emp} \wedge (n = \mathsf{null}) \\
& & \vee\ \exists x, y.\, n \mapsto x{,}y * \mathsf{binTree}(x) * \mathsf{binTree}(y)
\end{array}
$$

This predicate only describes the branch structure of a binary tree, but it would be quite simple to extend the tree with some data stored at each node. If we wanted to generalise our program, and its specification, to handle n-ary trees we would be better off using our tree module. Whilst we could provide an abstract predicate that takes a tree formula, or context formula, as a parameter, this would effectively just be encoding our tree module into the heap module. As we have already seen in chapter 6 such an encoding is not straight-forward and is also implementation dependent.

The binTree predicate is sufficient to describe the behaviour of our `parTreeDelete` program with the following specification:

$$
\begin{array}{c}
\left\{\ \mathsf{binTree}(n) * \mathtt{n} \Rightarrow n\ \right\} \\
\mathtt{parDeleteTree(n)} \\
\left\{\ \mathtt{n} \Rightarrow n\ \right\}
\end{array}
$$

We can show that this specification holds with a simple inductive proof. In the case where the input parameter $n = \mathsf{null}$ the $\mathsf{binTree}(n)$ predicate is equal to emp, so the `if` test fails and the program does noting more. This establishes that the base case of the induction holds. To prove the inductive step, we assume that the recursive calls to `parDeleteTree` satisfy the specification we are trying to prove. We can then complete the proof with the following derivation:

```
$\left\{\ \mathsf{binTree}(n) * \mathtt{n} \Rightarrow n\ \right\}$
local l, r in
  $\left\{\ \mathsf{binTree}(n) * \mathtt{n} \Rightarrow n * \mathtt{l} \Rightarrow - * \mathtt{r} \Rightarrow -\ \right\}$
  $\left\{\begin{array}{l} \mathsf{emp} \wedge (n = \mathsf{null}) \vee \exists x, y.\, n \mapsto x,y * \mathsf{binTree}(x) * \mathsf{binTree}(y) \\ * \mathtt{n} \Rightarrow n * \mathtt{l} \Rightarrow - * \mathtt{r} \Rightarrow - \end{array}\right\}$
  if n $\neq$ null then
    $\left\{\ \exists x, y.\, n \mapsto x,y * \mathsf{binTree}(x) * \mathsf{binTree}(y) * \mathtt{n} \Rightarrow n * \mathtt{l} \Rightarrow - * \mathtt{r} \Rightarrow -\ \right\}$
    l := n.left ;
    $\left\{\ \exists x, y.\, n \mapsto x,y * \mathsf{binTree}(x) * \mathsf{binTree}(y) * \mathtt{n} \Rightarrow n * \mathtt{l} \Rightarrow x * \mathtt{r} \Rightarrow y\ \right\}$
    r := n.right ;
    $\left\{\ \exists x, y.\, n \mapsto x,y * \mathsf{binTree}(x) * \mathsf{binTree}(y) * \mathtt{n} \Rightarrow n * \mathtt{l} \Rightarrow x * \mathtt{r} \Rightarrow y\ \right\}$
    $\begin{array}{c||c} \left\{\ \mathsf{binTree}(x) * \mathtt{l} \Rightarrow x\ \right\} & \left\{\ \mathsf{binTree}(y) * \mathtt{r} \Rightarrow y\ \right\} \\ \mathtt{parTreeDelete(l)} & \mathtt{parTreeDelete(r)} \\ \left\{\ \mathtt{l} \Rightarrow x\ \right\} & \left\{\ \mathtt{r} \Rightarrow y\ \right\} \end{array}$
    $\left\{\ \exists x, y.\, n \mapsto x,y * \mathtt{n} \Rightarrow n * \mathtt{l} \Rightarrow x * \mathtt{r} \Rightarrow y\ \right\}$
    dispose(n)
    $\left\{\ \exists x.\, y\mathtt{n} \Rightarrow n * \mathtt{l} \Rightarrow x * \mathtt{r} \Rightarrow y\ \right\}$
  $\left\{\ \mathtt{n} \Rightarrow n * \mathtt{l} \Rightarrow - * \mathtt{r} \Rightarrow -\ \right\}$
$\left\{\ \mathtt{n} \Rightarrow n\ \right\}$
```

Note when we pass resource to the parallel threads we are also choosing to frame off $n \mapsto x,y$ and $\mathtt{n} \Rightarrow n$ since neither of the threads requires this resource. We frame this resource back on when the parallel threads pass their resources back to the main thread. Many other divide and conquer style concurrent programs can be proven in a similar way with our reasoning system.

7.2.3 Shared Resource Concurrency

Disjoint concurrency, by design, assures that there will be no race conditions in a program. However, in practice many programs will want to share access to some data structure during their execution. In such cases we have to take more care to ensure that there are no race conditions. One common approach is to define shared resources and restrict access to these resources to be with mutual exclusion. That is, each resource may only be used by at most one thread at a time.

Definition 7.5 (Programming Language with Resources)**.** The programming language $\mathcal{L}'_{\text{CMD}}$ from definition 7.5 is further extended to the language $\mathcal{L}''_{\text{CMD}}$ by adding resource declarations and conditional critical region statements.

$$\mathbb{C} \;::=\; ... \mid \texttt{res r in } \mathbb{C} \mid \texttt{with r when } B \texttt{ do } \mathbb{C}$$

Resource declarations create a new region of mutual exclusion, called a critical region, and the `with` statements control access to these critical regions. Only one thread at a time may be inside a critical region for each resource `r`. In addition, we also require that a boolean expression B evaluates to true before a thread is allowed to access a critical region. If the expression does not evaluate to true then that thread must wait until such a time as the expression does evaluate to true. Threads which cannot enter a critical region, either due to mutual exclusion of a failed test, must busy-wait and try to access the region again later. We do not always need to provide a boolean condition to control entrance a critical region, that we are accessing the region with mutual exclusion can sometimes be enough to guarantee race freedom. In such cases we write `with r do` $\mathbb{C}$ to mean `with r when true do` $\mathbb{C}$.

Informally, we can give the semantics of these new program statements in the small-step style as above. First, we need to extend the program state γ, d, σ to include a lock environment $\rho : \textsc{Locks} \rightarrow \{\mathsf{free}, \mathsf{busy}\}$ that tracks when a resource is free or in use. The small-step style semantics can then be given as:

$$\frac{\mathtt{r} \notin \mathit{dom}(\rho)}{\mathtt{res\ r\ in}\ \mathbb{C}, \rho, \gamma, d, \sigma \rightsquigarrow \mathbb{C}, \rho[\mathtt{r} \to \mathsf{free}], \gamma, d, \sigma}$$

$$\frac{\mathtt{r} \in \mathit{dom}(\rho)}{\mathtt{res\ r\ in}\ \mathbb{C}, \rho, \gamma, d, \sigma \rightsquigarrow \lightning}$$

$$\frac{\rho(\mathtt{r}) = \mathsf{free} \quad \text{and} \quad \mathcal{B}[\![B]\!]\sigma = \mathsf{true}}{\mathtt{with\ r\ when}\ B\ \mathtt{do}\ \mathbb{C}, \rho, \gamma, d, \sigma \rightsquigarrow \mathbb{C}\ ;\ \mathtt{unlock\ r}, \rho[\mathtt{r} \to \mathsf{busy}], \gamma, d, \sigma}$$

$$\frac{\rho(\mathtt{r}) = \mathsf{busy} \quad \text{or} \quad \mathcal{B}[\![B]\!]\sigma = \mathsf{false}}{\mathtt{with\ r\ when}\ B\ \mathtt{do}\ \mathbb{C}, \rho, \gamma, d, \sigma \rightsquigarrow \mathtt{with\ r\ when}\ B\ \mathtt{do}\ \mathbb{C}, \rho, \gamma, d, \sigma}$$

$$\frac{\mathtt{r} \notin \rho}{\mathtt{with\ r\ when}\ B\ \mathtt{do}\ \mathbb{C}, \rho, \gamma, d, \sigma \rightsquigarrow \lightning}$$

$$\frac{}{\mathtt{unlock\ r}, \rho, \gamma, d, \sigma \rightsquigarrow \mathtt{skip}, \rho[\mathtt{r} \to \mathsf{free}], \gamma, d, \sigma}$$

The `res` – `in` block creates a new lock for controlling access to the resource. Each `with` – `do` block then acquires the lock, runs some commands and releases the lock. If the lock is already owned by another thread then the thread blocks until the lock is released by that thread.

In §7.3 we will give a formal treatment of the semantics of resource declaration and conditional critical regions in terms of traces. However, this description should be sufficient to understand our upcoming examples.

Extending our programming language with this more powerful form of concurrency lets us express several more common programming patterns. We consider two more example programs: one that controls read access to some shared tree node and one that uses the producer/consumer style of programming.

Example 7.6 (Shared Node Reading). As a simple example of how we can share resources between threads consider the `siblicide` program given below:

```
siblicide(n) ::= local l, r in
                   res c in
                     with c do            ‖  with c do
                       l := getLeft(n)    ‖    r := getRight(n)
                     deleteTree(l)        ‖  deleteTree(r)
```

This program runs two threads which read a value from a shared node `n` under mutual exclusion and then delete the corresponding subtree. As we have seen before, the disjointness of the two subtrees to be deleted is guaranteed by the data structure. Notice that the program does not need conditions on the critical regions in the two threads. This is because neither thread modifies the shared state and so the order in which the threads access the shared state is not important.

Example 7.7 (Producer/Consumer Pattern)**.** One common example of shared resource concurrency is that of the producer/consumer pattern. In this pattern some number of threads operate on some shared structure, such as a buffer, with some threads producing data and putting it into the shared structure and some threads taking data out of the shared structure and consuming it. We consider a program `prodCons` here, with just two threads, where one thread creates nodes and inserts them as children under some shared nodes and the other thread takes children out of the shared node and then deletes them. In practice it is likely that the second thread will actually make some use of the data it is extracting, but deletion is sufficient to establish the pattern we are interested in.

```
prodCons(p, n, m) ::=
     local c, x, y in
       c := 0 ;
       res r in
        while true do            ||  while true do
          //makedata             ||    with r when (c > 0) do
          newNodeAfter(p) ;      ||      y := getFirst(n) ;
          x := getRight(p) ;     ||      appendChild(m, y) ;
          with r do              ||      c := c − 1
            appendChild(n, x) ;  ||    //usedata
            c := c + 1           ||  c  deleteTree(y)
```

The left-hand thread repeatedly creates a new node to the right of node `p`, which is representative of producing some data. It then tries to access the shared region and when it gets access it appends the new node to the children under the shared store node `n` and increments the counter `c`. The right-hand thread repeatedly tries to access the shared node when it has at least one child ($\mathtt{c} > 0$). When it get access it removes the last node under `n`, placing it under its local node `m`, and decrements the counter `c`. It then locally (outside of the critical region) deletes this node, which is representative of consuming the data. Notice that whilst the right-hand thread may only access the critical region when there is at least one node beneath `n`, the left-hand thread is unrestricted as to when it may try and access the critical region. However, because the created data is always put onto the end of the list of children under `n` and the removed data is alway taken off of the front of the list of children under `n`, the order of data is preserved when passed through the shared state. Also notice that there is some resource transfer taking place in this program. The new nodes that are created are initially owned by the left-hand thread, but once they have been read out of the shared store they are then owned by the right-hand thread. We will see that this resource transfer is key in establishing the correctness of the program.

7.2.4 Reasoning About Shared Resource Concurrency

Just as with disjoint concurrency, we can use similar techniques to concurrent separation logic to reason about shared resource concurrency in segment logic. In order to work with shared resources we need to be able to provide resource invariants for these resources. In concurrent separation logic resource invariants describe the potential structure of some part of the heap. However, for concurrent segment logic we need more than this, we also need to know how the shared state links up with the rest of our data structure. For this reason our resource invariants must also contain a set of labels that link the resource with the rest of the data structure. We will quantify these labels with the hidden label quantification H when a thread enters a critical region and acquires access to a resource.

Definition 7.8 (Resource Environment)**.** A resource environment $\Delta \in \mathrm{R}\textsc{Env}$ is a finite partial function $\Delta : \textsc{Locks} \rightharpoonup_{\mathsf{fin}} \mathcal{P}(\mathrm{X}) \times \textsc{Pred}$ mapping resource/lock names `r` to pairs consisting of a set of labels Π and a precise predicate RI.

Recall that a segment logic predicate P is precise if, for every $e \in \textsc{Env}$, $(s, \sigma) \in \textsc{State}$ there is at most one $(s', \sigma') \in \textsc{State}$ such that $(s', \sigma') \in \mathcal{P}[\![P]\!]e$ with $s = (\bar{x})(s_0 + s')$ and $\sigma = \sigma_0 * \sigma'$ for some $\bar{x} \in \mathcal{P}_{\mathsf{fin}}(\mathcal{X})$, $s_0 \in \mathrm{S}$ and $\sigma_0 \in \Sigma$.

We can then define our inference rules that deal with our new programing constructs for shared resource reasoning.

Definition 7.9 (Shared Resource Concurrency Inference Rules)**.** The Hoare logic rules of our reasoning system, from definitions 4.13 and 7.4, are extended to include the following inference rules for resource declarations and conditional critical regions:

$$\textsc{Res}: \quad \frac{e, \Gamma, \Delta : (\mathtt{r} \to \Pi, RI) \vdash \{P\}\, \mathbb{C}\, \{Q\}}{e, \Gamma, \Delta \vdash \{\mathsf{H}\Pi.\, (P * RI)\}\, \mathtt{res\ r\ in}\ \mathbb{C}\, \{\mathsf{H}\Pi.\, (Q * RI)\}}$$

$$\text{CCR}: \quad \frac{e, \Gamma, \Delta \vdash \{\mathsf{H}\Pi'.\, (P * RI) \wedge \mathtt{B}\}\ \mathbb{C}\ \{\mathsf{H}\Pi'.\, (Q * RI)\} \quad \Pi' = \Pi \cap \mathsf{free}(P)}{e, \Gamma, \Delta : (\mathtt{r} \to \Pi, RI) \vdash \{P\}\, \mathtt{with\ r\ when}\ \mathtt{B}\ \mathtt{do}\ \mathbb{C}\, \{Q\}}$$

The existing inference rules do not interact with the resource environment Δ. We therefore treat rules that do not mention the resource environment as preserving it.

The resource declaration rule Res identifies some portion of the program state, described by RI and linked to the rest of the state by labels Π. It then passes ownership of the resource and the revelation of the labels to the shared resource `r`. The conditional critical regions rule CCR passes this ownership back to a thread when it successfully enters a critical region for `r`. Notice, however, that the thread only uses the labels that it shares with the resource (Π'). This ensures that the compression of the shared resource with the current threads resource is well defined. The CCR rules also requires that the thread is able to reestablish the resource invariant and pass ownership of it back to the resource. If the thread can not reestablish the resource invariant, then other threads might be able to access the resource in an unexpected state and the safety of their operation could not be guaranteed. Maintaining the resource invariant ensures that each thread accesses the shared resource in a consistent way.

Resource invariants can take many different forms, depending on the behaviour of the programs that share access to the resource. The simplest example of a resource invariant is a formula that describes a constant piece of state. This means that while many threads may access the shared state, none of them actually make any lasting modifications to it. To see an example of this in action, we return to our shared node reading program `siblicide` from Example 7.6. We wish to show that the `siblicide` program satisfies the following specification:

$$\left\{\ \alpha \leftarrow p[\text{tree}(ct)] \otimes n[\beta] \otimes q[\text{tree}(ct')] * \mathtt{n} \Rrightarrow n\ \right\}$$
$$\mathtt{siblicide(n)}$$
$$\left\{\ \alpha \leftarrow n[\beta] * \mathtt{n} \Rrightarrow n\ \right\}$$

Since the program makes use of a resource declaration and shared access via critical regions to this resource, we require a resource invariant for `r`. We choose to use the following label set and formula:

$$\begin{array}{rcl} \Pi & \overset{\text{def}}{=} & \{\gamma, \delta\} \\ RI & \overset{\text{def}}{=} & \alpha \leftarrow \gamma \otimes n[\beta] \otimes \delta * \mathtt{n} \Rightarrow n \end{array}$$

Notice that the invariant RI describes a fixed piece of state containing just a single node n, with its surrounding labels, and the variable $\mathtt{n}$ which maps to node identifier n. With this invariant we can then prove the specification of the program as shown by the proof sketch given in Figure 7.1.

In order to establish the choice of Π' for each thread's use of the CCR rule, notice that the free variables of the relevant predicates are:

$$\begin{array}{rcl} \mathsf{free}(\gamma \leftarrow p[\mathrm{tree}(ct)] * \mathtt{l} \Rightarrow -) & = & \{\gamma, p, ct\} \\ \mathsf{free}(\delta \leftarrow q[\mathrm{tree}(ct')] * \mathtt{r} \Rightarrow -) & = & \{\delta, q, ct'\} \end{array}$$

This example illustrates the need for the resource invariant to contain a set of labels Π as well as a formula RI. One might think that it is enough to simply join the resource's state with the threads state when it enters a CCR. However, this does not correctly account for the necessary compression of the segments that is often required to be able to reason about the code within the CCR. Consider the left hand thread in Figure 7.1. If we did not have the labels included in the resource invariant, then on entry to the CCR we would have the following formula:

$$\alpha \leftarrow \gamma \otimes n[\beta] \otimes \delta * \gamma \leftarrow p[\mathrm{tree}(ct)] * \mathtt{n} \Rightarrow n * \mathtt{l} \Rightarrow -$$

Notice that this is not enough to satisfy the precondition of the `l` := `getLeft(n)` command which requires more information about the relation between nodes p and q (namely that p is the left sibling of n):

$$\alpha \leftarrow p[\mathrm{tree}(ct)] \otimes n[\beta] \otimes \delta * \mathtt{n} \Rightarrow n * \mathtt{l} \Rightarrow -$$

We need to be able to compress the segments in order for our precondition to be in the correct form to use the axiom for `getLeft`. We use the hidden label quantification, which includes the use of revelation, to ensure the correct compression of the segments. It is important that we use hidden label quantification, and not just revelation, so that we can re-establish our resource invariant.

In the last example the shared state is not modified by the threads that access it. However, our reasoning can also handle programs that do make modifications to the shared state. As an example of this we return to our producer/consumer program

```
$\{\ \alpha{\leftarrow}p[\text{tree}(ct)] \otimes n[\beta] \otimes q[\text{tree}(ct')] * \texttt{n} \Rightarrow n\ \}$
local l, r in
  $\{\ \alpha{\leftarrow}p[\text{tree}(ct)] \otimes n[\beta] \otimes q[\text{tree}(ct')] * \texttt{n} \Rightarrow n * \texttt{l} \Rightarrow - * \texttt{r} \Rightarrow -\ \}$
  $\left\{ \begin{array}{l} \mathsf{H}\gamma,\delta.\,(\alpha{\leftarrow}\gamma \otimes n[\beta] \otimes \delta * \gamma{\leftarrow}p[\text{tree}(ct)] * \delta{\leftarrow}q[\text{tree}(ct')]) \\ * \texttt{n} \Rightarrow n * \texttt{l} \Rightarrow - * \texttt{r} \Rightarrow - \end{array} \right\}$
  res c in
    $\{\ \gamma{\leftarrow}p[\text{tree}(ct)] * \delta{\leftarrow}q[\text{tree}(ct')] * \texttt{l} \Rightarrow - * \texttt{r} \Rightarrow -\ \}$
      $\{\ \gamma{\leftarrow}p[\text{tree}(ct)] * \texttt{l} \Rightarrow -\ \}$
      with c do
        $\left\{ \begin{array}{l} \mathsf{H}\gamma.\left( \begin{array}{c} \alpha{\leftarrow}\gamma \otimes n[\beta] \otimes \delta \\ * \gamma{\leftarrow}p[\text{tree}(ct)] \end{array} \right) \\ * \texttt{n} \Rightarrow n * \texttt{l} \Rightarrow - \end{array} \right\}$
        $\left\{ \begin{array}{l} \alpha{\leftarrow}p[\text{tree}(ct)] \otimes n[\beta] \otimes \delta \\ * \texttt{n} \Rightarrow n * \texttt{l} \Rightarrow - \end{array} \right\}$
        l := getLeft(n)
        $\left\{ \begin{array}{l} \alpha{\leftarrow}p[\text{tree}(ct)] \otimes n[\beta] \otimes \delta \\ * \texttt{n} \Rightarrow n * \texttt{l} \Rightarrow p \end{array} \right\}$
        $\left\{ \begin{array}{l} \mathsf{H}\gamma.\left( \begin{array}{c} \alpha{\leftarrow}\gamma \otimes n[\beta] \otimes \delta \\ * \gamma{\leftarrow}p[\text{tree}(ct)] \end{array} \right) \\ * \texttt{n} \Rightarrow n * \texttt{l} \Rightarrow p \end{array} \right\}$
      $\{\ \gamma{\leftarrow}p[\text{tree}(ct)] * \texttt{l} \Rightarrow p\ \}$
      deleteTree(l)
      $\{\ \gamma{\leftarrow}\varnothing * \texttt{l} \Rightarrow p\ \}$
    ||
      $\{\ \delta{\leftarrow}q[\text{tree}(ct')] * \texttt{r} \Rightarrow -\ \}$
      with c do
        $\left\{ \begin{array}{l} \mathsf{H}\delta.\left( \begin{array}{c} \alpha{\leftarrow}\gamma \otimes n[\beta] \otimes \delta \\ * \delta{\leftarrow}q[\text{tree}(ct')] \end{array} \right) \\ * \texttt{n} \Rightarrow n * \texttt{r} \Rightarrow - \end{array} \right\}$
        $\left\{ \begin{array}{l} \alpha{\leftarrow}\gamma \otimes n[\beta] \otimes q[\text{tree}(ct')] \\ * \texttt{n} \Rightarrow n * \texttt{r} \Rightarrow - \end{array} \right\}$
        r := getRight(n)
        $\left\{ \begin{array}{l} \alpha{\leftarrow}\gamma \otimes n[\beta] \otimes q[\text{tree}(ct')] \\ * \texttt{n} \Rightarrow n * \texttt{r} \Rightarrow q \end{array} \right\}$
        $\left\{ \begin{array}{l} \mathsf{H}\delta.\left( \begin{array}{c} \alpha{\leftarrow}\gamma \otimes n[\beta] \otimes \delta \\ * \delta{\leftarrow}q[\text{tree}(ct')] \end{array} \right) \\ * \texttt{n} \Rightarrow n * \texttt{r} \Rightarrow q \end{array} \right\}$
      $\{\ \delta{\leftarrow}q[\text{tree}(ct')] * \texttt{r} \Rightarrow q\ \}$
      deleteTree(r)
      $\{\ \delta{\leftarrow}\varnothing * \texttt{r} \Rightarrow q\ \}$
    $\{\ \gamma{\leftarrow}\varnothing * \delta{\leftarrow}\varnothing * \texttt{l} \Rightarrow p * \texttt{r} \Rightarrow q\ \}$
  $\{\ \mathsf{H}\gamma,\delta.\,(\alpha{\leftarrow}\gamma \otimes n[\beta] \otimes \delta * \gamma{\leftarrow}\varnothing * \delta{\leftarrow}\varnothing) * \texttt{n} \Rightarrow n * \texttt{l} \Rightarrow p * \texttt{r} \Rightarrow q\ \}$
  $\{\ \alpha{\leftarrow}\varnothing \otimes n[\beta] \otimes \varnothing * \texttt{n} \Rightarrow n * \texttt{l} \Rightarrow p * \texttt{r} \Rightarrow q\ \}$
$\{\ \alpha{\leftarrow}n[\beta] * \texttt{n} \Rightarrow n\ \}$
```

Figure 7.1: Proof sketch for the `siblicide` program.

`prodCond` from Example 7.7. Notice that due to the use of the `while true` loops this program will never terminate, so its post-condition will be false. Whilst we cannot give the overall program a meaningful specification, we can still prove that the loops themselves are fault free. As with the previous example, we need to choose a resource invariant for resource `r` so we choose to use the following label set and formula:

$$\begin{aligned} \Pi &\stackrel{\text{def}}{=} \emptyset \\ RI &\stackrel{\text{def}}{=} \exists t, c.\, \beta{\leftarrow}n[\text{tree}(t)] * \mathtt{n} \Rightarrow n * \mathsf{c} \Rightarrow c \land \mathsf{len}(t) = c \end{aligned}$$

where the len predicate is defined as:

$$\begin{aligned} \mathsf{len}(\varnothing) &\stackrel{\text{def}}{=} 0 \\ \mathsf{len}(x) &\stackrel{\text{def}}{=} \text{undefined} \\ \mathsf{len}(n[ct]) &\stackrel{\text{def}}{=} 1 \\ \mathsf{len}(ct_1 \otimes ct_2) &\stackrel{\text{def}}{=} \mathsf{len}(ct_1) + \mathsf{len}(ct_2) \end{aligned}$$

In this example the label set Π is empty and the invariant RI describes a complete tree with root n, pointed to by variable $\mathtt{n}$, and a variable c, where c contains the number of children beneath n.

With this invariant we can then give the proof sketch shown in Figure 7.2. Notice that because the tree t beneath node n is always complete we know that $\mathsf{len}(t)$ will always be well-defined.

7.3 Soundness of Concurrent Segment Logic

In chapter 4 we proved soundness for our sequential reasoning framework with respect to a big-step operational semantics. We now wish to show that the inference rules that we have added to deal with concurrency are also sound. However, there is no elegant way of representing concurrency, in particular parallel threads, in the big-step semantics style. Instead we choose to provide a trace semantics, in the style of abstract separation logic [17][18], for a simplified programming language that concentrates on the concurrency constructs we have added. Since our sequential rules remain unaffected by our concurrency additions, their soundness still holds from our result in chapter 4.

Much of our setup in this section is very similar to the original abstract separation logic work it is based on. The main difference is the need to handle a logical environment and compression as we are now working with segments.

```
$\{\ \alpha{\leftarrow}p[\delta] * \beta{\leftarrow}n[\varnothing] * \gamma{\leftarrow}m[\varepsilon] * \mathtt{p} \Rightarrow p * \mathtt{n} \Rightarrow n * \mathtt{m} \Rightarrow m\ \}$
local c, x, y in
$\{\ \alpha{\leftarrow}p[\delta] * \beta{\leftarrow}n[\varnothing] * \gamma{\leftarrow}m[\varepsilon] * \mathtt{p} \Rightarrow p * \mathtt{n} \Rightarrow n * \mathtt{m} \Rightarrow m * \mathtt{c} \Rightarrow \mathsf{null} * \mathtt{x} \Rightarrow \mathsf{null} * \mathtt{y} \Rightarrow \mathsf{null}\ \}$
  c := 0 ;
$\{\ \alpha{\leftarrow}p[\delta] * \beta{\leftarrow}n[\varnothing] * \gamma{\leftarrow}m[\varepsilon] * \mathtt{p} \Rightarrow p * \mathtt{n} \Rightarrow n * \mathtt{m} \Rightarrow m * \mathtt{c} \Rightarrow 0 * \mathtt{x} \Rightarrow \mathsf{null} * \mathtt{y} \Rightarrow \mathsf{null}\ \}$
  res r in
    $\{\ \alpha{\leftarrow}p[\delta] * \gamma{\leftarrow}m[\varepsilon] * \mathtt{p} \Rightarrow p * \mathtt{m} \Rightarrow m * \mathtt{x} \Rightarrow \mathsf{null} * \mathtt{y} \Rightarrow \mathsf{null}\ \}$

  $\{\ \alpha{\leftarrow}p[\delta] * \mathtt{p} \Rightarrow p * \mathtt{x} \Rightarrow -\ \}$
  while true do
    $\{\ \alpha{\leftarrow}p[\delta] * \mathtt{p} \Rightarrow p * \mathtt{x} \Rightarrow -\ \}$
    //makedata
    newNodeAfter(p) ;
    $\{\ \exists x.\, \alpha{\leftarrow}p[\delta] \otimes x[\varnothing] * \mathtt{p} \Rightarrow p * \mathtt{x} \Rightarrow -\ \}$
    x := getRight(p) ;
    $\{\ \exists x.\, \alpha{\leftarrow}p[\delta] \otimes x[\varnothing] * \mathtt{p} \Rightarrow p * \mathtt{x} \Rightarrow x\ \}$
    with r do
      $\{\ \exists t, c, x.\, \alpha{\leftarrow}p[\delta] \otimes x[\varnothing]$
        $* \beta{\leftarrow}n[\mathrm{tree}(t)]$
        $* \mathtt{p} \Rightarrow p * \mathtt{x} \Rightarrow x$
        $* \mathtt{n} \Rightarrow n * \mathtt{c} \Rightarrow c$
        $\wedge \mathsf{len}(t) = c\ \}$
      appendChild(n, x) ;
      $\{\ \exists t, c, x.\, \alpha{\leftarrow}p[\delta]$
        $* \beta{\leftarrow}n[\mathrm{tree}(t) \otimes x[\varnothing]]$
        $* \mathtt{p} \Rightarrow p * \mathtt{x} \Rightarrow x$
        $* \mathtt{n} \Rightarrow n * \mathtt{c} \Rightarrow c$
        $\wedge \mathsf{len}(t) = c\ \}$
      c := c + 1
      $\{\ \exists t, c, x.\, \alpha{\leftarrow}p[\delta]$
        $* \beta{\leftarrow}n[\mathrm{tree}(t) \otimes x[\varnothing]]$
        $* \mathtt{p} \Rightarrow p * \mathtt{x} \Rightarrow x$
        $* \mathtt{n} \Rightarrow n * \mathtt{c} \Rightarrow c + 1$
        $\wedge \mathsf{len}(t) = c\ \}$

      $\{\ \exists t, c, x.\, \alpha{\leftarrow}p[\delta]$
        $* \beta{\leftarrow}n[\mathrm{tree}(t)]$
        $* \mathtt{p} \Rightarrow p * \mathtt{x} \Rightarrow x$
        $* \mathtt{n} \Rightarrow n * \mathtt{c} \Rightarrow c$
        $\wedge \mathsf{len}(t) = c\ \}$
    $\{\ \exists x.\, \alpha{\leftarrow}p[\delta] * \mathtt{p} \Rightarrow p * \mathtt{x} \Rightarrow x\ \}$
    $\{\ \alpha{\leftarrow}p[\delta] * \mathtt{p} \Rightarrow p * \mathtt{x} \Rightarrow -\ \}$
  $\{\ \mathsf{false}\ \}$

||

  $\{\ \gamma{\leftarrow}m[\varepsilon] * \mathtt{m} \Rightarrow m * \mathtt{y} \Rightarrow -\ \}$
  while true do
    $\{\ \gamma{\leftarrow}m[\varepsilon] * \mathtt{m} \Rightarrow m * \mathtt{y} \Rightarrow -\ \}$
    with r when (c > 0) do
      $\{\ \exists t, c.\, \gamma{\leftarrow}m[\varepsilon]$
        $* \beta{\leftarrow}n[\mathrm{tree}(t)]$
        $* \mathtt{m} \Rightarrow m * \mathtt{y} \Rightarrow -$
        $* \mathtt{n} \Rightarrow n * \mathtt{c} \Rightarrow c$
        $\wedge \mathsf{len}(t) = c \wedge c > 0\ \}$
      y := getFirst(n) ;
      $\{\ \exists a, t', t'', c.\, \gamma{\leftarrow}m[\varepsilon]$
        $* \beta{\leftarrow}n[\mathrm{tree}(a[t']) \otimes \mathrm{tree}(t'')]$
        $* \mathtt{m} \Rightarrow m * \mathtt{y} \Rightarrow a$
        $* \mathtt{n} \Rightarrow n * \mathtt{c} \Rightarrow c$
        $\wedge \mathsf{len}(t'') = c - 1\ \}$
      appendChild(m, y) ;
      $\{\ \exists a, t', t'', c.\, \gamma{\leftarrow}m[\varepsilon \otimes \mathrm{tree}(a[t'])]$
        $* \beta{\leftarrow}n[\mathrm{tree}(t'')]$
        $* \mathtt{m} \Rightarrow m * \mathtt{y} \Rightarrow a$
        $* \mathtt{n} \Rightarrow n * \mathtt{c} \Rightarrow c$
        $\wedge \mathsf{len}(t'') = c - 1\ \}$
      c := c − 1
      $\{\ \exists a, t', t'', c.\, \gamma{\leftarrow}m[\varepsilon \otimes \mathrm{tree}(a[t'])]$
        $* \beta{\leftarrow}n[\mathrm{tree}(t'')]$
        $* \mathtt{m} \Rightarrow m * \mathtt{y} \Rightarrow a$
        $* \mathtt{n} \Rightarrow n * \mathtt{c} \Rightarrow c - 1$
        $\wedge \mathsf{len}(t'') = c - 1\ \}$
    $\{\ \exists a, t'.\, \gamma{\leftarrow}m[\varepsilon \otimes \mathrm{tree}(a[t'])]$
      $* \mathtt{m} \Rightarrow m * \mathtt{y} \Rightarrow a\ \}$
    //usedata
    deleteTree(y)
    $\{\ \exists a.\, \gamma{\leftarrow}m[\varepsilon] * \mathtt{m} \Rightarrow m * \mathtt{y} \Rightarrow a\ \}$
    $\{\ \gamma{\leftarrow}m[\varepsilon] * \mathtt{m} \Rightarrow m * \mathtt{y} \Rightarrow -\ \}$
  $\{\ \mathsf{false}\ \}$

$\{\ \mathsf{false}\ \}$
```

Figure 7.2: Proof sketch for the `prodCons` program.

The programs of our simplified programming language are constructed as follows:

$$\mathbb{C} \;::=\; \varphi \mid \texttt{skip} \mid \mathbb{C}\,;\mathbb{C} \mid \mathbb{C}+\mathbb{C} \mid \mathbb{C}^* \mid \mathbb{C}||\mathbb{C} \mid \texttt{res r in } \mathbb{C} \mid \texttt{with r do } \mathbb{C}$$

where $\varphi \in \textsc{Cmd}$, ; is sequential composition, + is nondeterministic choice and $(\cdot)^*$ is Kleene-star (iterated ;). We use + and $(\cdot)^*$ instead of conditionals and while loops and omit the test on a `with` region to avoid explicitly considering boolean conditions. We also drop our program constructs for procedures and local variables. These choices all simplify our proof and allow us to concentrate on the soundness of the rules for our new concurrency constructs. Remember that we have already shown our sequential rules to be sound, we only want a soundness result for our concurrency constructs. It should not be too onerous to extend the results presented here to our full programming language.

We take the state of a program to be given by a pair consisting of a segment algebra $\mathcal{S}(\mathcal{M}_{\textsc{Store}})$ and a variable store Σ. i.e. $\textsc{State} = \mathcal{S}(\mathcal{M}_{\textsc{Store}}) \times \Sigma$ as in our axiomatic semantics given in chapter 4. As before, we also evaluate predicates to elements of the powerset $\mathcal{P}(\textsc{State})$. For ease of notation, we lift operations on programs states to sets of power states. That is, for $p, q \in \mathcal{P}(\textsc{State})$,

$$\begin{aligned} p + q &\overset{\text{def}}{=} \{(s_1 + s_2, \sigma_1 \uplus \sigma_2) \mid (s_1, \sigma_1) \in p \text{ and } (s_2, \sigma_2) \in q\} \\ (x)(p) &\overset{\text{def}}{=} \{((x)(s), \sigma) \mid (s, \sigma) \in p\} \end{aligned}$$

Following the style of the abstract separation logic work, we extend the powerset $\mathcal{P}(\textsc{State})$ with a new fault element $\top$ to enable us to model the semantics of programs as functions. Conceptually, we treat faulting as an inconsistent, or over determined value.

Definition 7.10 (States with Faults)**.** The powerset of program states including fault $\mathcal{P}(\textsc{State})^\top$ is obtained by adding a new greatest element $\top$ to $\mathcal{P}(\textsc{State})$ such that for all $p \in \mathcal{P}(\textsc{State})$, $p + \top = \top = \top + p$ and for all $x \in \mathrm{X}$, $(x)(\top) = \top$

In our semantics we treat programs as functions $f : \textsc{State} \to \mathcal{P}(\textsc{State})^\top$.

Definition 7.11 (Semantic Hoare Triple)**.** If $p, q \in \mathcal{P}(\textsc{State})$ and $f : \textsc{State} \to \mathcal{P}(\textsc{State})^\top$ then

$$\{p\}\, f\, \{q\} \text{ holds} \;\Leftrightarrow\; \text{for all } (s, \sigma) \in p.\, f(s, \sigma) \subseteq q$$

Note that this is a fault-avoiding interpretation as the postcondition q does not include the $\top$ element. We can then describe what it means for a function f to be a local action.

Definition 7.12 (Local Action)**.** A *local action* f : $\textsc{State} \to \mathcal{P}(\textsc{State})^\top$ is a function satisfying the following locality condition:
For any two disjoint program states $(s_1, \sigma_1), (s_2, \sigma_2) \in \textsc{State}$ and $\bar{x} \subseteq \mathrm{X}$,

$$f((\bar{x})(s_1 + s_2), \sigma_1 \uplus \sigma_2) \;\; \subseteq \;\; (\bar{x})(f(s_1, \sigma_1) + \{(s_2, \sigma_2)\}).$$

The set of local actions is denoted $\textsc{Lact}$.

Given any precondition p and postcondition q, we can define the best, or largest, local action satisfying the triple $\{p\} - \{q\}$.

Definition 7.13 (Best Local Action)**.** The *best local action* $\mathsf{bla}[p, q]$ is the function of type $\textsc{State} \to \mathcal{P}(\textsc{State})$ defined by,

$$\mathsf{bla}[p,q](s,\sigma) = \left\{ (\bar{x})(q + \{(s_2,\sigma_2)\}) \;\middle|\; \begin{array}{l} s = (\bar{x})(s_1 + s_2) \text{ and } \sigma = \sigma_1 \uplus \sigma_2 \\ \text{and } (s_1, \sigma_1) \in p \end{array} \right\}$$

7.3.1 Syntactic Trace Model

We define an interleaving semantics based on action traces. This is a completely syntactic model that resolves all of the occurrences of concurrency. To provide a semantics for the behaviour of our programs, we give an execution model the runs a trace on a give state. Each trace will be made up from the basic commands of our module, along with two additional lock and unlock operations used to model entry to and exit from critical regions. In order to define our trace semantics we require the notion of an atomic action. Recall from §7.1, that an atomic action is some indivisible unit of computation that occurs without any possible interference

Definition 7.14 (Atomic Action)**.** An *atomic action* $a \in \mathcal{A}$ is either a basic command, skip, a race check, a lock command, or an unlock command.

$$a \;\; ::= \;\; \varphi \mid \texttt{skip} \mid \texttt{check}(\varphi, \varphi) \mid \texttt{lock(r)} \mid \texttt{unlock(r)}$$

Notation: we refer to the `lock(r)` and `unlock(r)` commands (for a particular `r`) as `r`-actions.

Definition 7.15 (Trace)**.** A *trace* τ is a sequence of atomic actions:

$$\tau \;\; ::= \;\; a \,;\, \ldots \,;\, a$$

Notation: we write ϵ for the empty trace, $\tau - \mathtt{r}$ for the trace obtained by removing all $\mathtt{r}$-actions from τ and $\tau|_{\mathtt{r}}$ for the trace obtained by removing all non $\mathtt{r}$-actions from τ.

Definition 7.16 (Synchronised Trace)**.** A trace is $\mathtt{r}$-synchronised if $\tau|_{\mathtt{r}}$ is an element of the regular language $(\mathtt{lock(r)} \mathbin{;} \mathtt{unlock(r)})^*$.

We now define how to generate a set of traces for a program written in our language.

Definition 7.17 (Trace Semantics)**.** The set of traces of a program $\mathbb{C}$, denoted $T(\mathbb{C})$, is defined as follows:

$$\begin{array}{rcl}
T(\varphi) & \overset{\text{def}}{=} & \{\varphi\} \\
T(\mathtt{skip}) & \overset{\text{def}}{=} & \{\mathtt{skip}\} \\
T(\mathbb{C}_1 \mathbin{;} \mathbb{C}_2) & \overset{\text{def}}{=} & \{\tau_1 \mathbin{;} \tau_2 \mid \tau_1 \in T(\mathbb{C}_1) \text{ and } \tau_2 \in T(\mathbb{C}_2)\} \\
T(\mathbb{C}_1 + \mathbb{C}_2) & \overset{\text{def}}{=} & T(\mathbb{C}_1) \cup T(\mathbb{C}_2) \\
T(\mathbb{C}^*) & \overset{\text{def}}{=} & (T(\mathbb{C}))^* \\
T(\mathbb{C}_1 || \mathbb{C}_2) & \overset{\text{def}}{=} & \{\mathsf{zip}(\tau_1, \tau_2) \mid \tau_1 \in T(\mathbb{C}_1) \text{ and } \tau_2 \in T(\mathbb{C}_2)\} \\
T(\mathtt{res\ r\ in\ } \mathbb{C}) & \overset{\text{def}}{=} & \{(\mathtt{unlock(r)} \mathbin{;} \tau \mathbin{;} \mathtt{lock(r)}) - \mathtt{r} \mid \tau \in T(\mathbb{C}) \text{ is } \mathtt{r}\text{-synchronised}\} \\
T(\mathtt{with\ r\ do\ } \mathbb{C}) & \overset{\text{def}}{=} & \{\mathtt{lock(r)} \mathbin{;} \tau \mathbin{;} \mathtt{unlock(r)} \mid \tau \in T(\mathbb{C})\}
\end{array}$$

where $\mathsf{zip}(\tau_1, \tau_2)$ and its auxiliary $\mathsf{zip}'(\tau_1, \tau_2)$ are defined as:

$$\begin{array}{rcl}
\mathsf{zip}(\epsilon, \tau) & \overset{\text{def}}{=} & \tau \\
\mathsf{zip}(\tau, \epsilon) & \overset{\text{def}}{=} & \tau \\
\mathsf{zip}(\varphi_1 \mathbin{;} \tau_1, \varphi_2 \mathbin{;} \tau_2) & \overset{\text{def}}{=} & \mathsf{check}(\varphi_1, \varphi_2) \mathbin{;} \mathsf{zip}'(\varphi_1 \mathbin{;} \tau_1, \varphi_2 \mathbin{;} \tau_2) \\
\mathsf{zip}(\mathsf{com} \mathbin{;} \tau_1, \tau_2) & \overset{\text{def}}{=} & \mathsf{zip}'(\mathsf{com} \mathbin{;} \tau_1, \tau_2) \\
\mathsf{zip}(\tau_1, \mathsf{com} \mathbin{;} \tau_2) & \overset{\text{def}}{=} & \mathsf{zip}'(\tau_1, \mathsf{com} \mathbin{;} \tau_2)
\end{array}$$

$$\begin{array}{rcl}
\mathsf{zip}'(\epsilon, \tau) & \overset{\text{def}}{=} & \tau \\
\mathsf{zip}'(\tau, \epsilon) & \overset{\text{def}}{=} & \tau \\
\mathsf{zip}'(a_1 \mathbin{;} \tau_1, a_2 \mathbin{;} \tau_2) & \overset{\text{def}}{=} & (a_1 \mathbin{;} \mathsf{zip}(\tau_1, a_2 \mathbin{;} \tau_2)) \cup (a_2 \mathbin{;} \mathsf{zip}(a_1 \mathbin{;} \tau_1, \tau_2))
\end{array}$$

and where $\mathsf{com} ::= \mathtt{skip} \mid \mathtt{lock(r)} \mid \mathtt{unlock(r)}$.

Most of the above cases should be straightforward. The semantics of $\mathtt{res\ r\ in\ } \mathbb{C}$ starts with an $\mathtt{unlock(r)}$ and ends with a $\mathtt{lock(r)}$ to model the idea that when we declare a lock we pass some state into the resource that $\mathtt{r}$ holds, and when we destroy the lock we release this resource. The semantics of $\mathtt{with\ r\ do\ } \mathbb{C}$ just inserts `lock` (`r`) and `unlock` (`r`) commands before and after $\mathbb{C}$. The traces of $\mathbb{C}_1||\mathbb{C}_2$ are interleavings of each thread, except that whenever any two primitive actions may try to execute at the same time we insert a race check. Note that races are not detected at this stage, but they will be detected by the evaluation of check statements when we execute the traces.

From this point we choose to concentrate on `r`-synchronised traces as these capture all of the possible traces generated by well behaved programs. Any `lock` will have a matching `unlock` in the trace due to the way these actions are generated from the `with` regions from our programming language. `r`-synchronised traces do not capture nested regions for the same resource `r`, however, in such a case the inner region could never be executed, so the corresponding program would be non-terminating.

7.3.2 Executing Traces

An individual trace is just a sequence of simple commands. We choose to describe the behaviour of a trace in terms of a denotational semantics.

Assume we have a valuation $v : \textsc{Cmd} \to \textsc{LAct}$ that maps the basic commands to local actions which express their behaviour. Also assume that for all basic commands $\varphi \in \textsc{Cmd}$, $v(\varphi)$ satisfies all of the axioms in the set $\textsc{Ax}[\![\varphi]\!]$. That is, for all $e \in \textsc{Env}$ and $(P, Q) \in \textsc{Ax}[\![\varphi]\!]$ we have $\{\mathcal{P}[\![P]\!]e\}\, v(\varphi)\, \{\mathcal{P}[\![Q]\!]e\}$.

Definition 7.18 (Trace Execution)**.** The denotational semantics of trace execution is given as follows:

$$\begin{aligned}
[\![\varphi]\!]v, e, \Delta &\overset{\text{def}}{=} v(\varphi)\\
[\![\texttt{skip}]\!]v, e, \Delta, (s, \sigma) &\overset{\text{def}}{=} \{(s, \sigma)\}\\
[\![\mathbb{C}_1 \,;\, \mathbb{C}_2]\!]v, e, \Delta &\overset{\text{def}}{=} ([\![\mathbb{C}_1]\!]v, e, \Delta) \bullet ([\![\mathbb{C}_2]\!]v, e, \Delta)\\
[\![\texttt{lock}(\texttt{r})]\!]v, e, \Delta &\overset{\text{def}}{=} \textstyle\bigvee_P \mathsf{bla}[\mathcal{P}[\![P]\!]e, \mathcal{P}[\![\Delta(\texttt{r}) \circ P]\!]e]\\
[\![\texttt{unlock}(\texttt{r})]\!]v, e, \Delta &\overset{\text{def}}{=} \textstyle\bigvee_P \mathsf{bla}[\mathcal{P}[\![\Delta(\texttt{r}) \circ P]\!]e, \mathcal{P}[\![P]\!]e]\\
[\![\mathsf{check}(\varphi_1, \varphi_2)]\!]v, e, \Delta &\overset{\text{def}}{=} \mathsf{raceChk}(v(\varphi_1), v(\varphi_2))
\end{aligned}$$

where the composition $f \bullet g$ functionally composes f with the obvious lifting $g\!\uparrow$: $\mathcal{P}(\textsc{State})^\top \to \mathcal{P}(\textsc{State})^\top$, the resource composition $\Delta(\texttt{r}) \circ P$ is defined as:

$$\Delta(\texttt{r}) \circ P \overset{\text{def}}{=} \begin{cases} \mathsf{H}\Pi'.\,(P * RI) & \text{if } \Delta(\texttt{r}) = (\Pi, RI) \text{ and } \Pi' = \Pi \cap \mathsf{free}(P)\\ \text{undefined} & \text{otherwise} \end{cases}$$

and the race check function $\mathsf{raceChk}(f, g)$ is defined as:

$$\mathsf{raceChk}(f,g)(s,\sigma) \stackrel{\text{def}}{=} \begin{cases} \{(s,\sigma)\} & \text{if } \exists \bar{x}, s_1, s_2, \sigma_1, \sigma_2. \\ & \quad s = (\bar{x})(s_1 + s_2) \text{ and } \sigma = \sigma_1 \uplus \sigma_2 \\ & \quad \text{and } f(s_1,\sigma_1) \neq \top \text{ and } g(s_2,\sigma_2) \neq \top \\ \top & \text{otherwise} \end{cases}$$

The local action $\mathsf{raceChk}(f, g)$ faults if there is no partition of the program state into disjoint components which are sufficient to run f and g without faulting. If there is sufficient state for both actions to run disjointly then $\mathsf{raceChk}(f, g)$ simply returns the input state. The race check function is used to convert races into faults.

7.3.3 Soundness

Having set up our trace semantics we can now turn to proving the soundness of our concurrency reasoning rules. First we need to define what it means to relate our axiomatic reasoning system to our trace model.

Definition 7.19 (Semantic Consequence Relation)**.** Given a set of traces S, we define the semantics $[\![S]\!]v, e, \Delta \stackrel{\text{def}}{=} \bigvee_{\tau \in S} [\![\tau]\!]v, e, \Delta$. We then write

$$e, \Delta \vDash \{P\}\, \mathbb{C}\, \{Q\}$$

to mean that for all valuations v that satisfy the axioms of our basic commands, $\{\mathcal{P}[\![P]\!]e\}\ [\![T(\mathbb{C})]\!]v, e, \Delta\ \{\mathcal{P}[\![Q]\!]e\}$ holds.

In order to prove the soundness of our rules for concurrency, we require the following lemmas.

Lemma 7.20 (Zip)**.** If $s = s_1 + s_2$ and $\sigma = \sigma_1 \uplus \sigma_2$ with $[\![\tau_1]\!]v, e, \Delta, (s_1, \sigma_1) \subseteq \mathcal{P}[\![Q_1]\!]e$ and $[\![\tau_2]\!]v, e, \Delta, (s_2, \sigma_2) \subseteq \mathcal{P}[\![Q_2]\!]e$ and if $\tau = \mathsf{zip}(\tau_1, \tau_2)$ or $\tau = \mathsf{zip}'(\tau_1, \tau_2)$, then $[\![\tau]\!]v, e, \Delta, (s, \sigma) \subseteq \mathcal{P}[\![Q_1 * Q_2]\!]e$.

Proof. The proof is by induction on the definition of zip and zip'. Most of the cases are trivial, but there are two interesting cases.

The first interesting case is the race checking case of zip. Consider $\tau_1 = \varphi_1 \ ; \tau_1'$ and $\tau_2 = \varphi_2 \ ; \tau_2'$. Then $\tau = \mathsf{check}(\varphi_1, \varphi_2) \ ; \tau'$ for some $\tau' \in \mathsf{zip}'(\tau_1, \tau_2)$. By assumption $[\![\varphi_1 \ ; \ \tau_1']\!]v, e, \Delta, (s_1, \sigma_1) \subseteq \mathcal{P}[\![Q_1]\!]e$ and $[\![\tau_2]\!]v, e, \Delta, (s_2, \sigma_2) \subseteq \mathcal{P}[\![Q_2]\!]e$, so we have $[\![\varphi_1]\!]v, e, \Delta, (s_1, \sigma_1) \neq \top$ and $[\![\varphi_2]\!]v, e, \Delta, (s_2, \sigma_2) \neq \top$. Hence,

$$\mathsf{raceChk}(([\![\varphi_1]\!], v, e, \Delta), ([\![\varphi_2]\!]v, e, \Delta))(s_1 + s_2, \sigma_1 \uplus \sigma_2) \;=\; (s_1 + s_2, \sigma_1 \uplus \sigma_2) \;\neq\; \top$$

That is, $[\![\mathsf{check}(\varphi_1, \varphi_2)]\!]v, e, \Delta, (s, \sigma) = (s, \sigma)$.

By the induction hypothesis we have that $[\![\tau']\!]v, e, \Delta, (s, \sigma) \subseteq \mathcal{P}[\![Q_1 * Q_2]\!]e$ and the conclusion $[\![\mathsf{check}(\varphi_1, \varphi_2) \mathbin{;} \tau']\!]v, e, \Delta, (s, \sigma) \subseteq \mathcal{P}[\![Q_1 * Q_2]\!]e$ follows directly from this.

The other interesting case is the interleaving case of zip'. Consider $\tau_1 = a_1 \mathbin{;} \tau_1'$ and $\tau_2 = a_1 \mathbin{;} \tau_2'$, and suppose that $\tau \in (a_1 \mathbin{;} \mathsf{zip}(\tau_1', a_2 \mathbin{;} \tau_2'))$ (the other case is symmetrical). Then there is some $\tau' \in \mathsf{zip}(\tau_1', a_2 \mathbin{;} \tau_2')$ with $\tau = a_1 \mathbin{;} \tau'$.

By assumption $[\![a_1 \mathbin{;} \tau_1']\!]v, e, \Delta, (s_1, \sigma_1) \subseteq \mathcal{P}[\![Q_1]\!]e$, so $[\![\tau_1']\!]v, e, \Delta, (s_1', \sigma_1') \subseteq \mathcal{P}[\![Q_1]\!]e$ for each $(s_1', \sigma_1') \in v(a_1)(s_1, \sigma_1)$, where $v(a_1)(s_1, \sigma_1) \neq \top$, by the denotational semantics of sequential composition and the fact that $\mathcal{P}[\![Q_1]\!]e \neq \top$.

By the induction hypothesis, for any such (s_1', σ_1') we have that $[\![\tau']\!]v, e, \Delta, (s_1' + s_2, \sigma_1' \uplus \sigma_2) \subseteq \mathcal{P}[\![Q_1 * Q_2]\!]e$. Since a_1 must satisfy the locality condition we have $v(a_1)(s_1 + s_2, \sigma_1 \uplus \sigma_2) \subseteq v(a_1)(s_1, \sigma_1) + \{(s_1, \sigma_2)\}$ and so by the denotational semantics for sequential composition we can obtain $[\![\tau]\!]v, e, \Delta(s, \sigma) \subseteq \mathcal{P}[\![Q_1 * Q_2]\!]e$ as required. $\square$

Lemma 7.21 (r-Sync). If τ is an $\mathtt{r}$-synchronised trace,

$$[\![\tau - \mathtt{r}]\!]v, e, \Delta \quad \subseteq \quad [\![\mathtt{unlock}(\mathtt{r}) \mathbin{;} \tau \mathbin{;} \mathtt{lock}(\mathtt{r})]\!]v, e, \Delta$$

Proof. Before we prove the lemma we first choose to prove an additional property. For any local action f,

$$f \quad \subseteq \quad ([\![\mathtt{unlock}(\mathtt{r})]\!]v, e, \Delta) \bullet f \bullet ([\![\mathtt{lock}(\mathtt{r})]\!]v, e, \Delta)$$

We prove the inclusion for all (s, σ). If $[\![\mathtt{unlock}(\mathtt{r})]\!]v, e, \Delta, (s, \sigma) = \top$ the conclusion is immediate. Otherwise, let $s = (\bar{x})(s_1 + s_2)$, $\sigma = \sigma_1 \uplus \sigma_2$ and $\Delta(\mathtt{r}) = (\Pi, RI)$ with $e(\Pi) = \bar{x}$ and $(s_2, \sigma_2) \in \mathcal{P}[\![RI]\!]e$. We can then show the following:

$$\begin{array}{rcl}
f(s, \sigma) & = & f((\bar{x})(s_1 + s_2), \sigma_1 \uplus \sigma_2) \\
& \subseteq & (\bar{x})(f(s_1, \sigma_1) + \{(s_2, \sigma_2)\}) \\
& = & (f \bullet ([\![\mathtt{lock}(\mathtt{r})]\!]v, e, \Delta))(s_1, \sigma_1) \\
& \subseteq & (([\![\mathtt{unlock}(\mathtt{r})]\!]v, e, \Delta) \bullet f \bullet ([\![\mathtt{lock}(\mathtt{r})]\!]v, e\Delta))((\bar{x})(s_1 + s_2), \sigma_1 \uplus \sigma_2) \\
& = & (([\![\mathtt{unlock}(\mathtt{r})]\!]v, e, \Delta) \bullet f \bullet ([\![\mathtt{lock}(\mathtt{r})]\!]v, e, \Delta))(s, \sigma)
\end{array}$$

The proof of the lemma is by induction on the length of τ.

If τ does not contain any $\mathtt{r}$-actions, then $\tau - \mathtt{r} = \tau$. Now $[\![\tau]\!]v, e, \Delta$ is a local action, so we can show:

$$\begin{array}{rcl} [\![\tau - \mathtt{r}]\!]v, e, \Delta & = & [\![\tau]\!]v, e, \Delta \\ & \subseteq & ([\![\mathtt{unlock(r)}]\!]v, e, \Delta) \bullet [\![\tau]\!]v, e, \Delta \bullet ([\![\mathtt{lock(r)}]\!]v, e, \Delta) \\ & = & [\![\mathtt{unlock(r)}\ ;\ \tau\ ;\ \mathtt{lock(r)}]\!]v, e, \Delta \end{array}$$

The inclusion step follows from the property given above.

If τ does contain some $\mathtt{r}$-actions then, because τ is $\mathtt{r}$-synchronised by our assumption, τ must be of the form $\tau_1\ ;\ \mathtt{lock(r)}\ ;\ \tau_2\ ;\ \mathtt{unlock(r)}\ ;\ \tau'$ where τ_1 and τ_2 do not contain any $\mathtt{r}$-actions and τ' is $\mathtt{r}$-synchronised. Following the same argument as the base case we have

$$[\![\tau_1]\!]v, e, \Delta \quad \subseteq \quad [\![\mathtt{unlock(r)}\ ;\ \tau_1\ ;\ \mathtt{lock(r)}]\!]v, e, \Delta$$

and by the induction hypothesis, we also have

$$[\![\tau' - \mathtt{r}]\!]v, e, \Delta \quad \subseteq \quad [\![\mathtt{unlock(r)}\ ;\ \tau'\ ;\ \mathtt{lock(r)}]\!]v, e, \Delta$$

We can then show the following:

$$\begin{array}{l} [\![\tau - \mathtt{r}]\!]v, e, \Delta \\ \quad = [\![\tau_1\ ;\ \tau_2\ ;\ (\tau' - \mathtt{r})]\!]v, e, \Delta \\ \quad = [\![\tau_1]\!]v, e, \Delta \bullet [\![\tau_2]\!]v, e, \Delta \bullet [\![\tau' - \mathtt{r}]\!]v, e, \Delta \\ \quad \subseteq [\![\mathtt{unlock(r)}\ ;\ \tau_1\ ;\ \mathtt{lock(r)}]\!]v, e, \Delta \bullet [\![\tau_2]\!]v, e, \Delta \bullet [\![\tau' - \mathtt{r}]\!]v, e, \Delta \\ \quad \subseteq [\![\mathtt{unlock(r)}\ ;\ \tau_1\ ;\ \mathtt{lock(r)}]\!]v, e, \Delta \bullet [\![\tau_2]\!]v, e, \Delta \bullet [\![\mathtt{unlock(r)}\ ;\ \tau'\ ;\ \mathtt{lock(r)}]\!]v, e, \Delta \\ \quad = [\![\mathtt{unlock(r)}\ ;\ \tau_1\ ;\ \mathtt{lock(r)}\ ;\ \tau_2\ ;\ \mathtt{unlock(r)}\ ;\ \tau'\ ;\ \mathtt{lock(r)}]\!]v, e, \Delta \\ \quad = [\![\mathtt{unlock(r)}\ ;\ \tau\ ;\ \mathtt{lock(r)}]\!]v, e, \Delta \end{array}$$

□

We are now able to establish that our reasoning rules for concurrency preserve validity.

Theorem 7.22 (Soundness)**.** For all $e \in \textsc{Env}$, $\Delta \in \textsc{REnv}$, $P, Q \in \textsc{Pred}$ and $\mathbb{C} \in \mathcal{L}_{\textsc{Cmd}}$,

$$e, \Delta \vdash \{P\}\, \mathbb{C}\, \{Q\} \quad \Longrightarrow \quad e, \Delta \vDash \{P\}\, \mathbb{C}\, \{Q\}$$

Proof. The proof is by induction on the derivation of $e, \Delta \vdash \{P\}\, \mathbb{C}\, \{Q\}$. For the sequential rules of our framework the proof is straightforward. We concentrate on our rules for concurrency.

PAR case:

Assume $e, \Delta \vDash \{P_1\}\, \mathbb{C}_1 \,\{Q_1\}$ and $e, \Delta \vDash \{P_2\}\, \mathbb{C}_2 \,\{Q_2\}$. We need to show that $e, \Delta \vDash \{P_1 * P_2\}\, \mathbb{C}_1 || \mathbb{C}_2 \,\{Q_1 * Q_2\}$. Consider a valuation v that satisfies the axioms of our basic commands and a trace $\tau \in T(\mathbb{C}_1 || \mathbb{C}_2)$. We need to show that $\{\mathcal{P}[\![P_1 * P_2]\!]e\}\, [\![\tau]\!]v, e, \Delta \,\{\mathcal{P}[\![Q_1 * Q_2]\!]e\}$ holds. Take (s, σ) with $s = s_1 + s_2$ and $\sigma = \sigma_1 \uplus \sigma_2$ such that $(s_1, \sigma_1) \in \mathcal{P}[\![P_1]\!]e$ and $(s_2, \sigma_2) \in \mathcal{P}[\![P_2]\!]e$. We need to show that $[\![\tau]\!]v, e, \Delta, (s, \sigma) \subseteq \mathcal{P}[\![Q_1 * Q_2]\!]e$.

Since $\tau \in T(\mathbb{C}_1 || \mathbb{C}_2)$ we have $\tau = \mathsf{zip}(\tau_1, \tau_2)$ for some $\tau_1 \in T(\mathbb{C}_1)$ and $\tau_2 \in T(\mathbb{C}_2)$. By our assumption $[\![\tau_1]\!]v, e, \Delta, (s_1, \sigma_1) \subseteq \mathcal{P}[\![Q_1]\!]e$ and $[\![\tau_2]\!]v, e, \Delta, (s_2, \sigma_2) \subseteq \mathcal{P}[\![Q_2]\!]e$, so Lemma 7.20 gives us $[\![\tau]\!]v, e, \Delta, (s, \sigma) \subseteq \mathcal{P}[\![Q_1 * Q_2]\!]e$ as required.

RES case:

Assume $e, \Delta{:}(\mathtt{r} \mapsto \Pi, RI) \vDash \{P\}\, \mathbb{C} \,\{Q\}$. We need to show that $e, \Delta \vDash \{\mathsf{H}\Pi.\, (P * RI)\}\, \mathtt{res\ r\ in}\ \mathbb{C} \,\{\mathsf{H}\Pi.\, (Q * RI)\}$. Consider a valuation v that satisfies the axioms of our basic commands and a trace $\tau \in T(\mathtt{res\ r\ in}\ \mathbb{C})$. We need to show that $\{\mathcal{P}[\![\mathsf{H}\Pi.\, (P * RI)]\!]e\}\, [\![\tau]\!]v, e, \Delta \,\{\mathcal{P}[\![\mathsf{H}\Pi.\, (Q * RI)]\!]e\}$ holds. Take $(s, \sigma) \in \mathcal{P}[\![\mathsf{H}\Pi.\, (P * RI)]\!]e$ then $s = (\bar{x})(s_0 + s')$ and $\sigma = \sigma_0 \uplus \sigma'$ for $e(\Pi) = \bar{x}$, s_0 and σ_0 with $(s', \sigma') \in \mathcal{P}[\![P]\!]e$. We need to show that $[\![\tau]\!]v, e, \Delta, (s, \sigma) \in \mathcal{P}[\![\mathsf{H}\Pi.\, (Q * RI)]\!]e$.

Since $\tau \in T(\mathtt{res\ r\ in}\ \mathbb{C})$ we have $\tau = (\mathtt{unlock(r)}\ ;\ \tau'\ ;\ \mathtt{lock(r)}) - \mathtt{r}$ for some $\mathtt{r}$-synchronised trace $\tau' \in T(\mathbb{C})$. Now we know by our initial assumption that for $(s', \sigma') \in \mathcal{P}[\![P]\!]e$ we have $[\![\tau']\!]v, e, \Delta{:}(\mathtt{r} \mapsto \Pi, RI), (s', \sigma') \in \mathcal{P}[\![Q]\!]e$. By the semantics of $\mathtt{lock(r)}$ and $\mathtt{unlock(r)}$ we can deduce that $[\![\mathtt{unlock(r)}\ ;\ \tau'\ ;\ \mathtt{lock(r)}]\!]v, e, \Delta{:}(\mathtt{r} \mapsto \Pi, RI), (s, \sigma) \subseteq \mathcal{P}[\![\mathsf{H}\Pi.\, (Q * RI)]\!]e$. Now Lemma 7.21 gives

$$[\![\tau' - \mathtt{r}]\!]v, e, \Delta : (\mathtt{r} \mapsto \Pi, RI) \quad \subseteq \quad [\![\mathtt{unlock(r)}\ ;\ \tau'\ ;\ \mathtt{lock(r)}]\!]v, e, \Delta : (\mathtt{r} \mapsto \Pi, RI)$$

and since $\tau' - \mathtt{r}$ contains no $\mathtt{r}$-actions we know that $[\![\tau' - \mathtt{r}]\!]v, e, \Delta{:}(\mathtt{r} \mapsto \Pi, RI) = [\![\tau' - \mathtt{r}]\!]v, e, \Delta$. Finally, we observe that $\tau = (\mathtt{unlock(r)}\ ;\ \tau'\ ;\ \mathtt{lock(r)}) - \mathtt{r} = (\tau' - \mathtt{r})$, so it follows that $[\![\tau]\!]v, e, \Delta, (s, \sigma) \in \mathcal{P}[\![\mathsf{H}\Pi.\, (Q * RI)]\!]e$ as required.

CCR case:

Assume $e, \Delta \vDash \{\mathsf{H}\Pi'.\, (P * RI)\}\, \mathbb{C} \,\{\mathsf{H}\Pi'.\, (Q * RI)\}$ and $\Pi' = \Pi \cap \mathsf{free}(P)$. We need to show that $e, \Delta{:}(\mathtt{r} \mapsto \Pi, RI) \vDash \{P\}\, \mathtt{with\ r\ do}\ \mathbb{C} \,\{Q\}$. Consider a valuation v that satisfies the axioms of our basic commands and a trace $\tau \in T(\mathtt{with\ r\ do}\ \mathbb{C})$. We need to show that $\{\mathcal{P}[\![P]\!]e\}\, [\![\tau]\!]v, e, \Delta{:}(\mathtt{r} \mapsto \Pi, RI) \,\{\mathcal{P}[\![Q]\!]e\}$ holds.

Since $\tau \in T(\mathtt{with\ r\ do}\ \mathbb{C})$ we know $\tau = (\mathtt{lock(r)}\ ;\ \tau'\ ;\ \mathtt{unlock(r)})$ for some $\tau' \in T(\mathbb{C})$. By our assumption $\{\mathcal{P}[\![\mathsf{H}\Pi'.\, (P * RI)]\!]e\}\, [\![\tau']\!]v, e, \Delta \,\{\mathcal{P}[\![\mathsf{H}\Pi'.\, (Q * RI)]\!]e\}$.

Let $\Delta' = \Delta{:}(\mathtt{r} \mapsto \Pi, RI)$, then by the semantics of `lock(r)` and `unlock(r)` we can give the following proof outline:

$$\begin{array}{l}
\{\mathcal{P}[\![P]\!]e\} \\
\mathtt{lock(r)} \\
\{\mathcal{P}[\![\mathsf{H}\Pi'.\,(P * RI)]\!]e\} \\
[\![\tau']\!], v, e, \Delta \\
\{\mathcal{P}[\![\mathsf{H}\Pi'.\,(Q * RI)]\!]e\} \\
\mathtt{unlock(r)} \\
\{\mathcal{P}[\![Q]\!]e\}
\end{array}$$

By the rule of sequential composition (Seq) it follows that

$$\{\mathcal{P}[\![P]\!]e\}\ [\![\tau]\!]v, e, \Delta{:}(\mathtt{r} \mapsto \Pi, RI)\ \{\mathcal{P}[\![Q]\!]e\}$$

as required. □

The proof of the disjoint concurrency rule Par is almost exactly the same as the proof for the equivalent rule from the abstract separation logic work. This should not be surprising as the two rules are almost identical. The addition of compression to our model (and revelation to the logic) does not have any affect on the use of disjoint concurrency, which is only concerned with the separating conjunction $*$.

The proof of the resource rule Res and the conditional critical region rule CCR hinge on how we split up the state in a segment algebra. Rather than simply using disjointness, in the style of a separation algebra, we also make use of compression. We choose to split up the state (s, σ) such that $s = (\bar{x})(s_1 + s_2)$ and $\sigma = \sigma_1 \uplus \sigma_2$ for some $\bar{x}$, s_1, s_2, σ_1 and σ_2. We still have a notion of what it means for an action to behave locally on such a splitting, and it is this modified notion of locality that allows our reasoning rules to work.

7.4 Remarks

We have shown how to apply the techniques of concurrent separation logic to segment logic to develop a system for reasoning about abstract level concurrency. Segment logic's separating conjunction $*$ allows us to reason naturally about disjoint concurrency, and with some modifications we are also able to reason about critical regions and resource transfer.

Invariant Generation

Picking a resource invariant for a certain region `r` to obey is a lot like picking a loop invariant for a while loop. That is, it requires some intuition on the part of the prover. Just as choosing loop invariants is one of the significant hurdles to automating proof generation for sequential programs, choosing resource invariants is one of the significant hurdles to automating proof generation for concurrent programs.

In his thesis [59] Raza introduces a promising new technique for automatically generating resource invariants in concurrent separation logic proofs. Using labelled separation logic, Raza is able to analyse a concurrent program and construct ownership constraints for each resource `r` in a proof of the program. These constraints can then be solved, using the specifications of separation logic's primitive commands, to determine what part of the program state state must be owned by each resource in order for the program not to fault.

It would be interesting to see if a similar approach can be applied to the concurrent segment logic framework presented above.

Permissions

In our work on concurrency so far we have only considered access to resources in an all or nothing style. At any one time, each piece of program state is owned by exactly one thread or resource. However, in practice it is possible to share state in a more fine-grained fashion. As a particular example, it should be possible for any number of threads to have a read-only view of some piece of program state.

Boyland introduced fractional permissions in separation logic [9] to record splittings of heap cells. A permission $\pi \in (0,1)$ records that a cell is shared with other threads, while $\pi = 1$ records that it is held exclusively by one thread. Any fractional permission $x \overset{\pi}{\mapsto} v$ is enough to allow a thread to read from a heap cell, but to be able to modify the cell a thread must hold exclusive permission $x \overset{1}{\mapsto} v$ for that cell. This ensures that one thread's modifications to the heap do not invalidate other thread's views of the heap. Permissions are then split and combined via the separating conjunction $*$. For example,

$$x \overset{i}{\mapsto} v * x \overset{j}{\mapsto} v \quad \Leftrightarrow \quad x \overset{i+j}{\mapsto} v \quad \text{if } i + j \leq 1$$

The parallel rule then allows heap cells to be shared in a read-only sense between multiple threads.

It would seem that the analogous extension to segment logic would be to add permissions to address labels, that is $\alpha \overset{\pi}{\leftarrow} c$. Indeed, this does exhibit the desired behaviour with segment logic's separating conjunction. That is,

$$\alpha \overset{i}{\leftarrow} c * \alpha \overset{j}{\leftarrow} c \quad \Leftrightarrow \quad \alpha \overset{i+j}{\leftarrow} c \quad \text{if } i + j \leq 1$$

However, some care has to be taken with compression, in particular the use of the collapse/expand equivalence, in such a model. We cannot allow for segments to be compressed if their permission values are not the same. For example,

$$\mathsf{H}\beta.\,(\alpha \overset{1}{\leftarrow} n[\beta] * \beta \overset{\frac{1}{2}}{\leftarrow} m[\varnothing_{\mathrm{T}}]) \quad \not\Rightarrow \quad \alpha \overset{1}{\leftarrow} n[m[\varnothing]]$$

If we could derive such an implication then we would gain extra permission over the β segment that we should not have. In particular we would be able to modify the subtree $m[\varnothing_{\mathrm{T}}]$ which some other thread may be assuming is read-only. We should not be able to throw away or introduce permissions other than by the permissions splitting rule, otherwise our reasoning will be unsound.

An obvious solution to this problem would be to only allow collapse/expansion of a segment when a thread has exclusive permission on the segment. However, such a restriction would severely limit the utility of adding permissions to our logic. We have already seen how the CCR rule needs to make use of compression in order to arrange segments in the correct form to apply the small axioms of our basic commands. A similar requirement will occur if we work with permissions in our logic.

Allowing partial segments to be compressed introduces a need to track the labels that were used in the compression, so that the correct label may be used if the segment if broken apart again. What we mean by this is that,

$$\mathsf{H}\beta.\,(\alpha \overset{i}{\leftarrow} n[\beta] * \beta \overset{i}{\leftarrow} m[\varnothing_{\mathrm{T}}]) \quad \not\Rightarrow \quad \mathsf{H}\gamma.\,(\alpha \overset{i}{\leftarrow} n[\gamma] * \gamma \overset{i}{\leftarrow} m[\varnothing_{\mathrm{T}}]) \quad \text{if } i < 1$$

This is because the rest of the state will contain $\beta \overset{j}{\leftarrow} m[\varnothing_{\mathrm{T}}]$, with $i + j = 1$, and we must be able to recombine these segments later.

Adding permissions to the segment model would be interesting, but is clearly not a straightforward matter. One possible solution to the compression issue would be to add a frame-like rule to our reasoning system to enable us to locally compress a segment for reasoning purposes. [1]

$$\frac{e, \Gamma \vdash \{\mathsf{H}\beta.\,(P * \alpha \overset{i}{\leftarrow} R_1 \bullet_\beta R_2)\}\,\mathbb{C}\,\{\mathsf{H}\beta.\,(Q * \alpha \overset{i}{\leftarrow} R_1 \bullet_\beta R_2)\} \qquad 0 < i < 1 \quad \beta \in \mathit{free}(R_1)}{e, \Gamma \vdash \{\mathsf{H}\beta.\,(P * \alpha \overset{i}{\leftarrow} R_1 * \beta \overset{i}{\leftarrow} R_2)\}\,\mathbb{C}\,\{\mathsf{H}\beta.\,(Q * \alpha \overset{i}{\leftarrow} R_1 * \beta \overset{i}{\leftarrow} R_2)\}}$$

The idea behind this rule is that we should be able to treat partial segments as if they were compressed when we are reasoning about a program. However, we must be sure to restore the original labels and permissions at the end of the proof. This rule seems to capture our intuition of how permissions should work, but ensuring that it is sound may be quite tricky.

Refinement for Concurrent Programs

In chapter 6 we saw how to implement one fine-grained abstract module in terms of another in the sequential setting. An obvious question is does our theory cover the concurrent setting too? Unfortunately the answer is no.

In the sequential setting we do not have to consider the interference caused by the environment. In particular, this means that all of our assertions are implicitly stable: an assertion is stable if it is not modified by the actions of the environment. However, when we move into the concurrent setting, the interference of the environment becomes a very important factor. At the abstract level we do not have a problem, as we treat our basic commands as if they were atomic, which means the environment cannot interfere with them. At the concrete level, though, we can implement the basic commands with no atomic actions. These actions may interfere with one another, in particular when the access state that may be shared between two threads.

As an example, consider running two tree deletion operations in parallel on disjoint subtrees. At the abstract level these operations do not seem to interfere with one another. However, at the concrete level this is no longer the case as the operations have to perform pointer update in the surrounding state. Consider the case where the two trees are actually side by side in the tree and we are performing this pointer update. The first thread may get to run, it reads its right pointer, but then gets descheduled. The other thread is then scheduled and runs to completion removing the right tree , including the node read by the first thread. Now when the first thread gets scheduled again later it has a pointer to its old right node. If it tries to dereference this pointer it will fault, as this node no longer exists.

[1]Thanks to Adam Wright for discussions on this idea.

Our current technique for reasoning about module refinement only works because in the sequential setting we know that a command can not be interrupted part-way through its execution. In order to reason about concurrent module refinement we are going to have to introduce some sort of locking or atomic blocks to be able to rule out the bad interleavings, such as the case described above.

Relation to Concurrent Abstract Predicates

Based on existing work on abstract predicates [54], Dinsdale-Young, Dodds, Gardner and Parkinson have recently introduced the concept of concurrent abstract predicates [25]. The main focus of their work has be to allow the abstraction of concurrent program details in the same way as we abstract data structures.

Using abstract predicates, they have been able to provide abstract specifications for modules that allow concurrent manipulation of shared data structures. They have also provided refinements of these modules, in terms of permissions and actions, that enable them to show if a particular implementation satisfies their high-level specifications. This work is aiming at showing similar results as our abstraction and refinement work from chapter 6.

As an example, consider a concurrent access set s which stores unique values. There are three common operations that are called on such a set:

`search`(s, v)	Check if v is in the set s. If it is return true, otherwise return false.
`insert`(s, v)	Add v to the set s if it is not already in the set.
`remove`(s, v)	Remove v from the set s if it was initially in the set.

To represent the state of the set, we can provide a pair of abstract predicates $\mathsf{in}(h, v)$ and $\mathsf{out}(h, v)$ that describe if a particular value v is in the set s or not. We also need an axiom that states that only one in or out predicate can exist for each value v.

$$(\mathsf{in}(s, v) \vee \mathsf{out}(s, v)) * (\mathsf{in}(s, v) \vee \mathsf{out}(s, v)) \quad \Rightarrow \quad \mathsf{false}$$

This captures the idea that a value can't both be in the set and not in the set. This also forces the knowledge of a values status to be in one place, so multiple threads cannot observe the same value's status.

We can then provide specifications for our set commands in terms of these abstract

```
find(h, v) {
  local p, c in
    p := h ;
    lock p ;
    c := p.next ;
    while c.value < v do
      lock c ;
      unlock p ;
      p := c ;
      c := p.next ;
    return (p, c) ;
}
```

```
search(h, v) {
  local p, c, u in
    (p, c) := find(h, v) ;
    u := c.value ;
    unlock p ;
    return (u == v) ;
}
```

Figure 7.3: Linked list `search` implementation.

predicates. For example:

$$
\begin{array}{rcl}
\{\mathsf{in}(s,v)\} & r := \texttt{search}(s,v) & \{\mathsf{in}(s,v) \wedge (r = \mathsf{true})\} \\
\{\mathsf{out}(s,v)\} & \texttt{insert}(s,v) & \{\mathsf{in}(s,v)\} \\
\{\mathsf{in}(s,v)\} & \texttt{remove}(s,v) & \{\mathsf{out}(s,v)\}
\end{array}
$$

The remaining cases are analogous. These specifications allow us to reason about operations on a concurrent set at the abstract level. For example consider the following program and its proof sketch:

$$
\begin{array}{c}
\{\mathsf{in}(s,5) * \mathsf{out}(s,7)\} \\
\begin{array}{c|c}
\{\mathsf{in}(s,5)\} & \\
\texttt{remove}(s,5)\ ; & \{\mathsf{out}(s,7)\} \\
\{\mathsf{out}(s,5)\} & \texttt{insert}(s,7) \\
r := \texttt{search}(s,5) & \{\mathsf{in}(s,7)\} \\
\{\mathsf{out}(s,5) \wedge (r = \mathsf{false})\} &
\end{array} \\
\{\mathsf{out}(s,5) * \mathsf{in}(s,7) \wedge (r = \mathsf{false})\}
\end{array}
$$

This style of abstract reasoning is very similar to that presented by our segment logic framework. We could easily provide a segment algebra that is capable of reasoning about this concurrent set module in much the same style as presented above.

When it comes to reasoning about implementations of an abstract module, the CAP approach is reminiscent of our locality-breaking translations as introduced in chapter 6. To see this let us consider an implementation of the set module in terms of a linked list, using hand over hand lock to traverse the list to ensure that threads do not interfere with one another. An example implementation of the `search` command given in Figure 7.3. We wish to show that such an implementation satisfies the abstract specification of the `search` command, justifying that the abstraction is suitable for this implementation.

Assuming the existence of a list predicate $\mathsf{list}(s, X)$ which describes a linked list at s with contents X, we can provide concrete interpretations for the predicates of our abstract model.

$$\begin{aligned}\mathsf{in}(s, v) &::= \exists X, \pi > 0.\, \mathsf{isLock}(s, \pi) * [\mathit{change}(s, v)]_1^r * \boxed{\mathsf{list}(s, X) \wedge v \in X}_{\mathcal{A}}^{r} \\ \mathsf{out}(s, v) &::= \exists X, \pi > 0.\, \mathsf{isLock}(s, \pi) * [\mathit{change}(s, v)]_1^r * \boxed{\mathsf{list}(s, X) \wedge v \notin X}_{\mathcal{A}}^{r}\end{aligned}$$

The concrete interpretations of the predicates make use of a permissions model with $0 < \pi \leq 1$. The $\mathsf{isLock}(s, \pi)$ predicate gives partial permission on the knowledge that there is a lock for the head of the linked list s. This allows a thread to lock s which in turn allows the thread to lock the next node in the list, and so on. Owning the full permission ($i = 1$) on the token $[\mathit{change}(s, v)]_i^r$ gives the thread the exclusive right to modify if v is, or is not, in the set. The boxed assertion describes the state of the heap that is shared between the threads. In this case the shared state contains a list in the heap and whether the value v is in that list or not. Boxed assertions describe all of the shared state and behave additively under $*$, that is, $\boxed{P} * \boxed{Q} = \boxed{P \wedge Q}$. Additionally, the boxed assertion is parameterised a region name r and an interference environment $\mathcal{A}$ which captures the possible interference of the environment on the shared state. The region name is used to identify the region for tokens and actions. This is particularly important when there are multiple shared regions in use. The interference environment is defined as a set of actions on the shared state. Formally the actions would be defined as a set of state updates, but we shall just give the intuition behind the action set.

The environment $\mathcal{A}$ allows for the following actions in shared region r:

◇ Nodes in the list may be locked and unlocked, locking a node requires that the thread currently holds the lock on the nodes predecessor, unless it its the head node s;

◇ Nodes may be added to the list so long as the thread has the lock on the predecessor and the thread has the $[\mathit{change}(s, v)]_1^r$ token for the value v being added;

◇ Nodes may be removed from the list so long as the thread has the lock on the predecessor and the thread has the $[change(s, v)]_1^r$ token for the value v being removed.

With this action model and the concrete interpretations of the abstract index predicates it is possible to prove that the implementations in Figure 7.3 satisfy their respective abstract specifications and also that the abstract predicate axiom holds for the concrete implementations. The full details can be found in the Concurrent Abstract Predicates Technical Report [24].

Notice how the concrete reasoning breaks the locality of the abstract module. At the abstract level we are able to reason about individual elements of the set, but at the concrete level each of these predicates is interpreted over the whole program state (the boxed assertion). Thus, the CAP technique establishes a *fiction of locality* in much the same way as our locality-breaking translations do. One important difference here is that the CAP technique is able to handle reasoning about concurrency. This is managed by translating the abstract predicates to *stable* low level assertions. An assertion is said to be stable, with respect to an environment $\mathcal{A}$, if the truth of the assertion is unchanged by any of the actions that can be carried out by the environment.

Our module translating theory, which is currently only defined for sequential reasoning, in effect gets stability for free. In the sequential setting there is no possibility of interference, so the environment cannot modify the program state. If we want to extend our theory to handle reasoning about concurrency then we are going to have to deal more directly with the idea of stable assertions. One interesting approach to reasoning about concurrency refinement might be to translate from an abstract segment algebra into a concrete CAP model. However, this is only really applicable in the locality-breaking sense. To reason in a locality-preserving style we will have to deal with interference and assertion stability more directly. This will likely require us to include the idea of action capabilities in our reasoning framework.

CAP introduces regions names to identify portions of shared state, but these region names are not visible to the programmer. There are also rules for creating, destroying, splitting and joining regions. Our segment model of the heap, introduced in Example 3.46, also allows for regions of the heap to be labelled with abstract addresses that are not visible to the programmer. It would be interesting to further investigate the links between these two styles of logically identifying portions of heap.

8 Conclusions

We conclude this thesis by giving a summery of our main achievements. We also look at the applications of this work and discuss some avenues of future research that follow on from our work.

8.1 Summary of Thesis Achievements

The main achievement of this thesis has been to introduce segment logic for reasoning about structured data. We provided the model of this logic in terms of segment algebras which provide a general way of representing structured data. We have seen that segment algebras can be used to represent a wide range of data structures, such as trees, lists, heaps and DOM. Thus, our logic can similarly be tailored to reason about these various data structures.

Using segment logic we have been able to provide a framework for fine-grained abstract reasoning about programs. In particular, we have been able to develop a system of local Hoare reasoning which is able to work with smaller specifications than previous techniques allowed. We have seen that this reasoning system can be applied to a range of different program modules ranging in complexity from simple modules, such as heaps, to complex modules, such as featherweight DOM. One significant advantage of our framework is that we have a general soundness result for arbitrary choices of the underlying segment algebra.

An important part of any abstraction technique is to be able to link an abstraction with its concrete implementations. Building on existing work on abstraction and refinement, we have shown how to soundly implement one abstract module in terms of another. We have provided two general techniques for reasoning about such implementations: locality-breaking translations and locality-preserving translations. Each technique allows us to prove if a given implementation correctly satisfies some abstract specification.

Our final achievement has been to extend our reasoning framework to handle some simple forms of concurrency. In particular we are able to reason about programs that utilise disjoint concurrency or simple resource management. As with our sequential reasoning framework, we have been able to provide a general soundness result that does not depend on the exact segment algebra underlying our reasoning.

8.2 Applications

We have seen that segment logic can be used to reason about a number of different data structures, even complex structures such as that of featherweight DOM. Segment logic is already being used to help reason about other complex structures.

In his master's thesis [51] Ntzik has investigated using the segment model to represent graph structures. In particular he has considered representing a graph as a combination of disjoint spanning trees. Whilst this view of a graph may not always be the most useful, especially for graphs that have high numbers of cycles, it greatly simplifies the reasoning for graphs that are commonly accessed in a tree-like way, or where the majority of the graph is actually tree-like.

In upcoming work [58] Ntzik and Wright have been using segment logic to reason about file-system commands in the style of the Posix specification [42]. They have made some interesting modifications to the segment model, including annotating segment addresses with path information. Such annotations encapsulate some global information about how to reach a subtree from the root of the tree, but still allow the reasoning to be local. These annotations restrict the possible frames that can be added to a segment to those that agree with the path annotation. Such annotations offer an interesting way of reasoning locally with certain global knowledge.

Wright has been generalising this idea of locally expressing global properties in his work on strong local reasoning [36]. He uses formulae as annotations, rather than just simple paths, and allows annotations on both addresses and hole labels. The formula annotation on a segment address restricts the frame that may be added around the segment to those frames that satisfy the address annotation. Similarly, the formula annotation on a segment hole label restricts the frame that may fill that hole to those that satisfy the hole label annotation. Wright's techniques allow him to express a wide range of global properties in a local fashion, such as paths, number of siblings and uniqueness of names/elements.

Segment logic reasoning helps to simplify the axioms of the basic commands for many program module. This makes these modules more amenable to automated reasoning. In particular, Wright has been developing a proof assistant, based on segment logic, for reasoning about programs written in featherweight DOM. In discussions with Jacobs, he has also been investigating the possibility of linking this tool with the existing VeriFast tool [44].

8.3 Future Work

As discussed above we have begun to investigate using segment logic to reason about graph structures. However, our current best approach has been to treat graphs as trees wherever possible. Whilst this has proven to be quite successful for models that are largely tree-like, such as filesystems, it is less suitable for graph models that are highly connected or have large numbers of cycles. It would be interesting to see if we can find a more general model for reasoning about graphs.

We have developed a general theory for reasoning about implementations of a program module that preserve the module's locality. However, our locality-preserving translations have to make use of complex and quite ad-hoc permissions models to be able to establish the required 'fiction of disjointness'. We would really like to look at these permissions models in more detail and see if we can construct a general permissions model that will simplify our reasoning.

One of the most important next steps for the work in this thesis is to extend our abstraction and refinement theory to the concurrent setting. As a first step we want to extend our techniques so that we might implement sequential abstract programs with concurrent concrete programs. However, our real goal is to be able prove that a concrete concurrent implementation is correct with respect to a concurrent abstract specification. This is significantly more complex, as it involves having to translate abstract concurrency constructs into concrete concurrency constructs. It is not clear that abstract locks will translate directly to concrete locks, in fact the abstract locks may sometimes be unnecessary for highly concurrent implementations. Moreover, we will have to be very careful to ensure that our translations do not introduce live-lock or dead-lock issues. Our use of segments in the existing theory has given us a good starting point as we have already developed a framework for reasoning about abstract concurrency. Our existing translations also have a strong notion of what state is being shared between segments and this should help us to reason about sharing between threads.

With Raad we have already begun to look into abstraction and refinement for concurrency and in doing so have noticed some similarities between our techniques and those of the concurrent abstract predicates (CAP) work [25]. In particular, our locality-breaking translations seem to share a lot in common with the way that the CAP work takes abstract predicates and interprets them over the complete shared state. We believe it would be very interesting to look more closely at the links between our work and that of the CAP style of reasoning.

Our initial aim when setting out to reason about concurrency was to see if we could design and formally specify a concurrent XML update language. Such a language would enable web applications to make the most of the dynamic nature of XML. For example, with Wikipedia, users currently copy articles on to their browsers, before updating and returning them to Wikipedia to be integrated with the main site. Ideally we would like to be able to view Wikipedia (or some scientific data base or any information on the Cloud) as a shared XML memory store that can be concurrently updated by many clients. Currently, methods for safely performing such operations are poorly understood. Our work on concurrency lets us get some way to specifying such a language, but we are missing a key component: distributivity. In practice we do not know exactly what code might be being run on a shared web resource, which makes it very hard to reason about concurrent web languages. In order to achieve our goal we need to understand what it means to perform local reasoning for distributed systems.

Bibliography

[1] Ralph-Johan Back. Incremental software construction with refinement diagrams. In *Marktoberdorf*, volume 195 of *NATO Science Series*, pages 3–46. Springer, 2005.

[2] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. SLAM and static driver verifier: Technology transfer of formal methods inside Microsoft. In E. A. Boiten, J. Derrick, and G. Smith, editors, *Proc. Integrated Formal Methods, 4th International Conference, IFM 2004, Canterbury, UK*, volume 2999 of *LNCS*, pages 1–20. Springer, 2004.

[3] Thomas Ball and Sriram K. Rajamani. The SLAM toolkit. In *Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 260–264, Berlin, 2001. Springer-Verlag.

[4] Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter W. O'Hearn, Thomas Wies, and Hongseok Yang. Shape analysis for composite data structures. *Lecture Notes in Computer Science*, 4590, 2007.

[5] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. *Lecture Notes in Computer Science*, 4111, 2005.

[6] Josh Berdine, Byron Cook, Dino Distefano, and Peter W. O'Hearn. Automatic termination proofs for programs with shape-shifting heaps. *Lecture Notes in Computer Science*, 4144, 2006.

[7] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker blast. 2007.

[8] Richard Bornat, Cristiano Calcagno, and Hongseok Yang. Variables as resource in separation logic. In *Proceedings of MFPS XXI*, volume 155 of *ENTCS*, pages 247–276, 2006.

[9] John Boyland. Checking interference with fractional permissions. In *Static Analysis (SAS)*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72. Springer-Verlag, 2003.

[10] Stephen Brookes. A semantics for concurrent separation logic. In *Theoretical Computer Science*, volume 375, pages 227–270, 2007.

[11] C. Calcagno, P. Gardner, and U. Zarfaty. A context logic for tree update. In *LRPP: Workshop on Logics for Resources, Processes and Programs*, 2004.

[12] Cristiano Calcagno, Thomas Dinsdale-Young, and Philippa Gardner. Adjunct elimination in context logic for trees. In *APLAS*, volume 4807 of *Lecture Notes in Computer Science*, pages 255–270. Springer, 2007.

[13] Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *POPL*, ACM SIGPLAN Notices, pages 289–300. ACM, 2009.

[14] Cristiano Calcagno, Philippa Gardner, and Uri Zarfaty. Context logic and tree update. In *POPL*, volume 40 of *ACM SIGPLAN Notices*, pages 271–282, 2005.

[15] Cristiano Calcagno, Philippa Gardner, and Uri Zarfaty. Context logic as modal logic: completeness and parametric inexpressivity. *SIGPLAN*, 2007.

[16] Cristiano Calcagno, Philippa Gardner, and Uri Zarfaty. Local reasoning about data update. *Electronic Notes in Theoretical Computer Science*, 172, 2007.

[17] Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. Local action and abstract separation logic. In *LICS*, pages 366–378. IEEE Computer Society, 2007.

[18] Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. Local action and abstract separation logic, 2007. extended version of paper from LICS'07 http://www.dcs.qmul.ac.uk/ hyang/paper/asl.pdf.

[19] Luca Cardelli and Andrew D. Gordon. Anytime, anywhere, modal logics for mobile ambients. In *POPL*, pages 365–377, 2000.

[20] Luca Cardelli and Andrew D. Gordon. Ambient logic, 2003. Microsoft Research `http://lucacardelli.name/Papers/Ambient%20Logic.A4.pdf`.

[21] Pedro da Rocha Pinto. Reasoning about concurrent indexes. Master's thesis, Department of Computing, Imperial College London, 2010.

[22] W. P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison.* Cambridge Tracts in Theoretical Computer Science 47. Cambridge University Press, 1998.

[23] Thomas Dinsdale-Young. Abstract data and local reasoning. PhD Thesis, Department of Computing, Imperial College London, 2010.

[24] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew Parkinson, and Vikotr Vafeiadis. Concurrent abstract predicates. Technical Report UCAM-CL-TR-777, University of Cambridge, Computer Laboratory, April 2010.

[25] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In *ECOOP*, pages 504–528, 2010.

[26] Thomas Dinsdale-Young, Philippa Gardner, and Mark Wheelhouse. Abstraction and refinement for local reasoning. In *VSTTE*, volume 6217 of *Lecture Notes in Computer Science*, pages 199–215, 2010.

[27] Thomas Dinsdale-Young, Philippa Gardner, and Mark Wheelhouse. Abstraction and refinement for local reasoning. Technical report, Imperial College London, Department of Computing, April 2010.

[28] D. Distefano, P. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920 of *Lecture Notes in Computer Science*, pages 287–302. Springer-Verlag, 2006.

[29] Dino Distefano and Matthew Parkinson. jstar: towards practical verification for java. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, OOPSLA '08, pages 213–226. ACM, 2008.

[30] Mike Dodds, Xinyu Feng, Matthew J. Parkinson, and Viktor Vafeiadis. Deny-guarantee reasoning. In *ESOP*, volume 5502 of *Lecture Notes in Computer Science*, pages 363–377. Springer, 2009.

[31] Ivana Filipovic, Peter W. O'Hearn, Noah Torp-Smith, and Hongseok Yang. Blaming the client: on data refinement in the presence of pointers. *Formal Aspects of Computing*, 22(5):547–583, 2010.

[32] Murdoch J. Gabbay and Andrew M. Pitts. A new approach to abstract syntax with variable binding. In *Formal Aspects of Computing*, volume 13, pages 341–363, 2002.

[33] Philippa Gardner, Gareth Smith, Mark Wheelhouse, and Uri Zarfaty. DOM: Towards a formal specification. In *Plan-X*, 2008.

[34] Philippa Gardner, Gareth Smith, Mark Wheelhouse, and Uri Zarfaty. Local Hoare reasoning about DOM. In *PODS*, pages 261–270. ACM, 2008.

[35] Philippa Gardner and Mark Wheelhouse. Small specifications for tree update. In *WS-FM*, volume 6194 of *Lecture Notes in Computer Science*, pages 178–195. Springer, 2010.

[36] Philippa Gardner and Adam Wright. Strong local reasoning. *upcoming publication*, 2012.

[37] J. Guttag. *The Specification and Application to Programming of Abstract Data Types*. PhD thesis, University of Toronto, Department of Computer Science, 1975.

[38] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and λ-Calculus*. Cambridge University Press, Cambridge, 1986.

[39] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[40] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.

[41] Haruo Hosoya and Benjamin C. Pierce. Xduce: A statically typed XML processing language. In *TOIT*, volume 3, pages 117–148. ACM, 2003.

[42] IEEE. Posix specification. IEEE Standard, 2008. `http://www.opengroup.org/onlinepubs/9699919799/`.

[43] Samin Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *POPL*, volume 36 of *ACM SIGPLAN Notices*, pages 14–26, 2001.

[44] Bart Jacobs and Frank Piessens. The VeriFast program verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, August 2008.

[45] Cliff B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332, 1983.

[46] James Kearney. Concurrent segment logic for trees. Master's thesis, Department of Computing, Imperial College London, 2010.

[47] B. Liskov and S. Zilles. Programming with abstract data types. *ACM SIGPLAN Conference on Very High Level Languages, SIGPLAN Notices*, 9(4):50–59.

[48] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes I & II. In *Information and Computation*, volume 100, pages 1–77, 1992.

[49] Robin Milner. Pi-nets: A graphical form of π-calculus. In *ESOP*, volume 788 of *Lecture Notes in Computer Science*, pages 26–42. Springer, 1994.

[50] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM TOPLAS*, 10(3):470–502, 1988.

[51] Gian Ntzik. Local reasoning for filesystems. Master's thesis, Department of Computing, Imperial College London, 2010.

[52] Peter W. OHearn. Resources, concurrency and local reasoning. In *Theoretical Computer Science*, volume 375, pages 271–307, 2007.

[53] Peter W. O'Hearn, John Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *CSL*, Lecture Notes in Computer Science. Springer, 2001.

[54] Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. *SIGPLAN Not.*, 40(1):247–258, 2005.

[55] David L. Parnas. The secret history of information hiding. In *Software Pioneers — Contributions to Software Engineering*, pages 398–409. Springer-Verlag, 2002.

[56] Mike Dodds Philippa Gardner Pedro da Rocha Pinto, Thomas Dinsdale-Young and Mark Wheelhouse. A simple abstraction for complex concurrent indexes. In *OOPSLA*, pages 845–864. ACM, 2011.

[57] Philippa Gardner Pedro da Rocha Pinto, Thomas Dinsdale-Young and Mark Wheelhouse. Abstract reasoning for concurrent indexes. In *Verico*, 2011.

[58] Gian Ntzik Philippa Gardner and Adam Wright. Local reasoning for file-systems. *upcoming publication*, 2012.

[59] Mohammad Raza. Resource reasoning and labelled separation logic. PhD Thesis, Department of Computing, Imperial College London, 2010.

[60] Mohammad Raza and Pilippa Gardner. Footprints in local reasoning. In *FoSSaCS*, volume 4962, pages 201–215. Springer, 2008.

[61] Microsoft Research. Slayer: automatic verification tool, 2006. `http://research.microsoft.com/SLAyer/`.

[62] J C Reynolds. Types, abstraction and parametric polymorphism. In *IFIP'83, Paris, France*. North-Holland, 1983.

[63] J.C. Reynolds. Separation logic: a logic for shared mutable data structures. In *IEEE Symposium on Logic in Computer Science*, pages 55 – 74, 2002.

[64] Gareth Smith. Providing a formal specification for DOM core level 1. PhD Thesis, Department of Computing, Imperial College London, 2010.

[65] Squillante. Magic: A computer performance modeling tool based on matrix-geometric techniques. Technical Report 90-05-09, University of Washington, 1990.

[66] Viktor Vafeiadis. Concurrent separation logic and operational semantics. *Electr. Notes Theor. Comput. Sci*, 276:335–351, 2011.

[67] Viktor Vafeiadis and Matthew Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*, pages 256–271. Springer, 2007.

[68] W3C. Dom: Document object model. W3C recommendation, 1997 - 2005. `http://www.w3.org/DOM/`.

[69] W3C. Dom core level 1 specification. W3C recommendation, 1998. `http://www.w3.org/TR/REC-DOM-Level-1/`.

[70] Mark Wheelhouse. Dom: Towards a formal specification. Master's thesis, Department of Computing, Imperial College London, 2007.

[71] Uri Zarfaty. Context logic and tree update. PhD Thesis, Department of Computing, Imperial College London, 2007.