University of London
Imperial College of Science, Technology and Medicine
Department of Computing

Abstract Data and Local Reasoning

Thomas Dinsdale-Young

Submitted in part fulfilment of the requirements for the degree of Doctor of Philosophy in Computing of the University of London and the Diploma of Imperial College, November 2010

Abstract

This thesis tackles problems concerning abstract data structures — expressibility and decidability results for logics for reasoning about abstract data structures, and techniques for proving the correctness of programs that implement abstract data structures. The main expressivity issue addressed is the question of whether certain spatial adjunct connectives contribute to the expressive power of context logic for trees. The question is answered in the affirmative for context logic in its basic form, but in the negative for a multi-holed variant of context logic. This result is interesting, since the adjunct connectives play an important role in expressing the weakest preconditions of programs. The decidability results show that multi-holed context logic is decidable for trees, sequences and terms, by encoding logical formulae with automata.

Concerning the correctness of programs that implement abstract data structures, this thesis presents two techniques for justifying abstract local reasoning about such implementations in the sequential setting. In the concurrent setting, this thesis presents an approach to verifying implementations of abstract specifications that incorporate a fiction of disjointness using a fine-grained permissions model.

To all from whom I have ever learned, in the hope that I might reciprocate your generosity in some small part.

Declaration of Originality

This thesis represents my own work, except where otherwise acknowledged.

– Thomas Dinsdale-Young

Contents

List of Figures

List of Definitions

List of Theorems

Acknowledgements

I am eternally indebted to Philippa, my fantastic supervisor, for so much, not least her indefatigable enthusiasm and relentless drive for excellence; to my inspirational collaborators — Cristiano, Mark, Matt, Mike and Viktor — who have been a delight to work with; to my colleagues and friends at Imperial who make up the exciting and intellectual environment in which I have been privileged to work; and to my wonderful friends and family whose love and support I cherish above all.

Chapter 1

Introduction

This thesis is broadly concerned with the subject of abstract data structures. Within that remit, it is broadly divided into two parts: Reasoning about Data, and Reasoning about Programs. The first part addresses expressivity and decidability issues concerning context logic — a spatial logic for reasoning about structured data. The second part addresses the problem of verifying programs that implement such abstract data structures, particularly when they provide an abstract view of locality or disjointness. This introduction discusses each part of the work in its overall context.

1.1 Local Reasoning and Separation Logic

Hoare logic, introduced by Hoare in [Hoa69], was developed as a formal logical system for proving properties of programs. Assertions of Hoare logic, called Hoare triples, are denoted $\{P\}\ \mathbb{C}\ \{Q\}$, where P and Q are predicates describing machine states and $\mathbb{C}$ is a program. The interpretation of such a triple is, informally, that if the program $\mathbb{C}$ is run on from an initial state satisfying P (the precondition) then, if the program terminates it will do so in a state satisfying Q (the postcondition). The logic provides axioms and inference rules from which valid triples are proven.

Proving the correctness of programs that use dynamic allocation is difficult to do with traditional Hoare logic, however. This is because it is necessary to account for the possibility of multiple references to the same data. For example, proving that a program correctly reverses a list does not allow us to directly infer that it leaves a second list untouched. It is possible that the two lists overlap in memory, so the second would be updated by the operation on the first. Even if the two lists are completely disjoint in memory, it would be necessary to

account for this disjointness throughout the proof. When large amounts of data are involved, often only a small portion of that is being manipulated by a given part of the program, yet it is necessary to account for the data in its entirety.

O'Hearn, Reynolds and Yang introduced *separation logic* [IO01, Rey02] as a solution to this issue. Their key innovation was in treating heap memory as resource. For example, $2 \mapsto 5 * 3 \mapsto 8$ describes the piece of memory where addresses 2 and 3 are allocated and hold the values 5 and 8 respectively. This resource can be subdivided into the *disjoint* resources $2 \mapsto 5$ and $3 \mapsto 8$. Importantly, the $*$ connective, termed *separating conjunction*, enforces that the subdivisions are disjoint: $2 \mapsto 5 * 2 \mapsto 5$ does not describe any piece of memory at all, since both components refer to the same memory address and so are not disjoint. Basic heap operations, such as updating a heap cell or deallocating a block of memory, operate on specific *footprints* — the piece of resource that the operation requires for its execution. These operations are *local* to their footprints in that any additional resource is not modified by the operation.

Consider, for example, the operation of updating the memory location at address 2 to hold the value 7. This operation has the specification

$$\{2 \mapsto -\}\ [2] := 7\ \{2 \mapsto 7\}\,.$$

Since the operation is local, it would leave memory at address 3 unchanged, and so the following specification is also valid:

$$\{2 \mapsto - * 3 \mapsto 8\}\ [2] := 7\ \{2 \mapsto 7 * 3 \mapsto 8\}\,.$$

Separation logic allows this second specification to be deduced from the first using the *frame rule*:

$$\frac{\{P\}\ \mathbb{C}\ \{Q\}}{\{P * F\}\ \mathbb{C}\ \{Q * F\}}.$$

The frame rule of separation logic enables *local reasoning*: a program fragment can be proved purely in terms of the resource it manipulates, and any additional resource, which would not be affected by the program, can be "framed on".

(It is worth noting that the frame rule also allows the derivation of the triple

$$\{2 \mapsto - * 2 \mapsto 5\}\ [2] := 7\ \{2 \mapsto 7 * 2 \mapsto 5\}\,.$$

Recall, however, that $*$ enforces disjointness and so the precondition is not satisfiable. In fact, the triple is equivalent to

$$\{\text{false}\}\ [2] := 7\ \{\text{false}\}\,.$$

This triple is trivially true, since no initial state satisfies the precondition.)

The assertion language of separation logic was seen as an application of O'Hearn and Pym's *logic of bunched implications* (or *bunched logic*). Bunched logic includes both the additive conjunction of propositional logic ($\wedge$) and a multiplicative conjunction ($*$). Whereas the additive conjunction $P \wedge Q$ expresses that the state satisfies both P and Q, the multiplicative conjunction $P * Q$ expresses that the state can be divided into two disjoint substates, one satisfying P and the other satisfying Q. Many natural models of resource are models of bunched logic, including the heap model used by separation logic.

As well as multiplicative conjunction, Bunched logic also includes a multiplicative analogue ($-\!\!*$) of the additive implication ($\rightarrow$). The interpretation of the multiplicative implication $P -\!\!* Q$ is that the state is such that, when it is extended with any disjoint state satisfying P, the resulting state satisfies Q. Multiplicative implication is (right) *adjoint* to multiplicative conjunction, or more correctly, $F -\!\!* (\cdot)$ is the right adjoint of $F * (\cdot)$. (Multiplicative implication is referred to as an *adjoint* or as an *adjunct connective*.) Essentially what this means is

$$F * P \models Q \quad \iff \quad P \models F -\!\!* Q.$$

Note that additive implication and conjunction are also adjoint:

$$F \wedge P \models Q \quad \iff \quad P \models F \rightarrow Q.$$

In separation logic, the multiplicative implication is used to express weakest preconditions — that is, for arbitrary Q, the most general precondition P such that $\{P\}\ \mathbb{C}\ \{Q\}$ holds. In the case of $[2] := 7$, for example, the weakest precondition for arbitrary postcondition Q is $2 \mapsto - * (2 \mapsto 7 -\!\!* Q)$.

1.2 Context Logic and Tree Update

Calcagno, Gardner and Zarfaty introduced *context logic* [CGZ04, CGZ05], in order to reason locally about structured data update. That is, they wanted to adapt the local reasoning approach, which makes separation logic effective in reasoning about heap update, to reasoning about updating data that has intrinsic structure.

A prime example of such structured data is trees. Whereas in a heap, each component (a heap cell) is associated with a value whose interpretation could be the address of another cell, an integer or whatever else, in a tree, each component (a tree node) is associated with a parent node and a set of child nodes. Furthermore, certain well-formedness constraints apply to trees, such as the fact a node cannot be its own parent.

```
<html>
  <head>
    <title>Thomas's Page</title>
  </head>
  <body>
    <table>
      <tr>
        <td>
          <img src="me.jpg" alt="Me" id="mypic" />
        </td>
        <td>
          <h1>Thomas Dinsdale-Young</h1>
          <p>Department of Computing</p>
        </td>
      </tr>
    </table>
    <h2>Papers</h2>
    <ul>
      <li>
        <a href="adjelim.pdf">Adjunct Elimination</a>
      </li>
      <li>
        <a href="decidability.pdf">Decidability</a>
      </li>
    </ul>
  </body>
</html>
```

Figure 1.1: An HTML document

Another reason that trees are such an important example is that they are so common. The hypertext documents that make up the web are typical instances. Figure 1.1 shows the source of a simple web page, which could be rendered as Figure 1.2. The document textually represents the tree illustrated in Figure 1.2.

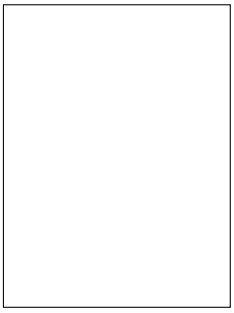
Updating document trees is foundational to web programming. For instance, the following JavaScript code could be used to change the image on the page:

```
var pic = document.getElementById("mypic");
pic.src = "me2.jpg";
```

Such an update is local: it affects only the unique "`mypic`" node of the tree and would perform the same function if the node were to appear in a different context.[1] Context logic was introduced to reason locally about such updates.

Cardelli and Gordon had previously introduced *ambient logic* for describing properties of processes in the ambient calculus, which are tree-like structures. As with separation logic, ambient logic includes connectives that allow processes

[1] Up to well-formedness, which in this case must ensure that no other node has the id "`mypic`".



Thomas Dinsdale-Young

Department of Computing

Papers

- Adjunct Elimination
- Decidability

Figure 1.2: A rendering of the HTML document of Figure 1.1

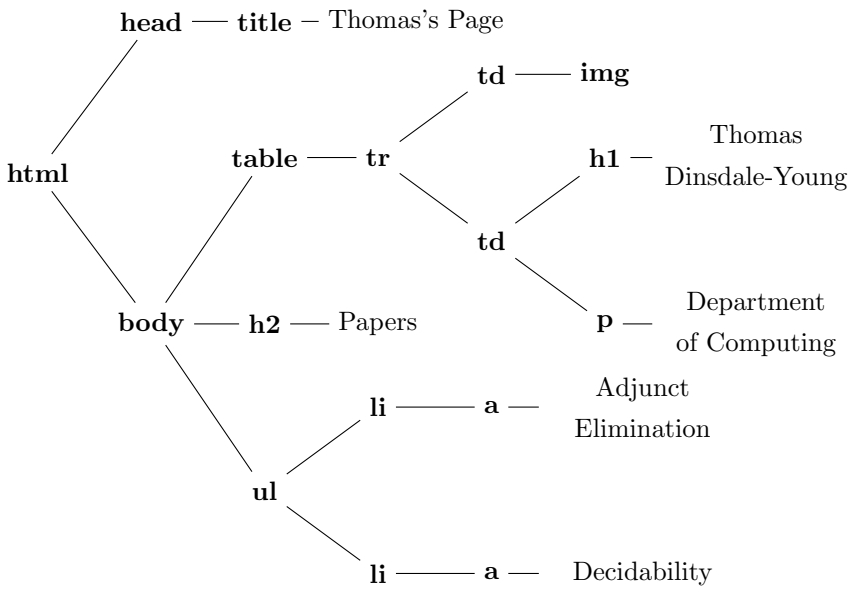


Figure 1.3: Tree representation of the HTML document of Figure 1.1

to be described in terms of their decompositions. In particular, $P|Q$ specifies that a process is the parallel composition of two processes at the root level, one satisfying P and the other satisfying Q, and $n[P]$ expresses that a process consists of a process satisfying P located at n. Ambient logic also includes the adjoints of these connectives: $P \triangleright Q$ specifies a process that, when it is composed in parallel with any process satisfying P, the resulting process satisfies Q; and $P@n$ specifies a process that, when it is placed in location n, the resulting process satisfies P. These adjunct connectives are important for specifying security properties.

Calcagno, Garder and Zarfaty attempted to use ambient logic for local Hoare reasoning about tree update, just as bunched logic had been successfully applied to reason about heap update in the form of separation logic. They found, however, that ambient logic lacked the expressive power necessary to describe weakest preconditions. Spurned on by this, they introduced *context logic*, which resolved the issue.

The key innovation of context logic was the introduction of *separating application* (or *context application*), and its adjoints: whereas ambient logic's connectives describe how a tree can be decomposed at the root, separating application describes how a tree can be decomposed at an arbitrary position into a context and a subtree. Expressed as $K \bullet P$, the connective specifies a tree that may be split into some context satisfying K and some subtree satisfying P. Whereas the $*$ and $|$ connectives of separation logic and ambient logic are commutative, $\bullet$ is non-commutative; indeed, K and P refer to different sorts of objects altogether. Consequently, $\bullet$ has two right adjoints: $\bullet\!\!-$ and $-\!\!\bullet$. The formula $K \bullet P'$ specifies a tree that, when any context satisfying K is applied to it, gives a tree satisfying P'. ($K \bullet\!\!- (\cdot)$ is the right adjoint of $K \bullet (\cdot)$.) The formula $P -\!\!\bullet P'$ specifies a context that, when applied to any tree satisfying P, gives a tree satisfying P'. ($P -\!\!\bullet (\cdot)$ is the right adjoint of $(\cdot) \bullet P$.)[2] As with separation logic, these adjoints (specifically $-\!\!\bullet$) are necessary for specifying weakest preconditions.

For local Hoare reasoning, context logic's separating application takes the place of separation logic's separating conjunction in the frame rule, which becomes

$$\frac{\{P\}\ \mathbb{C}\ \{Q\}}{\{K \bullet P\}\ \mathbb{C}\ \{K \bullet Q\}}.$$

The general theory of context logic was developed in [CGZ05], [CGZ06a] and [Zar07]. The logic has a proof theory which is sound and complete with respect to the general model theory. A specialisation of the general theory,

[2] My notation differs from the early context logic papers [CGZ05, CGZ06a], which used $K(P)$ where I use $K \bullet P$, $K \lhd P$ where I use $K \bullet\!\!- P$, and $P \rhd P'$ where I use $P -\!\!\bullet P'$.

context logic with zero, includes the concept of special zero elements. In the tree model, the empty tree plays this role. Bunched logic can be viewed as a specialisation of context logic with zero, where contexts and data structures are identified. Context logic has many interesting models, besides trees, such as terms (ranked trees) and sequences, as well as those inherited from bunched logic, such as heaps and multisets.

In Chapter 2, I will formally define context logic for a range of different models. I will also show extensions of the logic which permit context composition and contexts with multiple holes. Following [CGZ07], I will show how context logic can be viewed as a modal logic.

1.3 Expressivity and Adjunct Elimination

An interesting class of problems frequently studied for logics are expressivity results: which properties can and cannot be specified in a particular logic. Adjunct elimination is a among these problems: the question as to whether the logic is equally expressive with and without the adjunct connectives.

While the adjunct connectives of the aforementioned logics are important, for instance, to specify weakest preconditions, at an intuitive level it can be reasoned taht they add little to the expressivity of the logic: if a tree property can be expressed by considering the tree in a wider context, then surely it is enough to consider the tree itself. In [Loz03] and [Loz05], Lozes showed that the adjunct connectives of ambient logic and separation logic add no expressive power.

Lozes' method was based on *intensional bisimilarity*: an equivalence relation between static ambients (the tree-like models of ambient logic) that corresponds with the possibility of distinguishing between the ambients with some adjunct-free formula of a certain size. He showed that formulae with adjoints do not make greater distinctions between ambients than the equivalence relation, and consequently they do not add any expressive power. The results extend to the logic with the hiding quantifier, but are shown not to hold with classical quantifiers (existential and universal quantification). Lozes also showed that the loigc without the adjoints is minimal: no further connectives can be eliminated.

For separation logic, Lozes also showed that the adjoint connectives do not add expressive power by demonstrating a classical logic (*i.e.* one without spatial connectives such as $*$ and $-\!\!*$) that is as expressive as separation logic. His method again uses intensional bisimilarity.

Dawar, Gardner and Ghelli also studied adjunct elimination for ambient

logic in [DGG04]. Their approach was to use Ehrenfeucht-Fraïseé-style games, which resulted in a more modular proof (with respect to the connectives of the logic) of adjunct elimination. The games take place between two players, called Spoiler and Duplicator. At the start of each round of the game, the state consists of two trees. Spoiler wins the game by identifying some difference between the two, while Duplicator wins by preventing Spoiler from doing so. The games are limited by the number and type of moves that may be played, specified by the game rank, which decreases with each round. These moves directly correspond to the connectives of the logic, and so the rank relates games with formulae. For any starting state (including the rank), one or other of the players has a strategy that guarantees him or her a victory, regardless of how the other player plays.

In their work, the games take the place of Lozes' intensional bisimilarities: if Duplicator has a winning strategy for a game of finite rank played on two trees, then those two trees cannot be distinguished by an ambient logic formula of that rank. They show, for instance, that if Duplicatorhas a winning strategy for the games on trees (t_1, t'_1) and (t_2, t'_2) of some rank, then she also has a winning strategy for the game of the same rank on the trees $(t_1|t_2, t'_1|t'_2)$. This implies that the game move corresponding to the adjunct of composition does not help Spoiler to win a game, and hence that the connective does not add expressive power to the logic.

Dawar *et al.* also showed a stronger result than that of Lozes concerning the non-eliminability of adjoints in the presence of classical quantification: their counterexample (partly due to Yang) depends on fewer connectives of the logic.

In my Masters' thesis [DY06], I first studied adjunct elimination results for context logic. My key results were that, while one of the adjoints ($\bullet\!\!-$) may be eliminated from the standard presentation of Context Logic for trees, the other ($-\!\!\bullet$) cannot. The proof of the first result is through games in the style of [DGG04]. Since the $\bullet\!\!-$ connective can be seen as a generalisation of the adjunct connectives of ambient logic, it is not entirely surprising that it can be eliminated.

The $-\!\!\bullet$ connective, on the other hand, has no counterpart in ambient logic, and so the fact that it is not eliminable is interesting. The proof that $-\!\!\bullet$ is non-eliminable is by a direct counterexample: the context formula $\mathbf{0} -\!\!\bullet (\mathrm{True} \bullet \mathbf{a}[\mathbf{0}])$ has no adjunct-free equivalent. The formula essentially expresses that the label $\mathbf{a}$ occurs as a leaf or directly above the context hole. Without $-\!\!\bullet$, context formulae can only express properties of contexts that can be checked by examining the context to a fixed depth.

In Chapter 3, §3.2 I will give a proof of this counterexample (Theorem 13). Previously, it was not clear whether $-\!\bullet$ can be eliminated for tree formulae, since the counterexample is based on a context formula. After all, tree formulae *can* express properties at an arbitrary depth within the tree. I will demonstrate that $-\!\bullet$ cannot in fact be eliminated by showing a tree formula with no adjunct-free equivalent (Theorem 15). The proof of this fact is rather involved, and is again based on Ehrenfeucht-Fraïssé-style games, which I will formalise in §3.1.

Notably, both of these formulae *do* have adjunct-free equivalents when the logic is extended with context composition, since context composition can be used to analyze contexts at an arbitrary depth. It appears that adjunct elimination is likely to hold for context logic with composition, however, I have been unable to adapt the games-based proof technique to confirm this intuition. I will give details in §3.3, but the issue is that, while context composition permits directly splitting a subcontext from a context, it does not permit directly splitting a subtree from a context. While it is possible to perform such a splitting indirectly, this leads to a breakdown in the inductive argument for elimination.

This difficulty led me, with Calcagno and Gardner, to introduce multi-holed context logic and study adjunct elimination results for it [CDYG07, CDYG10]. In §3.4, I will show that adjunct elimination does indeed hold for multi-holed context logic for trees. This result once again makes use of Ehrenfeucht-Fraïssé-style games.

A feature of multi-holed context logic is quantification over hole labels, each of which may occur at most once in any given context. In §3.5 I show that existential (and hence universal) quantification over hole labels can be re-expressed in terms of Gabbay-Pitts fresh quantification (Lemma 31). Freshly quantified variables can take on any value, provided that it is not a value that is already in use — because the value is fresh, it does not make any difference exactly *which* value is used. Consequently, freshness quantification can be brought outside the other connectives in a formula without changing the formula's meaning (with certain caveats; see Lemma 32). This implies that multi-holed context logic is exactly as expressive when the logic is restricted to formulae in prenex normal form that only use freshness quantification.

1.3.1 Parametric Expressivity

The concept of *parametric expressivity* was introduced by Calcagno, Gardner and Zarfaty in [CGZ07]: expressivity in the presence of propositional variables. Parametric inexpressivity results help to ground the intuition that connectives are important by showing, for example, that formulae with adjuncts do not

have translations to formulae without adjuncts that are parametric in their subformulae.

To understand what is meant by this, consider for instance the 'somewhere' connective: $\Diamond P$ expresses that P holds for some subtree. This has a direct parametric translation into context logic as True $\bullet$ P — parametric since any substitution for F in both formulae maintains the logical equivalence. For separation logic, Calcagno, Gardner and Zarfaty's inexpressivity results show that $*$ and $-\!\!*$ cannot be parametrically translated into Lozes' classical fragment. For context logic, they show that the adjunct connective $-\!\!\bullet$ cannot be parametrically translated into an adjunct-free fragment for the sequence and tree models.

1.4 Formal Languages

The theory of formal languages has been studied extensively as one of the most fundamental aspects of theoretical computer science. Word languages, that is, sets of finite sequences of symbols, have received particular attention. For instance, a Turing machine is generally seen to define a language by the set of input tapes that it accepts. Word languages are classified according to the classes of formal grammars which generate them, or, often equivalently, by classes of automata which accept them. The Chomsky hierarchy [Cho56] is the best known classification.

1.4.1 Regular Languages

At the lowest level of the Chomsky hierarchy are the *regular languages*: those languages that are generated by regular grammars, or, equivalently, accepted by finite automata. Finite automata are defined formally in §4.1.1, but in essence they consist of a finite set of states and a (finite) table of rules for transitioning between these states depending on which symbol is encountered. The run of such an automaton on a word starts from a designated initial state and transitions according to the symbols in the word in sequence until no more symbols are left. The final state determines whether the word is accepted or not, and so an automata defines a language that consists of the words it accepts. An important property of automata is that, not only is it effective to determine whether a given word belongs to the language it defines, but it is also effective to determine whether any or all words belong to the language.

Other important characterisations of the class of regular languages are also known. Kleene's Theorem [Kle56] states that regular expressions define exactly

the regular languages. Regular expressions, $e, e_1, e_2, \ldots \in \mathsf{RegExp}$, are defined by the following grammar:

$$e ::= \emptyset \mid \varepsilon \mid a \mid (e_1 + e_2) \mid (e_1 \cdot e_2) \mid e^*$$

where $a \in \Omega$ is taken to range over the alphabet from which words are constructed. Their semantics as languages of words, given by $[\![(\cdot)]\!] : \mathsf{RegExp} \to \Omega^*$, is defined inductively as follows:

$$[\![\emptyset]\!] \stackrel{\text{def}}{=} \emptyset \qquad [\![\varepsilon]\!] \stackrel{\text{def}}{=} \{\varepsilon\}$$

$$[\![a]\!] \stackrel{\text{def}}{=} \{a\} \qquad [\![(e_1 + e_2)]\!] \stackrel{\text{def}}{=} [\![e_1]\!] \cup [\![e_2]\!]$$

$$[\![(e_1 \cdot e_2)]\!] \stackrel{\text{def}}{=} \{w_1 \cdot w_2 \mid w_1 \in [\![e_1]\!] \text{ and } w_2 \in [\![e_2]\!]\} \qquad [\![e^*]\!] \stackrel{\text{def}}{=} \bigcup_{n \in \mathbb{N}} [\![e]\!]^n$$

where

$$\mathcal{L}^0 \stackrel{\text{def}}{=} \{\varepsilon\} \qquad \mathcal{L}^{n+1} \stackrel{\text{def}}{=} \{w \cdot w' \mid w \in \mathcal{L} \text{ and } w' \in \mathcal{L}^n\}\,.$$

Another characterisation of the regular languages is as those having finite syntactic monoids [RS59]. A monoid $(M, \bullet)$ is said to recognise language $\mathcal{L} \subseteq \Omega^*$ if there is a monoid homomorphism $\phi : \Omega^* \to M$ (*i.e.* a function for which $\phi(w \cdot w') = \phi(w) \bullet \phi(w')$) such that $\mathcal{L} = \phi^{-1}N$ for some $N \subseteq M$. That is, $\mathcal{L}$ is a homomorphic preimage of a subset of M. The syntactic monoid of a language is the smallest monoid recognising the language. The concept is intimately related to the minimal automaton recognising a language.

Yet another characterisation of the regular languages is as exactly those which can be defined by sentences of monadic second-order logic [Büc60, Elg61]. In monadic second-order logic over words, first-order variables range over positions in a word, while second-order variables range over sets of such positions (*i.e.* monadic predicates). The signature includes unary predicates Q_a for each symbol $a \in \Omega$, with $Q_a(x)$ interpreted as "the symbol at position x is a", and the binary predicate S, with $S(x, y)$ interpreted as "y is the immediate successor of x".

The rich theory of regular languages makes relating context logic for sequences (CL_{Seq}) to the class of regular languages an attractive proposition. The syntax of CL_{Seq} has much in common with regular expressions, and so Kleene's Theorem gives a first suggestion that CL_{Seq} is embeddable in regular languages. The effectiveness of checking language emptiness with automata suggests a route to finding a decision procedure for CL_{Seq}. Indeed, in §4.1 I show that automata can be constructed that accept exactly the sequences which satisfy given CL_{Seq} formulae, just as Kleene's Theorem shows that automata can be constructed

corresponding to regular expressions. These constructions give a decision procedure for CL_{Seq}. Furthermore, they imply that CL_{Seq} is expressively contained in monadic second-order logic on sequences.

Star-free Regular Languages

An important subclass of the regular languages is the *star-free regular languages*: the languages that are expressible by regular expressions with the inclusion of complementation (with respect to Ω^*) and exclusion of Kleene star. Schützenberger characterised the star-free regular languages algebraically as those having finite, aperiodic syntactic monoids. A finite monoid $(M, \bullet)$ is aperiodic if there is some $n > 0$ such that, for all $m \in M$, $m^n = m^{n+1}$. A regular language $\mathcal{L}$ has an aperiodic syntactic monoid if and only if there exists some $n > 0$ such that for all $w_1, w_2, w_3 \in \Omega^*$, $w_1 \cdot w_2^n \cdot w_3 \in \mathcal{L}$ if and only if $w_1 \cdot w_2^{n+1} \cdot w_3 \in \mathcal{L}$.

McNaughton and Papert characterised the star-free regular languages as those expressible by sentences of first-order logic over words [MP71]. As with monadic second-order logic, the signature includes the predicates Q_a and S, which are interpreted as before, but also includes the binary predicates $=$, interpreted as equality on locations, and $<$, interpreted as the transitive closure of S.

The star-free regular languages can be shown to be exactly those expressible in context logic for sequences. On the one hand, context logic formulae can be constructed using analogues of all of the connectives of star-free regular expressions, and so all star-free regular languages are expressible as context logic formulae. On the other, the connectives of context logic can be seen to preserve aperiodicity, and therefore all context logic formulae define star-free regular languages. This correspondence was first noted in [CGZ06b].

1.4.2 Regular Term Languages

Language theory extends beyond words: languages of labelled trees and graphs, for instance, are also studied. Historically, terms (ranked trees)[3] have received more study than unranked trees, however, the recent rise of XML has stimulated greater interest in the latter.

The theory of languages of terms is quite extensive: for a comprehensive survey, see [GS97]. I mention a few results here.

[3] I use "term" to refer to ranked trees throughout, although in the literature they are typically referred to as "trees".

Regular term languages are defined as those accepted by finite term automata: terms are mapped to automaton states recursively, according to their root labels and the states that each immediate subterm is mapped to; the term belongs to the language recognised by the automaton if this state is one of the accepting states of the automaton. This description is that of deterministic, bottom-up finite term automata; non-deterministic bottom-up and non-deterministic top-down automata also recognise the regular term languages, while deterministic top-down automata recognise a subclass.

As with regular word languages, a number of equivalent characterisations exists for regular term languages: they are exactly those generated by regular term grammars; a counterpart of Kleene's Theorem states that they are exactly those expressible by term-regular expressions; they have finite syntactic monoids; and they are exactly those definable in monadic second-order logic. Subclasses of regular tree languages have also been studied, particularly with reference to logical definability (see [Tho97] for further details), which is of interest here.

In monadic second-order logic on terms, first- and second-order variables range over positions and sets of positions in the term respectively. The signature consists of unary labelling predicates Q_a (as for words), binary successor predicates S_n, with $S_n(x, y)$ interpreted as "y is the nth child of x", and a binary ancestor predicate $<$, with $x < y$ interpreted as "x is an ancestor of y". The standard signature for first-order logic on terms is the same.

While first-order definable languages, star-free regular languages and aperiodic languages all coincide for word languages, the case is significantly more complicated in the case of terms. Thomas [Tho84] showed that there are term languages that are star-free but not aperiodic, and so an analogue of Schützenberger's characterisation does not hold. On the other hand, Pothoff and Thomas [PT93] showed that all aperiodic regular tree languages are star-free.

Thomas [Tho84] showed that star-free regular languages correspond to a subclass of the monadic-second order definable languages called *antichain definable*. Antichain definable languages are those definable in monadic second-order logic when second-order quantification is restricted to antichains: sets of nodes for which no node is the ancestor of any other node.

Heuter [Heu91] showed that the first-order definable languages are exactly the *special star-free* regular term languages. These languages are those expressible by regular expressions without the Kleene star and using linear substitution symbols. Heuter's proof is based on Ehrenfeucht-Fraïssé games.

The special star-free regular term languages correspond directly to multi-

holed context logic formulae, excluding adjuncts and quantification. Together with adjunct elimination (which can be shown for terms in a similar fashion as for trees) and quantifier normalisation, Heuter's characterisation implies that multi-holed context logic for terms is equally expressive as first-order logic over terms.

Remark. For word languages, there is an effective procedure for determining if a regular language is definable in first-order logic, namely checking whether its syntactic monoid is aperiodic. No effective procedure is yet known for determining whether a regular term language is first-order definable, although an algebraic characterisation is known [ÉW03], as well as effective procedures for other logics including first-order logic without $<$ [BS09].

1.4.3 Regular Tree Languages

Throughout this thesis, I use the term "tree" to mean unranked, ordered trees with potentially multiple root nodes. In the literature, "tree" is generally reserved for such structures with a single root node; "forest" is used for my concept of tree in [Mur95, Tak75, Boj07], while "ramification" is used in [PQ68] and "hedge" in [BKMW01, CDG$^+$07].

Pair and Quere [PQ68] studied regular tree languages as "regular bilanguages". Their definition of regular languages is in terms of homomorphic preimages, analogous to the syntactic monoid characterisation of regular word languages. They showed various closure properties of regular tree languages and established and analogue of Kleene's Theorem. Their regular expression use the concept of a *graft*: attaching a copy of one tree beneath each occurrence of a particular symbol at the leaves of another tree.

Takahashi [Tak75] considered regular tree languages as generalisations of regular languages by defining them as projections of local languages (for words, these are languages that can be decided entirely by examining the first and last letters and the two-letter subwords).

Murata [Mur95] gave equivalent definitions of regular tree languages in terms of automata, regular expressions and grammars. His definition of regular expressions is closer to those used for ranked trees than that of Pair and Quere, in that they make use of substitution rather than grafting. Regular tree languages are also covered in [CDG$^+$07], which deals with automata, closure properties, encodings into terms and the connection to monadic second-order logic.

Bojańczyk and Walukiewicz [BW06] introduced the concept of a *forest algebra* in order to study algebraic characterisations of classes of tree languages,

in the way that monoids (or semigroups) are used for words. A forest algebra essentially consists of two semigroups: a concatenation semigroup and a composition semigroup, that latter of which acts on the former by application. In particular, trees with single-holed contexts form a forest algebra. Syntactic forest algebras arise as an analogue of syntactic monoids, and can be used to classifiy logics by their expressivity over trees. (Note, however, as with terms there is no known effective characterisation of first-order definability, although numerous other results are known.)

In [Boj07], Bojańczyk introduced *forest expressions*, which are essentially the same as context logic for trees with composition, but with the addition of two types of Kleene star and without the adjunct connectives. He has shown, by way of automata theory, that star-free forest expressions have the same expressive power as first-order logic.

Bojańczyk's result suggests once again that context logic and first-order logic have equal expressivity. However, as the expressive relationship between single- and multi-holed context logic is not fully known, and as it is still unclear whether adjunct elimination holds for the former, I cannot draw a firm conclusion in this regard.

1.5 Decidability

Given a logic, such as ambient logic or context logic, a natural question is whether or not it is decidable: that is, is there an algorithm that determines whether a given formula is valid (true in all models[4]) in a finite number of steps? For classical logics, having a decision procedure for validity is equivalent to having such a procedure for satisfiability (whether a formula is true in some model), since a formula is valid if its negation is unsatisfiable, and vice-versa. A third decision problem is that of model-checking: given a formula and a model, is the formula satisfied by the model? The decidability of such problems can be seen as a question of expressivity: can undecidable problems be expressed with the logic?

Calcagno, Yang and O'Hearn [CYO01] have studied the decidability and complexity of validity and model checking for fragments of separation logic. They have shown that both problems are undecidable in the presence of quantification, even for a restricted fragment of the logic. However, in the absence of quantification (although with the adjoint of composition), the problems are decidable, and indeed PSPACE-complete. Since the composition adjoint implic-

[4]By model, here, I mean, for instance, a tree.

itly quantifies over all heaps, a size argument is used to show that only a finitely bounded number of heaps need be considered.

Brotherston and Kanovich [BK10] have shown that separation logic is undecidable when uninterpreted propositional atoms occur. Their result simulates the execution of a simple (yet Turing-complete) machine in the logic, and thereby reduces the halting problem to validity in separation logic. Their results have undecidability implications for various forms of bunched logic.

For Ambient Logic, Cardelli and Gordon [CG00] originally provided a decision procedure for model checking a fragment of the logic that excludes the composition adjoint against ambients that do not include replication. Charatonik and Talbot [CT01] showed that model checking is undecidable when either of these restrictions is lifted. When the composition adjoint is included, the problem of satisfiability can be reduced to model checking, and satisfiability is undecidable even for a small fragment of the logic (with quantification). Charatonik *et al.* also show that model checking for Cardelli and Gordon's fragment is PSPACE-complete.

Calcagno, Cardelli and Gordon [CCG03] adapted the size-bounding approach of [CYO01] to a quantifier-free fragment of static Ambient Logic. Their result is by showing that the number of trees that need be considered to check satisfaction of an adjunct formula is finitely bounded. Since model checking and validity (and satisfiability) are equivalent in the presence of the adjoint connective, this leads to a proof for both results. They also provided a sound and complete proof system for validity, and establish a PSPACE lower bound on the complexity of validity and model checking. Conforti and Ghelli [CG04] established that adding Gabbay-Pitts freshness quantification does not break decidability by showing that fresh quantification can be extruded over the other connectives.

Dal Zilio, Lugiez and Meysonnier [DLM04] have shown the decidability of validity and model checking for a similar fragment of static ambient logic, including the composition adjoint, and extended with repetition (Kleene star). Their results are by effectively encoding formulae into a new logic, *sheaves logic*. They show that validity problem for sheaves logic can be decided by sheaves automata: tree automata with Presburger constraints.

Hague [Hag04] and Foster, Pierce and Schmitt [FPS07] have studied the practical implementations of the sheaves automata approach. Although validity and model checking have exponential complexity in the worst case, they have found optimisations whereby reasonable performance can be achieved in practice.

In Chapter 4 I show decidability for multi-holed context logic for sequences, terms and trees, including the adjunct connectives and quantification over hole labels, but not including general quantification or uninterpreted propositional atoms. The result is based on constructing finite automata corresponding to quantifier-free formulae. Simple algorithms can then determine whether a sequence, term or tree satisfies a formula, and whether the formula is valid or satisfiable in general on the basis of the corresponding automaton. For formulae with quantification over hole labels, the decision problems are reduced to the quantifier-free case by making use of the quantifier normalisation results established in §3.5.

The decision procedures given in Chapter 4 have NONELEMENTARY complexity. In particular, the number of states in an automaton implementing $K_1 \bullet\!\!-^{\exists} K_2$ has a number of states that is exponential in the number of states of the automaton implementing K_2.

1.6 Modular Reasoning

In designing and building complex systems, modularity is essential. By constructing the system from self-contained components, it can be robust to changes in configuration, and components can be reused and replaced. For example, consider a file server that maintains two data structures: a set of its clients and a tree hierarchy of the files it serves. These two structures are not intrinsically interdependent, and so they can be developed in isolation. Furthermore, they can be implemented by components from standard libraries, rather than needing to be implemented specifically for the file server. Moreover, if one of the components is replaced with a different implementation of the same functionality, the rest of the system is not affected.

Modularity is supported by two key concepts: locality and abstraction. For the components of a system, to be local means that that there are few dependencies between them. Dependencies between two components can arise indirectly if they both depend on a third component; for instance, if both components require some resource from the third, which can only supply a limited amount (*e.g.*, heap memory from a memory manager) — this kind of dependency can be mitigated if sufficient resource is assumed.

For the system, abstraction means that the individual components can be represented by what they do without consideration for how they actually do it. For instance, a set data structure can be viewed abstractly as a mathematical set, and its operations, such as adding and removing elements, can be seen

as operations on this abstract mathematical set. In reality, the set could be implemented in any number of ways: with a singly- or double-linked list, a bit vector, a red-black tree, or a skip list, to name a few.

Abstraction can support locality by exposing an abstract interface based on a resource model. For example, if a memory manager implementation allocates consecutive blocks of memory for consecutive allocation requests, then a component that uses the memory manager may depend on this assumption for two blocks it allocates. This introduces a dependency that no other component that also uses the memory manager will allocate a block in between the first component's allocation requests. By considering an abstract view of the memory manager which allocates blocks at nondeterministically chosen addresses, clients can no longer depend on the consecutive behaviour of the memory manager, which would violate locality by introducing indirect dependencies.

Locality in turn can support abstraction: if a component's subcomponents are independent, then the abstract view of the component can safely ignore their workings, whereas if their behaviour is sensitive to the wider context then the abstract view must account for this sensitivity. For example, if a component stores its data at a fixed location in memory, then this location must form part of the abstract view of the component, since there would be a dependency with any other component that used the same location; if the component were to store its data in a freshly-chosen location, however, there is no longer a potential dependency, and so the abstract view can be simplified. In essence, the abstract view of a component must declare its (potential) dependencies; locality minimises these, resulting in a simpler abstract interface.

In order to formally verify such complex systems, we need verification methods that also support modularity, reflecting the modularity of the systems themselves. I have already discussed how separation logic and context logic have been used to provide local reasoning about programs. In Chapter 5 I will present a formal program logic for verifying programs written in a simple imperative language that can be specialised with commands for updating and manipulating different kinds of resource.

Data refinement is the general process by which abstract component specifications are verifiably realised by concrete implementations [dE99]. The specification of the component includes an abstract data model and describes how the component's operations behave on these abstract data. An implementation is a data refinement if every program that uses the implementation *refines* the program which uses the abstract specification of the component — that is, all of its observable behaviours are contained within the possible behaviours of the

abstract component.

Various efforts have been made to combine the separation logic approach to local reasoning with some form of data refinement or abstraction. O'Hearn, Yang and Reynolds introduced the *hypothetical frame rule* [OYR04], which provides a degree of abstraction in reasoning about components by hiding the internal state of the component from clients. In their system, the internal state is encapsulated by a static module invariant that is assumed to hold on commencing each module operation and is re-established on completion of each operation. Beyond information hiding, the rule provides no abstraction in the sense that some details of the internal representation of the dynamic state are exposed to clients. For instance, if a set module's specification needs to make assertions about whether specific elements are in the set, then the specification must expose the internal representation that corresponds to those elements being in the set.

Parkinson and Bierman introduced *abstract predicates* [PB05, Par05] to separation logic, which encapsulate the details of a component's state. The client of a component is ignorant of the actual representation of the state, but is presented with an abstract view. For example, a set module might present the predicate $\mathsf{Set}(s, S)$ to the client to indicate that there is a representation of the (abstract mathematical) set S identified by s. The specification of a command for adding an element to such a set could then be given as follows:

$$\{\mathsf{Set}(s, S)\} \quad \mathtt{add}(s, v) \quad \{\mathsf{Set}(s, S \cup \{v\})\}$$

Since the client is oblivious to the concrete representation of the $\mathsf{Set}(s, S)$ predicate, the proof system only permits it to manipulate the set through the operations exposed by the module. This means that a client can equally well use another implementation that provides the same specification, although the actual representation may be radically different.

Abstract predicates, since they encapsulate resources, can be subject to local reasoning. For example, the above `add` operation can be seen to leave a second set $\mathsf{Set}(t, T)$ alone by introducing it with the frame rule thus:

$$\{\mathsf{Set}(s, S) * \mathsf{Set}(t, T)\} \quad \mathtt{add}(s, v) \quad \{\mathsf{Set}(s, S \cup \{v\}) * \mathsf{Set}(t, T)\}$$

This is simply repackaging locality that exists at the implementation level — the sets s and t are implemented by different pieces of heap. Abstract predicates do not provide a mechanism for exposing locality that is not inherited from the implementation level.

For example, the operation of adding a value, say 5, to the set s can be viewed as local not only to the set but also to the particular value: the operation does

not affect whether 3, say, is an element of the set. Thus, the operation of adding a value to a set could be specified abstractly as follows:

$$\{\mathsf{out}(s, v)\} \quad \mathtt{add}(s, v) \quad \{\mathsf{in}(s, v)\}$$

In this specification, $\mathsf{out}(s, v)$ denotes the resource "value v is not in the set s," and $\mathsf{in}(s, v)$ denotes the resource "value v is in the set s". Knowledge about other values can be added with the frame rule thus:

$$\{\mathsf{out}(s, v) * \mathsf{in}(v, w)\} \quad \mathtt{add}(s, v) \quad \{\mathsf{in}(s, v) * \mathsf{in}(v, w)\}$$

In order to prove that an implementation satisfies such a specification using abstract predicates, there must be some tangible resource at the implementation level that corresponds to $\mathsf{out}(s, v)$ that is sufficient to safely perform operations on s pertaining to value v, but that is at the same time disjoint from $\mathsf{out}(s, w)$ (where $w \neq v$). This is often not the case.

Consider the implementation of a set using an ordered linked list in the heap, starting from address s (the identifier of the set). In order to add the value 5 to the set, the operation must traverse the list from its head to find the place where 5 belongs and insert a new node containing the value. Thus, $\mathsf{out}(s, 5)$ must include this entire footprint. Yet $\mathsf{out}(s, 3)$ must, by the same logic, include the part of the list from its head to the location at which 3 belongs. The two are not disjoint, and so abstract predicates cannot establish that this abstract local specification holds for the set implementation.

In Chapter 6, I describe an approach that I call *locality refinement* which was developed by Gardner, Wheelhouse and me [DYGW10a] to establish that module implementations satisfy abstract local specifications, such as the set specification described above. The approach relates the abstract local axiomatic semantics of a module to the implementation of the module by establishing that all of the abstract inference rules, including the frame rule and axioms for each of the module operations, hold under a translation of the abstract data model to the data model on which the implementation is based. For example, to verify the list-based set, low-level local reasoning about heap update would be used to establish that every specification that can be proved about a client using high-level local reasoning about sets holds for some interpretation of the set data model in the heap.

I present two techniques for establishing locality refinement: *locality-preserving* and *locality-breaking* translations. Locality-preserving translations (§6.3) are best applied when there is a close correspondence between locality at the high level and locality at the low level. For example, an implementation of a tree

data structure in the heap might represent each node with a heap cell containing pointers to the node's parent, immediate sibling and first child; locality is closely preserved because the piece of heap required to represent a subtree can be separated from the heap that represents the rest of the context it appears in. Even when there is a close correspondence between locality at the high and low levels, there can still be some overlap between the representation of a tree (say) and the representation of its context. For example, disposing of a tree may require updating pointers in the nodes immediately adjacent to the three, which are nominally part of the context. This overlap is termed *crust*, and is accommodated by the locality-preserving technique.

Locality-breaking translations (§6.4) are suited to cases where there is not a close correspondence between locality at the high and low levels. In such cases, there is typically a greater burden to establish that the frame rule holds soundly at the high level. The locality-breaking technique simplifies the burden to just considering applications of the frame rule to the basic operations of the module.

1.6.1 Concurrency

A number of techniques have been put forward that help to support reasoning about concurrent programs in a modular way. Concurrency adds an extra dimension to the problem of developing and reasoning about programs: instead of only being at one code point at any given time as with a sequential program, a concurrent program's execution can have many threads, each with its own code point. Interactions between threads can be unpredictable because of the non-deterministic manner in which they are scheduled; therefore, it is necessary to account for every possible scheduling in reasoning about a concurrent program.

Linearisability, introduced by Herlihy and Wing [HW90], presents an abstract view of a component for which each of its operations appears to take effect instantaneously. This radically simplifies the interference possibilities that a client of the component has to account for. As with many modular reasoning techniques, linearisability depends on clients not breaching the abstraction boundary; if a client does, it may be able to observe some intermediate state between an operation being invoked and taking effect. Adding new operations to a component, therefore, may require rechecking all of the component's operations to ensure linearisability. Linearisability is conceptually simple, but in some cases it can be unnecessary, unattainable or impractical.

Rely/guarantee conditions, introduced to Hoare logic by Jones [Jon81], abstract the interference caused by concurrent threads with relations: the *rely* relation abstracts the interference that a thread must expect from others, while

the *guarantee* relation abstracts the interference that a thread may cause others. Key assertions in a proof are are required to be *stable*, that is, if they hold for a given state then they also hold for any state that it is related to by the rely relation. This means that the program can tolerate any interference that is abstracted by the rely. When two programs have compatible rely and guarantee conditions, they can be executed in parallel, as embodied by the following rule:

$$\frac{R \cup G_2, G_1 \vdash \{P\}\ \mathbb{C}_1\ \{Q_1\} \quad R \cup G_1, G_2 \vdash \{P\}\ \mathbb{C}_2\ \{Q_2\}}{R, G_1 \cup G_2 \vdash \{P\}\ \mathbb{C}_1 \parallel \mathbb{C}_2\ \{Q_1 \wedge Q_2\}}$$

One limitation of rely-guarantee reasoning is that it operates on an essentially global view of the state, thereby impinging on modularity. It may also be necessary to introduce auxiliary variables to programs in order to verify them. Auxiliary variables do not affect the behaviour of the program, but they do abstract the program's control flow, which can be important when the order of operations is significant to the interference.

Concurrent separation logic, introduced by O'Hearn [O'H04], applies the separation logic view of state as resource to the concurrent setting: if two threads use disjoint resources then they can be safely executed in parallel without any interference between them. This is encapsulated in the elegant parallel rule of concurrent separation logic:

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

Resources can be shared between threads through *resource invariants*; these resources can only be accessed inside atomic blocks, for which the resource invariant is assumed to hold on entering the block and required to be re-established on leaving the block. As with rely/guarantee, auxiliary variables may be needed to track control flow; in fact, they are needed to an even greater extent because invariants do not convey the evolution of the state in the way that relations do.

Bornat *et al.* [BCOP05] extended concurrent separation logic by introducing a permissions model for heap resource. This meant that two or more threads could have access to the same heap cell, *outside* of atomic blocks, provided that they do not write to it. Their fractional permission model, based on work of Boyland [Boy03], gives each thread requiring read-only access to a cell a fractional permission in the interval $(0, 1]$. The total permission for any cell, among all threads, cannot exceed 1, which corresponds to exclusive permission. A thread having permission 1 therefore can also safely write to the heap cell, since no other thread is expecting to read it.

RGSep, introduced by Vafeiadis and Parkinson [Vaf07, VP07], combines and subsumes rely/guarantee and concurrent separation logic, attaining the benefits

of a relational view of interference and a local view of state. The approach partitions the state into a shared portion, which is subject to interference described by rely and guarantee relations and is common to all threads, and a local portion, which is not subject to any interference and is exclusive to individual threads. Shared state assertions are denoted by boxes, as in $\boxed{P}$, and combining shared assertions with $*$ is equivalent to combining them with $\wedge$, *i.e.*, $\boxed{P} * \boxed{Q} \equiv \boxed{P} \wedge \boxed{Q} \equiv \boxed{P \wedge Q}$. This is essential to the nature of shared state: however the state is divided, the shared part is the same for all portions. RGSep's parallel rule is a combination of the rely/guarantee and separation logic rules:

$$\frac{R \cup G_2, G_1 \vdash \{P_1\}\ \mathbb{C}_1\ \{Q_1\} \quad R \cup G_1, G_2 \vdash \{P_2\}\ \mathbb{C}_2\ \{Q_2\}}{R, G_1 \cup G_2 \vdash \{P_1 * P_2\}\ \mathbb{C}_1 \parallel \mathbb{C}_2\ \{Q_1 * Q_2\}}$$

Deny-guarantee, introduced by Dodds *et al.* [DFPV09], builds on RGSep by introducing permissions on *actions* — relations which describe updates to shared state. These permissions constrain the rely and guarantee relations by determining whether actions are permitted to the thread (*i.e.*, in the guarantee), the environment (*i.e.*, in the rely), both, or neither. The read/write permissions for heap cells can be viewed as a special case of this: read permissions disallow both the thread and environment from modifying the cell, while full read/write permissions allow the thread to modify the cell, but not the environment.

A nice consequence of this view of permissions as resource is that they can be used to realise abstract resources. Returning to the set example, $\mathsf{out}(s, v)$ can be seen as a combination of the *knowledge* that value v is not in the set s and *exclusive permission* to change whether that value v is in the set. In a concurrent setting, this exclusive permission can be a complicated beast, entailing both exclusive and non-exclusive permissions to manipulate the underlying structure.

In Chapter 7 I present *concurrent abstract predicates* [DYDG$^+$10], an approach to providing abstract specifications for concurrent components in terms of disjoint resources, which are realised by a permissions model derived from deny-guarantee. The combination of abstract predicates and a rich underlying permissions model make it possible to verify concurrent implementations of abstract data structures which present a fiction of disjointness to clients.

Part I

Reasoning about Data

Chapter 2

Context Logic

In this chapter, I give the basic, formal definitions for context logic in the various forms I use it throughout this thesis. I begin by describing the tree model (§2.1.1), which is the main model used throughout Part I. Based on this model, I define three varieties of context logic: simple context logic CL^s_{Tree} (§2.1.3), context logic with composition CL^c_{Tree} (§2.1.5), and multi-holed context logic CL^m_{Tree} (§2.1.7).

Common to each of these logics is the ability to describe a tree in terms of the application of a context to a subtree: separating application. The context and subtree from which the tree is composed are described independently. Context logic with composition extends simple context logic by allowing contexts to be described in terms of the composition of two contexts. While context composition was included in an early version of context logic [CGZ04], it was not necessary to express the assertions required for reasoning about tree update and so was subsequently not considered an integral part of the basic logic. In his thesis [Zar07], Zarfaty considers a number of variations of context logic, both including and omitting context composition. Multi-holed context logic, which Calcagno, Gardner and I introduced in [CDYG07], further extends context logic by allowing contexts to be described as the composition of multi-holed contexts. Thus, each logic extends the previous in terms of how trees and contexts may be split apart.

In §2.2, I generalise the definitions to arbitrary models for context logic, which take the form of *context algebras*. I show a number of examples of data structures that fit the definitions of context algebra (§2.3). Although context algebras are less general than the Zarfaty's abstract models for context logic [Zar07], they are sufficiently general to cover the models considered in this thesis.

2.1 Context Logic for Trees

2.1.1 Trees

Trees, for the purposes of Part I, are finite, non-uniquely-labelled, unranked, ordered forests:

- they are *finite* in that only finite branching is permitted and branches must have finite depth;
- they are *non-uniquely-labelled* in that each node of the tree is associated with a label, and that label may not be unique to that node;
- they are *unranked* in that there is no enforced correspondence between the label that a node has and the number of branches that proceed from it;
- they are *ordered* in that the branches at each node have a specific total order;
- they are *forests* in that any number of nodes can occur at the root level.

(In Part II, §6.1.2, I will consider uniquely labelled trees: that is, trees for which each node has a distinctive label.)

In the following, let Σ be an (infinite) alphabet, the alphabet of *node labels*, ranged over by $\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{a}', \mathbf{a}_1, \ldots$.

Definition 2.1 (Trees)**.** The set of *trees* Tree, ranged over by $t, t_1, \ldots$, is defined inductively as follows:

$$t ::= \varnothing \;\mid\; \mathbf{a}[t] \;\mid\; t_1 \otimes t_2$$

where $\otimes$ is considered to be associative and to have $\varnothing$ as its identity.

Example 2.1. The following are all examples of trees:

$$\varnothing \tag{2.1}$$

$$\mathbf{a}[\mathbf{b}[\varnothing]] \tag{2.2}$$

$$\mathbf{a}[\mathbf{a}[\varnothing] \otimes \mathbf{b}[\varnothing]] \tag{2.3}$$

$$\mathbf{a}[\mathbf{a}[\varnothing] \otimes \mathbf{b}[\varnothing]] \otimes \mathbf{a}[\mathbf{b}[\varnothing] \otimes \mathbf{a}[\mathbf{b}[\varnothing]]]. \tag{2.4}$$

Figure 2.1 illustrates (2.4).

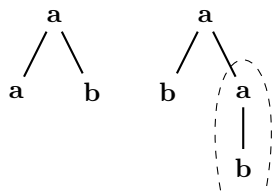


Figure 2.1: A graphical illustration of the tree (2.4), with the decomposition (2.12) indicated

2.1.2 Tree Contexts and Application

The inductive definition of trees leads to a very natural notion of a subtree of a given tree: a subtree is a tree that may be extended to the full tree by applying the tree constructors. For example, (2.1), (2.2), (2.3) and (2.4) are all subtrees of (2.4). The way that the subtree is extended is called a *tree context*, which is essentially a tree with a hole into which another tree may be placed. These tree contexts are formally defined in a manner that is analogous to the definition of trees themselves.

Definition 2.2 (Tree Contexts)**.** The set of *tree contexts* $\mathsf{C}_{\mathsf{Tree}}$, ranged over by $c, c_1, \ldots$, is defined inductively as follows:

$$c ::= - \;\mid\; \mathbf{a}[c] \;\mid\; c \otimes t \;\mid\; t \otimes c$$

where, again, $\otimes$ is considered to be associative and to have $\varnothing$ as its identity.

Example 2.2. The following are all examples of tree contexts:

$$-\tag{2.5}$$

$$- \otimes \mathbf{a}[\mathbf{b}[\varnothing] \otimes \mathbf{a}[\mathbf{b}[\varnothing]]]\tag{2.6}$$

$$\mathbf{a}[\mathbf{a}[\varnothing] \otimes \mathbf{b}[\varnothing]] \otimes \mathbf{a}[\mathbf{b}[\varnothing] \otimes -]\tag{2.7}$$

$$\mathbf{a}[\mathbf{a}[\varnothing] \otimes \mathbf{b}[-]] \otimes \mathbf{a}[\mathbf{b}[\varnothing] \otimes \mathbf{a}[\mathbf{b}[\varnothing]]]\tag{2.8}$$

$$\mathbf{a}[\mathbf{a}[\varnothing] \otimes \mathbf{b}[\varnothing]] \otimes - \otimes \mathbf{a}[\mathbf{b}[\varnothing] \otimes \mathbf{a}[\mathbf{b}[\varnothing]]].\tag{2.9}$$

The operation of extending a tree with a context is *context application*.

Definition 2.3 (Context Application)**.** The *context application* operator $\bullet : \mathsf{C}_{\mathsf{Tree}} \times \mathsf{Tree} \rightharpoonup \mathsf{Tree}$ is defined by induction on the structure of contexts as

follows:

$$
\begin{aligned}
- \bullet t &\stackrel{\text{def}}{=} t \\
\mathbf{a}[c] \bullet t &\stackrel{\text{def}}{=} \mathbf{a}[c \bullet t] \\
(c \otimes t') \bullet t &\stackrel{\text{def}}{=} (c \bullet t) \otimes t' \\
(t' \otimes c) \bullet t &\stackrel{\text{def}}{=} t' \otimes (c \bullet t).
\end{aligned}
$$

Remark. In the above definition, context application is in fact a total function. This would not be the case if we imposed a well-formedness condition, such as the uniqueness of node labels: the application would be undefined if the result would not be a well-formed tree.

Example 2.3. The tree $\mathbf{a}[\mathbf{a}[\varnothing] \otimes \mathbf{b}[\varnothing]] \otimes \mathbf{a}[\mathbf{b}[\varnothing] \otimes \mathbf{a}[\mathbf{b}[\varnothing]]]$ can be expressed as the application of a context to a tree in all of the following ways:

$$- \bullet \mathbf{a}[\mathbf{a}[\varnothing] \otimes \mathbf{b}[\varnothing]] \otimes \mathbf{a}[\mathbf{b}[\varnothing] \otimes \mathbf{a}[\mathbf{b}[\varnothing]]] \tag{2.10}$$

$$- \otimes \mathbf{a}[\mathbf{b}[\varnothing] \otimes \mathbf{a}[\mathbf{b}[\varnothing]]] \bullet \mathbf{a}[\mathbf{a}[\varnothing] \otimes \mathbf{b}[\varnothing]] \tag{2.11}$$

$$\mathbf{a}[\mathbf{a}[\varnothing] \otimes \mathbf{b}[\varnothing]] \otimes \mathbf{a}[\mathbf{b}[\varnothing] \otimes -] \bullet \mathbf{a}[\mathbf{b}[\varnothing]] \tag{2.12}$$

$$\mathbf{a}[\mathbf{a}[\varnothing] \otimes \mathbf{b}[-]] \otimes \mathbf{a}[\mathbf{b}[\varnothing] \otimes \mathbf{a}[\mathbf{b}[\varnothing]]] \bullet \varnothing \tag{2.13}$$

$$\mathbf{a}[\mathbf{a}[\varnothing] \otimes \mathbf{b}[\varnothing]] \otimes - \otimes \mathbf{a}[\mathbf{b}[\varnothing] \otimes \mathbf{a}[\mathbf{b}[\varnothing]]] \bullet \varnothing. \tag{2.14}$$

(This is not an exhaustive list.) Figure 2.1 illustrates (2.12).

2.1.3 The Logic CL^s_{Tree}

I now give the definition of a simple context logic for trees, CL^s_{Tree}. Formulae of the logic either describe trees or contexts. The logic includes connectives that are specific to the structure of trees, connectives that relate to the context application operation, and basic Boolean connectives.

Definition 2.4 (Context Logic Formulae)**.** The set of *tree formulae* $\mathsf{P}^s_{\mathsf{Tree}}$, ranged over by $P, Q, P_1, \ldots$, and the set of *tree context formulae* $\mathsf{K}^s_{\mathsf{Tree}}$, ranged

over by $K, K_1, \ldots$ are defined inductively as follows:

$$\begin{array}{lll} P ::= \mathbf{0} & & \text{tree-specific formulae} \\ \quad \mid K \bullet P \mid K \bullet\!\!- P & & \text{structural formulae} \\ \quad \mid \text{false} \mid P_1 \to P_2 & & \text{Boolean formulae} \\ \\ K ::= \mathbf{a}[K] \mid K \otimes P \mid P \otimes K & & \text{tree-specific formulae} \\ \quad \mid \mathbf{I} \mid P_1 \mathrel{-\!\!\bullet} P_2 & & \text{structural formulae} \\ \quad \mid \text{False} \mid K_1 \to K_2 & & \text{Boolean formulae.} \end{array}$$

When the superscripts and subscripts of $\mathsf{P}^s_{\mathsf{Tree}}$ and $\mathsf{K}^s_{\mathsf{Tree}}$ are clear from context I will omit them.

Notation (Derived Formulae)**.** For notational convenience, additional logical connectives are defined in terms of the basic logical connectives as follows:

$$\begin{array}{rlrl} \mathbf{a}[P] & \stackrel{\text{def}}{=} \mathbf{a}[\mathbf{I}] \bullet P & P_1 \otimes P_2 & \stackrel{\text{def}}{=} (P_1 \otimes \mathbf{I}) \bullet P_2 \\ \neg P & \stackrel{\text{def}}{=} P \to \text{false} & \neg K & \stackrel{\text{def}}{=} K \to \text{False} \\ \text{true} & \stackrel{\text{def}}{=} \neg\text{false} & \text{True} & \stackrel{\text{def}}{=} \neg\text{False} \\ P_1 \vee P_2 & \stackrel{\text{def}}{=} (\neg P_1) \to P_2 & K_1 \vee K_2 & \stackrel{\text{def}}{=} (\neg K_1) \to K_2 \\ P_1 \wedge P_2 & \stackrel{\text{def}}{=} \neg(P_1 \to \neg P_2) & K_1 \wedge K_2 & \stackrel{\text{def}}{=} \neg(K_1 \to \neg K_2) \\ K \bullet\!\!-^{\exists} P & \stackrel{\text{def}}{=} \neg(K \bullet\!\!- \neg P) & P_1 \mathrel{-\!\!\bullet}^{\exists} P_2 & \stackrel{\text{def}}{=} \neg(P_1 \mathrel{-\!\!\bullet} \neg P_2). \end{array}$$

(Defining these connectives in terms of a primitive set helps to simply the theory of the logic, since semantics can be given purely in terms of the primitive connectives.)

Notation (Precedence and Associativity)**.** Brackets will generally be used to avoid ambiguity in formulae, however, the following conventions are used to disambiguate formulae without brackets. The logical connectives bind in the following order (those on the left binding tighter than those to the right):

$$\neg, \bullet, \otimes, \wedge, \vee, \quad \begin{array}{c} \bullet\!\!- \\ \mathrel{-\!\!\bullet} \\ \bullet\!\!-^{\exists} \\ \mathrel{-\!\!\bullet}^{\exists} \end{array} \quad , \to .$$

The connectives associate to the right, that is $P_1 \to P_2 \to P_3$ is considered to be $P_1 \to (P_2 \to P_3)$, for example.

Example 2.4. The following are examples of tree formulae:

$$\mathbf{a}[\mathbf{b}[\mathbf{0}]] \tag{2.15}$$

$$\mathrm{True} \bullet \mathbf{a}[\mathbf{0}] \tag{2.16}$$

$$\neg\mathbf{0} \otimes \neg\mathbf{0} \tag{2.17}$$

$$(\mathbf{0} \multimap \neg(\mathrm{True} \bullet \mathbf{b}[\mathbf{0}])) \bullet \mathbf{b}[\mathbf{0}] \tag{2.18}$$

$$\mathbf{a}[\mathrm{true}] \otimes \mathbf{b}[\mathrm{true}] \rightarrow (\mathbf{a}[\_] \bullet\!\!- \mathrm{True} \bullet (\neg\mathbf{0} \otimes \neg\mathbf{0})). \tag{2.19}$$

The following are examples of context formulae:

$$\mathbf{I} \otimes \mathrm{true} \tag{2.20}$$

$$\mathbf{a}[\mathbf{0}] \multimap \mathbf{a}[\mathbf{0}] \tag{2.21}$$

$$\mathbf{a}[\mathbf{a}[\mathbf{0}] \otimes \mathrm{True}]. \tag{2.22}$$

I define satisfaction relations to give semantics to formulae. In general, satisfaction relations relate worlds with formulae; for CL^s_{Tree}, the worlds are trees and tree contexts. Both worlds and formulae are divided into these two sorts, as we have seen. The following definition formalises this.

Definition 2.5 (Sorts)**.** The set of *sorts* for CL^s_{Tree}, Sort, ranged over by ς, is defined as follows:

$$\varsigma ::= \mathtt{d} \mid \mathtt{c}.$$

That is, there are two sorts: $\mathtt{d}$, the sort of trees, and $\mathtt{c}$, the sort of tree contexts. The *family of worlds* for CL^s_{Tree}, $\{\mathsf{World}_\varsigma\}_{\varsigma\in\mathsf{Sort}}$, is defined by $\mathsf{World}_\mathtt{d} = \mathsf{Tree}$, $\mathsf{World}_\mathtt{c} = \mathsf{C}_{\mathsf{Tree}}$. The *family of formulae* for CL^s_{Tree}, $\{\mathsf{Formula}_\varsigma\}_{\varsigma\in\mathsf{Sort}}$, is defined by $\mathsf{Formula}_\mathtt{d} = \mathsf{P}^s_{\mathsf{Tree}}$, $\mathsf{Formula}_\mathtt{c} = \mathsf{K}^s_{\mathsf{Tree}}$.

Definition 2.6 (Satisfaction Relations)**.** The *satisfaction relations* $\models_\mathtt{d} \subseteq \mathsf{Tree} \times \mathsf{P}$ and $\models_\mathtt{c} \subseteq \mathsf{C}_{\mathsf{Tree}} \times \mathsf{K}$, which denote satisfaction of a tree formula by a tree and satisfaction of a context formula by a tree context respectively, are defined by induction on the structure of formulae as follows:

$$\begin{array}{lcl}
t \models_\mathtt{d} \mathbf{0} & \iff & t = \varnothing \\
t \models_\mathtt{d} K \bullet P & \iff & \text{there exist } c, t' \text{ s.t. } t = c \bullet t' \text{ and} \\
 & & \quad c \models_\mathtt{c} K \text{ and } t' \models_\mathtt{d} P \\
t \models_\mathtt{d} K \bullet\!\!- P & \iff & \text{for all } c, t',\ t' = c \bullet t \text{ and } c \models_\mathtt{c} K \implies t' \models_\mathtt{d} P \\
t \models_\mathtt{d} \mathrm{false} & & \text{never} \\
t \models_\mathtt{d} P_1 \rightarrow P_2 & \iff & t \models_\mathtt{d} P_1 \implies t \models_\mathtt{d} P_2
\end{array}$$

$$
\begin{array}{lcl}
c \models_{\mathsf{c}} \mathbf{a}[K] & \iff & \text{there exists } c' \text{ s.t. } c = \mathbf{a}[c'] \text{ and } c' \models_{\mathsf{c}} K \\
c \models_{\mathsf{c}} K \otimes P & \iff & \text{there exist } c', t \text{ s.t. } c = c' \otimes t \text{ and} \\
 & & \quad c' \models_{\mathsf{c}} K \text{ and } t' \models_{\mathsf{d}} P \\
c \models_{\mathsf{c}} P \otimes K & \iff & \text{there exist } t, c' \text{ s.t. } c = t \otimes c' \text{ and} \\
 & & \quad t \models_{\mathsf{d}} P \text{ and } c' \models_{\mathsf{c}} K \\
c \models_{\mathsf{c}} \mathbf{I} & \iff & c = - \\
c \models_{\mathsf{c}} P_1 \mathrel{-\!\!\bullet} P_2 & \iff & \text{for all } t_1, t_2,\ t_2 = c \bullet t_1 \text{ and } t_1 \models_{\mathsf{d}} P_1 \implies \\
 & & \quad t_2 \models_{\mathsf{d}} P_2 \\
c \models_{\mathsf{c}} \text{False} & & \text{never} \\
c \models_{\mathsf{c}} P_1 \to P_2 & \iff & c \models_{\mathsf{c}} P_1 \implies c \models_{\mathsf{c}} P_2.
\end{array}
$$

Example 2.5. The following are examples of trees and tree-contexts satisfying and not satisfying formulae:

$$\mathbf{a}[\mathbf{b}[\varnothing]] \models_{\mathsf{d}} \mathbf{a}[\mathbf{b}[\mathbf{0}]] \tag{2.23}$$

$$\mathbf{a}[\mathbf{a}[\varnothing] \otimes \mathbf{b}[\varnothing]] \models_{\mathsf{d}} \text{True} \bullet \mathbf{a}[\mathbf{0}] \tag{2.24}$$

$$\mathbf{a}[\mathbf{b}[\varnothing]] \not\models_{\mathsf{d}} \neg\mathbf{0} \otimes \neg\mathbf{0} \tag{2.25}$$

$$\mathbf{a}[\mathbf{a}[\varnothing] \otimes \mathbf{b}[\varnothing]] \models_{\mathsf{d}} \neg\mathbf{0} \otimes \neg\mathbf{0} \tag{2.26}$$

$$\mathbf{a}[\mathbf{a}[\varnothing] \otimes \mathbf{b}[\varnothing]] \models_{\mathsf{d}} (\mathbf{0} \mathrel{-\!\!\bullet} \neg(\text{True} \bullet \mathbf{b}[\mathbf{0}])) \bullet \mathbf{b}[\mathbf{0}] \tag{2.27}$$

$$\mathbf{b}[\mathbf{b}[\varnothing]] \not\models_{\mathsf{d}} (\mathbf{0} \mathrel{-\!\!\bullet} \neg(\text{True} \bullet \mathbf{b}[\mathbf{0}])) \bullet \mathbf{b}[\mathbf{0}] \tag{2.28}$$

$$\text{for all } t \in \mathsf{Tree},\ t \models_{\mathsf{d}} \mathbf{a}[\text{true}] \otimes \mathbf{b}[\text{true}] \to (\mathbf{a}[-] \bullet\!\!- \text{True} \bullet (\neg\mathbf{0} \otimes \neg\mathbf{0})) \tag{2.29}$$

$$- \otimes \mathbf{a}[\mathbf{b}[\varnothing] \otimes \mathbf{a}[\mathbf{b}[\varnothing]]] \models_{\mathsf{c}} \mathbf{I} \otimes \text{true} \tag{2.30}$$

$$\text{holes} \models_{\mathsf{c}} \mathbf{a}[\mathbf{0}] \mathrel{-\!\!\bullet} \mathbf{a}[\mathbf{0}] \tag{2.31}$$

$$\mathbf{a}[-] \not\models_{\mathsf{c}} \mathbf{a}[\mathbf{a}[\mathbf{0}] \otimes \mathbf{I}]. \tag{2.32}$$

2.1.4 Context Composition

Just as contexts can be applied to trees, there is also a natural manner in which contexts can be composed.

Definition 2.7 (Context Composition)**.** The *context composition* operator $\circ : \mathsf{C}_{\mathsf{Tree}} \times \mathsf{C}_{\mathsf{Tree}} \rightharpoonup \mathsf{C}_{\mathsf{Tree}}$ is defined by induction on the structure of contexts as

follows:

$$
\begin{aligned}
- \circ c &\stackrel{\text{def}}{=} c \\
\mathbf{a}[c'] \circ c &\stackrel{\text{def}}{=} \mathbf{a}[c' \circ c] \\
(c' \otimes t) \circ c &\stackrel{\text{def}}{=} (c' \circ c) \otimes t \\
(t \otimes c') \circ c &\stackrel{\text{def}}{=} t \otimes (c' \circ c).
\end{aligned}
$$

The following result is a basic consequence of the definitions of composition and application.

Lemma 1 (Associativity of Composition)**.** *Context composition is associative and is also associative with respect to context application. That is, for all* $c, c', c'' \in \mathsf{C}_{\mathsf{Tree}}$ *and* $t \in \mathsf{Tree}$*,*

$$
\begin{aligned}
c \circ (c' \circ c'') &= (c \circ c') \circ c'' \\
c \bullet (c' \bullet t) &= (c \circ c') \bullet t.
\end{aligned}
$$

2.1.5 The Logic CL^c_{Tree}

I now give an extension of the simple context logic which incorporates connectives for describing context composition, CL^c_{Tree}.

Definition 2.8 (Context Logic Formulae)**.** The set of *tree formulae* $\mathsf{P}^c_{\mathsf{Tree}}$, ranged over by $P, Q, P_1, \ldots$, and the set of *tree context formulae* $\mathsf{K}^c_{\mathsf{Tree}}$, ranged over by $K, K_1, \ldots$ are defined inductively as follows:

$P ::= \mathbf{0}$	tree-specific formulae
$\mid K \bullet P \mid K \bullet\!\!- P$	structural formulae
$\mid$ false $\mid P_1 \to P_2$	Boolean formulae
$K ::= \mathbf{a}[K] \mid K \otimes P \mid P \otimes K$	tree-specific formulae
$\mid \mathbf{I} \mid P_1 \mathrel{-\!\!\bullet} P_2$	structural formulae
$\mid K_1 \circ K_2 \mid K_1 \circ\!\!- K_2 \mid K_1 \mathrel{-\!\!\circ} K_2$	**composition formulae**
$\mid$ False $\mid K_1 \to K_2$	Boolean formulae.

Definition 2.9 (Sorts)**.** For CL^c_{Tree}, the set of *sorts*, Sort, and the *family of worlds*, $\{\mathsf{World}_\varsigma\}_{\varsigma \in \mathsf{Sort}}$, are as for CL^s_{Tree}. The *family of formulae* for CL^c_{Tree}, $\{\mathsf{Formula}_\varsigma\}_{\varsigma \in \mathsf{Sort}}$, is defined by $\mathsf{Formula}_{\mathsf{d}} = \mathsf{P}^c_{\mathsf{Tree}}$, $\mathsf{Formula}_{\mathsf{c}} = \mathsf{K}^c_{\mathsf{Tree}}$.

Definition 2.10 (Satisfaction Relations)**.** The *satisfaction relations* $\models_{\mathsf{d}} \subseteq \mathsf{Tree} \times \mathsf{P}$ and $\models_{\mathsf{c}} \subseteq \mathsf{C}_{\mathsf{Tree}} \times \mathsf{K}$, which denote satisfaction of a tree formula by

a tree and satisfaction of a context formula by a tree context respectively, are defined by induction on the structure of formulae. The cases for each connective inherited from CL^s_{Tree} are as in Def. 2.6; the remaining cases are as follows:

$$\begin{aligned}
c \models_{\mathsf{c}} K_1 \circ K_2 \quad &\iff \quad \text{there exist } c', c'' \text{ s.t. } c = c' \circ c'' \text{ and} \\
&\qquad\qquad c' \models_{\mathsf{c}} K_1 \text{ and } c'' \models_{\mathsf{c}} K_2 \\
c \models_{\mathsf{c}} K_1 \circ\!\!- K_2 \quad &\iff \quad \text{for all } c', c'',\ c'' = c' \circ c \text{ and } c' \models_{\mathsf{c}} K_1 \implies c'' \models_{\mathsf{c}} K \\
c \models_{\mathsf{c}} K_1 -\!\!\circ K_2 \quad &\iff \quad \text{for all } c', c'',\ c'' = c \circ c' \text{ and } c' \models_{\mathsf{c}} K_1 \implies c'' \models_{\mathsf{c}} K.
\end{aligned}$$

Example 2.6. The following are examples of trees and tree-contexts satisfying formulae with context composition:

$$\mathbf{a}[\mathbf{b}[-]] \models_{\mathsf{c}} \mathbf{a}[\mathbf{I}] \circ \mathbf{b}[\mathbf{I}] \tag{2.33}$$

$$\mathbf{b}[\mathbf{a}[\varnothing]] \otimes \mathbf{a}[\mathbf{a}[\varnothing]] \models_{\mathsf{c}} \neg(\top \circ \mathbf{b}[\top]) \bullet \mathbf{a}[\mathbf{0}]. \tag{2.34}$$

2.1.6 Multi-holed Tree Contexts and Composition

So far, I have only had contexts that have a single hole. A very natural generalisation would be to allow contexts to have multiple holes. Previously, the context hole has served to mark the position at which a tree was removed, with the intention of replacing it with another. In this spirit, it seems natural that holes in a multi-holed context should be distinguished from each other in order to maintain the relationship between a subtree and its location. Each hole, therefore, has a label, and that label is unique to it (within the context). In the following, let X be an (infinite) alphabet, the *alphabet of hole labels*, disjoint from Σ, and ranged over by $x, y, x', x_1, \ldots$.

Definition 2.11 (Multi-holed Tree Contexts)**.** The set of *multi-holed tree contexts* $\mathsf{C}^m_{\mathsf{Tree}}$, ranged over by $c, c_1, \ldots$, is defined inductively as follows:

$$c ::= \varnothing \;\mid\; x \;\mid\; \mathbf{a}[c] \;\mid\; c_1 \otimes c_2$$

where $\otimes$ is considered to be associative and to have $\varnothing$ as its identity, and each $x \in \mathrm{X}$ is allowed to occur *at most once* in a given context.

Definition 2.12 (Hole Labels)**.** The function $\mathrm{holes} : \mathsf{C}^m_{\mathsf{Tree}} \to \mathcal{P}(\mathrm{X})$ returns the set of hole labels appearing in a multi-holed context. Formally, it is defined by induction on the structure of contexts as follows:

$$\begin{aligned}
\mathrm{holes}\, \varnothing &\stackrel{\text{def}}{=} \emptyset \\
\mathrm{holes}\, x &\stackrel{\text{def}}{=} \{x\} \\
\mathrm{holes}\, \mathbf{a}[c] &\stackrel{\text{def}}{=} \mathrm{holes}\, c \\
\mathrm{holes}\, c_1 \otimes c_2 &\stackrel{\text{def}}{=} \mathrm{holes}\, c_1 \cup \mathrm{holes}\, c_2.
\end{aligned}$$

Definition 2.13 (Substitution)**.** Substitution of a context for a context hole is defined in standard fashion, by induction over the structure of the context as follows:

$$\begin{aligned}
\varnothing[c/x] &\stackrel{\text{def}}{=} \varnothing \\
x[c/x] &\stackrel{\text{def}}{=} c \\
y[c/x] &\stackrel{\text{def}}{=} y \\
\mathbf{a}[c'][c/x] &\stackrel{\text{def}}{=} \mathbf{a}[c'[c/x]] \\
(c' \otimes c'')[c/x] &\stackrel{\text{def}}{=} (c'[c/x]) \otimes (c''[c/x]).
\end{aligned}$$

Remark. If the hole labels of two contexts overlap then the substitution of one into the other may not result in a valid context. I shall avoid using substitution when such a result is possible.

Definition 2.14 (Multi-holed Context Composition)**.** For each hole label, $x \in \mathrm{X}$, the *context composition (with respect to x)* operator $\circledcirc x$: $\mathsf{C}^m_{\mathsf{Tree}} \times \mathsf{C}^m_{\mathsf{Tree}} \rightharpoonup \mathsf{C}^m_{\mathsf{Tree}}$ is defined in terms of substitution as follows:

$$c \circledcirc x\ c' \stackrel{\text{def}}{=} \begin{cases} c[c'/x] & \text{if } x \in \operatorname{holes} c \text{ and } \operatorname{holes} c \cap \operatorname{holes} c' \subseteq \{x\} \\ \textit{undefined} & \text{otherwise.} \end{cases}$$

The following results are basic consequences of the definition of multi-holed context composition.

Lemma 2 (Quasi-associativity of Composition)**.** *For all $c, c', c'' \in \mathsf{C}^m_{\mathsf{Tree}}$, for all $x \in \operatorname{holes} c$ and all $y \in \operatorname{holes} c'$ with either $y \notin \operatorname{holes} c$ or $y = x$,*

$$(c \circledcirc x\ c') \circledcirc y\ c'' = c \circledcirc x\ (c' \circledcirc y\ c'').$$

(Undefined terms are considered equal here.)

Lemma 3 (Quasi-commutativity of Composition)**.** *For all $c, c', c'' \in \mathsf{C}^m_{\mathsf{Tree}}$, for all $x, y \in \mathrm{X}$ with $x \notin \operatorname{holes} c''$ and $y \notin \operatorname{holes} c'$,*

$$(c \circledcirc x\ c') \circledcirc y\ c'' = (c \circledcirc y\ c'') \circledcirc x\ c'.$$

(Again, undefined terms are considered equal.)

2.1.7 The Logic CL^m_{Tree}

I now give a multi-holed version of context logic for trees, CL^m_{Tree}.

Whereas node labels are referenced directly in formulae, hole labels are referenced through hole variables. The reason for this is that, while the specific

value of a node label may carry information about the node, hole labels purely as a means to compose and decompose trees — their specific values are not important, just the knowledge of whether or not two hole labels are the same. In the following, let Θ be an (infinite) alphabet, the alphabet of *hole variables*, ranged over by $\alpha, \beta, \gamma, \alpha', \alpha_1, \ldots$.

Definition 2.15 (Context Logic Formulae)**.** The set of *formulae* $\mathsf{K}^m_{\mathsf{Tree}}$, ranged over by $K, K_1, \ldots$ is defined inductively as follows:

$$\begin{array}{lll} K ::= \mathbf{0} \mid \mathbf{a}[K] \mid K_1 \otimes K_2 & & \text{tree-specific formulae} \\ \quad \mid \alpha \mid K_1 \circ_\alpha K_2 \mid K_1 \circ\!\!-_\alpha K_2 & & \text{structural formulae} \\ \quad \mid K_1 -\!\!\circ_\alpha K_2 \mid \exists\alpha.\, K & & \\ \quad \mid \mathrm{False} \mid K_1 \to K_2 & & \text{Boolean formulae.} \end{array}$$

Notation (Derived Formulae)**.** For notational convenience, additional logical connectives are defined in terms of the basic logical connectives as follows:

$$\mapsto \stackrel{\text{def}}{=} \exists\alpha.\, \alpha$$
$$\circledcirc \stackrel{\text{def}}{=} \neg(\mathrm{True} \circ_\alpha \alpha).$$

A logical environment is a valuation of logical variables. In the case of CL^m_{Tree}, logical variables are elements of the set Θ and take hole labels from the set X as their values.

Definition 2.16 (Logical Environment)**.** The set of logical environments LEnv is defined to be the set of finite partial functions $\Theta \rightharpoonup_{\text{fin}} \mathrm{X}$. The set LEnv is ranged over by $\sigma, \sigma', \ldots$.

For CL^m_{Tree}, worlds do not consist only of a tree context, but also include a logical environment which is used to interpret the variables of formulae. Worlds are divided into sorts according to the domain of their logical environment. Formulae are divided into sorts according to their free variables.

Definition 2.17 (Sorts)**.** The set of *sorts* for CL^m_{Tree}, Sort, ranged over by ς, is defined as follows:

$$\varsigma ::= \mathsf{c}\phi$$

where $\phi \in \mathcal{P}_{\text{fin}}(\Theta)$ ranges over finite sets of hole variables. That is, there is a sort $\mathsf{c}\phi$ for each finite set of hole variables ϕ. The *family of worlds* for CL^m_{Tree}, $\{\mathsf{World}_\varsigma\}_{\varsigma \in \mathsf{Sort}}$, is defined by

$$\mathsf{World}_\varsigma = \{(c, \sigma) \in \mathsf{C}^m_{\mathsf{Tree}} \times \mathsf{LEnv} \mid \mathsf{c}(\mathrm{dom}\,\sigma) = \varsigma\}\,.$$

The *family of formulae* for CL^m_{Tree}, $\{\mathsf{Formula}_\varsigma\}_{\varsigma\in\mathsf{Sort}}$, is defined by

$$\mathsf{Formula}_\varsigma = \{K \in \mathsf{K}^m_{\mathsf{Tree}} \mid \text{there exists } \phi \text{ s.t. } \varsigma = \mathsf{c}\phi \text{ and } \mathrm{fv}\, K \subseteq \phi\}\,.$$

Definition 2.18 (Satisfaction Relations)**.** The *satisfaction relations* $\models_\varsigma \subseteq \mathsf{World}_\varsigma \times \mathsf{Formula}_\varsigma$, which denote satisfaction of a formula by a multi-holed tree context with respect to a logical environment (for each sort ς), are defined by induction on the structure of formulae as follows, where $\varsigma = \mathsf{c}\phi$ for some $\phi \in \mathcal{P}_{\mathrm{fin}}(\Theta)$, and, when applicable, $x = \sigma\alpha$ and $\alpha \in \phi$:

$$\begin{array}{lcl}
c, \sigma \models_\varsigma \mathbf{0} & \iff & c = \varnothing \\
c, \sigma \models_\varsigma \mathbf{a}[K] & \iff & \text{there exists } c' \text{ s.t. } c = \mathbf{a}[c'] \text{ and } c', \sigma \models_\varsigma K \\
c, \sigma \models_\varsigma K_1 \otimes K_2 & \iff & \text{there exist } c_1, c_2 \text{ s.t. } c = c_1 \otimes c_2 \text{ and} \\
 & & \quad c_1, \sigma \models_\varsigma K_1 \text{ and } c_2, \sigma \models_\varsigma K_2 \\
c, \sigma \models_\varsigma \alpha & \iff & c = x \\
c, \sigma \models_\varsigma K_1 \circ_\alpha K_2 & \iff & \text{there exist } c_1, c_2 \text{ s.t. } c = c_1 \mathbin{\textcircled{x}} c_2 \text{ and} \\
 & & \quad c_1 \models_\varsigma K_1 \text{ and } c_2 \models_\varsigma K_2 \\
c, \sigma \models_\varsigma K_1 \circ\!\!- _\alpha K_2 & \iff & \text{for all } c_1, c_2,\ c_2 = c_1 \mathbin{\textcircled{x}} c \text{ and } c_1, \sigma \models_\varsigma K_1 \implies \\
 & & \quad c_2, \sigma \models_\varsigma K_2 \\
c, \sigma \models_\varsigma K_1 -\!\!\circ_\alpha K_2 & \iff & \text{for all } c_1, c_2,\ c_2 = c \mathbin{\textcircled{x}} c_1 \text{ and } c_1, \sigma \models_\varsigma K_1 \implies \\
 & & \quad c_2, \sigma \models_\varsigma K_2 \\
c, \sigma \models_\varsigma \exists\beta.\, K & \iff & \text{there exists } y \text{ s.t. } c, \sigma[\beta \mapsto y] \models_{\mathsf{c}(\phi \cup \{\beta\})} K \\
c, \sigma \models_\varsigma \mathrm{False} & & \text{never} \\
c, \sigma \models_\varsigma K_1 \to K_2 & \iff & c, \sigma \models_\varsigma K_1 \implies c, \sigma \models_\varsigma K_2.
\end{array}$$

Freshness Quantification

The logic CL^m_{Tree} includes existential quantification of hole variables. It purpose is to allow us to forget about the actual identities of holes — it does not matter what a hole is called so long as we can refer to it in the appropriate places. Often, it is desirable that a hole label should be sufficiently fresh, *i.e.*, not already used for something else. The Gabbay-Pitts freshness quantifier, $\mathsf{И}$, achieves exactly this.

Notation (Freshness)**.** For $y \in \mathrm{X}$ and $c \in \mathsf{C}^m_{\mathsf{Tree}}$, $y \sharp c$ denotes that $y \notin \mathrm{holes}\, c$. For $y \in \mathrm{X}$ and $\sigma \in \mathsf{LEnv}$, $y \sharp \sigma$ denotes that $y \notin \mathrm{range}\, \sigma$. The notation $y \sharp c, \sigma$ denotes that $y \sharp c$ and $y \sharp \sigma$.

Definition 2.19 (Freshness Quantification)**.** Formulae of $CL^m_{\mathsf{Tree}} + \mathsf{И}$ are constructed according to the inductive rules for formulae of CL^m_{Tree} with the addition

of the following rule:

$$K ::= \ldots \mid \mathit{И}\alpha.\, K.$$

The satisfaction relations are defined as for CL^m_{Tree} but with the following additional case, where $\varsigma = \mathsf{c}\phi$:

$$c, \sigma \models_\varsigma \mathit{И}\beta.\, K \qquad \Longleftrightarrow \qquad \text{there exists } y \text{ s.t. } y \sharp c, \sigma \text{ and } c, \sigma[\beta \mapsto y] \models_{\mathsf{c}(\phi \cup \{\beta\})} K.$$

2.2 General Context Logic

The fundamental concept of context logic — that data can be split into subdata and context — is not exclusive to trees, but applies to a broad spectrum of abstract data structures. In §2.1.1, I show a number of examples of such data structures. First, however, I define context logic in terms of an arbitrary abstract data structure. This allows us to view context logic for each specific data structure as an instance of a general context logic.

2.2.1 Simple Context Logic

Definition 2.20 (Simple Context Algebra)**.** A *simple context algebra* $\mathcal{A} = (\mathsf{D}, \mathsf{C}, \bullet, \mathbf{I})$ consists of

- a non-empty set of abstract data structures, D,
- a non-empty set of contexts, C,
- an application operator, $\bullet : \mathsf{C} \times \mathsf{D} \rightharpoonup \mathsf{D}$, and
- a distinguished set of contexts, $\mathbf{I} \subseteq \mathsf{C}$,

for which, for all $s \in \mathsf{D}$, $i \bullet s$ is defined for some $i \in \mathbf{I}$, and, for all $i \in \mathbf{I}$, whenever $i \bullet s$ is defined, $i \bullet s = s$ (that is, $\mathbf{I}$ is a left identity of $\bullet$).

Definition 2.21 (Context Logic Formulae)**.** The set of *data formulae* P^s, ranged over by $P, Q, P_1, \ldots$, and the set of *context formulae* K^s, ranged over by

$K, K_1, \ldots$ are defined inductively as follows:

$$\begin{array}{lll} P ::= S_{\mathsf{P}} & & \text{specific formulae} \\ \quad \mid K \bullet P \mid K \bullet\!\!- P & & \text{structural formulae} \\ \quad \mid \text{false} \mid P_1 \to P_2 & & \text{Boolean formulae} \\ \\ K ::= S_{\mathsf{K}} & & \text{specific formulae} \\ \quad \mid \mathbf{I} \mid P_1 -\!\!\bullet P_2 & & \text{structural formulae} \\ \quad \mid \text{False} \mid K_1 \to K_2 & & \text{Boolean formulae.} \end{array}$$

The definition is parameterised by the definitions of S_{P} and S_{K}, which define formula constructions specific to the model under consideration.

Definition 2.22 (Sorts)**.** The set of *sorts* for CL^s, Sort, ranged over by ς, is defined as follows:

$$\varsigma ::= \mathtt{d} \mid \mathtt{c}.$$

That is, there are two sorts: $\mathtt{d}$, the sort of abstract data structures, and $\mathtt{c}$, the sort of contexts. The *family of worlds* for CL^s, $\{\mathsf{World}_\varsigma\}_{\varsigma \in \mathsf{Sort}}$, is defined by $\mathsf{World}_\mathtt{d} = \mathsf{D}$, $\mathsf{World}_\mathtt{c} = \mathsf{C}$. The *family of formulae* for CL^s, $\{\mathsf{Formula}_\varsigma\}_{\varsigma \in \mathsf{Sort}}$, is defined by $\mathsf{Formula}_\mathtt{d} = \mathsf{P}^s$, $\mathsf{Formula}_\mathtt{c} = \mathsf{K}^s$.

Definition 2.23 (Satisfaction Relations)**.** The *satisfaction relations* $\models_\mathtt{d} \subseteq \mathsf{D} \times \mathsf{P}$ and $\models_\mathtt{c} \subseteq \mathsf{C} \times \mathsf{K}$, which denote satisfaction of a data formula by a data structure and satisfaction of a context formula by a tree context respectively, are defined by induction on the structure of formulae as follows:

$$\begin{array}{lcl} s \models_\mathtt{d} K \bullet P & \iff & \text{there exist } c \in \mathsf{C}, s' \in \mathsf{D} \text{ s.t. } s = c \bullet s' \text{ and} \\ & & \quad c \models_\mathtt{c} K \text{ and } s' \models_\mathtt{d} P \\ s \models_\mathtt{d} K \bullet\!\!- P & \iff & \text{for all } c \in \mathsf{C}, s' \in \mathsf{D},\ s' = c \bullet s \text{ and } c \models_\mathtt{c} K \implies \\ & & \quad s' \models_\mathtt{d} P \\ s \models_\mathtt{d} \text{false} & & \text{never} \\ s \models_\mathtt{d} P_1 \to P_2 & \iff & s \models_\mathtt{d} P_1 \implies s \models_\mathtt{d} P_2 \\ c \models_\mathtt{c} \mathbf{I} & \iff & c \in \mathbf{I} \\ c \models_\mathtt{c} P_1 -\!\!\bullet P_2 & \iff & \text{for all } s_1, s_2 \in \mathsf{D},\ s_2 = c \bullet s_1 \text{ and } s_1 \models_\mathtt{d} P_1 \implies \\ & & \quad s_2 \models_\mathtt{d} P_2 \\ c \models_\mathtt{c} \text{False} & & \text{never} \\ c \models_\mathtt{c} P_1 \to P_2 & \iff & c \models_\mathtt{c} P_1 \implies c \models_\mathtt{c} P_2. \end{array}$$

The definition of satisfaction for model-specific formulae, S_{P} and S_{K}, is defined on a per-model basis.

2.2.2 Context Logic with Composition

Definition 2.24 (Compositional Context Algebra)**.** A *compositional context algebra* $\mathcal{A} = (\mathsf{D}, \mathsf{C}, \circ, \bullet, \mathbf{I})$ consists of

- a simple context algebra $(\mathsf{D}, \mathsf{C}, \bullet, \mathbf{I})$, and
- a context composition operator, $\circ : \mathsf{C} \times \mathsf{C} \rightharpoonup \mathsf{C}$,

for which the following properties hold: for all $c, c', c'' \in \mathsf{C}$, $s \in \mathsf{D}$, and $i' \in \mathbf{I}$

- $c \circ (c' \circ c'') = (c \circ c') \circ c''$ (that is, composition is associative);
- $c \circ (c' \bullet s) = (c \circ c') \bullet s$ (that is, composition associates with application);
- $i \circ c$ is defined for some $i \in \mathbf{I}$, and whenever $i \circ c$ is defined, $i' \circ c = c$ (that is, $\mathbf{I}$ is a left identity of $\circ$); and
- $c \circ i$ is defined for some $i \in \mathbf{I}$, and whenever $c \circ i'$ is defined, $c \circ i = c$ (that is, $\mathbf{I}$ is a right identity of $\circ$).

(Undefined terms are considered equal.)

Definition 2.25 (Context Logic Formulae)**.** The set of *data formulae* P^c, ranged over by $P, Q, P_1, \ldots$, and the set of *context formulae* K^c, ranged over by $K, K_1, \ldots$ are defined inductively as follows:

$P ::= S_{\mathsf{P}}$		specific formulae
$\mid K \bullet P$	$\mid K \bullet\!\!- P$	structural formulae
$\mid \text{false}$	$\mid P_1 \to P_2$	Boolean formulae
$K ::= S_{\mathsf{K}}$		specific formulae
$\mid \mathbf{I}$	$\mid P_1 -\!\!\bullet P_2$	structural formulae
$\mid K_1 \circ K_2$	$\mid K_1 \circ\!\!- K_2 \mid K_1 -\!\!\circ K_2$	composition formulae
$\mid \text{False}$	$\mid K_1 \to K_2$	Boolean formulae.

The definition is, again, parameterised by the definitions of S_{P} and S_{K}, which define formula constructions specific to the model under consideration.

Definition 2.26 (Sorts)**.** The set of *sorts* for CL^c, Sort, ranged over by ς, is defined as follows:

$$\varsigma ::= \mathsf{d} \mid \mathsf{c}.$$

That is, there are two sorts: d, the sort of abstract data structures, and c, the sort of contexts. The *family of worlds* for CL^c, $\{\mathsf{World}_\varsigma\}_{\varsigma \in \mathsf{Sort}}$, is defined by $\mathsf{World}_\mathsf{d} = \mathsf{D}$, $\mathsf{World}_\mathsf{c} = \mathsf{C}$. The *family of formulae* for CL^c, $\{\mathsf{Formula}_\varsigma\}_{\varsigma \in \mathsf{Sort}}$, is defined by $\mathsf{Formula}_\mathsf{d} = \mathsf{P}^c$, $\mathsf{Formula}_\mathsf{c} = \mathsf{K}^c$.

Definition 2.27 (Satisfaction Relations)**.** The *satisfaction relations* $\models_{\mathsf{d}} \subseteq \mathsf{D} \times \mathsf{P}$ and $\models_{\mathsf{c}} \subseteq \mathsf{C} \times \mathsf{K}$, which denote satisfaction of a data formula by a data structure and satisfaction of a context formula by a tree context respectively, are defined by induction on the structure of formulae as follows:

$$
\begin{array}{lcl}
s \models_{\mathsf{d}} K \bullet P & \iff & \text{there exist } c \in \mathsf{C}, s' \in \mathsf{D} \text{ s.t. } s = c \bullet s' \text{ and} \\
 & & \quad c \models_{\mathsf{c}} K \text{ and } s' \models_{\mathsf{d}} P \\
s \models_{\mathsf{d}} K \mathrel{\bullet\!\!-} P & \iff & \text{for all } c \in \mathsf{C}, s' \in \mathsf{D},\ s' = c \bullet s \text{ and } c \models_{\mathsf{c}} K \implies \\
 & & \quad s' \models_{\mathsf{d}} P \\
s \models_{\mathsf{d}} \text{false} & & \text{never} \\
s \models_{\mathsf{d}} P_1 \to P_2 & \iff & s \models_{\mathsf{d}} P_1 \implies s \models_{\mathsf{d}} P_2 \\
c \models_{\mathsf{c}} \mathbf{I} & \iff & c \in \mathbf{I} \\
c \models_{\mathsf{c}} P_1 \mathrel{-\!\!\bullet} P_2 & \iff & \text{for all } s_1, s_2 \in \mathsf{D},\ s_2 = c \bullet s_1 \text{ and } s_1 \models_{\mathsf{d}} P_1 \implies \\
 & & \quad s_2 \models_{\mathsf{d}} P_2 \\
c \models_{\mathsf{c}} K_1 \circ K_2 & \iff & \text{there exist } c', c'' \in \mathsf{C} \text{ s.t. } c = c' \circ c'' \text{ and} \\
 & & \quad c' \models_{\mathsf{c}} K_1 \text{ and } c'' \models_{\mathsf{c}} K_2 \\
c \models_{\mathsf{c}} K_1 \circ\!\!- K_2 & \iff & \text{for all } c', c'' \in \mathsf{C},\ c'' = c' \circ c \text{ and } c' \models_{\mathsf{c}} K_1 \implies \\
 & & \quad c'' \models_{\mathsf{c}} K \\
c \models_{\mathsf{c}} K_1 \mathrel{-\!\!\circ} K_2 & \iff & \text{for all } c', c'' \in \mathsf{C},\ c'' = c \circ c' \text{ and } c' \models_{\mathsf{c}} K_1 \implies \\
 & & \quad c'' \models_{\mathsf{c}} K \\
c \models_{\mathsf{c}} \text{False} & & \text{never} \\
c \models_{\mathsf{c}} P_1 \to P_2 & \iff & c \models_{\mathsf{c}} P_1 \implies c \models_{\mathsf{c}} P_2.
\end{array}
$$

The definition of satisfaction for model-specific formulae, S_{P} and S_{K}, is defined on a per-model basis.

2.2.3 Multi-holed Context Logic

Definition 2.28 (Multi-holed Context Algebra)**.** A *multi-holed context algebra* $\mathcal{A} = (\mathsf{C}, \text{holes}, \{\circledast_x\}_{x \in \mathrm{X}}, \{\mathbf{I}_x\}_{x \in \mathrm{X}})$ consists of

- a non-empty set of abstract data-structure contexts, C,
- a hole operator $\text{holes} : \mathsf{C} \to \mathcal{P}_{\text{fin}}(\mathrm{X})$,
- a composition operator $\circledast_x$: $\mathsf{C} \times \mathsf{C} \rightharpoonup \mathsf{C}$ for each $x \in \mathrm{X}$, and
- a distinguished set $\mathbf{I}_x \subseteq \mathsf{C}$ for each $x \in \mathrm{X}$,

for which the following properties hold:

- for all $c, c', c'' \in \mathsf{C}$, if $c \circledast_x c' = c''$ then

  - $x \in \text{holes}\, c$,
  - $\text{holes}\, c \cap \text{holes}\, c' \subseteq \{x\}$, and
  - $\text{holes}\, c'' = (\text{holes}\, c \setminus \{x\}) \cup \text{holes}\, c'$;

- for all $c \in \mathsf{C}$,
  - there exists $i_x \in \mathbf{I}_x$ such that $i_x \circledcirc{x} c$ is defined, and
  - for all $i_x \in \mathbf{I}_x$, if $i_x \circledcirc{x} c$ is defined, then $i_x \circledcirc{x} c = c$;

- for all $c \in \mathsf{C}$ and all $x \in \text{holes}\, c$,
  - there exists $i_x \in \mathbf{I}_x$ such that $c \circledcirc{x} i_x$ is defined, and
  - for all $i_x \in \mathbf{I}_x$, if $c \circledcirc{x} i_x$ is defined then $c \circledcirc{x} i_x = c$;

- for all $c \in \mathsf{C}$, all $x \in \text{holes}\, c$ and all $y \notin \text{holes}\, c$, there exists $i_y \in \mathbf{I}_y$ such that $c \circledcirc{x} i_y$ is defined;

- for all $c, c', c'' \in \mathsf{C}$, all $x \in \text{holes}\, c$ and all $y \in \text{holes}\, c'$ with either $y \notin \text{holes}\, c$ or $y = x$, $(c \circledcirc{x} c') \circledcirc{y} c'' = c \circledcirc{x} (c' \circledcirc{y} c'')$; and

- for all c, c', c'', all $x \notin \text{holes}\, c''$ and all $y \notin \text{holes}\, c'$, $(c \circledcirc{x} c') \circledcirc{y} c'' = (c \circledcirc{y} c'') \circledcirc{x} c'$.

(Undefined terms are considered equal.)

A number of interesting basic properties follow from the definition. Firstly, for every context $c \in \mathsf{C}$ there is exactly one $i_x \in \mathbf{I}_x$ with $i_x \circledcirc{x} c$ defined. To see this, suppose that $i'_x \in \mathbf{I}_x$ with $i'_x \circledcirc{x} c$ defined. By the axioms, $i'_x \circledcirc{x} c = i'_x \circledcirc{x} (i_x \circledcirc{x} c) = (i'_x \circledcirc{x} i_x) \circledcirc{x} c)$, and so $i'_x \circledcirc{x} i_x$ is defined. Furthermore, $i'_x = i'_x \circledcirc{x} i_x = i_x$, and so the choice of i_x was unique. By a similar argument, for every context $c \in \mathsf{C}$ and hole $x \in \text{holes}\, c$, there is exactly one $i_x \in \mathbf{I}_x$ with $c \circledcirc{x} i_x$.

The set $\mathbf{I}_x$ can thus be used to classify contexts and x-holes by type, determined by which element of $\mathbf{I}_x$ is a left or right x-identity. An x-hole of type $i_x \in \mathbf{I}_x$ will only accept a context of type i, since $c \circledcirc{x} c' = (c \circledcirc{x} i_x) \circledcirc{x} c' = c \circledcirc{x} (i_x \circledcirc{x} c')$. In order for no hole label to be distinctive, the classification of contexts according to $\mathbf{I}_x$ should be the same as the classification according to $\mathbf{I}_y$ for any two $x, y \in \mathrm{X}$. This is indeed the case. The see this, suppose that, for some $c, c' \in \mathsf{C}$, $i_x \in \mathbf{I}_x$ and $i_y, i'_y \in \mathbf{I}_y$, $i_x \circledcirc{x} c$, $i_x \circledcirc{x} c'$, $i_y \circledcirc{y} c$ and $i'_y \circledcirc{y} c'$ are all defined. Then $c = i_y \circledcirc{y} (i_x \circledcirc{x} c) = (i_y \circledcirc{y} i_x) \circledcirc{x} c$ and so $i_y \circledcirc{y} i_x$ is defined. By similar reasoning, $i'_y \circledcirc{y} i_x$ is defined, and so $i_y = i'_y$. Hence, the structural properties of a context are independent of its hole labels.

Definition 2.29 (Context Hole Substitution)**.** For $c \in \mathsf{C}$ and $x, y \in \mathrm{X}$ with $y \sharp c$, let $c[y/x]$ be the unique $c' \in \mathsf{C}$ such that $c = c' \circledx i_y$ for some $i_y \in \mathbf{I}_y$ if $x \in \operatorname{holes} c$, and c otherwise.

Definition 2.30 (Context Logic Formulae)**.** The set of *formulae* K^m, ranged over by $K, K_1, \ldots$ is defined inductively as follows:

$$
\begin{aligned}
K ::= {} & S && \text{specific formulae} \\
& \mid \alpha \mid K_1 \circ_\alpha K_2 \mid K_1 \circ\!\!-_\alpha K_2 && \\
& \mid K_1 -\!\!\circ_\alpha K_2 \mid \exists \alpha.\, K \;\big(\mid \text{И}\alpha.\, K\big) && \text{structural formulae} \\
& \mid \mathrm{False} \mid K_1 \to K_2 && \text{Boolean formulae.}
\end{aligned}
$$

The definition is parameterised by the definition of S, which defines formula constructions specific to the model under consideration.

Definition 2.31 (Sorts)**.** The set of *sorts* for CL^m, Sort, ranged over by ς, is defined as follows:

$$\varsigma ::= \mathsf{c}\phi$$

where $\phi \in \mathcal{P}_{\text{fin}}(\Theta)$ ranges over finite sets of hole variables. That is, there is a sort $\mathsf{c}\phi$ for each finite set of hole variables ϕ. The *family of worlds* for CL^m, $\{\mathsf{World}_\varsigma\}_{\varsigma \in \mathsf{Sort}}$, is defined by

$$\mathsf{World}_\varsigma = \{(c, \sigma) \in \mathsf{C}^m \times \mathsf{LEnv} \mid \mathsf{c}(\operatorname{dom} \sigma) = \varsigma\}\,.$$

The *family of formulae* for CL^m, $\{\mathsf{Formula}_\varsigma\}_{\varsigma \in \mathsf{Sort}}$, is defined by

$$\mathsf{Formula}_\varsigma = \{K \in \mathsf{K}^m \mid \text{there exists } \phi \text{ s.t. } \varsigma = \mathsf{c}\phi \text{ and } \operatorname{fv} K \subseteq \phi\}\,.$$

Definition 2.32 (Satisfaction Relations)**.** The *satisfaction relations* $\models_\varsigma \subseteq \mathsf{World}_\varsigma \times \mathsf{Formula}_\varsigma$, which denote satisfaction of a formula by a multi-holed context with respect to a logical environment (for each sort ς), is defined by induction on the structure of formulae as follows, where $\varsigma = \mathsf{c}\phi$ for some $\phi \in \mathcal{P}_{\text{fin}}(\Theta)$,

and, when applicable, $x = \sigma\alpha$ and $\alpha \in \phi$:

$$
\begin{array}{lcl}
c, \sigma \models_\varsigma \alpha & \iff & c \in \mathbf{I}_x \\
c, \sigma \models_\varsigma K_1 \circ_\alpha K_2 & \iff & \text{there exist } c_1, c_2 \text{ s.t. } c = c_1 \circledast_x c_2 \text{ and} \\
 & & \quad c_1 \models_\varsigma K_1 \text{ and } c_2 \models_\varsigma K_2 \\
c, \sigma \models_\varsigma K_1 \circ\!\!- _\alpha K_2 & \iff & \text{for all } c_1, c_2,\ c_2 = c_1 \circledast_x c \text{ and } c_1, \sigma \models_\varsigma K_1 \\
 & & \quad \implies c_2, \sigma \models_\varsigma K_2 \\
c, \sigma \models_\varsigma K_1 -\!\!\circ_\alpha K_2 & \iff & \text{for all } c_1, c_2,\ c_2 = c \circledast_x c_1 \text{ and } c_1, \sigma \models_\varsigma K_1 \\
 & & \quad \implies c_2, \sigma \models_\varsigma K_2 \\
c, \sigma \models_\varsigma \exists\beta.\, K & \iff & \text{there exists } y \text{ s.t. } c, \sigma[\beta \mapsto y] \models_{\mathsf{c}(\phi \cup \{\beta\})} K \\
\left(\begin{array}{l} c, \sigma \models_\varsigma \mathit{И}\beta.\, K \\ \\ \end{array} \right. & \iff & \left. \begin{array}{l} \text{there exists } y \text{ s.t. } y \sharp c, \sigma \text{ and} \\ \quad c, \sigma[\beta \mapsto y] \models_{\mathsf{c}(\phi \cup \{\beta\})} K \end{array} \right) \\
c, \sigma \models_\varsigma \text{False} & & \text{never} \\
c, \sigma \models_\varsigma K_1 \to K_2 & \iff & c, \sigma \models_\varsigma K_1 \implies c, \sigma \models_\varsigma K_2.
\end{array}
$$

The definition of satisfaction for model-specific formulae, S, is defined on a per-model basis.

Elementary Properties

The following two properties hold for all of the context logic models presented in this thesis. It is possible to define model-specific connectives in a pathological fashion that break these properties, however. In § 2.4.1 I define sufficient conditions on the interpretation of these connectives for the properties to hold.

Property 2.33 (Environment Extendability)**.** *For all* $\varsigma \in \mathsf{Sort}$*,* $(c, \sigma) \in \mathsf{World}_\varsigma$*,* $K \in \mathsf{Formula}_\varsigma$*,* $\alpha \in \Theta$ *and* $x \in \mathrm{X}$ *with* $\varsigma = \mathsf{c}\phi$ *for some* $\phi \in \mathcal{P}_{\mathrm{fin}}(\Theta)$*, and* $\alpha \notin \phi$*,*

$$c, \sigma \models_\varsigma K \quad \iff \quad c, \sigma[\alpha \mapsto x] \models_{\mathsf{c}(\phi \cup \{\alpha\})} K.$$

Property 2.34 (Hole Subsitutition)**.** *For all* $\varsigma \in \mathsf{Sort}$*,* $(c, \sigma) \in \mathsf{World}_\varsigma$*,* $K \in \mathsf{Formula}_\varsigma$*,* $x, y \in \mathrm{X}$ *with* $y \sharp c, \sigma$*,*

$$c, \sigma \models_\varsigma K \quad \iff \quad c[y/x], \sigma[y/x] \models_\varsigma K.$$

The following lemma is an important corollary of the hole substitution property.

Lemma 4 (Universal Characterisation of Fresh Quantification)**.** *For all* $\varsigma \in \mathsf{Sort}$*,* $(c, \sigma) \in \mathsf{World}_\varsigma$*,* $K \in \mathsf{Formula}_\varsigma$*,* $\beta \in \Theta$*,*

$$c, \sigma \models_\varsigma \mathit{И}\beta.\, K \quad \iff \quad \textit{for all } y,\ y \sharp c, \sigma \implies c, \sigma[\beta \mapsto y] \models_{\mathsf{c}(\phi \cup \{\beta\})} K$$

where $\varsigma = \mathsf{c}\phi$*.*

Proof. $\Longrightarrow$:
There is some $x \sharp c, \sigma$ with $c, \sigma[\beta \mapsto x] \models_{\mathsf{c}(\phi \cup \{\beta\})} K$. Fix some arbitrary $y \sharp c, \sigma$. By the freshness of x, $c = c[y/x]$ and $(\sigma[\beta \mapsto x])[y/x] = \sigma[\beta \mapsto y]$. Hence, since y is fresh, by the hole substitution property, $c, \sigma[\beta \mapsto x] \models_{\mathsf{c}(\phi \cup \{\beta\})} K$, as required.

$\Longleftarrow$:
This direction is trivial, since X is infinite and both holes c and range σ are finite. □

An important consequence of Lemma 4 is that $\mathsf{И}$ is self dual. That is, $\mathsf{И}\alpha.\, K \equiv \neg \mathsf{И}\alpha.\, \neg K$.

2.3 Context Logic Models

I present a number of examples of context algebras. Each is presented as a context algebra with composition, although the adaptation to a multi-holed context algebra is straightforward.

2.3.1 Sequences

Sequences are ordered, finite lists of labels. They can be viewed as a flat trees — trees in which all nodes are at the root level — and so in many ways are a special case of the tree model.

Definition 2.35 (Sequence Context Algebra)**.** The set of *sequences* Seq, ranged over by $s, s_1, s', \ldots$, and the set of *sequence contexts* $\mathsf{C}_{\mathsf{Seq}}$, ranged over by $c, c_1, c', \ldots$, are defined inductively as follows:

$$s ::= \varnothing \mid \mathbf{a} \mid s_1 \cdot s_2$$
$$c ::= \_ \mid s \cdot c \mid c \cdot s$$

where $\cdot$ is considered to be associative and to have $\varnothing$ as its identity. Context application, $\bullet : \mathsf{C}_{\mathsf{Seq}} \times \mathsf{Seq} \rightharpoonup \mathsf{Seq}$, and composition, $\circ : \mathsf{C}_{\mathsf{Seq}} \times \mathsf{C}_{\mathsf{Seq}} \rightharpoonup \mathsf{C}_{\mathsf{Seq}}$ are standard:

$$c \bullet s \stackrel{\text{def}}{=} c[s/\_] \qquad\qquad c_1 \circ c_2 \stackrel{\text{def}}{=} c_1[c_2/\_].$$

The *sequence context algebra* is $(\mathsf{Seq}, \mathsf{C}_{\mathsf{Seq}}, \circ, \bullet, \{\_\})$.

Definition 2.36 (Sequence-specific Formulae)**.** The *sequence-specific formulae* for context logic with composition are defined as follows:

$$S_{\mathsf{P}} ::= \mathbf{0} \mid \mathbf{a}$$
$$S_{\mathsf{K}} ::= P \cdot K \mid K \cdot P.$$

The satisfaction relations for these formulae are defined as follows:

$$
\begin{array}{lcl}
s \models_{\mathsf{d}} \mathbf{0} & \iff & s = \varnothing \\
s \models_{\mathsf{d}} \mathbf{a} & \iff & s = \mathbf{a} \\
c \models_{\mathsf{c}} P \cdot K & \iff & \text{there exist } s, c' \text{ s.t. } c = s \cdot c' \text{ and } s \models_{\mathsf{d}} P \text{ and } c' \models K \\
c \models_{\mathsf{c}} K \cdot P & \iff & \text{there exist } c', s \text{ s.t. } c = c' \cdot s \text{ and } c' \models K \text{ and } s \models_{\mathsf{d}} P.
\end{array}
$$

2.3.2 Heaps

The heap model of separation logic views heaps as finite partial functions from addresses to values. Heap separation is then the union of heaps that have disjoint domains. Here, I define heaps syntactically.

In the following, Addr, the set of *heap addresses*, ranged over by $a, a_1, a', \ldots$, is typically taken to be the positive integers, *i.e.* $\mathsf{Addr} = \mathbb{Z}^+$. The set of *values*, Val, ranged over by $v, w, v_1, v', \ldots$, is essentially arbitrary, but taken to include the set of heap addresses, *i.e.* $\mathsf{Addr} \subseteq \mathsf{Val}$.

Definition 2.37 (Heap Context Algebra)**.** The set of *heaps* Heap, ranged over by $h, h_1, h', \ldots$, is defined inductively as follows:

$$h ::= \text{emp} \;\mid\; a \mapsto v \;\mid\; h_1 * h_2$$

where $*$ is considered to be associative and commutative with identity emp, and heap addresses occur uniquely (as addresses) within any given heap. The heap context algebra is $(\mathsf{Heap}, \mathsf{Heap}, *, *, \{\text{emp}\})$.

This definition identifies heaps and heap contexts. It is quite possible to define heap contexts with explicit holes in the syntactic fashion I used to define tree and sequence contexts, however, the nature of $*$ (a commutative, partial monoid operator) would give a context algebra that is isomorphic to the one above.

In [COY07], Calcagno, O'Hearn and Yang consider abstract models for separation logic, called *separation algebras*, of which the heap model is an instance. Separation algebras are defined to be cancellative, partial commutative monoids, $(\mathsf{D}, \circ, u)$. Any such separation algebra gives rise to a context algebra $(\mathsf{D}, \mathsf{D}, \circ, \circ, \{u\})$.

Definition 2.38 (Heap-specific Formulae)**.** The *heap-specific formulae* for context logic with composition are defined as follows:

$$S_{\mathsf{P}} ::= \mathbf{0}$$

$$S_{\mathsf{K}} ::= a \mapsto v.$$

The satisfaction relations for these formulae are defined as follows:

$$h \models_{\mathsf{d}} \mathbf{0} \qquad \Longleftrightarrow \qquad h = \mathrm{emp}$$
$$h \models_{\mathsf{c}} a \mapsto v \qquad \Longleftrightarrow \qquad h = a \mapsto v.$$

2.3.3 Terms

Terms are finite, ranked trees. That is, each node of the term has a fixed number of children. Let Υ be a *ranked alphabet* from which the nodes of terms are to be labelled; that is $\Upsilon \subseteq \Sigma \times \mathbb{N}$ for some alphabet Σ. (Letters from Σ may be given more than one rank in Υ. Labels are considered to be different if they have different ranks.)

Definition 2.39 (Term Context Algebra)**.** The set of *terms* Term, ranged over by $r, r_1, r', \ldots$, and the set of *term contexts* $\mathsf{C}_{\mathsf{Term}}$, ranged over by $c, c_1, c', \ldots$, are defined inductively as follows:

$$t ::= \mathbf{a}(t_1, \ldots, t_n)$$
$$c ::= - \;\mid\; \mathbf{a}(t_1, \ldots, t_{i-1}, c_i, t_{i+1}, \ldots, t_n)$$

where $(\mathbf{a}, n) \in \Upsilon$ ranges over ranked terms and $1 \leq i \leq n$. Context application, $\bullet : \mathsf{C}_{\mathsf{Term}} \times \mathsf{Term} \rightharpoonup \mathsf{Term}$, and composition, $\circ : \mathsf{C}_{\mathsf{Term}} \times \mathsf{C}_{\mathsf{Term}} \rightharpoonup \mathsf{C}_{\mathsf{Term}}$ are standard:

$$c \bullet t \stackrel{\text{def}}{=} c[t/-] \qquad\qquad c_1 \circ c_2 \stackrel{\text{def}}{=} c_1[c_2/-].$$

The *term context algebra* is $(\mathsf{Term}, \mathsf{C}_{\mathsf{Term}}, \circ, \bullet, \{-\})$.

Definition 2.40 (Term-specific Formulae)**.** The *term-specific formula* for context logic with composition are defined as follows:

$$S_{\mathsf{P}} ::= \mathbf{a}(P_1, \ldots, P_n)$$
$$S_{\mathsf{K}} ::= \mathbf{a}(P_1, \ldots, P_{i-1}, K_i, K_{i+1}, \ldots, P_n)$$

where $(\mathbf{a}, n) \in \Upsilon$ ranges over ranked terms and $1 \leq i \leq n$. The satisfaction

relations for these formulae are defined as follows:

$$\begin{aligned} t \models_{\mathsf{d}} \mathbf{a}(P_1, \ldots, P_n) \quad \Longleftrightarrow \quad & \text{there exist } t_1, \ldots, t_n \in \mathsf{Term} \text{ s.t.} \\ & t = \mathbf{a}(t_1, \ldots, t_n) \text{ and} \\ & \text{for all } 1 \leq i \leq n,\ t_i \models_{\mathsf{d}} P_i \end{aligned}$$

$$\begin{aligned} c \models_{\mathsf{c}} \mathbf{a}(P_1, \ldots, P_{i-1}, K_i, P_{i+1}, \ldots, P_n) \\ \Longleftrightarrow \quad & \text{there exist } t_1, \ldots, t_{i-1}, \\ & t_{i+1}, \ldots, t_n \in \mathsf{Term}, c_i \in \mathsf{C}_{\mathsf{Term}} \text{ s.t.} \\ & c = \mathbf{a}(t_1, \ldots, t_{i-1}, c_i, t_{i+1}, \ldots, t_n) \text{ and} \\ & \text{for all } i \in \{1, \ldots, i-1, i+1, \ldots, n\},\ t_i \models_{\mathsf{d}} P_i \\ & \text{and } c_i \models_{\mathsf{c}} K_i. \end{aligned}$$

2.4 Context Logic as Modal Logic

Calcagno, Gardner and Zarfaty previously viewed context logic as a model logic in order to prove expressivity results [CGZ07]. In this presentation, the semantics of each non-Boolean connective of the logic, $\circledast$, is given by a Kripke semantics in which it is interpreted by a relation over worlds, $\mathfrak{M}_{\circledast}$. Taking $-\!\!\bullet^{\exists}$ and $\bullet\!\!-^{\exists}$ as primitive and viewing $-\!\!\bullet$ and $\bullet\!\!-$ as derived connectives in terms of these leads to the simplest modal presentation, in which each connective is a $\Diamond$-like modality. Specifically, for each n-ary connective $\circledast$, and formulae $F_1, \ldots, F_n$, and for each $w \in \mathsf{World}_{\varsigma}$

$$\begin{aligned} w \models_{\varsigma} \circledast(F_1, \ldots, F_n) \quad \Longleftrightarrow \quad & \text{there exist } w_1 \in \mathsf{World}_{\varsigma_1}, \ldots, w_n \in \mathsf{World}_{\varsigma_n} \text{ s.t.} \\ & (w_1, \ldots, w_n)\ \mathfrak{M}_{\circledast}\ w \text{ and} \\ & \text{for all } 1 \leq i \leq n,\ w_i \models_{\varsigma_i} F_i. \end{aligned}$$

The modal relations must be well behaved with respect to the sorts in the logic. In particular, for each n-ary connective $\circledast$ there must be some partial function $f_{\circledast} : \mathsf{Sort} \rightharpoonup \mathsf{Sort}^n$ such that for all sorts $\varsigma, \varsigma_1, \ldots, \varsigma_n$, and all $w \in \mathsf{World}_{\varsigma}$, $w_1 \in \mathsf{World}_{\varsigma_1}, \ldots, w_n \in \mathsf{World}_{\varsigma_n}$, if

$$(w_1, \ldots, w_n)\ \mathfrak{M}_{\circledast}\ w$$

then

$$f_{\circledast}(\varsigma) = (\varsigma_1, \ldots, \varsigma_n).$$

The sorts of formulae can be derived by the rule

$$\frac{f_{\circledast}(\varsigma) = (\varsigma_1, \ldots, \varsigma_n) \quad F_1 \in \mathsf{Formula}_{\varsigma_1} \quad \cdots \quad F_n \in \mathsf{Formula}_{\varsigma_n}}{\circledast(F_1, \ldots, F_n) \in \mathsf{Formula}_{\varsigma}}.$$

I give the modal relations for context logic for trees. For other models, similar presentations are possible, but I will not make use of them. Since the connectives of CL^s_{Tree} are all connectives of CL^c_{Tree}, I give the modalities for the latter; the modalities for the former are the same, with the exclusion of those for composition and its adjoints.

Definition 2.41 (Modalities for CL^c_{Tree})**.**

$$\begin{aligned}
\mathfrak{M}_{\mathbf{0}} &\stackrel{\text{def}}{=} \{t \in \mathsf{Tree} \mid t = \varnothing\} \\
\mathfrak{M}_{\mathbf{a}[]} &\stackrel{\text{def}}{=} \{(t_1, t) \in \mathsf{Tree} \times \mathsf{Tree} \mid t = \mathbf{a}[t_1]\} \\
\mathfrak{M}_{\otimes^{\text{left}}} &\stackrel{\text{def}}{=} \{((c_1, t_2), c) \in (\mathsf{C}_{\mathsf{Tree}} \times \mathsf{Tree}) \times \mathsf{C}_{\mathsf{Tree}} \mid c_1 \otimes t_2 = c\} \\
\mathfrak{M}_{\otimes^{\text{right}}} &\stackrel{\text{def}}{=} \{((t_1, c_1), c) \in (\mathsf{Tree} \times \mathsf{C}_{\mathsf{Tree}}) \times \mathsf{C} \mid t_1 \otimes c_2 = c)\} \\
\mathfrak{M}_{\mathbf{I}} &\stackrel{\text{def}}{=} \{c \in \mathsf{C}_{\mathsf{Tree}} \mid c = -\} \\
\mathfrak{M}_{\bullet} &\stackrel{\text{def}}{=} \{((c_1, t_2), t) \in (\mathsf{C}_{\mathsf{Tree}} \times \mathsf{Tree}) \times \mathsf{Tree} \mid c_1 \bullet t_2 = t\} \\
\mathfrak{M}_{\bullet\!-^{\exists}} &\stackrel{\text{def}}{=} \{((c_1, t_2), t) \in (\mathsf{C}_{\mathsf{Tree}} \times \mathsf{Tree}) \times \mathsf{Tree} \mid c_1 \bullet t = t_2\} \\
\mathfrak{M}_{-\!\bullet^{\exists}} &\stackrel{\text{def}}{=} \{((t_1, t_2), c) \in (\mathsf{Tree} \times \mathsf{Tree}) \times \mathsf{C}_{\mathsf{Tree}} \mid c \bullet t_1 = t_2\} \\
\mathfrak{M}_{\circ} &\stackrel{\text{def}}{=} \{((c_1, c_2), c) \in (\mathsf{C}_{\mathsf{Tree}} \times \mathsf{C}_{\mathsf{Tree}}) \times \mathsf{C}_{\mathsf{Tree}} \mid c_1 \circ c_2 = c\} \\
\mathfrak{M}_{\circ\!-^{\exists}} &\stackrel{\text{def}}{=} \{((c_1, c_2), c) \in (\mathsf{C}_{\mathsf{Tree}} \times \mathsf{C}_{\mathsf{Tree}}) \times \mathsf{C}_{\mathsf{Tree}} \mid c_1 \circ c = c_2\} \\
\mathfrak{M}_{-\!\circ^{\exists}} &\stackrel{\text{def}}{=} \{((c_1, c_2), c) \in (\mathsf{C}_{\mathsf{Tree}} \times \mathsf{C}_{\mathsf{Tree}}) \times \mathsf{C}_{\mathsf{Tree}} \mid c \circ c_1 = c_2\}\,.
\end{aligned}$$

Definition 2.42 (Modalities for CL^m_{Tree})**.**

$$\begin{aligned}
\mathfrak{M}_{\mathbf{0}} &\stackrel{\text{def}}{=} \{(c, \sigma) \in \mathsf{World}_{\varsigma} \mid c = \varnothing\} \\
\mathfrak{M}_{\mathbf{a}[]} &\stackrel{\text{def}}{=} \{((c_1, \sigma), (c, \sigma)) \in \mathsf{World}_{\varsigma} \times \mathsf{World}_{\varsigma} \mid c = \mathbf{a}[c_1]\} \\
\mathfrak{M}_{\otimes} &\stackrel{\text{def}}{=} \left\{(((c_1, \sigma), (c_2, \sigma)), (c, \sigma)) \in \mathsf{World}^2_{\varsigma} \times \mathsf{World}_{\varsigma} \;\middle|\; c = c_1 \otimes c_2\right\} \\
\mathfrak{M}_{\alpha} &\stackrel{\text{def}}{=} \{(c, \sigma) \in \mathsf{World}_{\mathsf{c}\phi} \mid \alpha \in \phi \text{ and } c = \sigma\alpha\} \\
\mathfrak{M}_{\circ_\alpha} &\stackrel{\text{def}}{=} \left\{ \begin{array}{l} (((c_1, \sigma), (c_2, \sigma)), (c, \sigma)) \\ \quad \in \mathsf{World}^2_{\mathsf{c}\phi} \times \mathsf{World}_{\mathsf{c}\phi} \end{array} \;\middle|\; \alpha \in \phi \text{ and } c_1 \mathbin{\textcircled{x}} c_2 = c \right\} \\
\mathfrak{M}_{\circ\!-^{\exists}_{\alpha}} &\stackrel{\text{def}}{=} \left\{ \begin{array}{l} (((c_1, \sigma), (c_2, \sigma)), (c, \sigma)) \\ \quad \in \mathsf{World}^2_{\mathsf{c}\phi} \times \mathsf{World}_{\mathsf{c}\phi} \end{array} \;\middle|\; \alpha \in \phi \text{ and } c_1 \mathbin{\textcircled{x}} c = c_2 \right\} \\
\mathfrak{M}_{-\!\circ^{\exists}_{\alpha}} &\stackrel{\text{def}}{=} \left\{ \begin{array}{l} (((c_1, \sigma), (c_2, \sigma)), (c, \sigma)) \\ \quad \in \mathsf{World}^2_{\mathsf{c}\phi} \times \mathsf{World}_{\mathsf{c}\phi} \end{array} \;\middle|\; \alpha \in \phi \text{ and } c \mathbin{\textcircled{x}} c_1 = c_2 \right\} \\
\mathfrak{M}_{\exists\alpha} &\stackrel{\text{def}}{=} \left\{ \begin{array}{l} ((c, \sigma[\alpha \mapsto x]), (c, \sigma)) \\ \in \mathsf{World}_{\mathsf{c}(\phi \cup \{\alpha\})} \times \mathsf{World}_{\mathsf{c}\phi} \end{array} \;\middle|\; x \in \mathrm{X} \right\} \\
\mathfrak{M}_{\text{Ƶ}\alpha} &\stackrel{\text{def}}{=} \left\{ \begin{array}{l} ((c, \sigma[\alpha \mapsto x]), (c, \sigma)) \\ \in \mathsf{World}_{\mathsf{c}(\phi \cup \{\alpha\})} \times \mathsf{World}_{\mathsf{c}\phi} \end{array} \;\middle|\; x \in \mathrm{X} \setminus (\operatorname{holes} c \cup \operatorname{range} \sigma) \right\}.
\end{aligned}$$

2.4.1 Elementary Properties

The following basic properties hold for all of the model-specific CL^m-modalities used in this thesis. They are simple, but sufficient for the environment extensibility and hole substitution properties to hold.

Property 2.43 (Environment Neutrality)**.** *For connective* $\circledast$ *of arity* n*,* $\mathfrak{M}_\circledast$ *satisfies the* environment neutrality property *if and only if there exists some* $\overline{\mathfrak{M}_\circledast} \subseteq \mathsf{C}^n \times \mathsf{C}$ *such that*

$$\mathfrak{M}_\circledast = \left\{ (((c_1, \sigma), \ldots, (c_n, \sigma)), (c, \sigma)) \;\middle|\; (c_1, \ldots, c_n)\ \overline{\mathfrak{M}_\circledast}\ c \text{ and } \sigma \in \mathsf{LEnv} \right\}.$$

Environment extensibility (Property 2.33) is a consequence of environment neutrality holding for each model-specific connective. The proof is by induction on the structure of formulae, the model-specific cases following by environment neutrality and the remaining cases holding trivially.

Property 2.44 (Hole Neutrality)**.** *For connective* $\circledast$ *of arity* n*,* $\mathfrak{M}_\circledast$ *satisfies the* hole neutrality property *if and only if, for all worlds* $w, w_1, \ldots, w_n \in \mathsf{C} \times \mathsf{LEnv}$ *with*

$$(w_1, \ldots, w_n)\ \mathfrak{M}_\circledast\ w,$$

for all $y \in \mathrm{X}$ *with* $y \sharp w, w_1, \ldots, w_n$*,*

$$(w_1[y/x], \ldots, w_n[y/x])\ \mathfrak{M}_\circledast\ w[y/x].$$

Hole substitution (Property 2.34) is a consequence of hole neutrality holding for each model-specific connective. The proof is by induction on the structure of formulae, the model-specific cases following directly from hole neutrality and the remaining cases holding by structural considerations concerning substitution.

Chapter 3

Expressivity

This chapter deals with expressivity questions about context logic for trees, in particular, the question of whether or not the adjunct connectives add expressive power to the logic. The main tool I use to study expressivity is Ehrenfeucht-Fraïssé Games, which are introduced in §3.1. In §3.2, I discuss to what extent adjunct elimination holds for CL^{s}_{Tree}, and prove that the $\multimap$ adjoint cannot be eliminated. In §3.3, I explain the difficulties in showing adjunct elimination for CL^{c}_{Tree}. Finally, in §3.4, I show that adjunct elimination does hold for CL^{m}_{Tree}.

Collaboration and Contribution

The work in this chapter was conducted in collaboration with Calcagno and Garder, much of which was previously published in [CDYG07] and [CDYG10]. Most of the technical work was my contribution, under the supervision of my collaborators.

3.1 Ehrenfeucht-Fraïssé Games

In this section, I give a general presentation of Ehrenfeucht-Fraïssé-style games that can be instantiated for each logic that I introduced in the previous chapter. For the purposes of this chapter, I will assume the modal presentation of each logic, in which we consider the connectives with existential semantics to be primitive. This significantly simplifies considerations for the soundness and completeness of games.

Central to the definition of games is the concept of a *rank*, which establishes a relationship between games and logical formulae — the same concept of rank abstracts the structure of both. Ranks are structured according to a *ranking*,

which captures enough of the structure of formulae for games to be sound and complete.

Definition 3.1 (Ranking)**.** A *ranking* for a logic is a bounded join semi-lattice $(\mathsf{Rank}, \sqsubseteq)$, together with a relation $\mathfrak{R}_{\circledast} \subseteq \mathsf{Rank}^n \times \mathsf{Rank}$ for each non-Boolean logical connective $\circledast$, where n is the arity of $\circledast$, having the properties that:

- for every non-Boolean connective $\circledast$ and ranks $r_1, \ldots, r_n$, where n is the arity of $\circledast$, there exists a rank r with $(r_1, \ldots, r_n)\ \mathfrak{R}_{\circledast}\ r$ — the ranking is *constructive*;
- for every non-Boolean connective $\circledast$ and ranks $r, r', r_1, \ldots, r_n$, where n is the arity of $\circledast$, if $(r_1, \ldots, r_n)\ \mathfrak{R}_{\circledast}\ r$ and $r \sqsubseteq r'$ then $(r_1, \ldots, r_n)\ \mathfrak{R}_{\circledast}\ r'$ — the ranking is *upwardly closed*;
- for every rank $r \in \mathsf{Rank}$, the set $\{(\circledast, \overrightarrow{r}) \mid \overrightarrow{r}\ \mathfrak{R}_{\circledast}\ r\}$ is finite — the ranking is *finitely branching*; and
- the relation $\lhd \subseteq \mathsf{Rank} \times \mathsf{Rank}$, defined to be the least relation such that, for every non-Boolean connective $\circledast$ and ranks $r, r_1, \ldots, r_n$, where n is the arity of $\circledast$, if $(r_1, \ldots, r_n)\ \mathfrak{R}_{\circledast}\ r$ then $r_i \lhd r$ for all $1 \leq i \leq n$, is a well-founded relation — the ranking is *inductive*.

3.1.1 Ranks of Formulae

The ranking relations, $\mathfrak{R}_{\circledast}$, are used to establish the ranks of formulae. Every formula has some rank (Lemma 6), yet each rank consists of only finitely many formulae, when considered up to logical equivalence (Lemma 7). A world can be characterised by the formulae of a given rank that it satisfies (Definition 3.3). This essentially establishes the concept of rank-r equivalence on worlds: two worlds are rank-r equivalent exactly when no formula of rank r discriminates between them.

Definition 3.2 (Rank of a Formula)**.** Formula F is said to have *rank* $r \in \mathsf{Rank}$ if:

- it is of the form $\circledast(F_1, \ldots, F_n)$ for some non-Boolean operator $\circledast$ and formulae $F_1, \ldots, F_n$ having ranks $r_1, \ldots, r_n$, with $(r_1, \ldots, r_n)\ \mathfrak{R}_{\circledast}\ r$; or
- it is a Boolean combination of such formulae.

The following lemma is trivial, but ensures that it is always possible to assign ranks to Boolean combinations of formulae.

Lemma 5 (Upward Closure for Formulae)**.** *If F has rank $r \in \mathsf{Rank}$ then, for any $r' \in \mathsf{Rank}$ with $r \sqsubseteq r'$, F has rank r'.*

Proof. The proof is by induction on the structure of F. There are two cases to consider.

Suppose that F is of the form $\circledast(F_1, \ldots, F_n)$ for some non-Boolean connective $\circledast$ and formulae $F_1, \ldots, F_n$ having ranks $r_1, \ldots, r_n$, with $(r_1, \ldots, r_n)\ \mathfrak{R}_{\circledast}\ r$. By the definition of ranking, $(r_1, \ldots, r_n)\ \mathfrak{R}_{\circledast}\ r'$, and so F has rank r'.

The remaining case is that F is a Boolean combination of formulae of rank r. By induction, these formulae also have rank r' and so F has rank r'. □

Lemma 6 (Rank Definedness)**.** *Every formula has at least one rank.*

Proof. Let F be an arbitrary formula. The proof is by induction on the structure of F. There are two cases to consider.

Suppose that F is of the form $\circledast(F_1, \ldots, F_n)$ for some non-Boolean connective $\circledast$ and formulae $F_1, \ldots, F_n$. By the inductive hypothesis, each formula F_i has some rank r_i. By the definition of ranking, there is some rank r such that $(r_1, \ldots, r_n)\ \mathfrak{R}_{\circledast}\ r$, and hence F has rank r.

The remaining case is that F is a Boolean combination of zero or more formulae, $F_1, \ldots, F_n$. By the inductive hypothesis, each F_i has some rank r_i. Since a ranking is a bounded join semi-lattice, there exists a least upperbound r of these ranks. For each i, since $r_i \sqsubseteq r$ it follows by Lemma 5 that F_i has rank r. Thus F also has rank r. □

Assumption 3.1. Logical equivalence is a congruence with respect to the connectives of the logic.

This assumption holds for all of our logics, as a consequence of the way their semantics is defined. (It would most likely be absurd for a logic not to satisfy this assumption.)

Lemma 7. *There are are only finitely many equivalence classes of formulae of each rank $r \in \mathsf{Rank}$.*

Proof. The proof is by well-founded induction on the rank r, using the relation $\lhd$. Formulae of rank r are boolean combinations of formulae of the form $\circledast(F_1, \ldots, F_n)$ for non-Boolean connectives $\circledast$ and formulae $F_1, \ldots, F_n$ having ranks $r_1, \ldots, r_n$ with $(r_1, \ldots, r_n)\ \mathfrak{R}_{\circledast}\ r$. By the definition of ranking, there are finitely many choices for $\circledast, r_1, \ldots, r_n$. Further, for each i, $r_i \lhd r$, and so by the inductive hypothesis there are only finitely many equivalence classes of formulae of rank r_i. Since logical equivalence is a congruence, there are only finitely

many equivalence classes of formulae of the form $\circledast(F_1, \ldots, F_n)$ where each F_i has rank r_i. There are only finitely many Boolean combinations that may be formed from finitely many equivalence classes of formulae. Therefore, there are only finitely many equivalence classes of formulae of rank r. □

Notation. For each sort ς and rank r, let $\mathcal{F}_{\varsigma,r}$ denote a set of sort-ς formulae of rank r, one from each equivalence class.

Definition 3.3 (Characteristic)**.** The *rank-r characteristic* of a world $w \in \mathsf{World}_\varsigma$ is the formula D^r_w, defined as follows

$$D^r_w = \bigwedge \{F \in \mathcal{F}_{\varsigma,r} \mid w \models F\}\,.$$

Lemma 8. *If $w' \models D^r_w$ then there is no formula of rank r that discriminates between w and w'.*

Proof. Suppose that there were some formula F of rank r that discriminates between w and w'. Without loss of generality, we can assume that $w \models F$ and $w' \not\models F$ (otherwise, consider the formula $\neg F$). There is a formula $F' \in \mathcal{F}_{\varsigma,r}$ that is equivalent to F, and hence $w \models F'$ and $w' \not\models F'$. Since $w' \models D^r_w$, which is, by definition, a conjunction of formulae including F', it must be that $w' \models F'$. This is a contradiction, therefore no discriminating formula exists. □

The following corollary, which characterises certain sets with formulae, is used to show the existence of equivalent formulae in my adjunct elimination results.

Corollary 9. *Let $W \subseteq \mathsf{World}_\varsigma$ be a set of worlds of sort ς such that, for any $w, w' \in \mathsf{World}_\varsigma$ with $w \in W$ and $w' \notin W$, there exists a formula F of rank r such that $w \models_\varsigma F$ and $w' \not\models_\varsigma F$. There exists a formula F_W of rank r such that, for every $w \in \mathsf{World}_\varsigma$, $w \models F_W$ if and only if $w \in W$.*

Proof. Let

$$F_W = \bigvee \{D^r_w \mid w \in W\}\,.$$

Since each characteristic formula of rank r is a conjunction of formulae from the finite set $\mathcal{F}_{\varsigma,r}$, there are only finitely many of them. Hence F_W is a finite disjunction (and so a proper, finite formula). Suppose that $w \in W$. Naturally, $w \models_\varsigma D^r_w$ and so $w \models F_W$. Suppose that $w \models F_W$. Since F_W is a disjunction, it must be that, for some $w' \in W$, $w \models D^r_{w'}$. By Lemma 8, there is no formula of rank r that discriminates between w and w'. Therefore $w \in W$. □

3.1.2 Games

A game state is a triple (w, w', r) where w and w' are worlds of the same sort and r is a rank. The game is played as a succession of rounds in which each of two players, called Spoiler and Duplicator, take turns to make choices that fulfil certain requirements. Eventually, one player is unable to make a choice that fulfils the requirements, thereby losing the game. When Spoiler wins the game, it is by finding a distinction between the two worlds in the state; when Duplicator wins, it is because Spoiler was unable to find such a distinction. Thus, while Spoiler's strategy is to find a property one world has but the other does not have, Duplicator's is to match Spoiler's property on the other world.

Definition 3.4 (Game)**.** From the game state (w, w', r), where $w, w' \in \mathsf{World}_\varsigma$ for some sort ς and $r \in \mathsf{Rank}$, the game proceeds as follows:

- Spoiler selects $(w^{\mathsf{S}}, w^{\mathsf{D}}) \in \{(w, w'), (w', w)\}$.
- Spoiler selects a logical connective $\circledast$ and ranks $r_1, \ldots, r_n \in \mathsf{Rank}$ (where n is the arity of $\circledast$) such that $(r_1, \ldots, r_n)\ \mathfrak{R}_\circledast\ r$. (If Spoiler cannot make such a choice, Duplicator wins.)
- Spoiler selects $w_1, \ldots, w_n$ such that $(w_1, \ldots, w_n)\ \mathfrak{M}_\circledast\ w^{\mathsf{S}}$. (Again, if Spoiler cannot make such a choice, Duplicator wins.)
- Duplicator selects $w'_1, \ldots, w'_n$ such that $(w'_1, \ldots, w'_n)\ \mathfrak{M}_\circledast\ w^{\mathsf{D}}$. (If Duplicator cannot make such a choice, Spoiler wins.)
- Spoiler selects $1 \leq i \leq n$ and the game proceeds from the state (w_i, w'_i, r_i). (If Spoiler cannot make such a choice, *i.e.*if $n = 0$, Duplicator wins.)

Every game must eventually end in victory for one of the players. If this were not the case, it would lead to an infinite sequence of game states (w_1, w'_1, r_1), $(w_2, w'_2, r_2), \ldots$ with the property that $r_{i+1} \lhd r_i$, which is impossible since $\lhd$ is well-founded.

Furthermore, from any initial state (w, w', r) there is a strategy for one of the players that enables that player to always win. To see this, suppose that Spoiler does not have a winning strategy. This means that however Spoiler chooses to play each round, Duplicator can respond so that she can still win the game. These responses are a winning strategy for Duplicator. Similarly, if Duplicator has no winning strategy then Spoiler must have one.

Let DW be the set of game states for which Duplicator has a winning strategy and let SW be the set of game states for which Spoiler has a winning strategy.

Clearly, $(w, w', r) \in \mathsf{SW}$ if and only if $(w', w, r) \in \mathsf{SW}$, since spoiler has the choice of which world to play with.

A useful property for games to have is *downward closure*:

Property 3.5 (Downward Closure)**.** *For worlds $w, w' \in \mathsf{World}_\varsigma$ (for some sort ς) and ranks $r, r' \in \mathsf{Rank}$ with $r \lhd^* r'$, if*

$$(w, w', r') \in \mathsf{DW}$$

then

$$(w, w', r) \in \mathsf{DW}$$

where $\lhd^*$ is the reflexive-transitive closure of $\lhd$.

This property does not hold in general, but the following is a sufficient condition on the ranking for downward closure to hold:

Definition 3.6 (Downwardly-Closed Ranking)**.** For a given ranking, let $\leq$ be the simulation preorder, that is, the most general relation such that, if $r \leq r'$ then for every non-Boolean connective $\circledast$ and ranks $r_1, \ldots, r_n$ with $(r_1, \ldots, r_n)\ \mathfrak{R}_\circledast\ r$, there exist $r'_1, \ldots, r'_n$ with $(r'_1, \ldots, r'_n)\ \mathfrak{R}_\circledast\ r'$ and $r_i \leq r'_i$ for each $1 \leq i \leq n$. We say that the ranking is *downwardly closed* if, $\lhd \subseteq \leq$.

Lemma 10. *For a downwardly-closed ranking, downward closure holds for games.*

Proof. Since $\lhd \subseteq \leq$ and $\leq$ is reflexive and transitive, it follows that $\lhd^* \subseteq \leq$. It is then sufficient to show that for every rank $r' \in \mathsf{Rank}$ and for all worlds $w, w' \in \mathsf{World}_\varsigma$ (for some sort ς) and every rank $r \in \mathsf{Rank}$ with $r \leq r'$, if

$$(w, w', r') \in \mathsf{DW}$$

then

$$(w, w', r) \in \mathsf{DW}.$$

The proof is by well-founded induction on the rank r'.

Consider the possible moves of Spoiler on the game (w, w', r). Assume that Spoiler selects some $(w^\mathsf{S}, w^\mathsf{D}) \in \{(w, w'), (w', w)\}$, some logical connective $\circledast$ and ranks $r_1, \ldots, r_n \in \mathsf{Rank}$ such that $(r_1, \ldots, r_n)\ \mathfrak{R}_\circledast\ r$ and some $w_1, \ldots, w_n$ such that $(w_1, \ldots, w_n)\ \mathfrak{M}_\circledast\ w^\mathsf{S}$ (otherwise, Duplicator wins by default). By the definition of $\leq$, there exist $r'_1, \ldots, r'_n$ with $(r'_1, \ldots, r'_n)\ \mathfrak{R}_\circledast\ r'$ and $r_i \leq r'_i$ for each $1 \leq i \leq n$. Hence, Spoiler could make the choice $(w^\mathsf{S}, w^\mathsf{D})$, $\circledast$, $r'_1, \ldots, r'_n$ and $w_1, \ldots, w_n$ on the game (w, w', r'). Duplicator's winning strategy gives some response to this move, say $w'_1, \ldots, w'_n$. Suppose that Duplicator responds

in this way on the game (w, w', r). Spoiler must then choose i and the game will continue from state (w_i, w'_i, r_i). Duplicator's winning strategy on the game (w, w', r') means that $(w_i, w'_i, r'_i) \in \mathsf{DW}$. By the inductive hypothesis (applicable since $r'_i \lhd r'$), since $r_i \leq r'_i$, $(w_i, w'_i, r_i) \in \mathsf{DW}$.

Therefore, $(w, w', r) \in \mathsf{DW}$, as required. □

The reason that these games are a useful tool for studying expressivity hinges on the fact that $(w, w', r) \in \mathsf{DW}$ if and only if w and w' are rank-r equivalent. This is established by soundness (the 'if' part) and completeness (the 'only if') part. The results are expressed in the contrapositive: w and w' are rank-r distinguishable if and only if $(w, w', r) \in \mathsf{SW}$.

3.1.3 Game Soundness

Theorem 11 (Soundness)**.** *Suppose that there is a rank-r formula F with $w \models F$ and $w' \not\models F$. Then $(w, w', r) \in \mathsf{SW}$.*

Game soundness is established by showing deriving a winning strategy for Spoiler from the formula F. Spoiler's choice of move will always match one of the outermost non-Boolean connectives of the formula. This leads to new pairs of worlds that are distinguished by subformulae, and so the result is established by induction.

Proof. The proof is by induction on the structure of the formula F, and cases on its outermost connective.

If the outermost connective is a Boolean connective then there is some rank-r subformula F' with either $w \models F'$ and $w' \not\models F'$ or $w' \models F'$ and $w \not\models F'$. By the inductive hypothesis, this implies that $(w, w', r) \in \mathsf{SW}$.

Otherwise, $F = \circledast(F_1, \ldots, F_n)$ for some non-Boolean connective $\circledast$ of arity n and some formulae $F_1, \ldots, F_n$ with ranks $r_1, \ldots, r_n$ where $(r_1, \ldots, r_n)\ \mathfrak{R}_\circledast\ r$. By definition,

$$w \models_\varsigma \circledast(F_1, \ldots, F_n) \iff \begin{array}{r}\text{there exist } w_1, \ldots, w_n \text{ s.t. } (w_1, \ldots, w_n)\ \mathfrak{M}_\circledast\ w \\ \text{and for all } i,\ w_i \models_{\varsigma_i} F_i.\end{array}$$

Suppose that Spoiler makes the choices:

- $(w^{\mathsf{S}}, w^{\mathsf{D}}) = (w, w')$;
- connective $\circledast$ and ranks $r_1, \ldots, r_n$ as given above; and
- $w_1, \ldots, w_n$ as given above.

If Duplicator cannot respond, then this is already a winning strategy for Spoiler. On the other hand, suppose that Duplicator does choose:

- some $w'_1, \ldots, w'_n$ with $(w'_1, \ldots, w'_n)\ \mathfrak{M}_{\circledast}\ w'$.

Then it must be the case that, for some $1 \leq j \leq n$, $w'_j \not\models F_j$ — for otherwise $w' \models F$, which we know not to be the case. Suppose then that Spoiler chooses:

- i to be this j, so that the game continues with (w_j, w'_j, r_j).

We have established that there is a formula, namely F_j, of rank r_j, with $w_j \models F_j$ and $w'_j \not\models F_j$. Therefore, by the inductive hypothesis, this gives Spoiler a winning strategy. Hence, $(w, w', r) \in \mathsf{SW}$. □

3.1.4 Game Completeness

Theorem 12 (Completeness)**.** *Suppose that $(w, w', r) \in \mathsf{SW}$. Then there is some rank-r formula F with $w \models F$ and $w' \not\models F$.*

Game completeness is established by using Spoiler's winning strategy to construct a formula. The outermost connective of the formula will correspond to Spoiler's choice of connective.

Proof. The proof is by induction on Spoiler on the rank r. Let ς be the sort of w and w'.

Spoiler's strategy must be to choose:

- some $(w^{\mathsf{S}}, w^{\mathsf{D}}) \in \{(w, w'), (w', w)\}$;
- some connective $\circledast$ and ranks $r_1, \ldots, r_n$ with $(r_1, \ldots, r_n)\ \mathfrak{R}_{\circledast}\ r$; and
- some worlds $w_1, \ldots, w_n$ with $(w_1, \ldots, w_n)\ \mathfrak{M}_{\circledast}\ w^{\mathsf{S}}$.

Let $F = \circledast(D^{r_1}_{w_1}, \ldots, D^{r_n}_{w_n})$. By definition, $w_i \models_{\varsigma_i} F_i$ for each i, and so $w^{\mathsf{S}} \models_{\varsigma} F$.

Suppose that also $w^{\mathsf{D}} \models_{\varsigma} F$. Then there must exist $w'_1, \ldots, w'_n$ with $(w'_1, \ldots, w'_n)\ \mathfrak{M}_{\circledast}\ w^{\mathsf{D}}$ and, for each i, $w'_i \models_{\varsigma_i} D^{r_i}_{w_i}$. By Lemma 8, this means that no rank-r_i formula discriminates between w_i and w_i, for each i. Hence, by the inductive hypothesis, $(w_i, w'_i, r_i) \in \mathsf{DW}$. Thus if Duplicator were to choose

- $w_1, \ldots, w_n$ as given above

it would give her a winning strategy, no matter which i Spoiler subsequently chose. This contradicts the fact that this is a winning strategy for Spoiler, and so it must be the case that $w^{\mathsf{D}} \not\models_{\varsigma} F$.

If $w^{\mathsf{S}} = w$ then $w \models F$ and $w' \not\models F$; if $w^{\mathsf{S}} = w'$ then $w \models \neg F$ and $w' \not\models \neg F$. Hence, there is a rank-r formula having the required property. □

3.2 Adjunct (Non)-elimination for CL^s_{Tree}

In my Masters' Thesis [DY06] I proved two results concerning adjunct elimination for CL^s_{Tree}. The first of these was a counterexample to the eliminability of the $\multimap\!\bullet$ connective. The second was a proof of the eliminability of the $\bullet\!\multimap$ connective. This first result is reproduced below.

3.2.1 Adjunct Elimination Counterexample

Theorem 13. *There is no adjunct-free formula of CL^s_{Tree} that is logically equivalent to the context formula $\mathbf{0} \multimap\!\bullet (\mathrm{True} \bullet \mathbf{a}[\mathbf{0}])$.*

Proof. Let $\mathbf{a}, \mathbf{b} \in \Sigma$ be distinct labels. Let $K = \mathbf{0} \multimap\!\bullet (\mathrm{True} \bullet \mathbf{a}[\mathbf{0}])$, and let c_i and c'_i be defined inductively as follows:

$$c_0 = \mathbf{a}[-] \qquad c'_0 = \mathbf{b}[-]$$
$$c_{i+1} = \mathbf{a}[c_i] \qquad c'_{i+1} = \mathbf{a}[c'_i].$$

It is clear that, for each i, $c_i \models K$ and $c'_i \not\models K$. We shall see that no adjunct-free formula shares this property. In particular, for all adjunct-free formulae K' there exists an n such that for all $j \geq n$, $c_j \models K'$ if and only if $c'_j \models K'$. The proof is by induction on the structure of K', taking cases on the outermost connective.

If $K' = \mathbf{a}'[K'']$ then either $\mathbf{a}' = \mathbf{a}$ or $c_j \not\models K'$ and $c'_j \not\models K'$ for all j. In the latter case, we are done. Let us assume then that $K' = \mathbf{a}[K'']$. By the inductive hypothesis, there is an n such that, for all $j \geq n$, $c_j \models K''$ if and only if $c'_j \models K''$. By construction, $c_{j+1} \models K'$ if and only if $c_j \models K''$, and $c'_{j+1} \models K'$ if and only if $c'_j \models K''$. Hence, for all $j \geq n+1$, $c_j \models K'$ if and only if $c'_j \models K'$.

If $K' = K'' \otimes P$ (or $K' = P \otimes K''$) then $c_j \models K'$ if and only if $c_j \models K''$ and $c'_j \models K'$ if and only if $c'_j \models K''$. This is since the only possible division of c_j as $c \otimes t$ is when $c = c_j$ and $t = \varnothing$. (Similarly for c'_j and for the division as $t \otimes c$.) By the inductive hypothesis, there is an n such that, for all $j \geq n$, $c_j \models K''$ if and only if $c'_j \models K''$. Consequently, for all $j \geq n$, $c_j \models K'$ if and only if $c'_j \models K'$.

If $K' = \mathbf{I}$ or $K' = \mathrm{False}$ then clearly $c_j \not\models K'$ and $c'_j \not\models K'$ for all j.

If $K' = K'' \to K'''$ then, by the inductive hypothesis there are n' and n'' such that for all $j \geq n'$, $c_j \models K''$ if and only if $c'_j \models K''$, and for all $j \geq n''$, $c_j \models K'''$ if and only if $c'_j \models K'''$. Let $n = \max(n', n'')$. It follows that, for all $j \geq n$, $c_j \models K'$ if and only if $c'_j \models K'$.

Since we have seen that, in every case of the structure of adjunct-free formula K', there is some n such that for every $j \geq n$ such that $c_j \models K'$ if and only if

$c'_j \models K'$. On the other hand, K does not have this property, and hence it is not equivalent to any adjunct-free formula. □

The key property of the logic that is exploited by Theorem 13 is that trees can be split arbitrarily while contexts cannot. Thus, inserting the empty tree into a context results in a tree which can be split in such a way as to specify properties that hold close to the context hole, regardless of how deep it was within the original context.

Although the proof is not presented in terms of games, the result originally arose from considerations in terms of games. The analysis is similar to constructing a winning strategy for Duplicator for the game (c_i, c'_i, r) where r does not permit Spoiler to play the connectives $\bullet\!\!-^{\exists}$ and $-\!\!\bullet^{\exists}$ and i is sufficiently large with respect to r. Such a strategy accounts for connectives Spoiler could play, just as the proof accounts for the possible outermost connectives of a potential discriminating formula. A winning strategy for Duplicator would establish the non-existence of a discriminating formula by game soundness, whereas the above proof does so directly.

3.2.2 Partial Adjunct Elimination

In my Masters' thesis [DY06], I established that the $\bullet\!\!-$ connective can be eliminated using games. I do not reproduce the result in full here, but report the key theorem underpinning the result. In order to prove adjunct elimination, it is sufficient to show that, if Duplicator has a winning strategy for a game that does not use $\bullet\!\!-^{\exists}$ then Duplicator also has a winning strategy for the same game when the $\bullet\!\!-^{\exists}$ connective may be played. This is shown by establishing that each time Spoiler plays the $\bullet\!\!-^{\exists}$ connective, Duplicator has a winning response given that she would have had a winning response if Spoiler had played any other connective. When Spoiler plays $\bullet\!\!-^{\exists}$, he introduces some new context, to which Duplicator is invited to respond with some context. The game will then continue with the two contexts or with the contexts applied to the trees. The key result is captured in the following theorem, which shows that Duplicator has a winning strategy by responding with *the same* context as Spoiler played:

Theorem 14. *For all ranks* $r \in \mathsf{Rank}$ *and for all* $t_1, t_2 \in \mathsf{Tree}$*, if*

$$(t_1, t_2, r) \in \mathsf{DW}$$

then for all contexts $c \in \mathsf{C}_{\mathsf{Tree}}$

$$(c \bullet t_1, c \bullet t_2, r) \in \mathsf{DW}.$$

The proof is intricate but is essentially founded on the principle that any decomposition of $c \bullet t_1$ (or indeed $c \bullet t_2$) that Spoiler chooses to do is a combination of some decomposition of t_1 (or t_2) and some decomposition of c. Duplicator can match these decompositions and induction is used to establish that she has a winning strategy when the game continues with the combination.

3.2.3 Adjunct Elimination Counterexample for Trees

While Theorem 13 established that the $\multimap\!\!\bullet$ connective does add expressivity to CL^s_{Tree}, it did so in terms of contexts. In particular, it does not show whether there is some *tree* formula that has no adjunct-free equivalent. This is an interesting problem, since contexts are really only a byproduct of reasoning about trees — it would matter little that adjoints are necessary to express context properties if they were not necessary to express tree properties. The context formula of Theorem 13 does not immediately suggest a tree formula counterexample, since trees permit the arbitrary-depth splitting that was key to the context formula counterexample. On the other hand, it would be difficult to adapt the games-based approach to adjunct elimination to show that $\multimap\!\!\bullet$ does not add expressive power just to tree formulae, since it works by inductively removing each usage at the point where it occurs — yet Theorem 13 shows that there can be subformulae from which $\multimap\!\!\bullet$ cannot be removed. I resolve the issue by showing that, in fact, there is a tree formula that uses $\multimap\!\!\bullet$, but which has no adjunct-free equivalent.

Theorem 15. *There is no adjunct-free formula of CL^s_{Tree} that is logically equivalent to the tree formula* $\neg(\mathbf{a}'[\mathbf{0}] \multimap\!\!\bullet (\mathrm{True} \bullet \mathbf{b}[\mathrm{True} \bullet \mathbf{a}'[\mathbf{0}]])) \bullet \mathbf{a}[\mathbf{0}]$.

The proof of this result makes use of Ehrenfeucht-Fraïssé games for CL^s_{Tree} to show that, for each rank r that does not permit the adjunct connectives, there are certain trees, namely $u(i,j)$ and $v(i,j)$ (defined below), for sufficiently large i and j, that are rank-r equivalent. These trees are defined in such a way that they are discriminated by the formula in Theorem 15. Therefore, there can be no adjunct-free formula equivalent to it.

The trees $u(i,j)$ and $v(i,j)$ are defined by mutual recursion as follows:

$$u(i,0) = \mathbf{c}^i[\mathbf{b}[\mathbf{a}[\varnothing]]] \qquad v(i,0) = \mathbf{c}^i[\mathbf{a}[\varnothing]]$$
$$u(i,j+1) = \mathbf{c}^i[\mathbf{b}[u(i,j) \otimes v(i,j)]] \qquad v(i,j+1) = \mathbf{c}^i[u(i,j) \otimes v(i,j)]$$

where $\mathbf{a}, \mathbf{b}, \mathbf{c} \in \Sigma$ are some fixed labels and $\mathbf{c}^i[t]$ denotes iterated labelling (that is, $\mathbf{c}^0[t] = t$ and $\mathbf{c}^{i+1}[t] = \mathbf{c}[\mathbf{c}^i[t]]$).

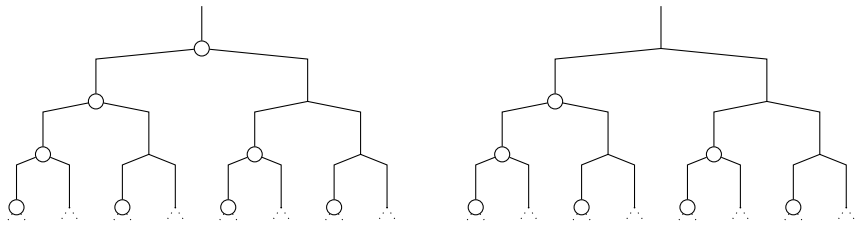


Figure 3.1: Schematic diagrams of $u(i,j)$ (left) and $v(i,j)$.

Figure 3.1 illustrates these trees schematically with the **b**-labelled nodes denoted by circles. In $u(i,j)$, every **a**-labelled leaf node has a **b**-labelled ancestor; in $v(i,j)$ there is one **a**-labelled leaf node which does not have any **b**-labelled ancestor (reached by taking the right-hand branch each time). It because of this that the formula $\neg(\mathbf{a}'[\mathbf{0}] \multimap (\mathrm{True} \bullet \mathbf{b}[\mathrm{True} \bullet \mathbf{a}'[\mathbf{0}]])) \bullet \mathbf{a}[\mathbf{0}]$ is be able to distinguish them.

While they are clearly distinct, the two trees only differ in the presence of a single **b**-labelled node at the first branching point. Any winning strategy for Spoiler would have to focus on this difference. However, even this first branching point will be buried sufficiently deeply within the tree that Spoiler will not be able to distinguish it from one of the second branching points on the opposite tree. The self-similarity of the trees is thus key to Duplicator having a winning strategy.

The games I use for this result are based on the following ranking:

Definition 3.7 (Adjunct-Distinction Ranking for CL^s_{Tree})**.** The *adjunct-distinction ranking* consists of the bounded join-semilattice $(\mathsf{Rank}, \sqsubseteq)$ given by:

- $\mathsf{Rank} = \mathbb{N} \times \mathbb{N} \times \mathcal{P}_{\mathrm{fin}}(\Sigma)$, and
- $(m,s,L) \sqsubseteq (m',s',L')$ if and only if $m \leq m'$, $s \leq s'$ and $L \subseteq L'$;

and ranking relations $\mathfrak{R}_{\circledast}$ given by:

- for $\circledast \in \{\mathbf{0}, \otimes, \bullet, \mathbf{I}\}$, $(r_1, \ldots, r_n)\ \mathfrak{R}_{\circledast}\ r$ if and only if $(m+1,s,L) \sqsubseteq r$ where $(m,s,L) = \bigsqcup\{r_i \mid 1 \leq i \leq n\}$,
- for $\circledast = \mathbf{a}[]$ (for some $\mathbf{a} \in \Sigma$), $((m,s,L))\ \mathfrak{R}_{\circledast}\ r$ if and only if $(m+1,s,L \cup \{\mathbf{a}\}) \sqsubseteq r$, and
- for $\circledast \in \{\bullet\!\!-^{\exists}, -\!\!\bullet^{\exists}\}$, $(r_1,r_2)\ \mathfrak{R}_{\circledast}\ r$ if and only if $(m,s+1,L) \sqsubseteq r$ where $(m,s,L) = r_1 \sqcup r_2$.

It is not difficult to verify that the above definition meets the requirements of a ranking (Definition 3.1). $(\mathsf{Rank}, \sqsubseteq)$ is a bounded join-semilattice since both $(\mathbb{N}, \leq)$ and $(\mathcal{P}_{\text{fin}}(\Sigma), \subseteq)$ are bounded join-semilattices. The minimal element is $(0, 0, \emptyset)$. The ranking is constructive by the fact that all joins exist (including of the empty set). The ranking is upwardly closed by the definition of $\mathfrak{R}_{\circledast}$ and the fact that $\sqsubseteq$ is transitive. The ranking is finitely branching by the fact that only finitely many operators can be used in the construction of a given rank (since the set component is restricted to being finite) and that the rank is always constructed from lesser ranks, of which there are only finitely many. The ranking is inductive by the fact that $\lhd \subseteq (\sqsubseteq) \setminus (=)$ by the definition of $\mathfrak{R}_{\circledast}$, and $(\sqsubseteq) \setminus (=)$ is well-founded (since $<$ and $\subset$ are well-founded). It is also clear that the above ranking is downwardly closed, since $\lhd \subseteq \sqsubseteq$.

The essential purpose of the ranking is that it distinguishes formulae that contain no adjunct connectives: they are the ones that have ranks of the form $(m, 0, L)$. Correspondingly, ranks of the form $(m, 0, L)$ disallow the use of the adjunct connectives in games.

The proof of Theorem 15 makes use of a number of auxiliary lemmata, as well as Theorem 14. These results are used to establish winning strategies for Duplicator in certain games. The first of these, Lemma 16, captures in terms of games the intuition behind Theorem 13: that the contexts $\mathbf{a}_1[\mathbf{a}_2[\cdots \mathbf{a}_n[c]]]$ and $\mathbf{a}_1[\mathbf{a}_2[\cdots \mathbf{a}_n[c']]]$ cannot be distinguished by a formula of rank $(m, 0, L)$. (This is not directly what is stated by the lemma, but is a consequence of its iterated application.)

Lemma 16. *If*

$$(c, c', (m, 0, L)) \in \mathsf{DW} \tag{3.1}$$

then, for any $\mathbf{a} \in \Sigma$*,*

$$(\mathbf{a}[c], \mathbf{a}[c'], (m + 1, 0, L)) \in \mathsf{DW}. \tag{3.2}$$

Proof. The proof is by induction on m. Consider the possible moves Spoiler could make on the game in (3.2). If Spoiler plays the $\mathbf{I}$ or $\mathbf{b}[]$ (for $\mathbf{b} \neq \mathbf{a}$) connectives, Duplicator wins immediately. If Spoiler plays the $\otimes$ connective then he splits one of the contexts, say $\mathbf{a}[c]$, into $\mathbf{a}[c]$ and $\varnothing$ (on one side or the other). If Duplicator responds by splitting $\mathbf{a}[c']$ into $\mathbf{a}[c']$ and $\varnothing$, then she has a winning strategy provided that

$$\begin{aligned}(\mathbf{a}[c], \mathbf{a}[c'], (m, 0, L)) &\in \mathsf{DW}\\ (\varnothing, \varnothing, (m, 0, L)) &\in \mathsf{DW}.\end{aligned}$$

The first of these holds by downward closure on (3.1) and the inductive hypothesis. The second holds by game completeness. If Spoiler plays the $\mathbf{a}[]$ connective then the Duplicator can ensure that the game continues as in (3.1), and so Duplicator has a winning strategy. This covers all of the possible moves Spoiler could make, and so Duplicator does have a winning strategy. □

The following lemma is used to establish winning strategies for Duplicator when contexts and trees are horizontally concatenated. The proviso that the contexts and trees must not themselves be horizontal concatenations (except of $\varnothing$ and themselves) constrains Spoiler's moves, facilitating a simple proof.

Lemma 17. *If*

$$(c, c', (m, 0, L)) \in \mathsf{DW} \tag{3.3}$$

$$(t, t', (m, 0, L)) \in \mathsf{DW} \tag{3.4}$$

and c and c' are each of the form $\mathbf{a}[c'']$ (for some label $\mathbf{a} \in \Sigma$ and context c'') or $-$, and t and t' are each of the form $\mathbf{a}[t'']$ (for some label $\mathbf{a} \in \Sigma$ and context c''), then

$$(c \otimes t, c' \otimes t', (m+1, 0, L)) \in \mathsf{DW} \tag{3.5}$$

$$(t \otimes c, t' \otimes c', (m+1, 0, L)) \in \mathsf{DW}. \tag{3.6}$$

Proof. The proof is by induction on m. Consider the case for (3.5); the case for (3.6) is analogous. Consider the possible moves that Spoiler could play in the game. If he players the $\mathbf{I}$ or $\mathbf{b}[]$ (for any $\mathbf{b} \in \Sigma$) connectives, Duplicator wins immediately. The remaining possibility is that Spoiler plays the $\otimes$ connective, splitting one of the contexts. If he splits $c \otimes t$ into c and t, then Duplicator has a winning response with c' and t' by (3.3) and (3.4). If he splits it into $\varnothing$ and $c \otimes t$, the Duplicator has a winning response with $\varnothing$ and $c' \otimes t'$ by game completeness and by downwards closure and the inductive hypothesis. The same analysis applies if Spoiler splits $c' \otimes t'$ instead. Since all cases lead to a winning strategy for Duplicator, it follows that (3.5) holds. □

The following lemma establishes that Duplicator can exploit the self-similarity present in $u(i, j)$ and $v(i, j)$ as part of a winning strategy.

Lemma 18. *For $i \geq m$ and $j \geq 2m$,*

$$(u(i,j) \otimes v(i,j), u(i,j+1) \otimes v(i,j+1), (m, 0, L)) \in \mathsf{DW}.$$

Proof. The proof is by induction on m and cases on the possible moves of Spoiler. The only move that Spoiler could usefully play is the $\bullet$ move, splitting either the smaller or larger of the two trees.

Suppose that Spoiler splits the smaller. Suppose that Spoiler splits the tree at the top level. The decomposition is then one of the following:

$$c = - \qquad t = u(i,j) \otimes v(i,j) \tag{3.7}$$
$$c = - \otimes u(i,j) \otimes v(i,j) \qquad t = \varnothing \tag{3.8}$$
$$c = - \otimes v(i,j) \qquad t = u(i,j) \tag{3.9}$$
$$c = u(i,j) \otimes - \otimes v(i,j) \qquad t = \varnothing \tag{3.10}$$
$$c = u(i,j) \otimes - \qquad t = v(i,j) \tag{3.11}$$
$$c = u(i,j) \otimes v(i,j) \otimes - \qquad t = \varnothing \tag{3.12}$$

By game completeness,

$$(-,-,(m-1,0,L)) \in \mathsf{DW} \tag{3.13}$$
$$(\varnothing,\varnothing,(m-1,0,L)) \in \mathsf{DW}. \tag{3.14}$$

By the inductive hypothesis, together with Theorem 14, applying the contexts $\mathbf{c}^i[\mathbf{b}[-]]$ and $\mathbf{c}^i[-]$ respectively,

$$(u(i,j), u(i,j+1), (m-1,0,L)) \in \mathsf{DW} \tag{3.15}$$
$$(v(i,j), v(i,j+1), (m-1,0,L)) \in \mathsf{DW}. \tag{3.16}$$

Applying Lemma 17 and downward closure to the above gives

$$(u(i,j) \otimes -, u(i,j+1) \otimes -, (m-1,0,L)) \in \mathsf{DW} \tag{3.17}$$
$$(v(i,j) \otimes -, v(i,j+1) \otimes -, (m-1,0,L)) \in \mathsf{DW} \tag{3.18}$$
$$(- \otimes u(i,j), - \otimes u(i,j+1), (m-1,0,L)) \in \mathsf{DW} \tag{3.19}$$
$$(- \otimes v(i,j), - \otimes v(i,j+1), (m-1,0,L)) \in \mathsf{DW}. \tag{3.20}$$

By the inductive hypothesis,

$$(u(i,j) \otimes v(i,j), u(i,j+1) \otimes v(i,j+1), (m-1,0,L)) \in \mathsf{DW}. \tag{3.21}$$

Consider the games

$$(- \otimes u(i,j) \otimes v(i,j), - \otimes u(i,j+1) \otimes v(i,j+1), (m',0,L)) \tag{3.22}$$
$$(u(i,j) \otimes - \otimes v(i,j), u(i,j+1) \otimes - \otimes v(i,j+1), (m',0,L)) \tag{3.23}$$
$$(u(i,j) \otimes v(i,j) \otimes -, u(i,j+1) \otimes v(i,j+1) \otimes -, (m',0,L)) \tag{3.24}$$

By induction on m', let us show that, for $m' < m$, Duplicator has a winning strategy for each of the above games. In the base case $m' = 0$ and so the result holds trivially. In the inductive case, the only relevant moves for Spoiler to play on any of the games are the $\otimes$ moves. If Duplicator imitates Spoiler's spitting on the other context, then the inductive hypothesis and the various results above, together with downward closure, establish that this gives Duplicator a winning strategy.

The above establish that Duplicator has a winning strategy by responding to Spoiler's splitting with the appropriate one of the following:

$$\begin{aligned}
&c' = _ && t' = u(i,j+1) \otimes v(i,j+1)\\
&c' = _ \otimes u(i,j+1) \otimes v(i,j+1) && t' = \varnothing\\
&c' = _ \otimes v(i,j+1) && t' = u(i,j+1)\\
&c' = u(i,j+1) \otimes _ \otimes v(i,j+1) && t' = \varnothing\\
&c' = u(i,j+1) \otimes _ && t' = v(i,j+1)\\
&c' = u(i,j+1) \otimes v(i,j+1) \otimes _ && t' = \varnothing.
\end{aligned}$$

If Spoiler does not split the tree at the top level then he splits inside one of the two branches. Suppose that Spoiler makes this splitting at a depth greater than m, that is, he chooses a context

$$c = \mathbf{c}^m[c_1] \otimes v(i,j) \text{ or} \tag{3.25}$$

$$c = u(i,j) \otimes \mathbf{c}^m[c_1], \tag{3.26}$$

for some context c_1, and some tree t such that $u(i,j) \otimes v(i,j) = c \bullet t$. In the case of (3.25), suppose that Duplicator responds with

$$c' = \mathbf{c}^i[\mathbf{b}[\mathbf{c}^m[c_1] \otimes v(i,j)]] \otimes v(i,j+1) \text{ and} \tag{3.27}$$

$$t' = t. \tag{3.28}$$

If Spoiler continues the game as $(t, t', (m-1, 0, L))$ then Duplicator has a winning strategy by game completeness. By iterated application of Lemma 16,

$$(\mathbf{c}^m[c_1], \mathbf{c}^i[\mathbf{b}[\mathbf{c}^m[c_1] \otimes v(i,j)]], (m-1, 0, L)) \in \mathsf{DW}. \tag{3.29}$$

By the inductive hypothesis (since we have already assumed $m \geq 1$, this implies $j \geq 2$),

$$(u(i,j-1) \otimes v(i,j-1), u(i,j) \otimes v(i,j), (m-1, 0, L)) \in \mathsf{DW}. \tag{3.30}$$

Applying Theorem 14 to (3.30) with the context $\mathbf{c}^i[\_]$ gives

$$(v(i,j), v(i,j+1), (m-1, 0, L)) \in \mathsf{DW}. \tag{3.31}$$

Applying Lemma 17 to (3.29) and (3.31) gives

$$(c, c', (m-1, 0, L)) \in \mathsf{DW}. \tag{3.32}$$

Hence Duplicator has a winning however Spoiler chooses to continue.

If Spoiler instead chose the context given in (3.26), suppose that Duplicator responds with

$$c' = u(i, j+1) \otimes \mathbf{c}^i[u(i,j) \otimes \mathbf{c}^m[c']] \text{ and} \tag{3.33}$$

$$t' = t. \tag{3.34}$$

By a similar argument to the above, it can again be established that this gives Duplicator a winning strategy.

If Spoiler splits the tree at a depth no greater than m then the splitting must be one of the following:

$$c = \mathbf{c}^k[\_] \otimes v(i,j) \qquad t = \mathbf{c}^{i-k}[\mathbf{b}[u(i,j-1) \otimes v(i,j-1)]] \tag{3.35}$$

$$c = u(i,j) \otimes \mathbf{c}^k[\_] \qquad t = \mathbf{c}^{i-k}[u(i,j-1) \otimes v(i,j-1)] \tag{3.36}$$

$$c = \mathbf{c}^k[\_ \otimes \mathbf{c}^{i-k}[\mathbf{b}[u(i,j-1) \otimes v(i,j-1)]]] \otimes v(i,j) \qquad t = \varnothing \tag{3.37}$$

$$c = \mathbf{c}^k[\mathbf{c}^{i-k}[\mathbf{b}[u(i,j-1) \otimes v(i,j-1)]] \otimes \_] \otimes v(i,j) \qquad t = \varnothing \tag{3.38}$$

$$c = u(i,j) \otimes \mathbf{c}^k[\_ \otimes \mathbf{c}^{i-k}[u(i,j-1) \otimes v(i,j-1)]] \qquad t = \varnothing \tag{3.39}$$

$$c = u(i,j) \otimes \mathbf{c}^k[\mathbf{c}^{i-k}[u(i,j-1) \otimes v(i,j-1)] \otimes \_] \qquad t = \varnothing \tag{3.40}$$

for some $1 \leq k \leq m$. In the case of (3.35), suppose that Duplicator responds with the splitting

$$c' = \mathbf{c}^k[\_] \otimes v(i,j+1) \qquad t' = \mathbf{c}^{i-k}[\mathbf{b}[u(i,j) \otimes v(i,j)]]. \tag{3.41}$$

Recalling (3.15) and since $(\mathbf{c}^k[\_], \mathbf{c}^k[\_], (m-1,0,L)) \in \mathsf{DW}$ by game completeness, applying Lemma 17 gives

$$(\mathbf{c}^k[\_] \otimes v(i,j), \mathbf{c}^k[\_] \otimes v(i,j+1), (m-1,0,L)) \in \mathsf{DW}. \tag{3.42}$$

Recalling (3.30) and applying Theorem 14 with the context $\mathbf{c}^{i-k}[\mathbf{b}[\_]]$ gives

$$(\mathbf{c}^{i-k}[\mathbf{b}[u(i,j-1) \otimes v(i,j-1)]], \mathbf{c}^{i-k}[\mathbf{b}[u(i,j) \otimes v(i,j)]], (m,0,L)) \in \mathsf{DW}. \tag{3.43}$$

Hence, this gives Duplicator a winning strategy. In the case of (3.36), similar reasoning gives Duplicator a winning strategy by choosing the splitting

$$c = u(i,j+1) \otimes \mathbf{c}^k[\_] \qquad t = \mathbf{c}^{i-k}[u(i,j) \otimes v(i,j)]. \tag{3.44}$$

For the remaining four cases, suppose that Duplicator responds with the corresponding splitting below:

$$c' = \mathbf{c}^k[- \otimes \mathbf{c}^{i-k}[\mathbf{b}[u(i,j) \otimes v(i,j)]]] \otimes v(i,j+1) \qquad t' = \varnothing \tag{3.45}$$

$$c' = \mathbf{c}^k[\mathbf{c}^{i-k}[\mathbf{b}[u(i,j) \otimes v(i,j)]] \otimes -] \otimes v(i,j+1) \qquad t' = \varnothing \tag{3.46}$$

$$c' = u(i,j+1) \otimes \mathbf{c}^k[- \otimes \mathbf{c}^{i-k}[u(i,j) \otimes v(i,j)]] \qquad t' = \varnothing \tag{3.47}$$

$$c' = u(i,j+1) \otimes \mathbf{c}^k[\mathbf{c}^{i-k}[u(i,j) \otimes v(i,j)] \otimes -] \qquad t' = \varnothing. \tag{3.48}$$

Duplicator has a winning strategy if Spoiler continues with the trees by game completeness. By the inductive hypothesis

$$(u(i,j-1) \otimes v(i,j-1), u(i,j) \otimes v(i,j), (m-1,0,L)) \in \mathsf{DW}. \tag{3.49}$$

Applying Theorem 14, Lemma 16, Lemma 17 (iteratively), and Lemma 16 again (recalling (3.15) and (3.16)), it can be established that Duplicator has a winning strategy with this response.

This covers all possibilities of Spoiler splitting the smaller tree. Suppose that Spoiler splits the larger tree. If Spoiler splits the tree at the top level then the same analysis applies as for Spoiler splitting the smaller tree in the same manner. This gives Duplicator a winning strategy. Assume Spoiler splits the tree at a point inside one of the second-level branches below depth m, for example, choosing

$$c = \mathbf{c}^i[\mathbf{b}[\mathbf{c}^m[c_1] \otimes v(i,j)]] \otimes v(i,j+1) \tag{3.50}$$

for some c_1, and some t. (The analysis for the other three cases of Spoiler making such a splitting is similar.) Duplicator could respond with

$$c' = \mathbf{c}^m[c_1] \otimes v(i,j) \tag{3.51}$$

$$t' = t. \tag{3.52}$$

By iterated application of Lemma 16,

$$(\mathbf{c}^m[c_2], \mathbf{c}^m[c_2'], (m-1,0,L)) \in \mathsf{DW} \tag{3.53}$$

for any c_2, c_2'. In particular,

$$(\mathbf{c}^i[\mathbf{b}[\mathbf{c}^m[c_1] \otimes v(i,j)]], \mathbf{c}^m[c_1], (m-1,0,L)) \in \mathsf{DW}. \tag{3.54}$$

Applying Lemma 17, recalling (3.16), gives

$$(c, c', (m-1,0,L)) \in \mathsf{DW}. \tag{3.55}$$

Hence, and by game completeness, Duplicator has a winning strategy in this case.

Assume Spoiler splits the tree inside one of the second-level branches above depth m. One possibility for such a splitting is

$$c = \mathbf{c}^i[\mathbf{b}[\mathbf{c}^k[\_] \otimes v(i,j)]] \otimes v(i,j+1) \tag{3.56}$$

$$t = \mathbf{c}^{i-k}[u(i,j-1) \otimes v(i,j-1)] \tag{3.57}$$

for some $1 \leq k \leq m$. There are three other cases (spitting in each of the second-level branches) that follow the same analysis as this one. Spoiler could also choose to split with $t = \varnothing$ and, for example,

$$c = \mathbf{c}^i[\mathbf{b}[\mathbf{c}^k[\_ \otimes \mathbf{c}^{i-k}[u(i,j-1) \otimes v(i,j-1)]]]] \otimes v(i,j+1) \tag{3.58}$$

for some $1 \leq k \leq m$. There are seven other cases (splitting in each of the second-level branches and putting the hole on either side of the tree) that follow the same analysis as this one.

Consider the splitting of (3.56) and (3.57). Duplicator could respond with

$$c' = \mathbf{c}^i[\mathbf{b}[\mathbf{c}^k[\_] \otimes v(i,j-1)]] \otimes v(i,j) \tag{3.59}$$

$$t' = \mathbf{c}^{i-k}[u(i,j-2) \otimes v(i,j-2)]. \tag{3.60}$$

By Lemma 17, recalling (3.53) and (3.16),

$$(c, c', (m-1, 0, L)) \in \mathsf{DW}. \tag{3.61}$$

By the inductive hypothesis, since $j \geq 2m$ means that $j - 2 \geq 2(m-1)$,

$$(u(i,j-2) \otimes v(i,j-2), u(i,j-1) \otimes v(i,j-1), (m,0,L)) \in \mathsf{DW}. \tag{3.62}$$

Applying Theorem 14 with the context $\mathbf{c}^{i-k}[\_]$ gives

$$(t, t', (m-1, 0, L)) \in \mathsf{DW}. \tag{3.63}$$

Hence, Duplicator has a winning strategy in this case. I omit the other, similar cases.

Consider the splitting given by $t = \varnothing$ and (3.58). Duplicator could respond with $t' = \varnothing$ and

$$c = \mathbf{c}^i[\mathbf{b}[\mathbf{c}^k[\_ \otimes \mathbf{c}^{i-k}[u(i,j-2) \otimes v(i,j-2)]]]] \otimes v(i,j). \tag{3.64}$$

Recalling (3.53) and (3.16), Lemma 17 gives

$$(c, c', (m-1, 0L)) \in \mathsf{DW}. \tag{3.65}$$

Hence, and by game completeness, Duplicator has a winning strategy in this case. I again omit the other, similar cases.

The remaining case is if Spoiler splits the larger tree in one of the first-level branches above the second-level branches. That is, either

$$c = \mathbf{c}[c_1] \otimes v(i, j+1) \qquad t = c_2 \bullet (u(i,j) \otimes v(i,j)) \text{ or} \tag{3.66}$$

$$c = u(i, j+1) \otimes \mathbf{c}[c_1] \qquad t = c_2 \bullet (u(i,j) \otimes v(i,j)) \tag{3.67}$$

for some c_1, c_2, or $t = \varnothing$ and

$$c = \mathbf{c}[c_1 \circ (\_ \otimes (c_2 \bullet (u(i,j) \otimes v(i,j))))] \otimes v(i, j+1) \tag{3.68}$$

$$c = \mathbf{c}[c_1 \circ ((c_2 \bullet (u(i,j) \otimes v(i,j))) \otimes \_)] \otimes v(i, j+1) \tag{3.69}$$

$$c = u(i, j+1) \otimes \mathbf{c}[c_1 \circ (\_ \otimes (c_2 \bullet (u(i,j) \otimes v(i,j))))] \text{ or} \tag{3.70}$$

$$c = u(i, j+1) \otimes \mathbf{c}[c_1 \circ ((c_2 \bullet (u(i,j) \otimes v(i,j))) \otimes \_)] \tag{3.71}$$

for some c_1, c_2.

In the case of (3.66), Duplicator can respond with

$$c' = \mathbf{c}[c_1] \otimes v(i,j) \qquad t = c_2 \bullet (u(i, j-1) \otimes v(i, j-1)). \tag{3.72}$$

By Lemma 17, recalling (3.16),

$$(c, c', (m-1, 0, L)) \in \mathsf{DW}. \tag{3.73}$$

By the inductive hypothesis and applying Theorem 14 with the context c_2,

$$(t, t', (m-1, 0, L)) \in \mathsf{DW}. \tag{3.74}$$

Hence, Duplicator has a winning strategy in this case. The case for (3.67) is similar, with Duplicator responding

$$c' = u(i,j) \otimes \mathbf{c}[c_1] \qquad t = c_2 \bullet (u(i, j-1) \otimes v(i, j-1)). \tag{3.75}$$

In the case of (3.68), Duplicator can respond with $t' = \varnothing$ and

$$c' = \mathbf{c}[c_1 \circ (\_ \otimes (c_2 \bullet (u(i, j-1) \otimes v(i, j-1))))] \otimes v(i,j). \tag{3.76}$$

Recalling the derivation of (3.74),

$$(c_2 \bullet (u(i,j) \otimes v(i,j)), c_2 \bullet (u(i, j-1) \otimes v(i, j-1)), (m-1, 0, L)) \in \mathsf{DW}. \tag{3.77}$$

Applying Lemma 17 (with both contexts being $\_$), iteratively applying Lemma 16 (since c_1 must be $\mathbf{c}^k[\_]$ for some k or $\mathbf{c}^i[\mathbf{b}[\_]]$), and applying Lemma 17 again (recalling (3.16)) gives

$$(c, c', (m-1, 0, L)) \in \mathsf{DW}. \tag{3.78}$$

Hence, and by game completeness, Duplicator has a winning strategy in this case. The cases for (3.69), (3.70) and (3.71) are similar.

For each possible move Spoiler could make, we have seen that Duplicator has a winning response. Hence,

$$(u(i,j) \otimes v(i,j), u(i,j+1) \otimes v(i,j+1), (m,0,L)) \in \mathsf{DW}.$$

□

The next lemma establishes that Duplicator has a winning strategy for the adjunct-free game played on $u(i,j)$ and $v(i,j)$ for i and j sufficiently large. The statement of the lemma allows the length of the top branch to vary independently of the i parameter in order facilitate the inductive argument.

Lemma 19. *For $i \geq m^2$, $j \geq 2m$, $k \geq m^2$,*

$$(\mathbf{c}^k[\mathbf{b}[u(i,j) \otimes v(i,j)]], \mathbf{c}^k[u(i,j) \otimes v(i,j)], (m,0,L)) \in \mathsf{DW}.$$

Proof. The proof is by induction on m and cases on the possible moves of Spoiler. The only move that Spoiler could usefully play is the $\bullet$ move, splitting one of the two trees.

If Spoiler splits the first tree into $\mathbf{c}^k[\mathbf{b}[c_1]]$ and t for some c_1, t then Duplicator can respond with $\mathbf{c}^k[c_1]$ and the same t. This leads to a winning strategy for Duplicator since, by iterated application of Lemma 16,

$$(\mathbf{c}^i[\mathbf{b}[c_1]], \mathbf{c}^i[c_1], (m-1,0,L)) \in \mathsf{DW},$$

and, by game completeness,

$$(t, t, (m-1,0,L)) \in \mathsf{DW}.$$

Conversely, if Spoiler splits the second tree into $\mathbf{c}^k[c_1]$ and t for some c_1, t, Duplicator has a winning strategy by responding with $\mathbf{c}^k[\mathbf{b}[c_1]]$ and t.

If Spoiler splits the first tree into $\mathbf{c}^n[c_1]$ and $c_2 \bullet (u(i,j) \otimes v(i,j))$ for some c_1, c_2, then Duplicator can respond with $\mathbf{c}^k[\mathbf{c}^n[c_1] \otimes v(i,j)]$ and $c_2 \bullet (u(i,j-1) \otimes v(i,j-1))$. By iterated application of Lemma 16,

$$(\mathbf{c}^n[c_1], \mathbf{c}^k[\mathbf{c}^n[c_1] \otimes v(i,j)], (m-1,0,L)) \in \mathsf{DW}.$$

By Lemma 18,

$$(u(i,j) \otimes v(i,j), u(i,j-1) \otimes v(i,j-1), (m-1,0,L)) \in \mathsf{DW}$$

and so, by applying Theorem 14 with the context c_2,

$$(c_2 \bullet (u(i,j) \otimes v(i,j)), c_2 \bullet (u(i,j-1) \otimes v(i,j-1)), (m-1,0,L)) \in \mathsf{DW}.$$

Hence, this gives Duplicator a winning strategy.

If Spoiler splits the second tree into $\mathbf{c}^n[c_1]$ and $c_2 \bullet (u(i,j) \otimes v(i,j))$ for some c_1, c_2, then Duplicator can respond with $\mathbf{c}^k[u(i,j) \otimes \mathbf{c}^n[c_1]]$ and $c_2 \bullet (u(i,j-1) \otimes v(i,j-1))$. By a similar argument to the above, this gives Duplicator a winning strategy.

If Spoiler splits the first tree into $\mathbf{c}^l[\_]$ and $\mathbf{c}^{k-l}[\mathbf{b}[u(i,j) \otimes v(i,j)]]$, for $l \leq m$, then Duplicator can respond with $\mathbf{c}^l[\_]$ and $\mathbf{c}^{k-l}[u(i,j) \otimes v(i,j)]$. By game completeness,

$$(\mathbf{c}^l[\_], \mathbf{c}^l[\_], (m-1, 0, L)) \in \mathsf{DW}.$$

Bearing in mind that $m \geq 1$ (for otherwise Spoiler could not play any move),

$$k - l \geq m^2 - m \geq m^2 - m + 1 - m = (m-1)^2.$$

Hence, by the inductive hypothesis,

$$(\mathbf{c}^{k-l}[\mathbf{b}[u(i,j) \otimes v(i,j)]], \mathbf{c}^{k-l}[u(i,j) \otimes v(i,j)], (m-1, 0, L)) \in \mathsf{DW}.$$

This therefore gives Duplicator a winning strategy.

If Spoiler splits the second tree into $\mathbf{c}^l[\_]$ and $\mathbf{c}^{k-l}[u(i,j) \otimes v(i,j)]$, for $l \leq m$, then Duplicator can respond with $\mathbf{c}^l[\_]$ and $\mathbf{c}^{k-l}[\mathbf{b}[u(i,j) \otimes v(i,j)]]$. This gives Duplicator a winning strategy, as before.

If Spoiler splits the first tree into $\mathbf{c}^l[\_ \otimes \mathbf{c}^{k-l}[\mathbf{b}[u(i,j) \otimes v(i,j)]]]$ and $\varnothing$, for $l \leq k$, then Duplicator can respond with $\mathbf{c}^l[\_ \otimes \mathbf{c}^{k-l}[u(i,j) \otimes v(i,j)]]$ and $\varnothing$. By iterated application of Lemma 16,

$$(\mathbf{c}^{k-l}[\mathbf{b}[u(i,j) \otimes v(i,j)]], \mathbf{c}^{k-l}[u(i,j) \otimes v(i,j)], (m-k-2, 0, L)) \in \mathsf{DW}.$$

By Lemma 17,

$$(\_ \otimes \mathbf{c}^{k-l}[\mathbf{b}[u(i,j) \otimes v(i,j)]], \_ \otimes \mathbf{c}^{k-l}[u(i,j) \otimes v(i,j)], (m-k-1, 0, L)) \in \mathsf{DW}.$$

By iterated application of Lemma 16, again,

$$(\mathbf{c}^k[\_ \otimes \mathbf{c}^{k-l}[\mathbf{b}[u(i,j) \otimes v(i,j)]]], \mathbf{c}^k[\_ \otimes \mathbf{c}^{k-l}[u(i,j) \otimes v(i,j)]], (m-1, 0, L)) \in \mathsf{DW}.$$

Naturally,

$$(\varnothing, \varnothing, (m-1, 0, L)) \in \mathsf{DW}.$$

Hence, Duplicator has a winning strategy.

Similar arguments hold for if Spoiler places the hole on the right, and if Spoiler splits the second of the trees in such a way.

This covers every way that Spoiler may split either tree, and Duplicator has a winning strategy for each. Hence, as required,

$$(\mathbf{c}^k[\mathbf{b}[u(i,j) \otimes v(i,j)]], \mathbf{c}^k[u(i,j) \otimes v(i,j)], (m, 0, L)) \in \mathsf{DW}.$$

□

The above result pertaining to games gives the following corollary pertaining to formulae by the soundness of games:

Corollary 20. *There is no of adjunct-free formula of CL^s_{Tree}, P, such that, for all $i, j \in \mathbb{N}$,*

$$u(i,j) \not\models_{\mathsf{d}} P$$
$$v(i,j) \models_{\mathsf{d}} P.$$

Proof. Any such formula would have some rank $(m, 0, L)$. By game soundness, this would mean that

$$(u(i,j), v(i,j), (m,0,L)) \in \mathsf{SW}$$

for every $i, j \in \mathbb{N}$. This contradicts Lemma 19, and so such a formula could not exist. □

Having established in Corollary 20 a property that no adjunct-free formula exhibits, to prove that the $\multimap\!\bullet$ connective adds expressive power it is sufficient to show that some formula that uses $\multimap\!\bullet$ does exhibit this property.

Proof of Theorem 15. Fix some $i, j \in \mathbb{N}$. Let $c \in \mathsf{C}_{\mathsf{Tree}}$ be such that $u(i,j) = c \bullet \mathbf{a}[\varnothing]$. By structural considerations, $c = \mathbf{c}^i[\mathbf{b}[c']]$ for some c'. Therefore, $c \bullet \mathbf{a}'[\varnothing] = c'' \bullet \mathbf{b}[c' \bullet \mathbf{a}'[\varnothing]]$, for some c''. This implies that

$$c \not\models_{\mathsf{d}} \neg(\mathbf{a}'[\mathbf{0}] \multimap\!\bullet (\mathrm{True} \bullet \mathbf{b}[\mathrm{True} \bullet \mathbf{a}'[\mathbf{0}]]))$$

and hence

$$u(i,j) \not\models_{\mathsf{d}} \neg(\mathbf{a}'[\mathbf{0}] \multimap\!\bullet (\mathrm{True} \bullet \mathbf{b}[\mathrm{True} \bullet \mathbf{a}'[\mathbf{0}]])) \bullet \mathbf{a}[\mathbf{0}]$$

for all $i, j \in \mathbb{N}$.

Define $v'(i,j)$ recursively as follows:

$$v'(i,0) = \mathbf{c}^i[\mathbf{a}'[\varnothing]] \qquad v'(i,j+1) = \mathbf{c}^i[u(i,j) \otimes v'(i,j)].$$

Now fix some $i, j \in \mathbb{N}$ and observe that there exists a $c \in \mathsf{C}_{\mathsf{Tree}}$ such that $v(i,j) = c \bullet \mathbf{a}[\varnothing]$ and $v'(i,j) = c \bullet \mathbf{a}'[\varnothing]$. By construction, the $\mathbf{a}'$ leaf in $v'(i,j)$ has no $\mathbf{b}$ ancestor, and so

$$v'(i,j) \not\models_{\mathsf{d}} \mathrm{True} \bullet \mathbf{b}[\mathrm{True} \bullet \mathbf{a}'[\mathbf{0}]].$$

Hence

$$v(i,j) \models_{\mathsf{d}} \neg(\mathbf{a}'[\mathbf{0}] \multimap\!\bullet (\mathrm{True} \bullet \mathbf{b}[\mathrm{True} \bullet \mathbf{a}'[\mathbf{0}]])) \bullet \mathbf{a}[\mathbf{0}]$$

for all $i, j \in \mathbb{N}$.

By Corollary 20, there can be no adjunct-free formula logically equivalent to $\neg(\mathbf{a}'[\mathbf{0}] \multimap (\text{True} \bullet \mathbf{b}[\text{True} \bullet \mathbf{a}'[\mathbf{0}]])) \bullet \mathbf{a}[\mathbf{0}]$. □

3.3 Adjunct Elimination for CL^c_{Tree}

Both of the counterexamples I have given to adjunct elimination for CL^s_{Tree} do not hold when context composition is added. In particular, the formula $\mathbf{0} \multimap (\text{True} \bullet \mathbf{a}[\mathbf{0}])$, is equivalent to the adjunct-free formula

$$\text{True} \circ (\mathbf{a}[\mathbf{I}] \vee (\text{True} \bullet \mathbf{a}[\mathbf{0}]) \otimes \text{true} \vee \text{true} \otimes (\text{True} \bullet \mathbf{a}[\mathbf{0}]))$$

and

$$\neg(\mathbf{a}'[\mathbf{0}] \multimap (\text{True} \bullet \mathbf{b}[\text{True} \bullet \mathbf{a}'[\mathbf{0}]])) \bullet \mathbf{a}[\mathbf{0}]$$

is equivalent to

$$(\text{True} \circ \mathbf{a}[\mathbf{0}]) \wedge (\neg(\text{True} \circ \mathbf{b}[\text{True}]) \bullet (\mathbf{a}[\mathbf{0}] \vee \mathbf{a}'[\mathbf{0}])).$$

The question that naturally arises is: does adjunct elimination hold for CL^c_{Tree}?

The trick underlying both counterexamples was that application is a powerful tool for splitting up trees that has no analogue for contexts in CL^s_{Tree}. The additional expressivity offered by $\multimap$ comes from effectively converting contexts into trees (by suitably filling their holes) which are then split using $\bullet$. Without $\multimap$, it is simply not possible to examine a context at arbitrary depth.

With context composition, on the other hand, it is possible to examine a context at arbitrary depth. However, there is a subtlety: context composition permits the splitting of an arbitrary subcontext, but not (directly) an arbitrary subtree. To see what I mean by this, consider that it is possible to express that the tree $\mathbf{a}[\mathbf{b}[\varnothing] \otimes \mathbf{b}[\mathbf{c}[\varnothing]]]$ has subtree $\mathbf{c}[\varnothing]$ in terms of application as

$$\mathbf{a}[\mathbf{b}[\varnothing] \otimes \mathbf{b}[\mathbf{c}[\varnothing]]] = \mathbf{a}[\mathbf{b}[\varnothing] \otimes \mathbf{b}[\_]] \bullet \mathbf{c}[\varnothing],$$

but that there is no directly analogous way to express that $\mathbf{c}[\varnothing]$ is a subtree of the context $\mathbf{a}[\mathbf{b}[\_] \otimes \mathbf{b}[\mathbf{c}[\varnothing]]]$. A direct analogue would require a context with two holes.

Yet even so, it is possible to express properties about arbitrary subtrees indirectly. Any subtree of a context has a lowest common ancestor with the context hole. The subcontext at this ancestor can be viewed as the horizontal concatenation of a context and a tree that contains the subtree of interest. Thus, the overall context is the composition of a context with the concatenation of

a context and the application of a context to the subtree of interest. In my example:

$$\mathbf{a}[\mathbf{b}[\_] \otimes \mathbf{b}[\mathbf{c}[\varnothing]]] = \mathbf{a}[\_] \circ (\mathbf{b}[\_] \otimes (\mathbf{b}[\_] \bullet \mathbf{c}[\varnothing])).$$

As long as we can sufficiently express the potential properties of the two-holed context in terms of the properties of the component contexts, it appears that $\multimap\!\!\bullet$ is unnecessary for describing such splittings.

However, my attempts to prove adjunct elimination using games for CL^{c}_{Tree} have been thwarted by this issue. To see why this is the case, consider the following conjecture, which could be used to establish adjunct elimination:[1]

Conjecture 21. *There exists some function $f : \mathbb{N} \to \mathbb{N}$ such that, for all ranks of the form $r = (m, 0, L)$, for all trees $t, t' \in \mathsf{Tree}$ and contexts $c, c', d, d' \in \mathsf{C}_{\mathsf{Tree}}$, if*

$$(t, t', (f(m), 0, L)) \in \mathsf{DW} \tag{3.79}$$

$$(c, c', (f(m), 0, L)) \in \mathsf{DW} \tag{3.80}$$

$$(d, d', (f(m), 0, L)) \in \mathsf{DW} \tag{3.81}$$

then

$$(c \bullet t, c' \bullet t', (m, 0, L)) \in \mathsf{DW} \tag{3.82}$$

$$(d \circ c, d' \circ c', (m, 0, L)) \in \mathsf{DW}. \tag{3.83}$$

The reason for the presence of the function f is to allow for the adjunct-free equivalent of a formula having a rank with a greater m component than the formula with adjoints that it is equivalent to. Without loss of generality, any such f can be taken to be a monotone function. (Suppose that for some m, m' with $m \leq m'$, $f(m) > f(m')$. By downward closure, the assumptions in the case of m imply the assumptions for m' and the conclusions for m' imply the conclusions for m. Thus, it would have been sufficient to take $f(m)$ to be the smaller $f(m')$.)

An inductive proof of the conjecture would establish (3.82) by considering all of $\mathsf{Spoiler}$'s moves in the game. In particular, it may be that $\mathsf{Spoiler}$ plays $\bullet$ and splits the tree $c \bullet t$ into a context c_1 and a subtree t_1 that was originally part of the context c. That is to say, for some $c_2, c_3, c_4 \in \mathsf{C}_{\mathsf{Tree}}$,

$$c = c_2 \circ (c_3 \otimes (c_4 \bullet t_1))$$

$$c_1 = c_2 \circ ((c_3 \bullet t) \otimes c_4).$$

[1] The adjunct-distinction ranking for CL^{c}_{Tree} is as in Definition 3.7 but with $\circ$ treated the same way as $\bullet$, and $\circ\!\!-^{\exists}$ and $-\!\!\circ^{\exists}$ treated the same way as $\bullet\!\!-^{\exists}$ and $-\!\!\bullet^{\exists}$.

As in the earlier example, describing t_1 as a subtree of c requires the use of three nested operators. It would be necessary to find some $c'_1 \in \mathsf{C}_{\mathsf{Tree}}$ and $t'_1 \in \mathsf{Tree}$ with $c'_1 \bullet t'_1 = c' \bullet t'$ and

$$(c_1, c'_1, (m-1, 0, L)) \in \mathsf{DW} \tag{3.84}$$

$$(t_1, t'_1, (m-1, 0, L)) \in \mathsf{DW} \tag{3.85}$$

in order show that Duplicator has a winning strategy. One would hope to do this by analyzing (3.79) and (3.80) and applying the inductive hypothesis. By considering Duplicator's winning strategies for (3.79) and (3.80), it is possible to establish that there exist $c'_1, c'_2, c'_3, c'_4 \in \mathsf{C}_{\mathsf{Tree}}$ and $t'_1 \in \mathsf{Tree}$ such that

$$c' = c'_2 \circ (c'_3 \otimes (c'_4 \bullet t'_1))$$
$$c'_1 = c'_2 \circ ((c'_3 \bullet t') \otimes c'_4)$$

and

$$(c_2, c'_2, (f(m)-1, 0, L)) \in \mathsf{DW} \tag{3.86}$$

$$(c_3, c'_3, (f(m)-2, 0, L)) \in \mathsf{DW} \tag{3.87}$$

$$(c_4, c'_4, (f(m)-3, 0, L)) \in \mathsf{DW} \tag{3.88}$$

$$(t_1, t'_1, (f(m)-3, 0, L)) \in \mathsf{DW}. \tag{3.89}$$

Assuming that $f(m-1) \leq f(m) - 1$, to establish (3.84) by the inductive hypothesis, it is necessary to show that

$$(((c_3 \bullet t) \otimes c_4), ((c'_3 \bullet t') \otimes c'_4), (f(m-1), 0, L)) \in \mathsf{DW}. \tag{3.90}$$

Showing this is probably at least as hard as showing that

$$(c_3 \bullet t, c'_3 \bullet t', (f(m-1)-1, 0, L)) \in \mathsf{DW}. \tag{3.91}$$

Showing that Duplicator has a winning strategy for this game (or one of an even larger rank) by the inductive hypothesis would most likely be a key step to establishing (3.90). This would require that

$$f(f(m-1)-1) \leq f(m) - 2.$$

Since f is monotone, this implies that

$$f(m-1) - 1 < m.$$

Together with the assumption that $f(m-1) \leq f(m)-1$ made earlier, and since m is arbitrary (only constrained that $m \geq 1$, by the fact that Spoiler played a move), for all $i > 0$,

$$f(i-1) < f(i) < i + 1.$$

This constraint means that it must be the case that $f(i) = i$, for all $i \in \mathbb{N}$. Yet in order to establish (3.85) from (3.89), it would be necessary that $f(m)-3 \geq m-1$. This gives contradictory constraints on f, and so there is no choice of f that would lead to a proof of Conjecture 21 in this fashion.

Of course, I have not conclusively ruled out the possibility that Conjecture 21 holds, but the above does show that the strategy employed in my other adjunct elimination results is a dead end. It is possible that the intuition behind the reasoning above may lead to a counterexample, although I have not been able to find one. On the other hand, there may be an alternative proof technique that I have not considered.

3.4 Adjunct Elimination for CL^m_{Tree}

The difficulty of showing adjunct elimination for context logic with composition suggests a possible direction for its solution: since the problem was that contexts could not be split, it makes sense to move to multi-holed context logic, where they can be split. I now show that the games-based technique does indeed give adjunct elimination for multi-holed context logic for trees.

The games I use for this result are based on the following ranking:

Definition 3.8 (Adjunct-Distinction Ranking for CL^m_{Tree})**.** The *adjunct-distinction ranking* consists of the bounded join-semilattice $(\mathsf{Rank}, \sqsubseteq)$ given by:

- $\mathsf{Rank} = \mathbb{N} \times \mathbb{N} \times \mathcal{P}_{\mathrm{fin}}(\Sigma) \times \mathcal{P}_{\mathrm{fin}}(\Theta)$, and
- $(m, s, L, V) \sqsubseteq (m', s', L', V')$ if and only if $m \leq m'$, $s \leq s'$, $L \subseteq L'$ and $V \subseteq V'$;

and ranking relations $\mathfrak{R}_{\circledast}$ given by:

- for $\circledast \in \{\mathbf{0}, \otimes\}$, $(r_1, \ldots, r_n)\ \mathfrak{R}_{\circledast}\ r$ if and only if $(m+1, s, L, V) \sqsubseteq r$ where $(m, s, L, V) = \bigsqcup\{r_i \mid 1 \leq i \leq n\}$,
- for $\circledast = \mathbf{a}[]$ (for some $\mathbf{a} \in \Sigma$), $((m, s, L))\ \mathfrak{R}_{\circledast}\ r$ if and only if $(m+1, s, L \cup \{\mathbf{a}\}) \sqsubseteq r$,
- for $\circledast \in \{\circ_\alpha, \exists\alpha\}$ (for some $\alpha \in \Theta$), $(r_1, \ldots, r_n)\ \mathfrak{R}_{\circledast}\ r$ if and only if $(m+1, s, L, V \cup \{\alpha\}) \sqsubseteq r$, and
- for $\circledast \in \{\bullet\!\!-^{\exists}, -\!\!\bullet^{\exists}\}$, $(r_1, r_2)\ \mathfrak{R}_{\circledast}\ r$ if and only if $(m, s+1, L, V) \sqsubseteq r$ where $(m, s, L, V) = r_1 \sqcup r_2$.

It is not difficult to verify that the above definition meets the requirements of a ranking (Definition 3.1), just as was the case with Definition 3.7.

A number of auxiliary lemmata are used in the adjunct elimination proof. The first of these establishes a certain correspondence between the holes in two contexts, given that Duplicator has a winning strategy for a game of sufficient rank. This lemma is particularly useful for ensuring that the composition of certain contexts is defined, by showing that their holes match the holes of other contexts for which composition is defined.

Lemma 22. *If*

$$((c, \sigma), (c', \sigma'), (m, s, L, V)) \in \mathsf{DW} \tag{3.92}$$

with $m \geq 2$, then, for $x = \sigma\alpha$ and $\acute{x} = \sigma'\alpha$, for some $\alpha \in V$,

$$x \in \operatorname{holes} c \quad \Longleftrightarrow \quad \acute{x} \in \operatorname{holes} c'.$$

Proof. Suppose that $x \in \operatorname{holes} c$. Spoiler could play the $\circ_\alpha$ connective on the game in (3.92), splitting $c = c \circledcirc{x} x$. Since Duplicator has a winning strategy for the game, there must exist c'_1, c'_2 such that $c' = c'_1 \circledcirc{\acute{x}} c'_2$ and

$$((x, \sigma), (c'_2, \sigma'), (m - 1, s, L, V)) \in \mathsf{DW}. \tag{3.93}$$

Since Spoiler could then play the α connective on the game in (3.93), it must be that $c'_2 = \sigma'\alpha = \acute{x}$. Therefore, $\acute{x} \in \operatorname{holes} c'$. The argument in the reverse direction is the same. □

The next lemma uses a winning strategy by Duplicator to ensure a certain structural similarity between two contexts.

Lemma 23. *Suppose that*

$$((c, \sigma), (c', \sigma'), (m, s, L, V)) \in \mathsf{DW} \tag{3.94}$$

with $m \geq 2$ and $\alpha \in V$. Then if $c = c_1 \otimes x$ for $x = \sigma\alpha$, $c_1 \in \mathsf{C}^m_{\mathsf{Tree}}$, then $c' = c'_1 \otimes \acute{x}$ for $\acute{x} = \sigma'\alpha$ and some $c'_1 \in \mathsf{C}^m_{\mathsf{Tree}}$. Similarly, if $c = x \otimes c_1$ then $c' = \acute{x} \otimes c'_1$.

Proof. Suppose that $c = c'_1 \otimes x$. Spoiler could play the $\circ_\alpha$ connective on the game in (3.94), splitting $c = c'_1 \otimes x$. Since Duplicator has a winning strategy for the game, there must exist c'_1, c'_2 such that $c' = c'_1 \otimes c'_2$ and

$$((x, \sigma), (c'_2, \sigma'), (m - 1, s, L, V)) \in \mathsf{DW}. \tag{3.95}$$

Since Spoiler could then play the α connective on the game in (3.95), it must be the case that $c'_2 = \sigma'\alpha = \acute{x}$. Thus, $c' = c'_1 \otimes \acute{x}$, as required. The proof for the other case is analogous. □

The following lemma establishes that removing a variable from the environments of the two worlds does not impede Duplicator's ability to win a game, all else being equal.

Lemma 24. *If*

$$((c, \sigma[\alpha \mapsto x]), (c', \sigma'[\alpha \mapsto \acute{x}]), r) \in \mathsf{DW} \tag{3.96}$$

then

$$((c, \sigma), (c', \sigma'), r) \in \mathsf{DW}. \tag{3.97}$$

Proof. Any winning strategy that Spoiler could have for the game in (3.97) would also be a winning strategy for the game in (3.96). Since Spoiler does not have a winning strategy for that game, he cannot have a winning strategy for the other, and so Duplicator does have a winning strategy. □

The next lemma makes it possible to reduce the case where environments contain variables that are not mentioned in the rank to the case where they do not: these variables play no role in the games.

Lemma 25. *For $\alpha \notin V$,*

$$((c, \sigma), (c', \sigma'), (m, s, L, V)) \in \mathsf{DW}$$

if and only if

$$((c, \sigma[\alpha \mapsto x]), (c', \sigma'[\alpha \mapsto \acute{x}]), (m, s, L, V)) \in \mathsf{DW}.$$

Proof. The proof is by induction on the rank. At each move of either game, α is irrelevant since $\alpha \notin V$. Therefore Duplicator's winning responses for one game will be, by the inductive hypothesis, winning responses for the other. □

The hole substitution lemma, below, establishes that substituting a hole label in one world with a fresh hole label preserves a winning strategy for Duplicator.

Lemma 26 (Hole Substitution Property for Games)**.** *Suppose that*

$$((c, \sigma), (c', \sigma'), r) \in \mathsf{DW} \tag{3.98}$$

and that $x \notin \operatorname{holes} c \cup \operatorname{range} \sigma$. Then

$$((c[x/y], \sigma[x/y]), (c', \sigma'), r) \in \mathsf{DW}. \tag{3.99}$$

Proof. The proof is by induction on the rank. If Spoiler plays a move on the game in (3.99), working with $(c[x/y], \sigma[x/y])$ then he could play the same move, but with x and y swapped, on the game in (3.98). If Duplicator plays with the same response as for that game, it gives her a winning strategy by the inductive hypothesis. On the other hand, if Spoiler plays a move on the game in (3.99), working with (c', σ') then he could play exactly the same move on the game in (3.98). If Duplicator plays with the same response as for that game, but with x and y swapped, it gives her a winning strategy by the inductive hypothesis. □

The next lemma essentially gives two sufficient conditions on Duplicator's response to Spoiler playing the $\exists\alpha$ connective in order for it to give her a winning strategy. The key part is that if Spoiler introduces a fresh hole label, Duplicator may respond by introducing *any* fresh hole label.

Lemma 27 (Interchangability of Fresh Labels)**.** *If*

$$((c, \sigma), (c', \sigma'), (m, s, L, V)) \in \mathsf{DW} \tag{3.100}$$

with $m \geq 3$ *and* $\alpha \in V$*, then*

$$((c, \sigma[\alpha \mapsto x]), (c', \sigma'[\alpha \mapsto \acute{x}]), (m-1, s, L, V)) \in \mathsf{DW} \tag{3.101}$$

if either

1. *for some* $\beta \in V$*,* $\beta \neq \alpha$*,* $x = \sigma\beta$ *and* $\acute{x} = \sigma'\beta$*; or*
2. $x \notin \operatorname{holes} c \cup \operatorname{range} \sigma$ *and* $\acute{x} \notin \operatorname{holes} c' \cup \operatorname{range} \sigma'$*.*

Proof. For the first case, assume that β, x and $\acute{x}$ satisfy the specified properties. Spoiler could play the $\exists\alpha$ connective on (3.100), assigning α the value x. Since Duplicator has a winning strategy, there exists a y such that

$$((c, \sigma[\alpha \mapsto x]), (c', \sigma'[\alpha \mapsto y]), (m-1, s, L, V)) \in \mathsf{DW}.$$

If Spoiler then plays the $\circ_\alpha$ connective, choosing to split $c = x \circledcirc c$, it must be that, for some $\bar{c}'$,

$$((x, \sigma[\alpha \mapsto x]), (\bar{c}', \sigma'[\alpha \mapsto y]), (m-2, s, L, V)) \in \mathsf{DW}.$$

Now, Spoiler could play the β connective and win unless $\bar{c}' = (\sigma'[\alpha \mapsto y])\beta = \acute{x}$ (since $x = (\sigma[\alpha \mapsto x])\beta$). Spoiler could also play the α connective and win unless $\bar{c}' = (\sigma'[\alpha \mapsto y])\alpha = y$. Consequently, $\acute{x} = y$, and so

$$((c, \sigma[\alpha \mapsto x]), (c', \sigma'[\alpha \mapsto \acute{x}]), (m-1, s, L, V)) \in \mathsf{DW},$$

as required.

For the second case, assume that x and $\acute{x}$ satisfy the specified properties. Spoiler could play the $\exists\alpha$ connective on (3.100), assigning α the value x. As before, there exists a y such that

$$((c, \sigma[\alpha \mapsto x]), (c', \sigma'[\alpha \mapsto y]), (m-1, s, L, V)) \in \mathsf{DW}.$$

By Lemma 22, since $x \notin \operatorname{holes} c$, it follows that $y \notin \operatorname{holes} c'$, and so $c'[\acute{x}/y] = c'$. If $y = \sigma'\beta$ for some $\beta \in V$ then, since $x \neq \sigma\beta$, Spoiler would have a winning strategy for the above game by playing the $\circ_\alpha$ connective (splitting $c' = y \circledy c'$) followed by the β connective (on the y part). Hence, for some $\beta_1, \ldots, \beta_n \notin V$, and some $\hat{\sigma}'$ with $y \notin \operatorname{range} \hat{\sigma}'$, $\sigma' = \hat{\sigma}'[\beta_1 \mapsto y] \cdots [\beta_n \mapsto y]$. For some $z_1, \ldots, z_n$, and some $\hat{\sigma}$, $\sigma = \hat{\sigma}[\beta_1 \mapsto z_1] \cdots [\beta_n \mapsto z_n]$. By iterated application of Lemma 25,

$$((c, \hat{\sigma}[\alpha \mapsto x]), (c', \hat{\sigma}'[\alpha \mapsto y]), (m-1, s, L, V)) \in \mathsf{DW}.$$

Since $y \notin \operatorname{range} \hat{\sigma}'$, $(\hat{\sigma}'[\alpha \mapsto y])[\acute{x}/y] = \hat{\sigma}'[\alpha \mapsto \acute{x}]$. Hence, by Lemma 26,

$$((c, \hat{\sigma}[\alpha \mapsto x]), (c', \hat{\sigma}'[\alpha \mapsto \acute{x}]), (m-1, s, L, V)) \in \mathsf{DW}.$$

By iterated application of Lemma 25, we conclude

$$((c, \sigma[\alpha \mapsto x]), (c', \sigma'[\alpha \mapsto \acute{x}]), (m-1, s, L, V)) \in \mathsf{DW},$$

as required. □

The next result, Proposition 28 is the key result for adjunct elimination. It states that, when Spoiler can play no adjunct moves, a winning strategy for Duplicator for the game on the composition of contexts follows from Duplicator's winning strategies for its components. A consequence, captured in Corollary 20, is that, if Duplicator has a winning strategy without adjunct moves, then she has a winning strategy with adjunct moves, since adjunct moves simply perform context composition.

Proposition 28 (One-step move elimination)**.** *For all ranks of the form $r = (m, 0, L, V)$, for all tree contexts $c_1, c'_1, c_2, c'_2 \in \mathsf{C}^m_{\mathsf{Tree}}$, for all domain-coincident environments $\sigma, \sigma' \in \mathsf{LEnv}$, if $|V \setminus \operatorname{dom} \sigma| \geq m$ and*

$$((c_1, \sigma), (c'_1, \sigma'), (3m, 0, L, V)) \in \mathsf{DW} \tag{3.102}$$

$$((c_2, \sigma), (c'_2, \sigma'), (3m, 0, L, V)) \in \mathsf{DW} \tag{3.103}$$

then, for all $\alpha \in V \cap \operatorname{dom} \sigma$ with $x = \sigma\alpha$, $\acute{x} = \sigma'\alpha$, if $c = c_1 \circledx c_2$ and $c' = c'_1 \circledx c'_2$ are defined then

$$((c, \sigma), (c', \sigma), r) \in \mathsf{DW}. \tag{3.104}$$

Proof. The proof is by induction on m and by cases on Spoiler's choice of move in the game of (3.104). The base case, where $m = 0$, is trivial, since Spoiler cannot make any move. In the inductive case, where $m > 0$, assume as the inductive hypothesis that the proposition holds for all lesser values of m. Assume without loss of generality that Spoiler selects the world (c, σ). Consider each connective Spoiler may choose.

0 connective. If Spoiler plays this connective then either $c = \varnothing$ and $c' \neq \varnothing$ or Duplicator wins. In the former case, given that $c = c_1 \circledS{x} c_2$, it must be that $c_1 = x$ and $c_2 = \varnothing$. By considering that Spoiler could play moves with the α and $\mathbf{0}$ connectives on the games in (3.102) and (3.103) respectively, but that Duplicator has a winning strategy for both, it must be that $c'_1 = \acute{x}$ and $c_2 = \varnothing$. Hence, $c' = \varnothing$, and so Duplicator must win after all.

β **connective.** If Spoiler plays this connective then either $c = y = \sigma\beta$ and $c' \neq \acute{y} = \sigma'\beta$ or Duplicator wins. In the former case, given that $c = c_1 \circledS{x} c_2$, one of the following must be the case:

1. $c_1 = x$ and $c_2 = y$;

2. $c'_1 = y \otimes x$ and $c_2 = \varnothing$;

3. $c'_1 = x \otimes y$ and $c_2 = \varnothing$.

In the first case, $c'_1 = \acute{x}$ and $c'_2 = \acute{y}$, for otherwise Spoiler could win the games in (3.102) and (3.103) by playing the connectives α and β respectively. Hence, $c' = \acute{y}$, and so Duplicator must win after all.

In the second case, from (3.102) there must be $\bar{c}'_1, \hat{c}'_1 \in \mathsf{C}^m_{\mathsf{Tree}}$ with $c'_1 = \bar{c}'_1 \otimes \hat{c}'_1$ and

$$((y, \sigma), (\bar{c}'_1, \sigma'), (3m - 1, 0, L, V)) \in \mathsf{DW}$$
$$((x, \sigma), (\hat{c}'_1, \sigma'), (3m - 1, 0, L, V)) \in \mathsf{DW}.$$

Hence $\bar{c}'_1 = \acute{y}$ and $\hat{c}'_1 = \acute{x}$. Also, by (3.103), $c'_2 = \varnothing$, for otherwise Spoiler could win the game by playing the connective $\mathbf{0}$. Therefore, $c' = c'_1 \circledS{\acute{x}} c'_2 = (\acute{y} \otimes \acute{x}) \circledS{\acute{x}} \varnothing = \acute{y}$, and so Duplicator must win after all.

The third case is analogous to the second.

Since Duplicator wins in each case, it follows that Duplicator has a winning strategy if Spoiler plays the β connective.

$\mathbf{a}[]$ **connective.** If Spoiler plays this connective then $c = \mathbf{a}[d]$ for some $d \in \mathsf{C}^m_{\mathsf{Tree}}$. Given that $c = c_1 \circledS{x} c_2$, one of the following must be the case:

1. $c_1 = \mathbf{a}[d_1]$ and $d = d_1 \circledS{x} c_2$;

2. $c_1 = \mathbf{a}[d] \otimes x$ and $c_2 = \varnothing$;

3. $c_1 = x \otimes \mathbf{a}[d]$ and $c_2 = \varnothing$.

In the first case, Spoiler could play the $\mathbf{a}[]$ connective on the game in (3.102), choosing context d_1. Hence, since Duplicator has a winning strategy for that game, $c'_1 = \mathbf{a}[d'_1]$ with

$$((d_1, \sigma), (d_2, \sigma), (3m - 1, 0, L, V)) \in \mathsf{DW}. \tag{3.105}$$

By downward closure and the inductive hypothesis, noting that $d'_1 \mathbin{⊘} c'_2$ is defined, since $\operatorname{holes} d'_1 = \operatorname{holes} c'_1$ and $c'_1 \mathbin{⊘} c'_2$ is defined, it follows that

$$((d_1 \mathbin{⊘} c_2, \sigma), (d'_1 \mathbin{⊘} c'_2, \sigma'), (m - 1, 0, L, V)) \in \mathsf{DW}. \tag{3.106}$$

By the definition of composition, $c' = \mathbf{a}[d']$, where $d' = d'_1 \mathbin{⊘} c'_2$. Hence, by (3.106), Duplicator has a winning strategy in this case.

In the second case, $c'_2 = \varnothing$ by (3.103). Furthermore, Spoiler could play the $\otimes$ connective on the game in (3.102), and so, since Duplicator has a winning strategy for that game, $c'_1 = d'_1 \otimes d'_2$ with

$$((\mathbf{a}[d], \sigma), (d'_1, \sigma'), (3m - 1, 0, L, V)) \in \mathsf{DW} \tag{3.107}$$

$$((x, \sigma), (d'_2, \sigma'), (3m - 1, 0, L, V)) \in \mathsf{DW}. \tag{3.108}$$

Since $m - 1 \geq 1$, it follows from (3.108) that $d'_2 = \acute{x}$. Also, Spoiler could play the $\mathbf{a}[]$ connective on the game in (3.107), so it must be that $d'_1 = \mathbf{a}[d']$ with

$$((d, \sigma), (d', \sigma'), (3m - 2, 0, L, V) \in \mathsf{DW}. \tag{3.109}$$

Now $c' = (\alpha[d'] \otimes \acute{x}) \mathbin{⊘} \varnothing = \alpha[d']$. Hence, Duplicator can respond with d' and the game continues as $((d, \sigma), (d', \sigma'), (m - 1, 0, L, V))$, giving Duplicator a winning strategy by (3.109) and downward closure.

The third case is analogous to the second.

Since Duplicator has a winning strategy in each case, it follows that Duplicator has a winning strategy if Spoiler plays the $\mathbf{a}[]$ connective.

$\otimes$ **connective.** If Spoiler plays this connective then he must make the splitting $c = d_1 \otimes d_2$ in one of the following ways:

1. Spoiler splits in c_1 to the left of the x: that is, $c_1 = d_1 \otimes d_3$ and $d_2 = d_3 \mathbin{⊘} c_2$;

2. Spoiler splits in c_1 to the right of the x: that is, $c_1 = d_3 \otimes d_2$ and $d_1 = d_3 \mathbin{⊘} c_2$;

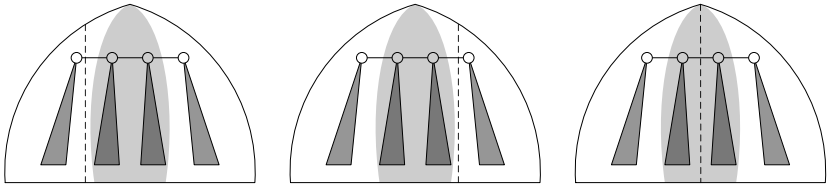


Figure 3.2: In left-to-right order, the three cases for splitting $c = d_1 \otimes d_2$

3. Spoiler splits in c_2. In order for this case to apply, the x must occur at the top level of c_1, so $c_1 = \bar{d}_3 \otimes x \otimes \bar{d}_4$, $c_2 = d_5 \otimes d_6$, $d_1 = \bar{d}_3 \otimes d_5$ and $d_2 = d_6 \otimes \bar{d}_4$.

These three cases are illustrated by Figure 3.2. The shaded area indicates the c_2 subtree and the dashed line indicates the splitting point. Note that the third case does not apply to every possible choice of c_1 and c_2, but the example shows a choice for which it does.

In the first case,

$$\begin{aligned} c_1 \mathbin{⊘} c_2 &= (d_1 \otimes d_3) \mathbin{⊘} c_2 \\ &= d_1 \otimes (d_3 \mathbin{⊘} c_2). \end{aligned}$$

Since Spoiler could play the $\otimes$ connective on the game in (3.102), it follows that $c'_1 = d'_1 \otimes d'_3$ such that

$$((d_1, \sigma), (d'_1, \sigma'), (3m-1, 0, L, V)) \in \mathsf{DW} \tag{3.110}$$

$$((d_3, \sigma), (d'_3, \sigma'), (3m-1, 0, L, V)) \in \mathsf{DW}. \tag{3.111}$$

Note that $\operatorname{holes} d'_3 \subseteq \operatorname{holes} c'_1$ and $\hat{x} \in \operatorname{holes} d'_3$ by Lemma 22 (since $x \in \operatorname{holes} d_3$), and so $d'_2 = d'_3 \mathbin{⊘} c'_2$ is defined. By downward closure on (3.111) and (3.103) and by the inductive hypothesis,

$$((d_3 \mathbin{⊘} c_2, \sigma), (d'_3 \mathbin{⊘} c'_2, \sigma'), (m-1, 0, L, V)) \in \mathsf{DW}. \tag{3.112}$$

Observe that

$$\begin{aligned} c' &= c'_1 \mathbin{⊘} c'_2 \\ &= (d'_1 \otimes d'_3) \mathbin{⊘} c'_2 \\ &= d'_1 \otimes (d'_3 \mathbin{⊘} c'_2) \\ &= d'_1 \otimes d'_2. \end{aligned}$$

Thus responding with d'_1 and d'_2 gives Duplicator a winning strategy in this case, by downward closure on (3.110) and by (3.112).

The second case is analogous to the first.

In the third case,

$$\begin{aligned} c_1 \mathbin{⊛} c_2 &= d_1 \otimes d_2 \\ &= (d_3 \mathbin{⊛} d_5) \otimes (d_4 \mathbin{⊛} d_6) \\ d_3 &= \bar{d}_3 \otimes x \\ d_4 &= x \otimes \bar{d}_5 \\ c_1 &= d_3 \mathbin{⊛} d_5 \\ &= (\bar{d}_3 \otimes x) \mathbin{⊛} (x \otimes \bar{d}_4) \\ c_2 &= d_5 \otimes d_6. \end{aligned}$$

Spoiler could play the $\circ_\alpha$ connective on the game in (3.102), so $c'_1 = d'_3 \mathbin{⊛} d'_4$ with

$$((d_3, \sigma), (d'_3, \sigma'), (3m-1, 0, L, V)) \in \mathsf{DW} \tag{3.113}$$

$$((d_4, \sigma), (d'_4, \sigma'), (3m-1, 0, L, V)) \in \mathsf{DW}. \tag{3.114}$$

Also, Spoiler could play the $\otimes$ connective on the game in (3.103), so $c'_2 = d'_5 \otimes d'_6$ with

$$((d_5, \sigma), (d'_5, \sigma'), (3m-1, 0, L, V)) \in \mathsf{DW} \tag{3.115}$$

$$((d_6, \sigma), (d'_6, \sigma'), (3m-1, 0, L, V)) \in \mathsf{DW}. \tag{3.116}$$

Since $c'_1 = d'_3 \mathbin{⊛} d'_4$ and $c'_2 = d'_5 \otimes d'_6$, it follows that $\acute{x} \in \operatorname{holes} d'_3 \subseteq \operatorname{holes} c'_1$, $\acute{x} \in \operatorname{holes} d'_4 \subseteq \operatorname{holes} c'_1$, $\operatorname{holes} d'_5 \subseteq \operatorname{holes} c'_2$ and $\operatorname{holes} d'_6 \subseteq \operatorname{holes} c'_2$. Hence, $d_1 = d'_3 \mathbin{⊛} d'_5$ and $d'_2 = d'_4 \mathbin{⊛} d'_6$ are defined. By downward closure and the inductive hypothesis on (3.113) and (3.115), and on (3.114) and (3.116),

$$((d_3 \mathbin{⊛} d_5, \sigma), (d'_3 \mathbin{⊛} d'_5, \sigma'), (m-1, 0, L, V) \in \mathsf{DW} \tag{3.117}$$

$$((d_4 \mathbin{⊛} d_6, \sigma), (d'_4 \mathbin{⊛} d'_6, \sigma'), (m-1, 0, L, V) \in \mathsf{DW}. \tag{3.118}$$

It remains to show that $c' = d'_1 \otimes d'_2$. For this to be the case, it is sufficient that $d'_3 = \bar{d}'_3 \otimes \acute{x}$ and $d'_4 = \acute{x} \otimes \bar{d}'_4$, which hold by applying Lemma 23 to (3.113) and (3.114) respectively. Thus, by structural considerations,

$$\begin{aligned} c' &= c'_1 \mathbin{⊛} c'_2 \\ &= (d'_3 \mathbin{⊛} d'_4) \mathbin{⊛} (d'_5 \otimes d'_6) \\ &= ((\bar{d}'_3 \otimes \acute{x}) \mathbin{⊛} (\acute{x} \otimes \bar{d}'_4)) \mathbin{⊛} (d'_5 \otimes d'_6) \\ &= \bar{d}'_3 \otimes d'_3 \otimes d'_6 \otimes \bar{d}'_4 \\ &= (d'_3 \mathbin{⊛} d'_5) \otimes (d'_4 \mathbin{⊛} d'_6) \\ &= d'_1 \otimes d'_2. \end{aligned}$$

Hence, by (3.117) and (3.118), Duplicator has a winning strategy if she responds by splitting c' as $d'_1 \otimes d'_2$.

Thus, Duplicator has a winning strategy whenever Spoiler plays the $\otimes$ connective.

$\circ_\beta$ **connective.** Let $y = \sigma\beta$ and $\acute{y} = \sigma'\beta$. Spoiler splits c as $d_1 \mathbin{\textcircled{y}} d_2$. Note that Spoiler cannot play this connective as the final move of a winning strategy, so it is safe to assume that $n \geq 2$. (If $n = 1$, Duplicator would have a winning strategy by responding with the splitting $c' = \acute{y} \mathbin{\textcircled{y}} c'$, for instance.)

There are four cases for how Spoiler can make the splitting $c = d_1 \mathbin{\textcircled{y}} d_2$. These are illustrated in Figures 3.3, 3.4, 3.5 and 3.6. In the diagrams, the darker subtree denotes the c_2 part of $c = c_1 \mathbin{\textcircled{x}} c_2$ and the dashed outline denotes the d_2 part of $c = d_1 \mathbin{\textcircled{y}} d_2$. The cases are:

1. Spoiler splits c within c_2 (Figure 3.3), so that $c_2 = d_3 \mathbin{\textcircled{y}} d_2$ and $d_1 = c_1 \mathbin{\textcircled{x}} d_3$.

2. Spoiler splits c outside c_2, including all of c_2 (Figure 3.4), so that $c_1 = d_1 \mathbin{\textcircled{y}} d_3$ and $d_2 = d_3 \mathbin{\textcircled{x}} c_2$.

3. Spoiler splits c so that d_2 consists of part of c_1 and part (but not all) of c_2 (Figure 3.5). Here, the part of c_1 must be a subtree adjacent to the x hole, and the part of c_2 must be a subtree at the root of c_2 and on the appropriate side.

4. Spoiler splits c so that d_2 is a subtree of c_1 that is completely disjoint from the x hole (Figure 3.6). Here, $c_1 = d_3 \mathbin{\textcircled{y}} d_2$ and $d_1 = d_3 \mathbin{\textcircled{x}} c_2$, providing $x \neq y$. I shall also consider the case when $x = y$.

Let us consider each case individually.

Case 1: Spoiler splits inside c_2, as

$$\begin{aligned} c_1 \mathbin{\textcircled{x}} d_2 &= c_1 \mathbin{\textcircled{x}} (d_3 \mathbin{\textcircled{y}} d_2) \\ &= (c_1 \mathbin{\textcircled{x}} d_3) \mathbin{\textcircled{y}} d_2 \\ &= d_1 \mathbin{\textcircled{y}} d_2 \\ c_2 &= d_3 \mathbin{\textcircled{y}} d_2 \\ d_1 &= c_1 \mathbin{\textcircled{x}} d_3. \end{aligned}$$

Note that $y \notin \operatorname{holes} c_1$, since otherwise this type of splitting is not applicable.

Spoiler would be able to play the $\circ_\beta$ connective on the game in (3.103), so

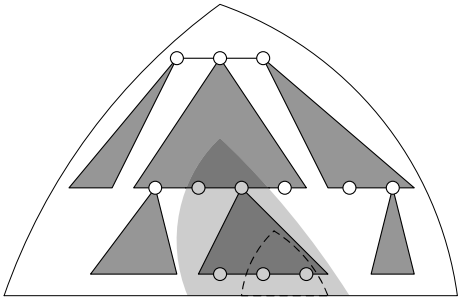


Figure 3.3: Splitting type 1

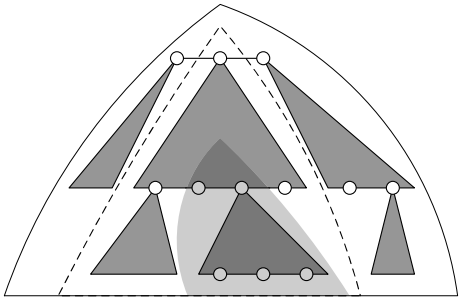


Figure 3.4: Splitting type 2

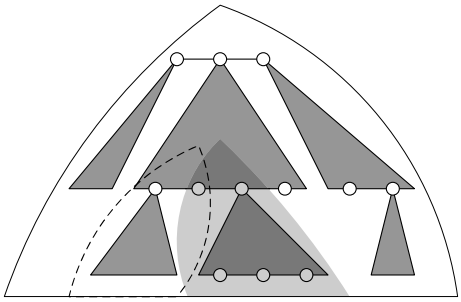


Figure 3.5: Splitting type 3

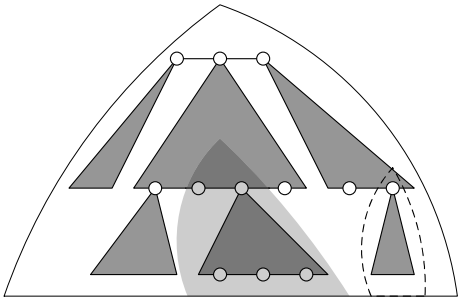


Figure 3.6: Splitting type 4

Duplicator must be able to split c'_2 as $d'_3 \mathbin{⊛}_{\acute{y}} d'_2$ such that

$$((d_3, \sigma), (d'_3, \sigma'), (3m-1, 0, L, V)) \in \mathsf{DW} \tag{3.119}$$

$$((d_2, \sigma), (d'_2, \sigma'), (3m-1, 0, L, V)) \in \mathsf{DW}. \tag{3.120}$$

Note that $\operatorname{holes} d'_3 \subseteq \operatorname{holes} c'_2 \cup \{\acute{y}\}$. Also, by Lemma 22, $\acute{y} \notin \operatorname{holes} c'_1$ since $y \notin \operatorname{holes} c_1$. Hence, $d'_1 = c'_2 \mathbin{⊛}_{\acute{x}} d'_3$ is defined. By downward closure on (3.102) and (3.119) and by the inductive hypothesis,

$$((c_1 \mathbin{⊛}_x d_3, \sigma), (c'_1 \mathbin{⊛}_{\acute{x}} d'_3, \sigma'), (m-1, 0, L, V)) \in \mathsf{DW}. \tag{3.121}$$

By structural considerations, since $\acute{y} \notin \operatorname{holes} c'_1$,

$$\begin{aligned} c'_1 \mathbin{⊛}_{\acute{x}} c'_2 &= c'_1 \mathbin{⊛}_{\acute{x}} (d'_3 \mathbin{⊛}_{\acute{y}} d'_2) \\ &= (c'_1 \mathbin{⊛}_{\acute{x}} d'_3) \mathbin{⊛}_{\acute{y}} d'_2 \\ &= d'_1 \mathbin{⊛}_{\acute{y}} d'_2. \end{aligned}$$

Hence, by (3.121) and by downward closure on (3.120), Duplicator has a winning strategy if she splits c' as $d'_1 \mathbin{⊛}_{\acute{y}} d'_2$.

Case 2: Spoiler splits outside c_2, including all of c_2 itself:

$$\begin{aligned} c_1 \mathbin{⊛}_x c_2 &= (d_1 \mathbin{⊛}_y d_3) \mathbin{⊛}_x c_2 \\ &= d_1 \mathbin{⊛}_y (d_3 \mathbin{⊛}_x c_2) \\ &= d_1 \mathbin{⊛}_y d_2 \\ c_1 &= d_1 \mathbin{⊛}_y d_3 \\ d_2 &= d_3 \mathbin{⊛}_x c_2. \end{aligned}$$

Spoiler would be able to play the $\circ_\beta$ connective on the game in (3.102), so Duplicator must be able to split c'_1 as $d'_1 \mathbin{⊛}_{\acute{x}} d'_3$ such that

$$((d_1, \sigma), (d'_1, \sigma'), (3m-1, 0, L, V)) \in \mathsf{DW} \tag{3.122}$$

$$((d_3, \sigma), (d'_3, \sigma'), (3m-1, 0, L, V)) \in \mathsf{DW}. \tag{3.123}$$

Note that $\operatorname{holes} d'_3 \subseteq \operatorname{holes} c'_1$ and that, by Lemma 22, $\acute{x} \in \operatorname{holes} d'_3$ since $x \in \operatorname{holes} d_3$. Thus $d'_2 \mathbin{⊛}_{\acute{x}} c'_2$ is defined. By downward closure on (3.123) and (3.103) and by the inductive hypothesis,

$$((d_3 \mathbin{⊛}_x c_2, \sigma), (d'_3 \mathbin{⊛}_{\acute{x}} c'_2, \sigma'), (m-1, 0, L)) \in \mathsf{DW}. \tag{3.124}$$

By structural considerations, since either $\acute{x} = \acute{y}$ or $\acute{x} \notin \operatorname{holes} d'_1$ (since $\acute{x} \in \operatorname{holes} d'_3$ and $c'_1 = d'_1 \mathbin{⊛}_{\acute{y}} d'_3$),

$$\begin{aligned} c'_1 \mathbin{⊛}_{\acute{x}} c'_2 &= (d'_1 \mathbin{⊛}_{\acute{y}} d'_3) \mathbin{⊛}_{\acute{x}} c'_2 \\ &= d'_1 \mathbin{⊛}_{\acute{y}} (d'_3 \mathbin{⊛}_{\acute{x}} c'_2) \\ &= d'_1 \mathbin{⊛}_{\acute{y}} d'_2. \end{aligned}$$

Hence, by downward closure on (3.122) and by (3.124), Duplicator has a winning strategy if she splits c' as $d'_1 \mathbin{⊛} d'_2$.

Case 3: Spoiler splits part of c_1 and part of c_2:

$$c_1 = d_3 \mathbin{⊛} d_4 \qquad\qquad c_2 = d_5 \mathbin{⊛} d_6$$

where

$$d_1 = d_3 \mathbin{⊛} d_5 \qquad\qquad d_2 = d_4 \mathbin{⊛} d_6$$

with either

$$d_4 = \bar{d}_4 \otimes x \qquad\qquad d_5 = y \otimes \bar{d}_5$$

or

$$d_4 = x \otimes \bar{d}_4 \qquad\qquad d_5 = \bar{d}_5 \otimes y.$$

Assume that $d_4 = \bar{d}_4 \otimes x$ and $d_5 = y \otimes \bar{d}_5$; the proof in the other case is directly analogous. Now,

$$\begin{aligned} c_1 \mathbin{⊛} c_2 &= (d_3 \mathbin{⊛} d_4) \mathbin{⊛} (d_5 \mathbin{⊛} d_6) \\ &= (d_3 \mathbin{⊛} (\bar{d}_4 \otimes x)) \mathbin{⊛} ((y \otimes \bar{d}_5) \mathbin{⊛} d_6) \\ &= d_3 \mathbin{⊛} (\bar{d}_4 \otimes d_6 \otimes \bar{d}_5) \\ &= (d_3 \mathbin{⊛} (y \otimes \bar{d}_5)) \mathbin{⊛} ((\bar{d}_4 \otimes x) \mathbin{⊛} d_6) \\ &= (d_3 \mathbin{⊛} d_5) \mathbin{⊛} (d_4 \mathbin{⊛} d_6) \\ &= d_1 \mathbin{⊛} d_2. \end{aligned}$$

Spoiler could play the $\circ_\alpha$ move on the game in (3.102) splitting $c_1 = d_3 \mathbin{⊛} d_4$, so $c'_1 = d'_3 \mathbin{⊛} d'_4$ such that

$$((d_3, \sigma), (d'_3, \sigma'), (3m-1, 0, L, V)) \in \mathsf{DW} \tag{3.125}$$

$$((d_4, \sigma), (d'_4, \sigma'), (3m-1, 0, L, V)) \in \mathsf{DW}. \tag{3.126}$$

Similarly, Spoiler could play $\circ_\beta$, splitting $c_2 = d_5 \mathbin{⊛} d_6$, on the game in (3.103), so $c'_2 = d'_5 \mathbin{⊛} d'_6$ such that

$$((d_5, \sigma), (d'_5, \sigma'), (3m-1, 0, L, V)) \in \mathsf{DW} \tag{3.127}$$

$$((d_6, \sigma), (d'_6, \sigma'), (3m-1, 0, L, V)) \in \mathsf{DW}. \tag{3.128}$$

Note that $\acute{x} \in \operatorname{holes} d'_3 \subset \operatorname{holes} c'_1$ and $\operatorname{holes} d'_5 \subseteq \operatorname{holes} d'_2 \cup \{\acute{y}\}$. Furthermore, either $y = x$ and so $\acute{y} = \acute{x}$ (since otherwise Spoiler could win the game in (3.102)

by playing $\circ_\alpha$ followed by β), or $y \notin d_3$ and so, by Lemma 22, $\acute{y} \notin d'_3$. Thus $d'_1 = d'_3 \mathbin{\textcircled{\scriptsize $\acute{x}$}} d'_5$ is defined. Similarly, $\acute{x} \in \operatorname{holes} d'_4$ and $\operatorname{holes} d'_6 \subseteq \operatorname{holes} c'_2$, so $d'_2 = d'_4 \mathbin{\textcircled{\scriptsize $\acute{x}$}} d'_6$ is defined. Hence, by downward closure on (3.126), (3.127), (3.126) and (3.128), and by the inductive hypothesis,

$$((d_3, \textcircled{x}\, d_5, \sigma), (d'_3 \mathbin{\textcircled{\scriptsize $\acute{x}$}} d'_5, \sigma'), (m-1, 0, L, V)) \in \mathsf{DW} \tag{3.129}$$

$$((d_4, \textcircled{x}\, d_6, \sigma), (d'_4 \mathbin{\textcircled{\scriptsize $\acute{x}$}} d'_6, \sigma'), (m-1, 0, L, V)) \in \mathsf{DW}. \tag{3.130}$$

It remains to show that $c'_1 \mathbin{\textcircled{\scriptsize $\acute{x}$}} d'_2 = d'_1 \mathbin{\textcircled{\scriptsize $\acute{y}$}} d'_2$. By structural considerations

$$\begin{aligned} c'_1 \mathbin{\textcircled{\scriptsize $\acute{x}$}} c'_2 &= (d'_3 \mathbin{\textcircled{\scriptsize $\acute{x}$}} d'_4) \mathbin{\textcircled{\scriptsize $\acute{x}$}} (d'_5 \mathbin{\textcircled{\scriptsize $\acute{y}$}} d'_6) \\ &= d'_3 \mathbin{\textcircled{\scriptsize $\acute{x}$}} (d'_4 \mathbin{\textcircled{\scriptsize $\acute{x}$}} (d'_5 \mathbin{\textcircled{\scriptsize $\acute{y}$}} d'_6)). \end{aligned}$$

Since $d_4 = \bar{d}_4 \otimes x$ and $d_5 = y \otimes \bar{d}_5$, it must be the case that $d'_4 = \bar{d}'_4 \otimes \acute{x}$ and $d'_5 = \acute{y} \otimes \bar{d}'_5$. Thus,

$$\begin{aligned} d'_4 \mathbin{\textcircled{\scriptsize $\acute{x}$}} (d'_5 \mathbin{\textcircled{\scriptsize $\acute{y}$}} d'_6) = \bar{d}'_4 \otimes d'_6 \otimes \bar{d}'_5 \\ = d'_5 \mathbin{\textcircled{\scriptsize $\acute{y}$}} (d'_4 \mathbin{\textcircled{\scriptsize $\acute{x}$}} d'_6). \end{aligned}$$

Hence,

$$\begin{aligned} c'_1 \mathbin{\textcircled{x}} c'_2 &= d'_3 \mathbin{\textcircled{\scriptsize $\acute{x}$}} (d'_5 \mathbin{\textcircled{\scriptsize $\acute{y}$}} (d'_4 \mathbin{\textcircled{\scriptsize $\acute{x}$}} d'_6)) \\ &= (d'_3 \mathbin{\textcircled{\scriptsize $\acute{x}$}} d'_5) \mathbin{\textcircled{\scriptsize $\acute{y}$}} (d'_4 \mathbin{\textcircled{\scriptsize $\acute{x}$}} d'_6) \\ &= d'_1 \mathbin{\textcircled{\scriptsize $\acute{y}$}} d'_2, \end{aligned}$$

as required. Thus, by (3.129) and (3.130), Duplicator has a winning strategy if she splits c' as $d'_1 \mathbin{\textcircled{\scriptsize $\acute{y}$}} d'_2$.

Case 4: Spoiler splits part of c_1 disjoint from c_2. There are two subcases, which I shall consider separately: (a) $y \neq x$ and (b) $y = x$.

(a) $y \neq x$:

$$\begin{aligned} c_1 \mathbin{\textcircled{x}} c_2 &= (d_3 \mathbin{\textcircled{y}} d_2) \mathbin{\textcircled{x}} c_2 \\ &= (d_3 \mathbin{\textcircled{x}} c_2) \mathbin{\textcircled{y}} d_2 \\ &= d_1 \mathbin{\textcircled{y}} d_2 \\ c_1 &= d_3 \mathbin{\textcircled{y}} d_2 \\ d_1 &= d_3 \mathbin{\textcircled{x}} c_2. \end{aligned}$$

Since $y \neq x$, it must be that $\acute{y} \neq \acute{x}$, for otherwise Spoiler would have a winning strategy for (3.102) by playing $\circ_\alpha$ followed by β. Spoiler could play the $\circ_\beta$ move on the game in (3.102), splitting $c_1 = d_3 \mathbin{\textcircled{y}} d_2$, so $c'_1 = d'_3 \mathbin{\textcircled{\scriptsize $\acute{y}$}} d'_2$ such that

$$((d_3, \sigma), (d'_3, \sigma'), (3m-1, 0, L, V)) \in \mathsf{DW} \tag{3.131}$$

$$((d_2, \sigma), (d'_2, \sigma'), (3m-1, 0, L, V)) \in \mathsf{DW}. \tag{3.132}$$

Note that $\operatorname{holes} d'_3 \subseteq \operatorname{holes} c'_1 \cup \{\acute{y}\}$. Also, by Lemma 22, $\acute{x} \in \operatorname{holes} d'_3$ and $\acute{y} \notin \operatorname{holes} c'_2$. Thus $d'_1 = d'_3 \mathbin{\textcircled{\scriptsize $\acute{x}$}} c'_2$ is defined. By downward closure on (3.131) and (3.103), and by the inductive hypothesis,

$$((d_3 \mathbin{\textcircled{\scriptsize x}} c_2, \sigma), (d'_3 \mathbin{\textcircled{\scriptsize $\acute{x}$}} c'_2, \sigma'), (m-1, 0, L, V)) \in \mathsf{DW}. \tag{3.133}$$

By structural considerations, since $\acute{x} \in \operatorname{holes} d'_3$,

$$(d'_3 \mathbin{\textcircled{\scriptsize $\acute{y}$}} d'_2) \mathbin{\textcircled{\scriptsize $\acute{x}$}} c'_2 = (d'_3 \mathbin{\textcircled{\scriptsize $\acute{x}$}} c'_2) \mathbin{\textcircled{\scriptsize y}} d'_2.$$

Hence, by (3.133) and downward closure on (3.132), Duplicator has a winning strategy if she splits c' as $d'_1 \mathbin{\textcircled{\scriptsize $\acute{y}$}} d'_2$.

(b) $y \neq x$: The reason I give this case separate consideration is that the construction for (a) would give a d_3 with two holes labelled x. This problem can be avoided by working with context $\bar{c}_1$, which is just c_1 with the x hole replaced with a z hole, where z is some sufficiently fresh hole name. Choosing some $z \notin \operatorname{holes} c_1 \cup \operatorname{holes} c_2 \cup \operatorname{range} \sigma$,

$$\begin{aligned}
c &= ((d_3 \mathbin{\textcircled{\scriptsize x}} d_2) \mathbin{\textcircled{\scriptsize z}} x) \mathbin{\textcircled{\scriptsize x}} c_2 \\
&= (d_3 \mathbin{\textcircled{\scriptsize x}} d_2) \mathbin{\textcircled{\scriptsize z}} c_2 \\
&= (d_3 \mathbin{\textcircled{\scriptsize z}} c_2) \mathbin{\textcircled{\scriptsize x}} d_2 \\
&= d_1 \mathbin{\textcircled{\scriptsize x}} d_2 \\
c_1 &= \bar{c}_1 \mathbin{\textcircled{\scriptsize z}} x \\
\bar{c}_1 &= d_3 \mathbin{\textcircled{\scriptsize x}} d_2 \\
d_1 &= d_3 \mathbin{\textcircled{\scriptsize z}} c_2.
\end{aligned}$$

Since $y = x$, it must be that $\acute{y} = \acute{x}$, for otherwise Spoiler would have a winning strategy for (3.102) by playing $\circ_\alpha$ followed by β. By Lemma 27, chosing some $\gamma \in V \setminus \operatorname{dom} \sigma$ and $\acute{z} \notin \operatorname{holes} c'_1 \cup \operatorname{holes} c'_2 \cup \operatorname{range} \sigma'$,

$$((c_1, \sigma[\gamma \mapsto z]), (c'_1, \sigma'[\gamma \mapsto \acute{z}]), (3m-1, 0, L, V)) \in \mathsf{DW} \tag{3.134}$$

$$((c_2, \sigma[\gamma \mapsto z]), (c'_2, \sigma'[\gamma \mapsto \acute{z}]), (3m-1, 0, L, V)) \in \mathsf{DW}. \tag{3.135}$$

Spoiler could play $\circ_\gamma$ on the game in (3.134), splitting c_1 as $\bar{c}_1 \mathbin{\textcircled{\scriptsize z}} x$, and so $c'_1 = \bar{c}'_1 \mathbin{\textcircled{\scriptsize $\acute{z}$}} \hat{c}'_1$ such that

$$((\bar{c}_1, \sigma[\gamma \mapsto z]), (\bar{c}'_1, \sigma'[\gamma \mapsto \acute{z}]), (3m-2, 0, L, V)) \in \mathsf{DW} \tag{3.136}$$

$$((x, \sigma[\gamma \mapsto z]), (\hat{c}'_1, \sigma'[\gamma \mapsto \acute{z}]), (3m-2, 0, L, V)) \in \mathsf{DW}. \tag{3.137}$$

Since $3m - 2 \geq 1$, (3.137) implies that $\hat{c}'_1 = \acute{x}$. Furthermore, Spoiler could play $\circ_\alpha$ on the game in (3.136), splitting $\bar{c}_1$ as $d_3 \mathbin{\textcircled{\scriptsize x}} d_2$, and so $\bar{c}'_1 = d'_3 \mathbin{\textcircled{\scriptsize $\acute{x}$}} d'_2$ such

that

$$((d_3, \sigma[\gamma \mapsto z]), (d'_3, \sigma'[\gamma \mapsto \acute{z}]), (3m-3, 0, L, V)) \in \mathsf{DW} \tag{3.138}$$

$$((d_2, \sigma[\gamma \mapsto z]), (d'_2, \sigma'[\gamma \mapsto \acute{z}]), (3m-3, 0, L, V)) \in \mathsf{DW}. \tag{3.139}$$

By construction and by Lemma 22 (recalling that $m \geq 2$),

$$\{\acute{x}, \acute{z}\} \subseteq \operatorname{holes} d'_3 \subseteq (\operatorname{holes} c' \setminus \operatorname{holes} c'_2) \cup \{\acute{x}, \acute{z}\}$$

Also, by Lemma 22 and by definition, neither $\acute{x}$ nor $\acute{z}$ occurs in c'_2. Hence $d'_1 = d'_3 \mathbin{⊘}_{\acute{z}} c'_2$ is defined. Now, by (3.138) and downward closure on (3.135), and by the inductive hypothesis,

$$((d_3 \mathbin{⊘}_z c_2, \sigma[\gamma \mapsto z]), (d'_3 \mathbin{⊘}_{\acute{z}} c'_2, \sigma[\gamma \mapsto \acute{z}]), (m-1, 0, L, V)) \in \mathsf{DW}. \tag{3.140}$$

By Lemma 24 and downward closure on (3.140) and (3.139),

$$((d_1, \sigma), (d'_1, \sigma'), (m-1, 0, L, V)) \in \mathsf{DW} \tag{3.141}$$

$$((d_2, \sigma), (d'_2, \sigma'), (m-1, 0, L, V)) \in \mathsf{DW}. \tag{3.142}$$

By construction, and by Lemma 22, $\acute{x}, \acute{z} \notin \operatorname{holes} d'_2$ and $\acute{x} \notin \operatorname{holes} c'_2$. Thus, by structural considerations,

$$\begin{aligned}
c' &= (\bar{c}'_1 \mathbin{⊘}_{\acute{z}} \acute{x}) \mathbin{⊘}_{\acute{x}} c'_2 \\
&= ((d'_3 \mathbin{⊘}_{\acute{x}} d'_2) \mathbin{⊘}_{\acute{z}} \acute{x}) \mathbin{⊘}_{\acute{x}} c'_2 \\
&= (d'_3 \mathbin{⊘}_{\acute{x}} d'_2) \mathbin{⊘}_{\acute{z}} (\acute{x} \mathbin{⊘}_{\acute{x}} c'_2) \\
&= (d'_3 \mathbin{⊘}_{\acute{x}} d'_2) \mathbin{⊘}_{\acute{z}} c'_2 \\
&= (d'_3 \mathbin{⊘}_{\acute{z}} c'_2) \mathbin{⊘}_{\acute{x}} d'_2 \\
&= d'_1 \mathbin{⊘}_{\acute{x}} d'_2.
\end{aligned}$$

Hence Duplicator has a winning strategy if she splits c' as $d'_1 \mathbin{⊘}_{\acute{y}} d'_2$.

For each possible splitting that Spoiler could make with the $\circ_\beta$ connective, Duplicator has a corresponding splitting that gives her a winning strategy. Therefore, Duplicator has a winning strategy if Spoiler plays the $\circ_\beta$ connective.

$\exists\beta$ connective. In playing this move, Spoiler chooses to instantiate β as y, say. If $m = 1$, any choice gives Duplicator a winning strategy, so assume that $m \geq 2$. Assume that $\operatorname{dom}\sigma = \operatorname{dom}\sigma' \subseteq V$. (Any other case can be reduced to this case by using Lemma 25 to add any assignments for variables not in V to (3.104) and to remove them from (3.102) and (3.103).)

There are four mutually-exclusive cases for Spoiler's choice of y:

1. $y \in \operatorname{range}\sigma$;

2. $y \in \text{holes}\, c_1$ but $y \notin \text{range}\, \sigma$;

3. $y \in \text{holes}\, c_2$ but $y \notin \text{range}\, \sigma$;

4. y is fresh, that is, $y \notin \text{holes}\, c_1 \cup \text{holes}\, c_2 \cup \text{range}\, \sigma$.

Case 1: In this case, $y = \sigma\gamma$ for some γ, and Duplicator can respond with $\acute{y} = \sigma'\gamma$. By the first case of Lemma 27,

$$((c_1, \sigma[\beta \mapsto y]), (c'_1, \sigma'[\beta \mapsto \acute{y}]), (3m - 1, 0, L, V)) \in \mathsf{DW} \tag{3.143}$$

$$((c_2, \sigma[\beta \mapsto y]), (c'_2, \sigma'[\beta \mapsto \acute{y}]), (3m - 1, 0, L, V)) \in \mathsf{DW} \tag{3.144}$$

and so, by downward closure and the inductive hypothesis,

$$((c, \sigma[\beta \mapsto y]), (c', \sigma'[\beta \mapsto \acute{y}]), (m - 1, 0, L, V)) \in \mathsf{DW}. \tag{3.145}$$

Hence, choosing $\acute{y}$ gives Duplicator a winning strategy in this case.

Case 2: Spoiler could play the $\exists\beta$ connective on the game in (3.102). Since Duplicator has a winning strategy, there is some $\acute{y}$ such that

$$((c_1, \sigma[\beta \mapsto y]), (c'_1, \sigma'[\beta \mapsto \acute{y}]), (3m - 1, 0, L, V)) \in \mathsf{DW}. \tag{3.146}$$

Since $y \notin \text{range}\, \sigma$ and $3m-2 \geq 2$, $\acute{y} \notin \text{range}\, \sigma'$. (To see this, suppose that Spoiler were to play the $\circ_\beta$ connective on the game in (3.146), splitting $c'_1 = \acute{y} \circledcirc c'_1$. Duplicator's responding choice of context must naturally contain a y hole, so if there were some γ with $\acute{y} = \sigma'\gamma$ and $y \neq \sigma\gamma$ then Spoiler would have a winning strategy by playing the γ connective.) Also, since $y \in \text{holes}\, c_1$ and $3m - 2 \geq 2$, $\acute{y} \in \text{holes}\, c'_1$ by Lemma 22. Thus $y \notin \text{holes}\, c_2 \cup \text{range}\, \sigma$ and $\acute{y} \notin \text{holes}\, c'_2 \cup \text{range}\, \sigma'$, and hence, by the second case of Lemma 27,

$$((c_2, \sigma[\beta \mapsto y]), (c'_2, \sigma'[\beta \mapsto \acute{y}]), (3m - 1, 0, L, V)) \in \mathsf{DW}. \tag{3.147}$$

So, by downward closure and the inductive hypothesis,

$$((c, \sigma[\beta \mapsto y]), (c', \sigma'[\beta \mapsto \acute{y}]), (m - 1, 0, L, V)) \in \mathsf{DW}. \tag{3.148}$$

Hence, choosing $\acute{y}$ gives Duplicator a winning strategy in this case.

Case 3: This case is essentially the same as case 2, except that Duplicator's choice of $\acute{y}$ is derived from her winning response for the game in (3.103).

Case 4: This case admits the same proof as case 2.

Since Duplicator has a winning strategy in every case, Duplicator has a winning strategy if Spoiler plays the $\exists\beta$ connective.

For each move that Spoiler could play on the game in (3.104), I have shown how Duplicator can respond to give her a winning strategy. Therefore (3.104) holds. □

Corollary 29 (Multi-step Move Elimination)**.** *For all ranks $r = (m, s, L, V)$, for all $c, c' \in \mathsf{C}^m_{\mathsf{Tree}}$, and for all domain-coincident environments $\sigma, \sigma' \in \mathsf{LEnv}$, if*

$$((c, \sigma), (c', \sigma'), (3^s(m+1), 0, L, V)) \in \mathsf{DW} \tag{3.149}$$

then

$$((c, \sigma), (c', \sigma'), (m, s, L, V)) \in \mathsf{DW}. \tag{3.150}$$

Proof. The proof is by induction on s.

If $s = 0$ then the conclusion follows by downward closure.

For $s > 0$, consider how an instance of the game in (3.150) would proceed. Until Spoiler first plays an adjunct connective, Duplicator may respond to any strategy of Spoiler's as she would for her winning strategy for the game in (3.149), preventing Spoiler from winning up to that point. Spoiler first plays an adjunct connective for, say, the $(k+1)^{\text{th}}$ move (so $k \leq m$). At this stage, for some $c_1, c_1' \in \mathsf{C}^m_{\mathsf{Tree}}$ and $\sigma_1, \sigma_1' \in \mathsf{LEnv}$, the game state is

$$((c_1, \sigma_1), (c_1', \sigma_1'), (m-k, s, L, V)) \tag{3.151}$$

and we know that

$$((c_1, \sigma_1), (c_1', \sigma_1'), (3^s(m+1) - k, 0, L, V)) \in \mathsf{DW}. \tag{3.152}$$

Spoiler now plays either the $\bullet\!\!-\!\!^{\exists}_{\alpha}$ or $-\!\!\bullet^{\exists}_{\alpha}$ connective on (3.151) for some $\alpha \in V$.

$\bullet\!\!-\!\!^{\exists}_{\alpha}$ connective. Spoiler chooses one of c_1, c_1'; assume without loss of generality that he picks c_1. Let $x = \sigma_1\alpha$ and $\acute{x} = \sigma_1'\alpha$. Spoiler also chooses $d_1, d_2 \in \mathsf{C}^m_{\mathsf{Tree}}$ with $d_2 = d_1 \mathbin{⊛} c_1$. By downward closure on (3.152),

$$((c_1, \sigma_1), (c_1', \sigma_1'), (3 \cdot 3^{s-1}(m-k+1), 0, L, V)) \in \mathsf{DW}. \tag{3.153}$$

Note that, for $\beta, \gamma \in V$, $\sigma_1\beta = \sigma_1\gamma$ if and only if $\sigma_1'\beta = \sigma_1'\gamma$. This follows from (3.153) by considering that, if $x = \sigma_1\beta = \sigma_1\gamma$, Spoiler could play the $\circ_\beta$ move and split $c_1 = x \mathbin{⊛} c_1$. Duplicator's response for her winning strategy must split $c_1' = \acute{x} \mathbin{⊛} c_1'$ with $\acute{x} = \sigma_1'\beta$ and $\acute{x} = \sigma_1'\gamma$, or else Spoiler would be able to win by playing either the β or γ connective.

Now let d_1' be d_1 with the hole labels renamed as follows: for each $\beta \in \operatorname{dom}\sigma_1 \cap V$, $\sigma_1\beta$ is renamed to $\sigma_1'\beta$; and the remaining hole labels (which are distinct from holes $c_1 \cup \sigma_1 V$ are renamed to be fresh with respect to holes $c_1' \cup \sigma_1' V$. By construction, for every $K \in \mathsf{K}^m_{\mathsf{Tree}}$ with $\operatorname{fv} K \subseteq V$, $d_1, \sigma \models_{\varsigma} K$ if and only if $d_1', \sigma' \models_{\varsigma} K$. Thus, by game completeness,

$$((d_1, \sigma_1), (d_1, \sigma_1'), (3 \cdot 3^{s-1}(m-k+1), 0, L, V)) \in \mathsf{DW} \tag{3.154}$$

$$((d_1, \sigma_1), (d_1, \sigma_1'), (m-k, s-1, L, V)) \in \mathsf{DW}. \tag{3.155}$$

Notice that $d'_2 = d'_1 \mathbin{⊛_{\hat{x}}} c'_1$ is defined by construction. By Proposition 28, from (3.153) and (3.154),

$$((d_1 \mathbin{⊛_x} c_1, \sigma_1), (d'_1 \mathbin{⊛_{\hat{x}}} c'_1, \sigma'_1), (3^{s-1}(m-k+1), 0, L, V)) \in \mathsf{DW}. \tag{3.156}$$

Hence, by the inductive hypothesis,

$$((d_1 \mathbin{⊛_x} c_1, \sigma_1), (d'_1 \mathbin{⊛_{\hat{x}}} c'_1, \sigma'_1), (m-k, s-1, L, V)) \in \mathsf{DW}. \tag{3.157}$$

From (3.155) and (3.157), Duplicator has a winning strategy by responding with d'_1 and d'_2.

$\multimap^{\exists}_{\alpha}$ **connective.** Again, without loss of generality assume that Spoiler chooses c_1, and let $x = \sigma_1\alpha$ and $\hat{x} = \sigma'_1\alpha$. Spoiler also chooses $d_1, d_2 \in \mathsf{C}^m_{\mathsf{Tree}}$ with $d_2 = c_1 \mathbin{⊛_x} d_1$. As before,

$$((c_1, \sigma_1), (c'_1, \sigma'_1), (3 \cdot 3^{s-1}(m-k+1), 0, L, V)) \in \mathsf{DW}. \tag{3.158}$$

Let d'_1 be the relabelling of d_1 as in the previous case. By construction and by Lemma 22 (using (3.153), since $3^s(m+1) - k \geq 3$), $d'_2 = c'_1 \mathbin{⊛_{\hat{x}}} d'_1$ is defined. Also, by game completeness,

$$((d_1, \sigma_1), (d_1, \sigma'_1), (3 \cdot 3^{s-1}(m-k+1), 0, L, V)) \in \mathsf{DW} \tag{3.159}$$

$$((d_1, \sigma_1), (d_1, \sigma'_1), (m-k, s-1, L, V)) \in \mathsf{DW}. \tag{3.160}$$

Thus, by Proposition 28,

$$((c_1 \mathbin{⊛_x} d_2, \sigma_1), (c'_1 \mathbin{⊛_{\hat{x}}} d'_1, \sigma'_1), (3^{s-1}(m-k+1), 0, L, V)) \in \mathsf{DW}. \tag{3.161}$$

Hence, by the inductive hypothesis,

$$((c_1 \mathbin{⊛_x} d_2, \sigma_1), (c'_1 \mathbin{⊛_{\hat{x}}} d'_1, \sigma'_1), (m-k, s-1, L, V)) \in \mathsf{DW}. \tag{3.162}$$

So, by (3.160) and (3.162), Duplicator has a winning strategy in this case also, by playing d'_1 and d'_2. □

The following theorem translates this result on games into a result on logical formulae, finally establishing that adjunct elimination holds for CL^m_{Tree}.

Theorem 30 (Adjunct Elimination)**.** *For any sort-ς formula of rank $r = (m, s, L, V)$, there exists an equivalent formula of rank $r' = (3^s(m+1), 0, L, V)$.*

Proof. Suppose that K is a sort-ς formula of rank r. Let

$$W = \left\{ w \in \mathsf{World}_\varsigma \;\middle|\; w \models_\varsigma K \right\}.$$

By game soundness, if $w \in W$ and $w' \notin W$ then $(w, w', r) \in \mathsf{SW}$. By Corollary 20, this means that $(w, w', r') \in \mathsf{SW}$. Hence, by game completeness, there is a formula $K_{w,w'}$ of rank r' which discriminates between w and w'.

Therefore, by Corollary 9, there is a sort-ς formula K' of rank r' such that, for $w \in \mathsf{World}_\varsigma$, $w \models_\varsigma K'$ if and only if $w \in W$. Hence, K' is equivalent to K. □

3.5 Quantifier Normalisation for CL^m

In this section, I present results that allow a multi-holed context logic formula to be rewritten in such a way that all quantifiers are $\mathsf{И}$ and appear at the head of the formula. (The one caveat is that $\mapsto$ is introduced as a subformula, which, though technically defined to be $\exists\alpha.\,\alpha$, I treat here as being primitive.) The eventual benefit of this normalisation procedure is that formulae with deeply-nested quantifiers typically present a barrier to decidability, while decidability is often more readily attainable for formulae with restricted forms of quantification.

For these results, I assume the modal presentation of CL^m, and that environment neutrality (Property 2.43) and hole neutrality (Property 2.44) hold for all model-specific connectives.

Lemma 31 (Encoding Existential with Freshness)**.** *For all $K \in \mathsf{K}^m$,*

$$\exists\alpha.\,K \equiv \mathsf{И}\alpha.\,K \circ_\alpha \left(\mapsto \wedge \neg \bigvee_{\beta \in (\mathrm{fv}\,K)\setminus\{\alpha\}} \beta \right) \vee K \vee \bigvee_{\beta \in (\mathrm{fv}\,K)\setminus\{\alpha\}} K[\beta/\alpha].$$

Consequently, every formula can be rewritten to an equivalent formula that contains no existential quantifiers.

Proof. Fix some $K \in \mathsf{K}^m$ with $(\exists\alpha.\,K) \in \mathsf{Formula}_\varsigma$, for some $\varsigma \in \mathsf{Sort}$. Let

$$K' = K \circ_\alpha \left(\mapsto \wedge \neg \bigvee_{\beta \in (\mathrm{fv}\,K)\setminus\{\alpha\}} \beta \right) \vee K \vee \bigvee_{\beta \in (\mathrm{fv}\,K)\setminus\{\alpha\}} K[\beta/\alpha].$$

Fix some $(c, \sigma) \in \mathsf{World}_\varsigma$. It suffices to show that

$$c, \sigma \models_\varsigma \exists\alpha.\,K \quad \iff \quad c, \sigma \models_\varsigma \mathsf{И}\alpha.\,K'. \tag{3.163}$$

By environment extendability (Property 2.33), assume without loss of generality that $\varsigma = \mathsf{c}(\mathrm{fv}\,K \setminus \{\alpha\})$.

$\Longrightarrow$:

Suppose that

$$c, \sigma \models_\varsigma \exists\alpha.\,K$$

and hence

$$\text{there exists } x \text{ s.t. } c, \sigma[\alpha \mapsto x] \models_{\mathsf{c}(\mathrm{fv}\, K)} K.$$

Consider the possible cases for x.

If $x \sharp c, \sigma$ then $c, \sigma \models_{\varsigma} \mathsf{И}\alpha.\, K$ and so $c, \sigma \models_{\varsigma} \mathsf{И}\alpha.\, K'$.

If $x \in \operatorname{range} \sigma$ (and so $x\sigma\beta$ for some β) then $c, \sigma \models_{\mathsf{c}(\mathrm{fv}\, K)} K[\beta/\alpha]$ (by induction of the structure of K). Hence $c, \sigma \models_{\varsigma} \mathsf{И}\alpha.\, K'$.

If $x \sharp \sigma$ but $x \in \operatorname{holes} c$ then for any $y \sharp \sigma, c$, and for some $i_x \in \mathbf{I}_x$,

$$c = c[y/x] \mathbin{ⓘ}_x$$
$$c[y/x], \sigma[\alpha \mapsto y] \models_{\mathsf{c}(\mathrm{fv}\, K)} K$$
$$x, \sigma[\alpha \mapsto y] \models_{\mathsf{c}(\mathrm{fv}\, K)} \mapsto \wedge \neg \bigvee_{\beta \in (\mathrm{fv}\, K) \setminus \{\alpha\}} \beta$$

and so

$$c, \sigma[\alpha \mapsto y] \models_{\mathsf{c}(\mathrm{fv}\, K)} K \circ_\alpha \left(\mapsto \wedge \neg \bigvee_{\beta \in (\mathrm{fv}\, K) \setminus \{\alpha\}} \beta \right)$$
$$\therefore \quad c, \sigma[\alpha \mapsto y] \models_{\mathsf{c}(\mathrm{fv}\, K)} K'$$
$$\therefore \quad c, \sigma \models_{\varsigma} \mathsf{И}\alpha.\, K'.$$

These three cases cover all possible choices of x, and hence $c, \sigma \models_{\varsigma} \mathsf{И}\alpha.\, K'$, as required.

$\Longleftarrow$:

Suppose that

$$c, \sigma \models_{\varsigma} \mathsf{И}\alpha.\, K'$$

and hence

$$\text{there exists } x \text{ s.t. } x \sharp c, \sigma \text{ and } c, \sigma[\alpha \mapsto x] \models_{\mathsf{c}(\mathrm{fv}\, K)} K'.$$

One of the disjuncts of K' must be satisfied; consider the possible cases.

If

$$c, \sigma[\alpha \mapsto y] \models_{\mathsf{c}(\mathrm{fv}\, K)} K \circ_\alpha \left(\mapsto \wedge \neg \bigvee_{\beta \in (\mathrm{fv}\, K) \setminus \{\alpha\}} \beta \right)$$

then there exists c', y with $y \sharp \sigma$ and $i_y \in \mathbf{I}_y$ such that

$$c = c' \mathbin{ⓧ} i_y$$
$$c', \sigma[\alpha \mapsto x] \models_{\mathsf{c}(\mathrm{fv}\, K)} K.$$

By hole substitution (Property 2.34), it follows that

$$c, \sigma[\alpha \mapsto y] \models_{\mathsf{c}(\mathrm{fv}\, K)} K,$$

and hence $c, \sigma \models_\varsigma \exists\alpha.\, K$.

If $c, \sigma[\alpha \mapsto x] \models_{\mathsf{c}(\mathrm{fv}\, K)} K$ then by definition $c, \sigma \models_\varsigma \exists\alpha.\, K$.

If $c, \sigma[\alpha \mapsto x] \models_{\mathsf{c}(\mathrm{fv}\, K)} K[\beta/\alpha]$ where $\sigma\beta = y$, say, then $c, \sigma[\alpha \mapsto y] \models_{\mathsf{c}(\mathrm{fv}\, K)} K$ (by induction on the structure of K). Hence, $c, \sigma \models_\varsigma \exists\alpha.\, K$. □

The above lemma shows that we can replace existential quantification with freshness quantification, but at the cost of enlarging the formula. A valid question is: can we do better than this? For specific choices of K, the answer is certainly 'yes', however, in the general case, the proof tends to justify the size of K'.

In particular, for the '$\Longrightarrow$' direction in the proof, each of the three cases for x requires a different disjunct — if any of them is left out, it is not difficult to conceive a counterexample: $K = \alpha$ requires the first disjunct (when $\exists\alpha.\, \alpha$ is satisfied, α is bound to a hole label that occurs in the context, but not necessarily in the environment); $K = \mathrm{True} \circ_\alpha \mathbf{a}[\mathbf{0}]$ requires the second disjunct (in this case, α gets bound to something that is fresh with respect to the context, and, assuming the environment is empty, also the environment); and $K = \alpha \wedge \beta$ requires the third disjunct (here, α gets bound to the same thing as β).

Less obvious is the need for the subformula

$$\neg \bigvee_{\beta \in (\mathrm{fv}\, K) \setminus \{\alpha\}} \beta$$

in the first disjunct of K'. In the '$\Longleftarrow$' direction of the proof, this subformula guarantees that the hole label y is fresh with respect to σ, and so the hole substitution property can be applied. If the subformula is omitted, the formula $K = \alpha \wedge \neg\beta$ is a counterexample to the desired equivalence:

$$x, [\beta \mapsto x] \not\models_\varsigma \exists\alpha.\, \alpha \wedge \neg\beta$$

but

$$x, [\beta \mapsto x] \models_\varsigma \mathsf{И}\alpha.\, (\alpha \wedge \neg\beta) \circ_\alpha \mapsto.$$

The next result shows that the freshness quantifiers in a formula can be brought to the outside; that is, formulae with only freshness quantification can be re-written as equivalent formulae in *prenex normal form*. Conforti and Ghelli take a similar approach in [CG04] to prove that ambient logic with freshness quantification is decidable.

Lemma 32 (Prenex Normalisation)**.** *The following logical equivalences hold, where α is assumed not to occur free on either side.*

For each model-specific connective $\circledast$ having arity n and each $1 \leq i \leq n$,

$$\circledast(K_1, \ldots, K_{i-1}, \mathsf{И}\alpha.\, K_i, K_{i+1}, \ldots, K_n) \equiv \mathsf{И}\alpha.\, \circledast(K_1, \ldots, K_n) \tag{3.164}$$

$$K_1 \circ_\beta (\mathsf{И}\alpha.\, K_2) \equiv \mathsf{И}\alpha.\, K_1 \circ_\beta K_2 \tag{3.165}$$

$$(\mathsf{И}\alpha.\, K_1) \circ_\beta K_2 \equiv \mathsf{И}\alpha.\, K_1 \circ_\beta K_2 \tag{3.166}$$

$$K_1 \multimap^{\exists}_\beta (\mathsf{И}\alpha.\, K_2) \equiv \mathsf{И}\alpha.\, K_1 \multimap^{\exists}_\beta (K_2 \wedge \textcircled{\scriptsize \#}) \tag{3.167}$$

$$(\mathsf{И}\alpha.\, K_1) \multimap^{\exists}_\beta K_2 \equiv \mathsf{И}\alpha.\, (K_1 \wedge \textcircled{\scriptsize \#}) \multimap^{\exists}_\beta K_2 \tag{3.168}$$

$$K_1 \circ\!\!-^{\exists}_\beta (\mathsf{И}\alpha.\, K_2) \equiv \mathsf{И}\alpha.\, K_1 \circ\!\!-^{\exists}_\beta (K_2 \wedge \textcircled{\scriptsize \#}) \tag{3.169}$$

$$(\mathsf{И}\alpha.\, K_1) \circ\!\!-^{\exists}_\beta K_2 \equiv \mathsf{И}\alpha.\, (K_1 \wedge \textcircled{\scriptsize \#}) \circ\!\!-^{\exists}_\beta K_2 \tag{3.170}$$

$$K_1 \to (\mathsf{И}\alpha.\, K_2) \equiv \mathsf{И}\alpha.\, K_1 \to K_2 \tag{3.171}$$

$$(\mathsf{И}\alpha.\, K_1) \to K_2 \equiv \mathsf{И}\alpha.\, K_1 \to K_2. \tag{3.172}$$

Consequently, every $\exists$-free formula can be rewritten to give an equivalent formula in which all quantifiers appear at the head of the formula — the prenex normal form.

Proof. Fix $(c, \sigma) \in \mathsf{World}_\varsigma$ for some appropriate choice of ς. Let $\varsigma' = \mathsf{c}(\phi \cup \{\alpha\})$ where $\varsigma = \mathsf{c}\phi$.

Equivalence (3.164):

$$
\begin{aligned}
& c, \sigma \models_\varsigma \circledast(K_1, \ldots, K_{i-1}, \mathsf{И}\alpha.\, K_i, K_{i+1}, \ldots, K_n) \\
\iff\ & \text{there exist } c_1, \ldots, c_n \text{ s.t.} \\
& \quad (c, \sigma)\ \mathfrak{M}_\circledast\ ((c_1, \sigma), \ldots, (c_n, \sigma)) \text{ and} \\
& \quad \text{for } j \neq i,\ c_j, \sigma \models_\varsigma K_j \text{ and} \\
& \quad c_i, \sigma \models_\varsigma \mathsf{И}\alpha.\, K_i \\
\iff\ & \text{there exist } c_1, \ldots, c_n \text{ s.t.} \\
& \quad (c, \sigma)\ \mathfrak{M}_\circledast\ ((c_1, \sigma), \ldots, (c_n, \sigma)) \text{ and} \\
& \quad \text{for } j \neq i,\ c_j, \sigma \models_\varsigma K_j \text{ and} \\
& \quad \text{there exists } x \text{ s.t. } x \sharp c_i, \sigma \text{ and } c_i, \sigma[\alpha \mapsto x] \models_{\varsigma'} K_i \\
& \text{(by hole substitution)} \\
\iff\ & \text{there exist } c_1, \ldots, c_n \text{ s.t.} \\
& \quad \text{there exists } z \text{ s.t. } z \sharp c, \sigma, c_1, \ldots, c_n \text{ and} \\
& \quad (c, \sigma)\ \mathfrak{M}_\circledast\ ((c_1, \sigma), \ldots, (c_n, \sigma)) \text{ and} \\
& \quad \text{for } j \neq i,\ c_j, \sigma \models_\varsigma K_j \text{ and} \\
& \quad c_i, \sigma[\alpha \mapsto z] \models_{\varsigma'} K_i \\
& \text{(by environment extensibility)} \\
\iff\ & \text{there exist } c_1, \ldots, c_n \text{ s.t.} \\
& \quad \text{there exists } z \text{ s.t. } z \sharp c, \sigma, c_1, \ldots, c_n \text{ and} \\
& \quad (c, \sigma)\ \mathfrak{M}_\circledast\ ((c_1, \sigma), \ldots, (c_n, \sigma)) \text{ and} \\
& \quad \text{for } 1 \leq j \leq n,\ c_j, \sigma[\alpha \mapsto z] \models_{\varsigma'} K_j \\
& \text{(by hole substitution and hole neutrality)} \\
\iff\ & \text{there exists } y \text{ s.t. } y \sharp c, \sigma \text{ and} \\
& \quad \text{there exist } c_1, \ldots, c_n \text{ s.t.} \\
& \quad (c, \sigma)\ \mathfrak{M}_\circledast\ ((c_1, \sigma), \ldots, (c_n, \sigma)) \text{ and} \\
& \quad \text{for } 1 \leq j \leq n,\ c_j, \sigma[\alpha \mapsto y] \models_{\varsigma'} K_j \\
& \text{(by environment neutrality)} \\
\iff\ & \text{there exists } y \text{ s.t. } y \sharp c, \sigma \text{ and} \\
& \quad \text{there exist } c_1, \ldots, c_n \text{ s.t.} \\
& \quad (c, \sigma[\alpha \mapsto y])\ \mathfrak{M}_\circledast\ ((c_1, \sigma[\alpha \mapsto y]), \ldots, (c_n, \sigma[\alpha \mapsto y])) \text{ and} \\
& \quad \text{for } 1 \leq j \leq n,\ c_j, \sigma[\alpha \mapsto y] \models_{\varsigma'} K_j \\
\iff\ & c, \sigma \models_\varsigma \mathsf{И}\alpha.\, \circledast(K_1, \ldots, K_n).
\end{aligned}
$$

Equivalence (3.165): Let $y = \sigma\beta$.

$$
\begin{array}{ll}
 & c, \sigma \models_\varsigma K_1 \circ_\beta (\mathsf{И}\alpha.\, K_2) \\
\iff & \text{there exist } c_1, c_2 \text{ s.t. } c = c_1 \mathbin{\textcircled{y}} c_2 \text{ and} \\
 & \quad c_1, \sigma \models_\varsigma K_1 \text{ and} \\
 & \quad \text{there exists } x \text{ s.t. } x \sharp c_2, \sigma \text{ and } c_2, \sigma[\alpha \mapsto x] \models_{\varsigma'} K_2 \\
 & \text{(by hole substitution and environment extensibility)} \\
\iff & \text{there exists } z \text{ s.t. } z \sharp c, \sigma \text{ and} \\
 & \quad \text{there exist } c_1, c_2 \text{ s.t. } c = c_1 \mathbin{\textcircled{y}} c_2 \text{ and} \\
 & \quad c_1, \sigma[\alpha \mapsto z] \models_{\varsigma'} K_1 \text{ and } c_2, \sigma[\alpha \mapsto z] \models_{\varsigma'} K_2 \\
\iff & c, \sigma \models_\varsigma \mathsf{И}\alpha.\, K_1 \circ_\beta K_2.
\end{array}
$$

Equivalence (3.166): Let $y = \sigma\beta$.

$$
\begin{array}{ll}
 & c, \sigma \models_\varsigma (\mathsf{И}\alpha.\, K_1) \circ_\beta K_2 \\
\iff & \text{there exist } c_1, c_2 \text{ s.t. } c = c_1 \mathbin{\textcircled{y}} c_2 \text{ and} \\
 & \quad \text{there exists } x \text{ s.t. } x \sharp c_1, \sigma \text{ and } c_1, \sigma \models_{\varsigma'} K_1 \text{ and} \\
 & \quad c_2, \sigma \models_\varsigma K_2 \\
\iff & \text{there exists } z \text{ s.t. } z \sharp c, \sigma \text{ and} \\
 & \quad \text{there exist } c_1, c_2 \text{ s.t. } c = c_1 \mathbin{\textcircled{y}} c_2 \text{ and} \\
 & \quad c_1, \sigma[\alpha \mapsto z] \models_{\varsigma'} K_1 \text{ and } c_2, \sigma[\alpha \mapsto z] \models_{\varsigma'} K_2 \\
\iff & c, \sigma \models_\varsigma \mathsf{И}\alpha.\, K_1 \circ_\beta K_2.
\end{array}
$$

Equivalence (3.167): Let $y = \sigma\beta$.

$$
\begin{array}{ll}
 & c, \sigma \models_\varsigma K_1 \multimap^{\exists}_\beta (\mathsf{И}\alpha.\, K_2) \\
\iff & \text{there exist } c_1, c_2 \text{ s.t. } c_2 = c \mathbin{\textcircled{x}} c_1 \text{ and} \\
 & \quad c_1, \sigma \models_\varsigma K_1 \text{ and} \\
 & \quad \text{there exists } x \text{ s.t. } x \sharp c_2, \sigma \text{ and } c_2, \sigma[\alpha \mapsto x] \models_{\varsigma'} K_2 \\
\iff & \text{there exist } c_1, c_2 \text{ s.t. } c_2 = c \mathbin{\textcircled{x}} c_1 \text{ and} \\
 & \quad c_1, \sigma \models_\varsigma K_1 \text{ and} \\
 & \quad \text{there exists } x \text{ s.t. } x \sharp \sigma \text{ and } c_2, \sigma[\alpha \mapsto x] \models_{\varsigma'} K_2 \wedge \circledcirc \\
\iff & \text{there exists } z \text{ s.t. } z \sharp c, \sigma \text{ and} \\
 & \quad \text{there exist } c_1, c_2 \text{ s.t. } c_2 = c \mathbin{\textcircled{x}} c_1 \text{ and} \\
 & \quad c_1, \sigma[\alpha \mapsto z] \models_{\varsigma'} K_1 \text{ and } c_2, \sigma[\alpha \mapsto z] \models_{\varsigma'} K_2 \wedge \circledcirc \\
\iff & c, \sigma \models_\varsigma \mathsf{И}\alpha.\, K_1 \multimap^{\exists}_\beta (K_2 \wedge \circledcirc).
\end{array}
$$

Equivalence (3.168): Let $y = \sigma\beta$.

$$
\begin{aligned}
& c, \sigma \models_\varsigma (\mathsf{И}\alpha.\, K_1) \multimap^{\exists}_{\beta} K_2 \\
\iff\ & \text{there exist } c_1, c_2 \text{ s.t. } c_2 = c \circledast_x c_1 \text{ and} \\
& \quad \text{there exists } x \text{ s.t. } x \sharp c_1, \sigma \text{ and } c_1, \sigma[\alpha \mapsto x] \models_{\varsigma'} K_1 \text{ and} \\
& \quad c_2, \sigma \models_\varsigma K_2 \\
\iff\ & \text{there exist } c_1, c_2 \text{ s.t. } c_2 = c \circledast_x c_1 \text{ and} \\
& \quad \text{there exists } x \text{ s.t. } x \sharp c_1, \sigma \text{ and } c_1, \sigma[\alpha \mapsto x] \models_{\varsigma'} K_1 \wedge \circledcirc \text{ and} \\
& \quad c_2, \sigma \models_\varsigma K_2 \\
\iff\ & \text{there exists } z \text{ s.t. } z \sharp c, \sigma \text{ and} \\
& \quad \text{there exist } c_1, c_2 \text{ s.t. } c_2 = c \circledast_x c_1 \text{ and} \\
& \quad c_1, \sigma[\alpha \mapsto z] \models_{\varsigma'} K_1 \wedge \circledcirc \text{ and } c_2, \sigma[\alpha \mapsto z] \models_{\varsigma'} K_2 \\
\iff\ & c, \sigma \models_\varsigma \mathsf{И}\alpha.\, (K_1 \wedge \circledcirc) \multimap^{\exists}_{\beta} K_2.
\end{aligned}
$$

Equivalence (3.169): Let $y = \sigma\beta$.

$$
\begin{aligned}
& c, \sigma \models_\varsigma K_1 \circ\!\!-^{\exists}_{\beta} (\mathsf{И}\alpha.\, K_2) \\
\iff\ & \text{there exist } c_1, c_2 \text{ s.t. } c_2 = c_1 \circledast_x c \text{ and} \\
& \quad c_1, \sigma \models_\varsigma K_1 \text{ and} \\
& \quad \text{there exists } x \text{ s.t. } x \sharp c_2, \sigma \text{ and } c_2, \sigma[\alpha \mapsto x] \models_{\varsigma'} K_2 \\
\iff\ & \text{there exist } c_1, c_2 \text{ s.t. } c_2 = c_1 \circledast_x c \text{ and} \\
& \quad c_1, \sigma \models_\varsigma K_1 \text{ and} \\
& \quad \text{there exists } x \text{ s.t. } x \sharp \sigma \text{ and } c_2, \sigma[\alpha \mapsto x] \models_{\varsigma'} K_2 \wedge \circledcirc \\
\iff\ & \text{there exists } z \text{ s.t. } z \sharp c, \sigma \text{ and} \\
& \quad \text{there exist } c_1, c_2 \text{ s.t. } c_2 = c_1 \circledast_x c \text{ and} \\
& \quad c_1, \sigma[\alpha \mapsto z] \models_{\varsigma'} K_1 \text{ and } c_2, \sigma[\alpha \mapsto z] \models_{\varsigma'} K_2 \wedge \circledcirc \\
\iff\ & c, \sigma \models_\varsigma \mathsf{И}\alpha.\, K_1 \circ\!\!-^{\exists}_{\beta} (K_2 \wedge \circledcirc).
\end{aligned}
$$

Equivalence (3.170): Let $y = \sigma\beta$.

$$
\begin{aligned}
& c, \sigma \models_\varsigma (\mathsf{И}\alpha.\, K_1) \circ\!\!-\!\!\!\!\!\!^{\exists}_{\beta}\, K_2 \\
\iff\ & \text{there exist } c_1, c_2 \text{ s.t. } c_2 = c_1 \textcircled{x} c \text{ and} \\
& \quad \text{there exists } x \text{ s.t. } x \sharp c_1, \sigma \text{ and } c_1, \sigma[\alpha \mapsto x] \models_{\varsigma'} K_1 \text{ and} \\
& \quad c_2, \sigma \models_\varsigma K_2 \\
\iff\ & \text{there exist } c_1, c_2 \text{ s.t. } c_2 = c_1 \textcircled{x} c \text{ and} \\
& \quad \text{there exists } x \text{ s.t. } x \sharp c_1, \sigma \text{ and } c_1, \sigma[\alpha \mapsto x] \models_{\varsigma'} K_1 \wedge \textcircled{\scriptsize /} \text{ and} \\
& \quad c_2, \sigma \models_\varsigma K_2 \\
\iff\ & \text{there exists } z \text{ s.t. } z \sharp c, \sigma \text{ and} \\
& \quad \text{there exist } c_1, c_2 \text{ s.t. } c_2 = c_1 \textcircled{x} c \text{ and} \\
& \quad c_1, \sigma[\alpha \mapsto z] \models_{\varsigma'} K_1 \wedge \textcircled{\scriptsize /} \text{ and} \\
& \quad c_2, \sigma[\alpha \mapsto z] \models_{\varsigma'} K_2 \\
\iff\ & c, \sigma \models_\varsigma \mathsf{И}\alpha.\, (K_1 \wedge \textcircled{\scriptsize /}) \circ\!\!-\!\!\!\!\!\!^{\exists}_{\beta}\, K_2.
\end{aligned}
$$

Equivalence (3.171):

$$
\begin{aligned}
& c, \sigma \models_\varsigma K_1 \to (\mathsf{И}\alpha.\, K_2) \\
\iff\ & c, \sigma \models_\varsigma K_1 \implies \text{there exists } x \text{ s.t. } x \sharp c, \sigma \text{ and } c, \sigma[\alpha \mapsto x] \models_{\varsigma'} K_2 \\
\iff\ & \text{there exists } x \text{ s.t. } x \sharp c, \sigma \text{ and} \\
& \quad c, \sigma[\alpha \mapsto x] \models_{\varsigma'} K_1 \implies c, \sigma[\alpha \mapsto x] \models_{\varsigma'} K_2 \\
\iff\ & c, \sigma \models_\varsigma \mathsf{И}\alpha.\, K_1 \to K_2.
\end{aligned}
$$

Equivalence (3.172):

$$
\begin{aligned}
& c, \sigma \models_\varsigma (\mathsf{И}\alpha.\, K_1) \to K_2 \\
\iff\ & \big(\text{there exists } x \text{ s.t. } x \sharp c, \sigma \text{ and } c, \sigma[\alpha \mapsto x] \models_{\varsigma'} K_1\big) \\
& \quad \implies c, \sigma \models_\varsigma K_2 \\
\iff\ & \text{for all } x,\ x \sharp c, \sigma \implies \\
& \quad c, \sigma[\alpha \mapsto x] \models_{\varsigma'} K_1 \implies c, \sigma[\alpha \mapsto x] \models_{\varsigma'} K_2 \\
\iff\ & c, \sigma \models_\varsigma \mathsf{И}\alpha.\, K_1 \to K_2.
\end{aligned}
$$

□

The above lemmata lead to the following proposition, which sums up quantifier normalisation for multi-holed context logic formulae.

Proposition 33 (Quantifier Normalisation)**.** *Every formula of CL^m can be rewritten to an equivalent prenex normal form that uses only $\mathsf{И}$ quantification. This rewriting is effective.*

Proof. The formula is first rewritten to only include fresh quantification by applying the equivalence of Lemma 31 to each existentially quantified subformula exhaustively. The equivalence rules of Lemma 32 are then applied exhaustively to the result, leading to an equivalent formula in prenex normal form with only $\forall$ quantification. It is not difficult to see that these strategies terminate. □

Chapter 4

Decidability

I now show how context logic over sequences, terms and trees can be decided. The decision procedures for each model are implemented using finite automata. Since the conceptual complexity of automata for terms and trees are greater than for sequences, I have chosen to first show the procedure for sequences, then for terms and finally for trees.

Initially, I make two constraints. Firstly, I assume that the labelling alphabets Σ and X are finite. This ensures that the automata are indeed finite. Secondly, I exclude quantification over hole labels (both $\exists$ and M). This is because the quantifiers do not have direct automaton constructions. Later, I show how these constraints can be lifted to give decision procedures where the labelling alphabets are infinite and for formulae with quantification over hole labels.

Collaboration and Contribution

The work in this chapter was conducted in collaboration with Calcagno and Gardner. The idea of implementing sequence formulae was present in an unpublished note of Calgno, Gardner and Zarfaty [CGZ06b]. Most of the technical work was my contribution, under the supervision of my collaborators.

4.1 Sequences

For a given formula P and logical environment σ, we are interested in the set of sequences which satisfy the formula: $\mathcal{L}_{P,\sigma} = \left\{ s \;\middle|\; s, \sigma \models_{\mathsf{c}(\mathrm{dom}\,\sigma)} P \right\}$. The question of whether the formula P is satisfiable (with respect to σ) is exactly the question of whether $\mathcal{L}_{P,\sigma} \neq \emptyset$. I can answer this question by leveraging the

theory of formal languages: $\mathcal{L}_{P,\sigma}$ is a language of words over $\Omega = \Sigma \cup \mathrm{X}$ and '$\mathcal{L}_{P,\sigma} \neq \emptyset$' is an instance of the language emptiness problem.

In fact, we shall see that each $\mathcal{L}_{P,\sigma}$ is a regular language, that is, a language that is recognised by a finite automaton. I show this by constructing automata corresponding to each context logic formula — that is, for a given P and σ, I define an automaton $\mathcal{A}_{P,\sigma}$ that accepts exactly the language $\mathcal{L}_{P,\sigma}$. These automata can be used to decide the language membership and emptiness problems, and hence decide model-checking and satisfiability for context logic for sequences.

In the following, assume that multi-holed sequence contexts, $\mathsf{C}^m_{\mathsf{Seq}}$, are defined over finite Σ and X. Let $\Omega = \Sigma \cup \mathrm{X}$, and let Ω^*, ranged over by $w, w', w_1, \ldots$, be the set of words over Ω (that is, finite sequences labelled from Ω). Note that $\mathsf{C}^m_{\mathsf{Seq}} \subseteq \Omega^*$.

4.1.1 Automata

Before I give these construction, I formally define finite automata and informally describe their operation.

Definition 4.1 (ε-NFA)**.** A *non-deterministic finite automaton with ε-transitions* (abbreviated ε-NFA) is a tuple $\mathcal{A} = (Q, e, \{f^a\}_{a \in \Omega \cup \{\varepsilon\}}, A)$ where:

- Q is the set of states, a finite set;
- $e \in Q$ is the initial state;
- for each $a \in \Omega$, $f^a \subseteq Q \times Q$ is the state transition relation for a (there is one for each);
- $f^\varepsilon \subseteq Q \times Q$ is the non-consuming state transition relation; and
- $A \subseteq Q$ is the set of accepting states.

Notation. For a given $q \in Q$, I write $f^a(q)$ for the set $\{q' \in Q \mid (q, q') \in f^a\}$.

Definition 4.2 (Forms of Automata)**.** An ε-NFA having $f^\varepsilon = \emptyset$ is a *non-deterministic finite automaton* (NFA). An NFA for which f^a is a partial function for all $a \in \Omega$ is a *deterministic finite automaton* (DFA). A DFA for which f^a is a total function for all $a \in \Omega$ is a *complete* DFA. A *pre-automaton* is an automaton without a set of accepting states, *i.e.* $\hat{\mathcal{A}} = (Q, e, \{f^a\}_{a \in \Omega \cup \{\varepsilon\}})$.

To formally define the language recognised by an automaton, I make some auxiliary definitions.

Definition 4.3 (ε-closure)**.** The ε-closure of a state is the set of states reachable by any number of ε-transitions. That is to say, ε-closure is the reflexive-transitive closure of f^{ε}:

$$\varepsilon\text{-closure} = (f^{\varepsilon})^*.$$

Definition 4.4 (Automaton-induced Mappings)**.** An automaton $\mathcal{A}$ induces a function $[\![(\cdot)]\!]_{\mathcal{A}} : \Omega^* \to \mathcal{P}(Q)$ that maps words $w \in \Omega^*$ to sets of states $[\![w]\!]_{\mathcal{A}} \subseteq Q$ according to the following definition:

$$\begin{aligned} [\![\varnothing]\!]_{\mathcal{A}} &= \varepsilon\text{-closure}(e) \\ [\![w \cdot a]\!]_{\mathcal{A}} &= \{q \mid \text{there exists } q' \in [\![w]\!]_{\mathcal{A}} \text{ s.t. } (q', q) \in (f^a \fatsemi \varepsilon\text{-closure})\}. \end{aligned}$$

An automaton $\mathcal{A}$ also induces a function $(\!|(\cdot)|\!)_{\mathcal{A}} : \Omega^* \to \mathcal{P}(Q \times Q)$ that maps words $w \in \Omega^*$ to relations on states $(\!|w|\!)_{\mathcal{A}} \subseteq Q \times Q$ according to the following definition:

$$\begin{aligned} (\!|\varnothing|\!)_{\mathcal{A}} &= \varepsilon\text{-closure} \\ (\!|w \cdot a|\!)_{\mathcal{A}} &= (\!|w|\!)_{\mathcal{A}} \fatsemi f^a \fatsemi \varepsilon\text{-closure}. \end{aligned}$$

Definition 4.5 (Acceptance)**.** A word w is said to be *accepted* by automaton $\mathcal{A}$ if $[\![w]\!]_{\mathcal{A}} \cap A \neq \emptyset$. The *language accepted by* $\mathcal{A}$, $\mathcal{L}_{\mathcal{A}}$, is the set of words accepted by $\mathcal{A}$:

$$\mathcal{L}_{\mathcal{A}} = \{w \in \Omega^* \mid [\![w]\!]_{\mathcal{A}} \cap A \neq \emptyset\}.$$

A run of an automaton is easiest to understand in the deterministic case. The automaton begins in state e and consumes the word, w, one letter at a time from left to right. When a letter, say a, is consumed, the automaton transitions from its current state, q, to the state $f^a(q)$. If the automaton is complete, then $f^a(q)$ is always defined, but otherwise it may be undefined. If the transitions for the word that the automaton is being run on are all defined, then once the entire word has been consumed, the automaton will be in some state, q' (for which $[\![w]\!]_{\mathcal{A}} = \{q'\}$). If $q' \in A$, the automaton accepts the word w. If $q' \notin A$, or if the transition is undefined at some point in the run, then the automaton does not accept the word w.

The non-deterministic case generalises this. One way to think of this generalisation is that multiple runs are possible for a given word, with each transition relation allowing a number of possible choices of transition from each state. The automaton accepts the word w if *any* of the runs, after consuming the entire word, ends in a state $q' \in A$.

An alternative view is that the automaton can be in multiple states at a time, beginning with $\{e\}$. At each step, after consuming letter a, the automaton is

in state q' (among others) if and only if there is some state q that it was in before such that $(q, q') \in f^a$. The automaton accepts the word w if, after it has consumed w in its entirety, it is in some state $q' \in A$ (possibly among others, which may or may not also be elements of A).

This view is useful for seeing how an NFA may be reduced to a (complete) DFA using the powerset construction. The states of the constructed DFA are the sets of states of the NFA, and the next state of the DFA is obtained by taking the union of the application of the appropriate transition relation of the NFA to each element of the DFA's state.

An ε-NFA generalises an NFA further by allowing transitions in which no letter is consumed (ε-transitions). The view of the automaton as being in multiple states still applies. However, the set of states after consuming a is not simply the set of all q' such that $(q, q') \in f^a$, for some state q' that the automaton was in, but it is the closure of this set under f^ε.

I have described an intuition for the following result, which is well known in the literature. A proof can be found in [Yu97].

Lemma 34. *For every ε-NFA, a complete DFA may be constructed that accepts exactly the same language.*

Although each type of automaton I have described can express the same languages, when constructing automata that accept specific languages, different types of automata can be better suited to the task, depending on the construction in question. For example, given a complete DFA $(Q, e, \{f^a\}_{a \in \Omega \cup \{\varepsilon\}}, A)$ accepting language $\mathcal{L}$, an automaton accepting the language $\Omega^* \setminus \mathcal{L}$ can be constructed by simply replacing the set of accepting states with $Q \setminus A$. (This construction works for complete DFA because they map each word to a single state. The words that were not accepted by the original automaton are exactly those that are mapped to non-accepting states.) Adapting this construction for NFA requires first determinising the NFA. On the other hand, ε-transitions permit a compact and intuitive automaton construction corresponding to language concatenation.

For a given word, w, it is useful to consider the set of states that runs of the automaton on that word end in. This set is given by $[\![w]\!]_{\mathcal{A}}$. It is, however, also useful to consider how the automaton would behave when run from different initial states. This is captured by the relation $(\!|w|\!)_{\mathcal{A}}$: $(q_1, q_2) \in (\!|w|\!)_{\mathcal{A}}$ if and only if, when the automaton is in state q_1 and proceeds to consume the word w it is possible for it to end in state q_2. The relation $(\!|w|\!)_{\mathcal{A}}$ effectively describes how the word w is interpreted by the automaton in any context, and hence it is a useful concept in constructing automata for context logic.

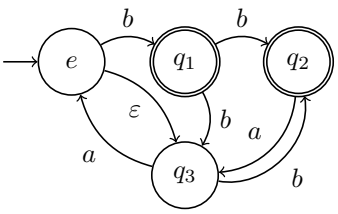


Figure 4.1: Representation of an ε-NFA

It should be noted that $(\!|(\cdot)|\!)_{\mathcal{A}}$ is a monoid homomorphism[1]; that is $(\!|w_1 \cdot w_2|\!)_{\mathcal{A}} = (\!|w_1|\!)_{\mathcal{A}} \fatsemi (\!|w_2|\!)_{\mathcal{A}}$ — relational composition corresponds to concatenation of words.

Finite automata are depicted as finite, edge-labelled, directed graphs. Each node of the graph represents a state, with the initial state identified by an arrow that does not originate at any node, and the accepting states identified by double circles. An a-labelled edge between two nodes indicates that the connected pair of states belongs to the relation f^a.

Example 4.1. Figure 4.1 illustrates the automaton $\mathcal{A} = (Q, e, \{f^a\}_{a \in \{a,b,\varepsilon\}}, A)$ where:

- $Q = \{e, q_1, q_2, q_3\}$;
- $f^a = \{(q_2, q_3), (q_3, e)\}$;
- $f^b = \{(e, q_1), (q_1, q_2), (q_1, q_3), (q_3, q_2)\}$;
- $f^\varepsilon = \{(e, q_3)\}$; and
- $A = \{q_2, q_3\}$.

For this automaton:

- $\varepsilon\text{-}\mathsf{closure} = \{(e, e), (e, q_3), (q_1, q_1), (q_3, q_3)\}$;
- $[\![\varnothing]\!]_{\mathcal{A}} = \{e, q_3\}$ and, since this set is disjoint form A, the word $\varnothing$ is not accepted by the automaton;
- $[\![b \cdot b]\!]_{\mathcal{A}} = \{q_2, q_3\}$ and, since $q_2 \in [\![b \cdot b]\!]_{\mathcal{A}} \cap A$, $b \cdot b$ is accepted by the automaton;
- $(\!|b \cdot a|\!)_{\mathcal{A}} = \{(e, q_3), (q_1, e), (q_1, q_3), (q_3, q_3)\}$.

[1] from the set of words under concatenation to the set of ε-closed binary relations on Q under composition

With all the types of finite automata, language membership and emptiness are decidable. For membership, it is sufficient to consider the run (or runs) of an automaton on a given word. For emptiness, it is sufficient to determine whether whether an accepting state is reachable from the initial state by any combination of transitions. This is effectively an instance of the reachability problem for finite directed graphs.

Definition 4.6 (Reachable States)**.** Given automaton $\mathcal{A}$, let $f : \mathcal{P}(Q) \to \mathcal{P}(Q)$ be the *one-step reachability function* given by

$$f(\mathcal{R}) = \mathcal{R} \cup \{e\} \cup \{q \mid \text{there exists } a \in \Omega \cup \{\varepsilon\}, q' \in \mathcal{R} \text{ s.t. } q \in f^a(q')\}.$$

This function is monotone and so, by the Knaster-Tarski theorem [Tar55], it has a least fixed-point. Let $\mathsf{reachable}$ be this least fixed-point, the *set of reachable states of* $\mathcal{A}$. Since $\mathcal{P}(Q)$ is finite, $\mathsf{reachable}$ is computable. Indeed, $\mathsf{reachable} = (f)^n(\emptyset)$ for some finite n.

Theorem 35. *Given automaton,* $\mathcal{A}$*, for all* $q \in Q$*,*

$$q \in \mathsf{reachable} \iff \textit{there exists } w \in \Omega^* \textit{ s.t. } q \in [\![w]\!]_{\mathcal{A}}.$$

Proof. $\Longrightarrow$:
Let us show that, for all m, for all $q \in (f)^m(\emptyset)$ there exists some $w \in \Omega^*$ such that $q \in [\![w]\!]_{\mathcal{A}}$. The proof is by induction on m. The base case, where $m = 0$, is trivial. For the inductive case, let $q \in (f)^m(\emptyset)$. One of the following is the case:

- $q \in (f)^{m-1}$ and the conclusion holds by the inductive hypothesis;
- $q \in \{e\}$ — so $q \in [\![\varnothing]\!]_{\mathcal{A}}$;
- for some $a \in \Omega$ and some $q' \in (f)^{m-1}$, $q \in f^a(q')$ — so by the inductive hypothesis, there is a $w' \in \Omega^*$ such that $q' \in [\![w']\!]_{\mathcal{A}}$, and so $q \in [\![w \cdot a]\!]_{\mathcal{A}}$; or
- for some $q' \in (f)^{m-1}$, $q \in f^{\varepsilon}(q')$ — so by the inductive hypothesis, there is some $w \in \Omega^*$ such that $q' \in [\![w]\!]_{\mathcal{A}}$, and so also $q \in [\![w]\!]_{\mathcal{A}}$.

Since $\mathsf{reachable} = (f)^n$ for some n, the implication holds.
$\Longleftarrow$:
The proof in this direction is by induction on the structure of the word w. Note first that $\mathsf{reachable}$ is closed under ε-transitions (since $f^{\varepsilon} \subseteq f$). In the base case, $w = \varnothing$ and so if $q \in [\![w]\!]_{\mathcal{A}}$ then $q \in \varepsilon\text{-}\mathsf{closure}(e)$. Clearly, $e \in \mathsf{reachable}$ and so $q \in \mathsf{reachable}$ also. In the inductive case, $w = w' \cdot a$ for some $w' \in \Omega^*$, $a \in \Omega$, so if $q \in [\![w]\!]_{\mathcal{A}}$ then there is some $q' \in [\![w']\!]_{\mathcal{A}}$ with $q \in (f^a \fatsemi \varepsilon\text{-}\mathsf{closure})(q')$. By the

inductive hypothesis, $q' \in \mathsf{reachable}$, and by the definition of f and the closure of $\mathsf{reachable}$ under ε-transitions, $q \in \mathsf{reachable}$ also. □

The class of languages accepted by automata is the class of *regular languages*. An important property of regular languages is that they are closed under union, intersection, complementation with respect to Ω^*, and language concatenation. This result forms part of Kleene's theorem, which states that the regular languages are exactly those corresponding to regular expressions [Kle56]. That is, they constitute the smallest class of languages that includes the empty language and every single-element language, and that is closed under the above operations, as well as repetition (Kleene star $*$). For these results, it is also significant that $\mathsf{C}^m_{\mathsf{Seq}}$ itself is a regular language. A fuller exposition of regular languages and automata may be found in [Yu97][2].

An important subclass of the regular languages are the *star-free regular languages*: the smallest class of languages with the same properties as regular languages except for closure under Kleene star.

4.1.2 Basic Constructions

The aforementioned closure properties of regular languages give a means to implement many of the connectives of context logic with automata — constructions for them are well-known in the literature (*e.g.* [Yu97]). I present constructions for disjunction (union), conjunction (intersection) and concatenation as examples. In constructions that are based on multiple automata, I assume that the state sets of these automata are disjoint. In any event, the state sets can be renamed so that this is the case.

Definition 4.7 (Union Construction)**.** Given ε-NFA

$$\mathcal{A}_1 = (Q_1, e_1, \{f_1^a\}_{a \in \Omega \cup \{\varepsilon\}}, A_1) \text{ and}$$
$$\mathcal{A}_2 = (Q_2, e_2, \{f_2^a\}_{a \in \Omega \cup \{\varepsilon\}}, A_2),$$

the ε-NFA

$$\mathcal{A}_1 \cup \mathcal{A}_2 = (Q, e, \{f^a\}_{a \in \Omega \cup \{\varepsilon\}}, A)$$

is defined as follows:

- $Q = Q_1 \cup Q_2 \cup \{e\}$;
- for $a \in \Omega$, $f^a = f_1^a \cup f_2^a$;

[2]Note that in [Yu97], λ is used instead of ε.

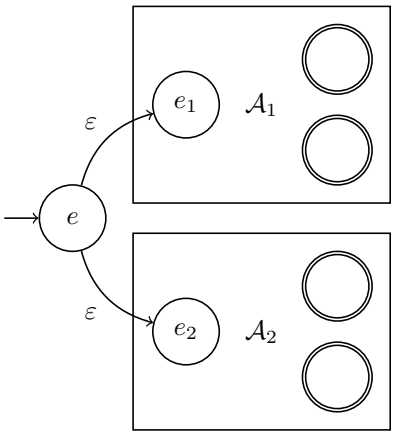


Figure 4.2: Representation of ε-NFA $\mathcal{A}_1 \cup \mathcal{A}_2$

- $f^\varepsilon = f_1^\varepsilon \cup f_2^\varepsilon \cup \{(e, e_1), (e, e_2)\}$; and
- $A = A_1 \cup A_2$,

where e is fresh.

Proposition 36 (Correctness of Union Construction)**.** *Given automata $\mathcal{A}_1$ and $\mathcal{A}_2$, accepting languages $\mathcal{L}_1$ and $\mathcal{L}_2$ respectively, the automaton $\mathcal{A}_1 \cup \mathcal{A}_2$ accepts the language $\mathcal{L}_1 \cup \mathcal{L}_2$.*

Proof sketch. The constructed automaton consists of a copy of $\mathcal{A}_1$ and $\mathcal{A}_2$ and starts with a non-deterministic choice as to which automaton to proceed in. An accepting run on either of the original automata will result in an accepting run in $\mathcal{A}$ by prepending the run with the transition from e to the initial state of the appropriate automaton. Conversely, an accepting run on $\mathcal{A}$ will result in an accepting run on one of the original automata by removing the initial transition (and beginning from the initial state of the appropriate automaton). □

Figure 4.2 illustrates this automaton construction.

Definition 4.8 (Product Pre-automaton)**.** Given pre-automata

$$\hat{\mathcal{A}}_1 = (Q_1, e, \{f_1^a\}_{a\Omega\cup\{\varepsilon\}\in)} \text{ and}$$
$$\hat{\mathcal{A}}_2 = (Q_2, e, \{f_2^a\}_{a\Omega\cup\{\varepsilon\}\in)},$$

the *product pre-automaton*

$$\hat{\mathcal{A}}_1 \times \hat{\mathcal{A}}_2 = (Q, e, \{f^a\}_{a\in\Omega\cup\{\varepsilon\}})$$

is defined as follows:

- $Q = Q_1 \times Q_2$;
- $e = (e_1, e_2)$;
- for $a \in \Omega$, $f^a(q_1, q_2) = \{(q_1', q_2') \mid q_1' \in f_1^a(q_1), q_2' \in f_2^a(q_2)\}$; and
- $f^\varepsilon(q_1, q_2) = \{(q_1', q_2), (q_1, q_2') \mid q_1' \in f_1^\varepsilon(q_1) \text{ and } q_2' \in f_2^\varepsilon(q_2)\}$.

The product construction simulates the operation of two automata in parallel. This construction is useful as the basis of other constructions, such as the intersection construction below.

Proposition 37 (Correctness of Product Construction)**.** *Given pre-automata $\hat{\mathcal{A}}_1$ and $\hat{\mathcal{A}}_2$, for all $w \in \Omega^*$, $q_1 \in Q_1$ and $q_2 \in Q_2$,*

$$(\!|w|\!)_{\hat{\mathcal{A}}_1 \times \hat{\mathcal{A}}_2}((q_1, q_2)) = \left((\!|w|\!)_{\hat{\mathcal{A}}_1}(q_1)\right) \times \left((\!|w|\!)_{\hat{\mathcal{A}}_2}(q_2)\right) \tag{4.1}$$

$$[\![w]\!]_{\hat{\mathcal{A}}_1 \times \hat{\mathcal{A}}_2} = [\![w]\!]_{\hat{\mathcal{A}}_1} \times [\![w]\!]_{\hat{\mathcal{A}}_2}. \tag{4.2}$$

Proof. Let $\hat{\mathcal{A}} = \hat{\mathcal{A}}_1 \times \hat{\mathcal{A}}_2$. For (4.1), the proof is by induction on the structure of w. Observe first that $\varepsilon\text{-}\mathsf{closure}((q_1, q_2)) = \varepsilon\text{-}\mathsf{closure}_1(q_1) \times \varepsilon\text{-}\mathsf{closure}_2(q_2)$.

Base case: $w = \varnothing$. In this case,

$$\begin{aligned}(\!|\varnothing|\!)_{\hat{\mathcal{A}}_1 \times \hat{\mathcal{A}}_2}((q_1, q_2)) &= \varepsilon\text{-}\mathsf{closure}((q_1, q_2)) \\ &= \varepsilon\text{-}\mathsf{closure}_1(q_1) \times \varepsilon\text{-}\mathsf{closure}_2(q_2) \\ &= \left((\!|\varnothing|\!)_{\hat{\mathcal{A}}_1}(q_1)\right) \times \left((\!|\varnothing|\!)_{\hat{\mathcal{A}}_2}(q_2)\right).\end{aligned}$$

Inductive case: $w = w' \cdot a$ (for some $w' \in \Omega^*$, $a \in \Omega$). In this case,

$$\begin{aligned}(\!|w' \cdot a|\!)_{\hat{\mathcal{A}}_1 \times \hat{\mathcal{A}}_2}((q_1, q_2)) &= \left((\!|w'|\!)_{\hat{\mathcal{A}}_1 \times \hat{\mathcal{A}}_2} \fatsemi f^a \fatsemi \varepsilon\text{-}\mathsf{closure}\right)((q_1, q_2)) \\ \text{(IH)} &= (f^a \fatsemi \varepsilon\text{-}\mathsf{closure})\left(\left((\!|w'|\!)_{\hat{\mathcal{A}}_1}(q_1)\right) \times \left((\!|w'|\!)_{\hat{\mathcal{A}}_2}\right)\right) \\ &= \varepsilon\text{-}\mathsf{closure}\left(\left((\!|w'|\!)_{\hat{\mathcal{A}}_1} \fatsemi f_1^a\right)(q_1)\right) \times \left(\left((\!|w'|\!)_{\hat{\mathcal{A}}_2} \fatsemi f_2^a\right)(q_2)\right)\right) \\ &= \left(\left((\!|w'|\!)_{\hat{\mathcal{A}}_1} \fatsemi f_1^a \fatsemi \varepsilon\text{-}\mathsf{closure}_1\right)(q_1)\right) \times \\ &\qquad \left(\left((\!|w'|\!)_{\hat{\mathcal{A}}_2} \fatsemi f_2^a \fatsemi \varepsilon\text{-}\mathsf{closure}_2\right)(q_2)\right) \\ &= \left((\!|w' \cdot a|\!)_{\hat{\mathcal{A}}_1}(q_1)\right) \times \left((\!|w' \cdot a|\!)_{\hat{\mathcal{A}}_2}(q_2)\right).\end{aligned}$$

For (4.2),

$$\begin{aligned}[\![w]\!]_{\hat{\mathcal{A}}_1 \times \hat{\mathcal{A}}_2} &= (\!|w|\!)_{\hat{\mathcal{A}}_1 \times \hat{\mathcal{A}}_2}((e_1, e_2)) \\ &= \left((\!|w|\!)_{\hat{\mathcal{A}}_1}(e_1)\right) \times \left((\!|w|\!)_{\hat{\mathcal{A}}_2}(e_2)\right) \\ &= [\![w]\!]_{\hat{\mathcal{A}}_1} \times [\![w]\!]_{\hat{\mathcal{A}}_2}.\end{aligned}$$

□

Definition 4.9 (Intersection Construction)**.** Given ε-NFA

$$\mathcal{A}_1 = (Q_1, e_1, \{f_1^a\}_{a \in \Omega \cup \{\varepsilon\}}, A_1) \text{ and}$$
$$\mathcal{A}_2 = (Q_2, e_2, \{f_2^a\}_{a \in \Omega \cup \{\varepsilon\}}, A_2),$$

the ε-NFA

$$\mathcal{A}_1 \cap \mathcal{A}_2 = (Q, e, \{f^a\}_{a \in \Omega \cup \{\varepsilon\}}, A)$$

is defined as follows:

- $(Q, e, \{f^a\}_{a \in \Omega \cup \{\varepsilon\}}) = (Q_1, e_1, \{f_1^a\}_{a \in \Omega \cup \{\varepsilon\}}) \times (Q_2, e_2, \{f_2^a\}_{a \in \Omega \cup \{\varepsilon\}})$; and
- $A = A_1 \times A_2$.

Proposition 38 (Correctness of Intersection Construction)**.** *Given automata $\mathcal{A}_1$ and $\mathcal{A}_2$, accepting languages $\mathcal{L}_1$ and $\mathcal{L}_2$ respectively, the automaton $\mathcal{A}_1 \cap \mathcal{A}_2$ accepts the language $\mathcal{L}_1 \cap \mathcal{L}_2$.*

Proof. Let $\mathcal{A} = \mathcal{A}_1 \cap \mathcal{A}_2$.

$$\begin{aligned}
& [\![w]\!]_{\mathcal{A}} \cap A \neq \emptyset \\
\iff & ([\![w]\!]_{\mathcal{A}_1} \times [\![w]\!]_{\mathcal{A}_2}) \cap (A_1 \times A_2) \neq \emptyset \\
\iff & [\![w]\!]_{\mathcal{A}_1} \cap A_1 \neq \emptyset \text{ and } [\![w]\!]_{\mathcal{A}_2} \cap A_2 \neq \emptyset \\
\iff & w \in \mathcal{L}_1 \text{ and } w \in \mathcal{L}_2 \\
\iff & w \in \mathcal{L}_1 \cap \mathcal{L}_2.
\end{aligned}$$

□

Remark. A similar construction for union is possible, based on the product pre-automaton, using the accept set $A_1 \times Q_2 \cup Q_1 \times A_2$. A possible benefit of such a construction is that it does not introduce any non-determinism. However, the construction does have a much greater state space than that of Definition 4.7.

Definition 4.10 (Concatenation Construction)**.** Given ε-NFA

$$\mathcal{A}_1 = (Q_1, e_1, \{f_1^a\}_{a \in \Omega \cup \{\varepsilon\}}, A_1) \text{ and}$$
$$\mathcal{A}_2 = (Q_2, e_2, \{f_2^a\}_{a \in \Omega \cup \{\varepsilon\}}, A_2),$$

the ε-NFA

$$\mathcal{A}_1 \cdot \mathcal{A}_2 = (Q, e, \{f^a\}_{a \in \Omega \cup \{\varepsilon\}}, A)$$

is defined as follows:

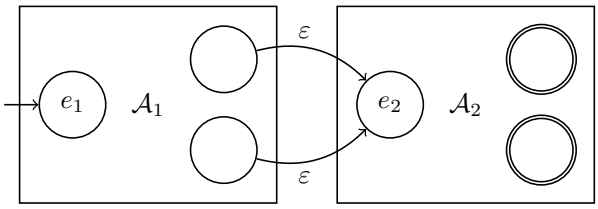


Figure 4.3: Representation of ε-NFA $\mathcal{A}_1 \cdot \mathcal{A}_2$

- $Q = Q_1 \cup Q_2$;
- $e = e_1$;
- for $a \in \Omega$, $f^a = f_1^a \cup f_2^a$;
- $f^\varepsilon = f_1^\varepsilon \cup f_2^\varepsilon \cup \{(q_1, e_2) \mid q_1 \in A_1\}$; and
- $A = A_2$.

Proposition 39 (Correctness of Concatenation Construction)**.** *Given automata $\mathcal{A}_1$ and $\mathcal{A}_2$, accepting languages $\mathcal{L}_1$ and $\mathcal{L}_2$ respectively, the automaton $\mathcal{A}_1 \cdot \mathcal{A}_2$ accepts the language $\mathcal{L}_1 \cdot \mathcal{L}_2$ — the language concatenation of $\mathcal{L}_1$ and $\mathcal{L}_2$.*

Proof sketch. An accepting run of $\mathcal{A}_1 \cdot \mathcal{A}_2$ consists of an accepting run of $\mathcal{A}_1$ followed by an accepting run of $\mathcal{A}_2$. Hence if w is accepted by $\mathcal{A}_1 \cdot \mathcal{A}_2$ then $w = w_1 \cdot w_2$ for some w_1 that is accepted by $\mathcal{A}_1$ and w_2 that is accepted by $\mathcal{A}_2$. Conversely, if w_1 is accepted by $\mathcal{A}_1$ and w_2 is accepted by $\mathcal{A}_2$ then the accepting runs of those automata can be combined to give an accepting run of $\mathcal{A}_1 \cdot \mathcal{A}_2$ on the concatenation $w_1 \cdot w_2$. □

Figure 4.3 illustrates this automaton construction.

Remark. Non-determinism means that every possible splitting of a candidate word between $\mathcal{L}_1$ and $\mathcal{L}_2$ is considered. For example, consider $\mathcal{L}_1 = \{\varnothing, a, a \cdot a\}$, $\mathcal{L}_2 = \{\varnothing, a \cdot b, a \cdot a \cdot b\}$. Clearly, the word $a \cdot a \cdot b$ is in $\mathcal{L}_1 \cdot \mathcal{L}_2$. Given $\mathcal{A}_1$ and $\mathcal{A}_2$ accepting these languages, one accepting run of $\mathcal{A}_1 \cdot \mathcal{A}_2$ on $a \cdot a \cdot b$ transitions from $\mathcal{A}_1$ to $\mathcal{A}_2$ after having consumed $\varnothing$, while another accepting run transitions after having read a. No accepting run transitions after having read $a \cdot a$, since b is not accepted by $\mathcal{A}_2$. No accepting run transitions after having read $a \cdot a \cdot b$, since that word is not accepted by $\mathcal{A}_1$.

Definition 4.11 (Complementation Construction)**.** Given an ε-NFA

$$\mathcal{A}_1 = (Q_1, e_1, \{f_1^a\}_{a \in \Omega \cup \{\varepsilon\}}, A_1),$$

the ε-NFA (actually, a complete DFA)

$$\overline{\mathcal{A}_1} = (Q, e, \{f^a\}_{a \in \Omega \cup \{\varepsilon\}}, A)$$

is defined as follows:

- $Q = \mathcal{P}(Q_1)$;
- $e = \varepsilon\text{-}\mathsf{closure}_1(e_1)$;
- for $a \in \Omega$, $f^a(q) = \left\{\varepsilon\text{-}\mathsf{closure}_1\left(\bigcup_{q_1 \in q} f_1^a(q_1)\right)\right\}$;
- $f^\varepsilon = \emptyset$; and
- $A = \{q \in Q \mid q \cap A_1 = \emptyset\}$.

Proposition 40 (Correctness of Complementation Construction)**.** *Given automaton $\mathcal{A}_1$, accepting language $\mathcal{L}_1$, the automaton $\overline{\mathcal{A}_1}$ accepts the language $\Omega^* \setminus \mathcal{L}_1$ — the complement of $\mathcal{L}_1$ with respect to Ω^*.*

Proof sketch. The automaton construction for $\overline{\mathcal{A}_1}$ implements the determinisation procedure described previously: each state of $\overline{\mathcal{A}_1}$, q, is a set of states from the original automaton representing the states reachable by every possible run of the original automaton on the word consumed to reach that state. If any $q_1 \in q$ is also contained in A_1, then the $\mathcal{A}_1$ would have accepted the input word; exactly such words are rejected by $\overline{\mathcal{A}_1}$. □

4.1.3 Generalisations of Composition

I must still give automaton constructions to implement the structural connectives $\circ$, $\circ\!\!-^{\exists}$ and $-\!\!\circ^{\exists}$. Although we are only really interested in sequence contexts, the criterion that an automaton should only accept sequence contexts, as opposed to words in general (in which hole labels may occur more than once), does not correspond to a useful structural property. Therefore it is sensible to implement the connectives in terms of operations that are defined for words. For this, I use non-deterministic linear substitution, as defined below.

Definition 4.12 (Non-deterministic Linear Substitution)**.** Given words w_1, $w_2 \in \Omega^*$ and label $a \in \Omega$, the *non-deterministic linear substitution* $w_1 \oslash_a w_2$ is defined to be the set of words obtained by replacing exactly one occurrence of a in w_1 by the word w_2. Given languages $\mathcal{L}_1, \mathcal{L}_2 \subseteq \Omega^*$, non-deterministic linear

substitution and the existential duals of its adjoints are defined as follows:

$$\begin{aligned}
\mathcal{L}_1 \oslash_a \mathcal{L}_2 &= \bigcup_{\substack{w_1 \in \mathcal{L}_1 \\ w_2 \in \mathcal{L}_2}} w_1 \oslash_a w_2 \\
\mathcal{L}_1 \oslash\!\!-\!\!^{\exists}_a \mathcal{L}_2 &= \{w \in \Omega^* \mid \text{there exists } w' \in \mathcal{L}_1 \text{ s.t. } (w' \oslash_a w) \cap \mathcal{L}_2 \neq \emptyset\} \\
\mathcal{L}_1 -\!\!\oslash^{\exists}_a \mathcal{L}_2 &= \{w \in \Omega^* \mid \text{there exists } w' \in \mathcal{L}_1 \text{ s.t. } (w \oslash_a w') \cap \mathcal{L}_2 \neq \emptyset\}\,.
\end{aligned}$$

A word in $\mathcal{L}_1 \oslash_a \mathcal{L}_2$ is a word from $\mathcal{L}_1$ in which exactly one occurrence of a has been replaced by a word from $\mathcal{L}_2$. This corresponds to context composition when restricted to proper, linear contexts, so $\oslash$ is a convenient operation with which to implement context composition. However, there are two other, more obvious generalisations of context composition to the non-linear case. Firstly, there is the operation of replacing each occurrence of a in a word of one language by a word of a second language, with each instance being replaced by the same word: uniform substitution.

Definition 4.13 (Uniform Substitution)**.** Given languages $\mathcal{L}_1, \mathcal{L}_2 \subseteq \Omega^*$ and label $a \in \Omega$, *uniform substitution* and the existential duals of its adjoints are defined as follows:

$$\begin{aligned}
\mathcal{L}_1 \odot_a \mathcal{L}_2 &= \{w_1[w_2/a] \mid w_1 \in \mathcal{L}_1 \text{ and } w_2 \in \mathcal{L}_2\} \\
\mathcal{L}_1 \odot\!\!-\!\!^{\exists}_a \mathcal{L}_2 &= \{w \in \Omega^* \mid \text{there exists } w_1 \in \mathcal{L}_1 \text{ s.t. } w_1[w/a] \in \mathcal{L}_2\} \\
\mathcal{L}_1 -\!\!\odot^{\exists}_a \mathcal{L}_2 &= \{w \in \Omega^* \mid \text{there exists } w_1 \in \mathcal{L}_1 \text{ s.t. } w[w_1/a] \in \mathcal{L}_2\}\,.
\end{aligned}$$

A word in $\mathcal{L}_1 \odot_a \mathcal{L}_2$ is a word from $\mathcal{L}_1$ in which each occurrence of a has been replaced by a word from $\mathcal{L}_2$, with each replacement being the same (hence, uniform). Although this may seem to be a more natural operation than non-deterministic linear substitution, neither star-free regular languages and regular languages are closed under uniform substitution. In fact, it is not possible in general to decide the emptiness problem for languages constructed with uniform substitution together with the basic constructions of star-free regular languages. This can be shown by the fact that the decision problem of whether a first order sentence has a finite model can be encoded as the emptiness problem. By Trakhtenbrot's Theorem [Tra50], this problem is undecidable.

A second alternative is the operation of replacing each occurrence of a in a word of one language by (possibly different) words of a second language: non-uniform substitution.

Definition 4.14 (Non-uniform Substitution)**.** Given languages $\mathcal{L}_1, \mathcal{L}_2 \subseteq \Omega^*$ and label $a \in \Omega$, the *non-uniform substitution* $\mathcal{L}_1 \circledcirc_a \mathcal{L}_2$ is defined to be the

set of words obtained by replacing each occurrence of a in words from $\mathcal{L}_1$ by a word in $\mathcal{L}_2$. Each occurrence of a may be replaced by a different word from $\mathcal{L}_2$ (hence, non-uniform).

The existential dual of the adjoint of $(\cdot) \circledcirc_a \mathcal{L}_1$ is defined as follows:

$$\mathcal{L}_1 \mathrel{-\!\!\circledcirc\!\!\!\!^{\exists}_{a}} \mathcal{L}_2 = \{w \in \Omega^* \mid (\{w\} \circledcirc_a \mathcal{L}_1) \cap \mathcal{L}_2 \neq \emptyset\}\,.$$

Unlike uniform substitution, the class of regular languages is closed under non-uniform substitution. Note, however, that star-free regular languages are not closed under non-uniform substitution. In particular, the languages $(a \cdot x)^*$ and a have aperiodicity number 2, and so are star-free, by Schützenberger [Sch65], while the language $(a \cdot a)^* = (a \cdot x)^* \circledcirc_x a$ is not aperiodic, and so is not star-free.

More significantly, non-uniform substitution does not have two corresponding adjoints, but only one. This is because $\circledcirc_a$ does not distribute over $\cup$ in its second argument. Consequently, it does not immediately suggest an operation with which to implement the $\circ\!\!-\!\!\!\!^{\exists}$ connective.

I shall show that non-deterministic linear substitution does not suffer from the issues arising with uniform and non-uniform substitution, and so I elect to use it in implementing the connectives of context logic. The disadvantage of this choice is that the automaton constructions are not the simplest required to implement the logic. However, I hope that the uniformity this choice affords my approach leads to a greater clarity of exposition.

4.1.4 Complex Constructions

I now present automaton constructions for the operations $\oslash$, $\oslash\!\!-\!\!\!\!^{\exists}$ and $-\!\!\oslash^{\exists}$.

Definition 4.15 ($\oslash$ Construction)**.** Given ε-NFA

$$\mathcal{A}_1 = (Q_1, e_1, \{f_1^a\}_{a \in \Omega \cup \{\varepsilon\}}, A_1) \text{ and}$$
$$\mathcal{A}_2 = (Q_2, e_2, \{f_2^a\}_{a \in \Omega \cup \{\varepsilon\}}, A_2),$$

the ε-NFA

$$\mathcal{A}_1 \oslash_x \mathcal{A}_2 = (Q, e, \{f^a\}_{a \in \Omega \cup \{\varepsilon\}}, A)$$

is defined as follows:

- $Q = Q_1 \times (Q_2 \cup \{0, 1\})$;
- $e = (e_1, 0)$;

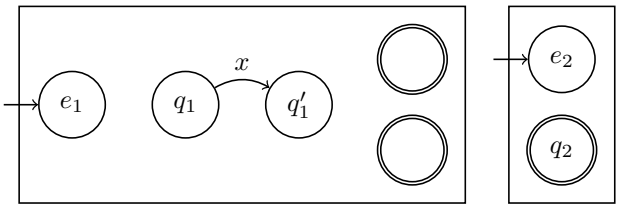


Figure 4.4: Partial representation of ε-NFA $\mathcal{A}_1$ (left) and $\mathcal{A}_2$

- for $a \in \Omega$, f^a is the smallest relation satisfying:
  - $(q_1', n) \in f^a((q_1, n))$ whenever $q_1' \in f_1^a(q_1)$ and for $n \in \{0, 1\}$, and
  - $(q_1, q_2') \in f^a((q_1, q_2))$ whenever $q_2' \in f_2^a(q_2)$ and for $q_1 \in Q_1$;
- f^ε is the smallest relation satisfying:
  - $(q_1', n) \in f^\varepsilon((q_1, n))$ whenever $q_1' \in f_1^\varepsilon(q_1)$ and for all $n \in \{0, 1\}$,
  - $(q_1, q_2') \in f^\varepsilon((q_1, q_2))$ whenever $q_2' \in f_2^\varepsilon(q_2)$ and for all $q_1 \in Q_1$,
  - $(q_1, e_2) \in f^\varepsilon((q_1, 0))$, and
  - $(q_1', 1) \in f^\varepsilon((q_1, q_2))$ whenever $q_1' \in f_1^x(q_1)$ and $q_2 \in A_2$; and
- $A = A_1 \times \{1\}$.

The essence of this construction is illustrated by Figures 4.4 and 4.5.

Consider the automaton $\mathcal{A} = \mathcal{A}_1 \oslash_x \mathcal{A}_2$, where $\mathcal{A}_1$ and $\mathcal{A}_2$ are automata accepting languages $\mathcal{L}_1$ and $\mathcal{L}_2$ respectively. When this automaton is run on a word, it initially behaves like $\mathcal{A}_1$; the state has the form $(q_1, 0)$. At some point in the run, the automaton may switch to behave like $\mathcal{A}_2$ from its initial state by making a ε-transition and keeping a record of the state of $\mathcal{A}_1$ that it was previously in; the state then has the form (q_1, q_2). If the automaton eventually reaches an accepting state of $\mathcal{A}_2$, the automaton may switch back to behave like $\mathcal{A}_1$ as if it had just read x instead of the word from $\mathcal{L}_2$ (the language accepted by $\mathcal{A}_2$) that it actually read; the state then has the form $(q_1, 1)$. Once the run is completed, if the automaton is in an accepting state then it has read a word of the form $w_1' \cdot w_2 \cdot w_1''$ where $w_1' \cdot x \cdot w_1'' \in \mathcal{L}_1$ and $w_2 \in \mathcal{L}_2$.

The above correctness argument is formalised in the following lemmata.

Lemma 41. *For all $w \in \Omega^*$, $q \in Q_1$,*

$$(q_1, 0) \in [\![w]\!]_{\mathcal{A}} \iff q_1 \in [\![w]\!]_{\mathcal{A}_1}.$$

Proof. Both directions are by induction on the structure of w.

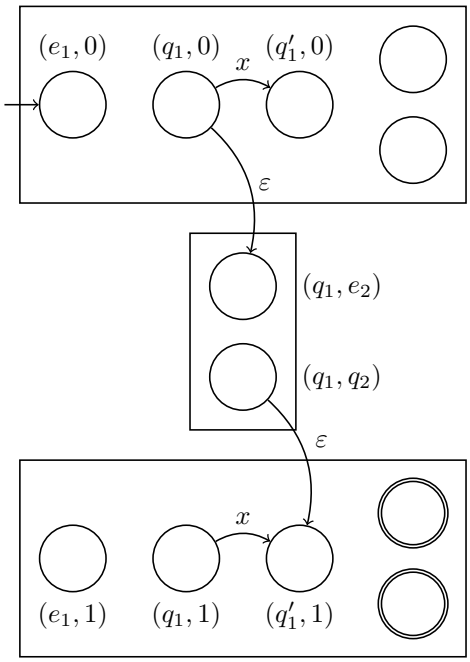


Figure 4.5: Partial representation of ε-NFA $\mathcal{A}_1 \oslash_x \mathcal{A}_2$

$\Longrightarrow$:

Base case: $w = \varnothing$ so $(q_1, 0) \in \varepsilon\text{-}\mathsf{closure}((e_1, 0))$, and hence $q_1 \in \varepsilon\text{-}\mathsf{closure}_1(e_1)$, and $q_1 \in [\![\varnothing]\!]_{\mathcal{A}_1}$.

Inductive case: $w = w' \cdot a$ (for some $w' \in \Omega^*$, $a \in \Omega$). In this case, $(q_1, 0) \in (f^a \fatsemi \varepsilon\text{-}\mathsf{closure})(q')$ for some $q' \in [\![w']\!]_{\mathcal{A}}$. This implies that $q' = (q_1', 0)$ for some $q_1' \in Q_1$ with $q_1 \in (f_1^a \fatsemi \varepsilon\text{-}\mathsf{closure}_1)(q_1')$ and, by the inductive hypothesis, $q_1' \in [\![w']\!]_{\mathcal{A}_1}$. Hence, $q_1 \in [\![w]\!]_{\mathcal{A}_1}$.

$\Longleftarrow$:

Base case: $w = \varnothing$ so $q_1 \in \varepsilon\text{-}\mathsf{closure}_1(e_1)$, and hence $(q_1, 0) \in \varepsilon\text{-}\mathsf{closure}(e)$.

Inductive case: $w = w' \cdot a$ (for some $w' \in \Omega^*$, $a \in \Omega$). In this case, $q_1 \in (f_1^a \fatsemi \varepsilon\text{-}\mathsf{closure}_1)(q_1')$ for some $q_1' \in [\![w']\!]_{\mathcal{A}_1}$. Since, by the inductive hypothesis, $(q_1', 0) \in [\![w']\!]_{\mathcal{A}}$, it follows that $(q_1, 0) \in [\![w]\!]_{\mathcal{A}}$. □

Lemma 42. *For all $w \in \Omega^*$, $q_1, \in Q_1$ and $q_2 \in Q_2$,*

$$(q_1, q_2) \in [\![w]\!]_{\mathcal{A}}$$
$$\Longleftrightarrow$$

there exist $w_1, w_2 \in \Omega^*$ s.t. $w = w_1 \cdot w_2$ and $q_1 \in [\![w_1]\!]_{\mathcal{A}_1}$ and $q_2 \in [\![w_2]\!]_{\mathcal{A}_2}$.

Proof. Both directions are by induction on the structure of w.

$\Longrightarrow$:

Base case: $w = \varnothing$ so $(q_1, q_2) \in \varepsilon\text{-}\mathsf{closure}((e_1, 0))$. Thus, by the definition of $\mathcal{A}$, it must be the case that:

- $(q_1, q_2) \in \varepsilon\text{-}\mathsf{closure}((q_1, e_2))$, and hence $q_2 \in \varepsilon\text{-}\mathsf{closure}_2(e_2)$, and $q_2 \in [\![\varnothing]\!]_{\mathcal{A}_2}$;
- $(q_1, e_2) \in f^{\varepsilon}((q_1, 0))$;
- $(q_1, 0) \in [\![\varnothing]\!]_{\mathcal{A}}$ and so $q_1 \in [\![\varnothing]\!]_{\mathcal{A}_1}$ (by Lemma 41).

The choice $w_1 = w_2 = \varnothing$ therefore fulfils the requirements.

Inductive case: $w = w' \cdot a$ (for some $w' \in \Omega^*$, $a \in \Omega$). In this case, $(q_1, q_2) \in (f^a \fatsemi \varepsilon\text{-}\mathsf{closure})(q')$ for some $q' \in Q$. It must be that either $q' = (q_1, q_2')$ or $q' = (q_1', 0)$ for some $q_2' \in Q_2$, $q_1' \in Q_1$.

In the former case, $q_2 \in (f_2^a \fatsemi \varepsilon\text{-}\mathsf{closure}_2)(q_2')$, by the definition of $\mathcal{A}$. By the inductive hypothesis, there are w_1, w_2' with $w = w_1 \cdot w_2' \cdot a$, $q_1 \in [\![w_1]\!]_{\mathcal{A}_1}$ and $q_2' \in [\![w_2']\!]_{\mathcal{A}_2}$. Hence $q_2 \in [\![w_2' \cdot a]\!]_{\mathcal{A}_2}$, and so the choice of w_1 and $w_2 = w_2' \cdot a$ fulfils the requirements.

In the latter case, it must be that:

- $(q_1, q_2) \in \varepsilon\text{-}\mathsf{closure}((q_1, e_2))$, and hence $q_2 \in \varepsilon\text{-}\mathsf{closure}_2(e_2)$, and $q_2 \in\in [\![\varnothing]\!]_{\mathcal{A}_2}$;
- $(q_1, e_2) \in f^{\varepsilon}((q_1, 0))$; and
- $(q_1, 0) \in [\![w]\!]_{\mathcal{A}}$, and so $q_1 \in [\![w]\!]_{\mathcal{A}_1}$ (by Lemma 41).

Therefore, the choice of $w_1 = w$ and $w_2 = \varnothing$ fulfils the requirements.

$\Longleftarrow$:

Base case: $w = \varnothing$. In this case $w_1 = \varnothing$ and $w_2 = \varnothing$. By Lemmma 41, $(q_1, 0) \in [\![\varnothing]\!]_{\mathcal{A}}$, and so, since $(q_1, e_2) \in f^{\varepsilon}((q_1, 0))$, $(q_1, e_2) \in [\![\varnothing]\!]_{\mathcal{A}}$. Now, it must be the case that $q_2 \in \varepsilon\text{-}\mathsf{closure}_2(e_2)$ and hence $(q_1, q_2) \in [\![\varnothing]\!]_{\mathcal{A}}$ as required.

Inductive case: $w = w' \cdot a$ (for some $w' \in \Omega^*$, $a \in \Omega$). Here, either $w_2 = w_2' \cdot a$, or $w_2 = \varnothing$ and $w_1 = w = w' \cdot a$.

In the former case, $q_2 \in (f_2^a \fatsemi \varepsilon\text{-}\mathsf{closure}_2)(q_2')$ for some q_2' with $q_2' \in [\![w_2']\!]_{\mathcal{A}_2}$. Hence, by the inductive hypothesis, $(q_1, q_2') \in [\![w_1 \cdot w_2']\!]_{\mathcal{A}}$. By the definition of $\mathcal{A}$, it follows that $(q_1, q_2) \in [\![w_1 \cdot w_2]\!]_{\mathcal{A}}$ as required.

In the latter case, $q_1 \in [\![w]\!]_{\mathcal{A}_1}$ and so, by Lemma 41, $(q_1, 0) \in [\![w]\!]_{\mathcal{A}}$. It follows then that $(q_1, e_2) \in [\![w]\!]_{\mathcal{A}}$. Further, since $q_2 \in [\![\varnothing]\!]_{\mathcal{A}_2} = \varepsilon\text{-}\mathsf{closure}_2(e_2)$, it follows that $(q_1, q_2) \in [\![w]\!]_{\mathcal{A}}$, as required. $\square$

Lemma 43. *For all $w \in \Omega^*$, $q_1, \in Q_1$ and $q_2 \in Q_2$,*

$$(q_1, 1) \in [\![w]\!]_{\mathcal{A}}$$
$$\iff$$

there exist $w_1, w_2 \in \Omega^*$ s.t. $w = w_1 \oslash_x w_2$ and $q_1 \in [\![w_1]\!]_{\mathcal{A}_1}$ and $w_2 \in \mathcal{L}_2$.

Proof. Both directions are by induction on the structure of w.

$\Longrightarrow$:

Base case: $w = \varnothing$. It must be the case that there are some q_1', q_2'', q_2 with:

- $(q_1'', q_2) \in [\![\varnothing]\!]_{\mathcal{A}}$, and hence, by Lemma 42, $q_1'' \in [\![\varnothing]\!]_{\mathcal{A}_1}$ and $q_2 \in [\![\varepsilon]\!]_{\mathcal{A}_2}$;
- $(q_1', 1) \in f^{\varepsilon}((q_1'', q_2))$, and hence $q_1' \in f_1^x(q_1'')$ and $q_2 \in A_2$, so $q_1' \in [\![x]\!]_{\mathcal{A}_2}$ and $\varnothing \in \mathcal{L}_2$; and
- $(q_1, 1) \in \varepsilon\text{-}\mathsf{closure}((q_1', 1))$, and hence $q_1 \in \varepsilon\text{-}\mathsf{closure}_1(q_1')$, so $q_1 \in [\![x]\!]_{\mathcal{A}_2}$.

Thus, $w_1 = x$ and $w_2 = \varnothing$ fit the requirements: $\varepsilon \in x \oslash_x \varnothing$.

Inductive case: $w = w' \cdot a$ (for some $w' \in \Omega^*$, $a \in \Omega$). There must be some q_1' with either:

- $(q_1, 1) \in (f^a \fatsemi \varepsilon\text{-}\mathsf{closure})((q_1', 1))$ and $(q_1', 1) \in [\![w']\!]_{\mathcal{A}}$; or
- $(q_1, 1) \in \varepsilon\text{-}\mathsf{closure}((q_1', 1))$ and $(q_1', 1) \in f^{\varepsilon}((q_1'', q_2))$ for some q_1'', q_2 with $(q_1'', q_2) \in [\![w]\!]_{\mathcal{A}}$.

In the former case, by the inductive hypothesis, there are w_1' and w_2 with $w' \in w_1' \oslash_x w_2$, $q_1' \in [\![w_1']\!]_{\mathcal{A}_1}$, and $w_2' \in \mathcal{L}_2$. By the definition of $\mathcal{A}$, $q_1 \in (f_1^a \fatsemi \varepsilon\text{-}\mathsf{closure}_1)(q_1)$ and so $q_1 \in [\![w_1' \cdot a]\!]_{\mathcal{A}_1}$. Observing that $w \in (w_1' \oslash_x w_2) \cdot \{a\} \subseteq (w_1' \cdot a) \oslash_x w_2$, the words $w_1 = w_1' \cdot a$ and w_2 fit the requirements.

In the latter case, by Lemma 42, there are w_1', w_2 with $w = w_1' \cdot w_2$, $q_1'' \in [\![w_1']\!]_{\mathcal{A}_1}$ and $q_2 \in [\![w_2]\!]_{\mathcal{A}_2}$. It follows, by the definition of f^{ε}, that $q_1' \in f_{\mathcal{A}_1}^{w_1' \cdot x}$ and $q_2 \in A_2$. Thus $w_1 = w_1' \cdot x$ and w_2 fit the requirements: $w = w_1' \cdot w_2 \in (w_1' \cdot x) \oslash_x w_2$, $q_1 \in \varepsilon\text{-}\mathsf{closure}_1(q_1') \subseteq [\![w_1']\!]_{\mathcal{A}_1}$ and $w_2 \in \mathcal{L}_2$.

$\Longleftarrow$:

Base case: $w = \varnothing$. In this case, $w_1 = x$ and $w_2 = \varnothing$. Since $q_1 \in [\![w_1]\!]_{\mathcal{A}_1}$, it follows that $q_1 \in (f_1^x \fatsemi \varepsilon\text{-}\mathsf{closure}_1)(q_1')$ for some $q_1' \in [\![\varnothing]\!]_{\mathcal{A}_1}$. Hence, since $w_2 \in \mathcal{L}_2$ there is a $q_2 \in [\![\varnothing]\!]_{\mathcal{A}_2} \cap A_2$, and so, by Lemma 42, $(q_1', q_2) \in [\![\varnothing]\!]_{\mathcal{A}}$. Thus, by the definition of f^{ε}, $(q_1, 1) \in \varepsilon\text{-}\mathsf{closure}((q_1', q_2))$ and so $(q_1, 1) \in [\![w]\!]_{\mathcal{A}}$ as required.

Inductive case: $w = w' \cdot a$ (for some $w' \in \Omega^*$, $a \in \Omega$). Either:

- $w = w_1'' \cdot w_2$ and $w_1 = w_1'' \cdot x$ for some $w_1'' \in \Omega^*$; or
- $w' \in w_1' \oslash_x w_2$ and $w_1 = w_1' \cdot a$ for some $w_1' \in \Omega^*$.

In the former case, there is a $q_1'' \in [\![w_1'']\!]_{\mathcal{A}_1}$ such that $q_1 \in (f_1^x \fatsemi \varepsilon\text{-}\mathsf{closure}_1)(q_1'')$, and a $q_1 \in [\![w_2]\!]_{\mathcal{A}_2} \cap A_2$. By Lemma 42, $(q_1'', q_2) \in [\![w_1'' \cdot w_2]\!]_{\mathcal{A}} = [\![w]\!]_{\mathcal{A}}$. It follows, using the definition of f^ε, that $(q_1, 1) \in \varepsilon\text{-}\mathsf{closure}((q_1'', q_2))$, and hence $(q_1, 1) \in [\![w]\!]_{\mathcal{A}}$, as required.

In the latter case, there is a $q_1' \in [\![w_1']\!]_{\mathcal{A}_1}$ such that $q_1 \in (f_1^a \fatsemi \varepsilon\text{-}\mathsf{closure}_1)(q_1')$. By the inductive hypothesis, $(q_1', 1) \in [\![w']\!]_{\mathcal{A}}$. By the definition $\mathcal{A}$, $(q_1, 1) \in (f^a \fatsemi \varepsilon\text{-}\mathsf{closure})((q_1', 1))$, and hence $(q_1, 1) \in [\![w]\!]_{\mathcal{A}}$ as required. □

Proposition 44 (Correctness of $\oslash$ Construction)**.** *The automaton $\mathcal{A} = \mathcal{A}_1 \oslash_x \mathcal{A}_2$ accepts the language $\mathcal{L}_1 \oslash_x \mathcal{L}_2$.*

Proof.

$$
\begin{aligned}
& w \in \mathcal{L}_1 \oslash_x \mathcal{L}_2 \\
\iff\ & \text{there exist } w_1, w_2 \in \Omega^* \text{ s.t. } w \in w_1 \oslash_x w_2 \\
& \qquad\qquad \text{and } w_1 \in \mathcal{L}_1 \text{ and } w_2 \in \mathcal{L}_2 \\
\text{(L. 43)}\quad \iff\ & \text{there exists } q_1 \in A_1 \text{ s.t. } (q_1, 1) \in [\![w]\!]_{\mathcal{A}} \\
\iff\ & A \cap [\![w]\!]_{\mathcal{A}} \neq \emptyset.
\end{aligned}
$$

□

Definition 4.16 ($\oslash\!\!-^{\exists}$ Construction)**.** Given ε-NFA

$$
\begin{aligned}
\mathcal{A}_1 &= (Q_1, e_1, \{f_1^a\}_{a \in \Omega \cup \{\varepsilon\}}, A_1) \text{ and} \\
\mathcal{A}_2 &= (Q_2, e_2, \{f_2^a\}_{a \in \Omega \cup \{\varepsilon\}}, A_2),
\end{aligned}
$$

the ε-NFA

$$
\mathcal{A}_1 \oslash\!\!-_x^{\exists} \mathcal{A}_2 = (Q, e, \{f^a\}_{a \in \Omega \cup \{\varepsilon\}}, A)
$$

is defined as follows:

- $Q = \mathcal{P}(Q_2 \times Q_2)$;
- $e = \varepsilon\text{-}\mathsf{closure}_2$;
- for $a \in \Omega$, $f^a(q) = \{q \fatsemi f_2^a \fatsemi \varepsilon\text{-}\mathsf{closure}_2\}$;
- $f^\varepsilon = \emptyset$; and
- $q \in A$ if and only if there exists $w' \in \Omega^*$ s.t. $[\![w']\!]_{\hat{\mathcal{A}}_1 \times \hat{\mathcal{A}}_q} \cap A_1 \times A_q \neq \emptyset$, where
  - $\mathcal{A}_q = (Q_2 \times \{0,1\}, (e_2, 0), \{f_q^a\}_{a \in \Omega \cup \{\varepsilon\}}, A_q)$,

– for $a \neq x$, $f_q^a = \{((q_2, n), (q_2', n)) \mid (q_2, q_2') \in f_2^a \text{ and } n \in \{0, 1\}\}$,

– $f_q^x = \{((q_2, n), (q_2', n)) \mid (q_2, q_2') \in f_2^x \text{ and } n \in \{0, 1\}\} \cup \{((q_2, 0), (q_2', 1)) \mid (q_2, q_2') \in q\}$,

– $A_q = A_2 \times \{1\}$.

Consider the automaton $\mathcal{A} = \mathcal{A}_1 \mathbin{\bigcirc\!\!\!\!-}_x^{\exists} \mathcal{A}_2$, where $\mathcal{A}_1$ and $\mathcal{A}_2$ are automata accepting languages $\mathcal{L}_1$ and $\mathcal{L}_2$ respectively. This automaton is deterministic, and the state on reading a word w is a relation expressing the effect of reading w in $\mathcal{A}_2$ from any given state. This is, $[\![w]\!]_{\mathcal{A}} = \{(\!|w|\!)_{\mathcal{A}_2}\}$. The automaton $\mathcal{A}_q$, where $q = (\!|w|\!)_{\mathcal{A}_2}$ accepts a word w' if and only if $(w' \mathbin{\bigcirc\!\!\!\!-}_x w) \cap \mathcal{L}_2 \neq \emptyset$. This is since, to reach a state of the form $(q_2, 1)$, the automaton must at some point read x and make a transition in the first component of the state equivalent to reading the word w. The condition that $[\![w']\!]_{\hat{\mathcal{A}}_1 \times \hat{\mathcal{A}}_q} \cap A_1 \times A_q \neq \emptyset$ is then equivalent to the condition that $w' \in \mathcal{L}_1$ and $(w' \mathbin{\bigcirc\!\!\!\!-}_x w) \cap \mathcal{L}_2 \neq \emptyset$. The existence of such a w' can be decided, for each q, by reachability in the pre-automaton $\mathcal{A}_1 \times \mathcal{A}_q$.

The correctness argument for the above construction is formalised in the following lemmata.

Lemma 45. *For all $w \in \Omega^*$,*

$$[\![w]\!]_{\mathcal{A}} = \{(\!|w|\!)_{\mathcal{A}_2}\}\,.$$

Proof. The proof is by induction on the structure of w. Note that, since $f^{\varepsilon} = \emptyset$, ε-closure is the identity relation.

Base case: $w = \varnothing$.

$$\begin{aligned}
[\![\varnothing]\!]_{\mathcal{A}} &= \varepsilon\text{-closure}(e) \\
&= \varepsilon\text{-closure}(\varepsilon\text{-closure}_2) \\
&= \{\varepsilon\text{-closure}_2\} \\
&= \{(\!|\varnothing|\!)_{\mathcal{A}_2}\}\,.
\end{aligned}$$

Inductive case: $w = w' \cdot a$ (for some $w' \in \Omega^*$, $a \in \Omega$).

$$\begin{aligned}
[\![w \cdot a]\!]_{\mathcal{A}} &= \{q \mid \text{there exists } q' \in [\![w']\!]_{\mathcal{A}} \text{ s.t. } q \in (f^a \fatsemi \varepsilon\text{-closure})(q')\} \\
\text{(IH)} \qquad &= \{q \mid q \in (f^a \fatsemi \varepsilon\text{-closure})((\!|w'|\!)_{\mathcal{A}_2})\} \\
&= f^a((\!|w'|\!)_{\mathcal{A}_2}) \\
&= \{(\!|w'|\!)_{\mathcal{A}_2} \fatsemi f_2^a \fatsemi \varepsilon\text{-closure}_2\} \\
&= \{(\!|w' \cdot a|\!)_{\mathcal{A}_2}\}\,.
\end{aligned}$$

□

For $q \in Q$, let $\mathcal{A}_q$ be as given in Definition 4.16.

Lemma 46. *Suppose that $q = (\!|w|\!)_{\mathcal{A}_2}$ for some $w \in \Omega^*$. Then for $w_1 \in \Omega^*$, and $q_2 \in Q_2$,*

$$(q_2, 1) \in [\![w_1]\!]_{\mathcal{A}_q}$$
$$\iff$$
there exists $w_2 \in \Omega^*$ s.t. $q_2 \in [\![w_2]\!]_{\mathcal{A}_2}$ and $w_2 \in w_1 \oslash_x w$.

Proof. $\Longrightarrow$:
Supposing that $(q_2, 1) \in [\![w_1]\!]_{\mathcal{A}_q}$, it must be the case that there exist $q_2', q_2'' \in Q_2$ and $w_1', w_1'' \in \Omega^*$ such that

$$\begin{aligned} w_1 &= w_1'' \cdot x \cdot w_1' \\ (q_2, 1) &\in (\!|w_1'|\!)_{\mathcal{A}_q}((q_2', 1)) \\ (q_2', 1) &\in f_q^x((q_2'', 0)) \\ (q_2'', 0) &\in [\![w_1'']\!]_{\mathcal{A}_q}. \end{aligned}$$

By the definition of $\mathcal{A}_q$, it follows that $q_2'' \in [\![w_1'']\!]_{\mathcal{A}_2}$. Similarly, $q_2' \in q(q_2'') = (\!|w|\!)_{\mathcal{A}_2}(q_2'')$ and so $q_2' \in [\![w_1'' \cdot w]\!]_{\mathcal{A}_2}$. Furthermore, $q_2 \in (\!|w_1'|\!)_{\mathcal{A}_2}(q_1')$ and so $q_2 \in [\![w_1'' \cdot w \cdot w_1']\!]_{\mathcal{A}_2}$. Taking $w_2 = w_1'' \cdot w \cdot w_1'$, gives both $q_1 \in [\![w_2]\!]_{\mathcal{A}_2}$ and $w_2 \in w_1 \oslash_x w$, as required.

$\Longleftarrow$:
Supposing that $w_2 \in \Omega^*$ is some word such that $q_2 \in [\![w_2]\!]_{\mathcal{A}_2}$ and $w_2 \in w_1 \oslash_x w$, it must be the case that there exist $q_2', q_2'' \in Q_2$ and $w_2', w_2'' \in \Omega^*$ such that

$$\begin{aligned} w_1 &= w_1'' \cdot x \cdot w_1' \\ w_2 &= w_1'' \cdot w \cdot w_1' \\ q_2 &\in (\!|w_1'|\!)_{\mathcal{A}_2}(q_2') \\ q_2' &\in (\!|w|\!)_{\mathcal{A}_2}(q_2'') \\ q_2'' &\in [\![w_1'']\!]_{\mathcal{A}_2}. \end{aligned}$$

It follows from the definiton of $\mathcal{A}_q$ that $(q_2'', 0) \in [\![w_1'']\!]_{\mathcal{A}_q}$. Similarly, $(q_2', 1) \in f_q^x((q_2'', 0))$ and so $(q_1', 1) \in [\![w_1'' \cdot x]\!]_{\mathcal{A}_q}$. Furthermore, $(q_2, 1) \in (\!|w_1'|\!)_{\mathcal{A}_q}(q_2')$ and so $(q_2, 1) \in [\![w_1'' \cdot x \cdot w_2']\!]_{\mathcal{A}_q} = [\![w_1]\!]_{\mathcal{A}_q}$, as required. □

Proposition 47 (Correctness of $\oslash\!\!-^{\exists}$ Construction)**.** *The automaton $\mathcal{A} = \mathcal{A}_1 \oslash\!\!-_x^{\exists} \mathcal{A}_2$ accepts the language $\mathcal{L}_1 \oslash\!\!-_x^{\exists} \mathcal{L}_2$.*

Proof.

$$
\begin{aligned}
& w \in \mathcal{L}_1 \mathbin{\oslash\!\!-\!\!\!\!^{\exists}_{x}} \mathcal{L}_2 \\
\iff\ & \text{there exist } w_1, w_2 \in \Omega^* \text{ s.t. } w_1 \in \mathcal{L}_1 \text{ and} \\
& \qquad w_2 \in \mathcal{L}_2 \text{ and } w_2 \in w_1 \oslash_x w \\
\iff\ & \text{there exist } w_1, w_2 \in \Omega^*, q_2 \in Q_2 \text{ s.t. } w_1 \in \mathcal{L}_1 \text{ and} \\
& \qquad q_2 \in [\![w_2]\!]_{\mathcal{A}_2} \text{ and } q_2 \in A_2 \text{ and } w_2 \in w_1 \oslash_x w \\
\iff\ & \text{there exists } q \in Q \text{ s.t. } q = (\!|w|\!)_{\mathcal{A}_2} \text{ and} \\
& \qquad \text{there exist } w_1, w_2 \in \Omega^*, q_2 \in Q_2 \text{ s.t. } w_1 \in \mathcal{L}_1 \text{ and} \\
& \qquad q_2 \in [\![w_2]\!]_{\mathcal{A}_2} \text{ and } q_2 \in A_2 \text{ and } w_2 \in w_1 \oslash_x w \\
\iff\ & \text{there exists } q \in Q \text{ s.t. } q = (\!|w|\!)_{\mathcal{A}_2} \text{ and} \\
& \qquad \text{there exist } w_1 \in \Omega^*, q_2 \in Q_2 \text{ s.t. } w_1 \in \mathcal{L}_1 \text{ and} \\
& \qquad (q_2, 1) \in [\![w_1]\!]_{\mathcal{A}_q} \text{ and } (q_2, 1) \in A_q \\
\iff\ & \text{there exists } q \in Q \text{ s.t. } q = (\!|w|\!)_{\mathcal{A}_2} \text{ and} \\
& \qquad \text{there exists } w_1 \in \Omega^* \text{ s.t. } [\![w_1]\!]_{\mathcal{A}_1} \cap A_1 \neq \emptyset \text{ and} \\
& \qquad [\![w_1]\!]_{\mathcal{A}_q} \cap A_q \neq \emptyset \\
\iff\ & \text{there exists } q \in Q \text{ s.t. } q = (\!|w|\!)_{\mathcal{A}_2} \text{ and } q \in A \\
\iff\ & \text{there exists } q \in Q \text{ s.t. } q = [\![w]\!]_{\mathcal{A}} \text{ and } q \in A \\
\iff\ & [\![w]\!]_{\mathcal{A}} \cap A \neq \emptyset.
\end{aligned}
$$

□

Definition 4.17 ($-\!\!\oslash^{\exists}$ Construction)**.** Given ε-NFA

$$
\begin{aligned}
\mathcal{A}_1 &= (Q_1, e_1, \{f_1^a\}_{a \in \Omega \cup \{\varepsilon\}}, A_1) \text{ and} \\
\mathcal{A}_2 &= (Q_2, e_2, \{f_2^a\}_{a \in \Omega \cup \{\varepsilon\}}, A_2),
\end{aligned}
$$

the ε-NFA

$$
\mathcal{A}_1 \mathbin{-\!\!\oslash^{\exists}_{x}} \mathcal{A}_2 = (Q, e, \{f^a\}_{a \in \Omega \cup \{\varepsilon\}}, A)
$$

is defined as follows:

- $Q = Q_2 \times \{0, 1\})$;
- $e = (e_2, 0)$;
- for $a \in \Omega \cup \{\varepsilon\}$, f^a is the smallest relation satisfying:
  - $(q_2', n) \in f^a((q_2, n))$ whenever $q_2' \in f_2^a(q_2)$ and for $n \in \{0, 1\}$, and

  - if $a = x$ then $(q_2', 1) \in f^a((q_2, 0))$ whenever $q_2' \in (\!|w'|\!)_{\mathcal{A}_2}(q_2)$ for some $w' \in \mathcal{L}_1$; and

- $A = A_2 \times \{1\}$.

Consider the automaton $\mathcal{A} = \mathcal{A}_1 \multimap\!\!\otimes_x^{\exists} \mathcal{A}_2$, where $\mathcal{A}_1$ and $\mathcal{A}_2$ are automata accepting languages $\mathcal{L}_1$ and $\mathcal{L}_2$ respectively. When this automaton is run on a word, it starts in state $(e_2, 0)$ and proceeds to read the word as $\mathcal{A}_2$ would. Eventually, it may be in state $(q_2, 0)$ having so far read w_2', say, and be about to read the label x. On reading the x, the automaton may transition to the state $(q_2', 1)$ if there is some $w_1 \in \mathcal{L}_1$ with $q_2' \in (\!|w_1|\!)_{\mathcal{A}_2}(q_2)$. At this point, the automaton has consumed $w_2' \cdot x$ and is in state $(q_2', 1)$ where $q_2' \in [\![w_2' \cdot w_1]\!]_{\mathcal{A}_2}$ for some $w_2 \in \mathcal{L}_1$. The automaton then proceeds to read the remainder of the word, call it w_2'', as $\mathcal{A}_2$ would, eventually reaching a state $(q_2'', 1)$, say, where $q_2'' \in [\![w_2' \cdot w_1 \cdot w_2'']\!]_{\mathcal{A}_2}$ for some $w_1 \in \mathcal{L}_1$. If this is an accepting state, that signifies that the automaton has read $w_2' \cdot x \cdot w_2''$ for some w_2', w_2'' with $w_2' \cdot w_1 \cdot w_2'' \in \mathcal{L}_2$ for some $w_1 \in \mathcal{L}_1$.

In order for the construction of $\mathcal{A}$ to be effective, it must be possible to determine whether there is some $w_1 \in \mathcal{L}_1$ with $q_2' \in [\![w_1]\!]_{\mathcal{A}_2}(q_2)$ for any given $q_2, q_2' \in Q_2$. This may be done by considering the product pre-automaton $\hat{\mathcal{A}}_1 \times \hat{\mathcal{A}}_2$. Since this pre-automaton behaves like $\mathcal{A}_1$ and $\mathcal{A}_2$ run in parallel, there is a path in $\hat{\mathcal{A}}_1 \times \hat{\mathcal{A}}_2$ from state (e_1, q_2) to state (q_1, q_2'), for some accepting $q_1' \in A_1$, if and only if there is a some $w_1 \in \mathcal{L}_1$ with $q_2' \in (\!|w_1|\!)_{\mathcal{A}_2}(q_2)$. Since automata are finite, determining the existence of such a path is decidable, and so the construction is effective.

Lemma 48. *For all $w \in \Omega^*$ and $q_2 \in Q_2$,*

$$(q_2, 1) \in [\![w]\!]_{\mathcal{A}}$$
$$\iff$$
there exist $w_1, w_2 \in \Omega^*$ s.t. $w_2 \in \mathcal{L}_1$ and $q_2 \in [\![w_2]\!]_{\mathcal{A}_2}$ and $w_2 \in w \otimes_x w_1$.

Proof. $\implies$:
Supposing that $(q_2, 1) \in [\![w]\!]_{\mathcal{A}}$, it must be the case taht there exist $q_2', q_2'' \in Q_2$ and $w', w'' \in \Omega^*$ such that

$$w = w'' \cdot x \cdot w'$$
$$(q_2, 1) \in (\!|w'|\!)_{\mathcal{A}}((q_2', 1))$$
$$(q_2', 1) \in f^x((q_2'', 0))$$
$$(q_2'', 0) \in [\![w'']\!]_{\mathcal{A}}.$$

By the definition of $\mathcal{A}$, it follows that $q_2'' \in [\![w'']\!]_{\mathcal{A}_2}$. Similarly, there exists some $w_1 \in \mathcal{L}_1$ such that $q_2' \in (\!|w_1|\!)_{\mathcal{A}_2}$, and hence $q_1' \in [\![w'' \cdot w_1]\!]_{\mathcal{A}_2}$. Futher, $q_2 \in (\!|w'|\!)_{\mathcal{A}_2}(q_2')$ and so $q_2 \in [\![w'' \cdot w_1 \cdot w']\!]_{\mathcal{A}_2}$. Let $w_2 = w'' \cdot w_1 \cdot w'$. Clearly, $w_2 \in w \oslash_x w_1$. Thus w_1 and w_2 fit the requirements.

$\Longleftarrow$:

Supposing that there are w_1 and w_2 with $w_1 \in \mathcal{L}_1$, $q_2 \in [\![w_2]\!]_{\mathcal{A}_2}$ and $w_2 \in w \oslash_x w_1$, it must be the case that there exist $q_2', q_2'' \in Q_2$ and $w', w'' \in \Omega^*$ such that

$$
\begin{aligned}
w &= w'' \cdot x \cdot w' \\
w_2 &= w'' \cdot w_1 \cdot w' \\
q_2 &\in (\!|w'|\!)_{\mathcal{A}_2}(q_2') \\
q_2' &\in (\!|w_2|\!)_{\mathcal{A}_2}(q_2'') \\
q_2'' &\in [\![w'']\!]_{\mathcal{A}_2}.
\end{aligned}
$$

It follows from the definition of $\mathcal{A}$ that $(q_2'', 0) \in [\![w'']\!]_{\mathcal{A}}$. Similarly, since $w_1 \in \mathcal{L}_1$, $(q_2', 1) \in f^x((q_2'', 0))$ and so $(q_2', 1) \in [\![w'' \cdot x]\!]_{\mathcal{A}}$. Further, $(q_2, 1) \in [\![w'' \cdot xw']\!]_{\mathcal{A}} = [\![w]\!]_{\mathcal{A}}$ as required. □

Proposition 49 (Correctness of $-\!\oslash^{\exists}$ Construction)**.** *The automaton* $\mathcal{A} = \mathcal{A}_1 -\!\oslash_x^{\exists} \mathcal{A}_2$ *accepts the language* $\mathcal{L}_1 -\!\oslash_x^{\exists} \mathcal{L}_2$.

Proof.

$$
\begin{aligned}
& & & w \in \mathcal{L}_1 \oslash_x \mathcal{L}_2 \\
& \iff & & \text{there exist } w_1, w_2 \in \Omega^* \text{ s.t. } w_1 \in \mathcal{L}_1 \text{ and } w_2 \in \mathcal{L}_2 \text{ and} \\
& & & \qquad w_2 \in w \oslash_x w_1 \\
& \iff & & \text{there exists } q_2 \in A_2 \text{ s.t. there exist } w_1, w_2 \in \Omega^* \text{ s.t.} \\
& & & \qquad w_1 \in \mathcal{L}_1 \text{ and } q_2 \in [\![w_2]\!]_{\mathcal{A}_2} \text{ and } w_2 \in w \oslash_x w_1 \\
\text{(L. 48)} \quad & \iff & & \text{there exists } q_2 \in A_2 \text{ s.t. } (q_2, 1) \in [\![w]\!]_{\mathcal{A}} \\
& \iff & & [\![w]\!]_{\mathcal{A}} \cap A \neq \emptyset.
\end{aligned}
$$

□

4.1.5 Decidability

Given automata $\mathcal{A}_\emptyset$, $\mathcal{A}_{\mathsf{C}^m_{\mathsf{Seq}}}$ and $\mathcal{A}_{\{w\}}$ recognising the languages $\emptyset$, $\mathsf{C}^m_{\mathsf{Seq}}$ and $\{w\}$ respectively (for each $w \in \Omega^*$), together with the automaton constructions defined above, a formula $K \in \mathsf{K}^m_{\mathsf{Seq}}$ and environment $\sigma \in \mathsf{LEnv}$ are encoded as

an automaton $\mathcal{A}_{K,\sigma}$ as follows (where $x = \sigma\alpha$):

$$\begin{aligned}
\mathcal{A}_{\mathbf{0},\sigma} &= \mathcal{A}_{\{\varnothing\}} \\
\mathcal{A}_{\mathbf{a},\sigma} &= \mathcal{A}_{\{\mathbf{a}\}} \\
\mathcal{A}_{K_1 \cdot K_2,\sigma} &= (\mathcal{A}_{K_1,\sigma} \cdot \mathcal{A}_{K_2,\sigma}) \cap \mathcal{A}_{\mathsf{C}^m_{\mathsf{Seq}}} \\
\mathcal{A}_{\alpha,\sigma} &= \mathcal{A}_{\{x\}} \\
\mathcal{A}_{K_1 \circ_\alpha K_2,\sigma} &= (\mathcal{A}_{K_1,\sigma} \oslash_x \mathcal{A}_{K_2,\sigma}) \cap \mathcal{A}_{\mathsf{C}^m_{\mathsf{Seq}}} \\
\mathcal{A}_{K_1 \circ\!\!- _\alpha K_2,\sigma} &= \left(\mathcal{A}_{K_1,\sigma} \oslash\!\!\!- \overset{\exists}{x} \mathcal{A}_{K_2,\sigma}\right) \cap \mathcal{A}_{\mathsf{C}^m_{\mathsf{Seq}}} \\
\mathcal{A}_{K_1 -\!\!\circ_\alpha K_2,\sigma} &= \left(\mathcal{A}_{K_1,\sigma} -\!\!\oslash \overset{\exists}{x} \mathcal{A}_{K_2,\sigma}\right) \cap \mathcal{A}_{\mathsf{C}^m_{\mathsf{Seq}}} \\
\mathcal{A}_{\mathrm{False},\sigma} &= \mathcal{A}_{\emptyset} \\
\mathcal{A}_{K_1 \to K_2,\sigma} &= \left(\overline{\mathcal{A}_{K_1,\sigma}} \cap \mathcal{A}_{\mathsf{C}^m_{\mathsf{Seq}}}\right) \cup \mathcal{A}_{K_2,\sigma}.
\end{aligned}$$

Since these constructions accept exactly the languages of sequence contexts that satisfy the corresponding formulae, in order to determine if a context satisfies a formula in a given environment it is enough to check if the context is accepted by the corresponding automaton. Thus, model-checking is decidable.

Theorem 50. *Given sort $\varsigma \in \mathsf{Sort}$, quantifier-free formula $K \in \mathsf{Formula}_\varsigma$, environment $\sigma \in \mathsf{LEnv}$, and multi-holed sequence context $c \in \mathsf{C}^m_{\mathsf{Seq}}$ with $(c, \sigma) \in \mathsf{World}_\varsigma$, it is decidable whether*

$$c, \sigma \models_\varsigma K.$$

Furthermore, satisfiability of a formula (for a given environment) is decidable since reachability can be used to determine if there is any word that is accepted by the corresponding automaton. The automata are constructed so that any such word must be a multi-holed sequence context.

Theorem 51. *Given sort $\varsigma \in \mathsf{Sort}$, quantifier-free formula $K \in \mathsf{Formula}_\varsigma$, and environment $\sigma \in \mathsf{LEnv}$, it is decidable whether there exists a multi-holed sequence context $c \in \mathsf{C}^m_{\mathsf{Seq}}$ with $(c, \sigma) \in \mathsf{World}_\varsigma$ such that*

$$c, \sigma \models_\varsigma K.$$

Furthermore, it is possible to decide whether there is a pair of context and environment which satisfy a given formula. This is since for any environment for which the formula is satisfiable, by environment extendability (Property 2.33) and hole substitution (Property 2.34), the formula is also satisfiable for one of a finite number of environments. (The number of environments that need be considered is the number of ways of partitioning $\mathrm{fv}\, K$.)

Corollary 52. *Given sort* $\varsigma \in \mathsf{Sort}$ *and quantifier-free formula* $K \in \mathsf{Formula}_\varsigma$*, it is decidable whether there exists a pair of multi-holed sequence context* $c \in \mathsf{C}^m_{\mathsf{Seq}}$ *and environment* $\sigma \in \mathsf{LEnv}$ *with* $(c, \sigma) \in \mathsf{World}_\varsigma$ *such that*

$$c, \sigma \models_\varsigma K.$$

4.2 Terms

The theory of automata can be adapted to accommodate more elaborately-structured forms of data, such as terms. In the literature on the subject, terms as I have defined them are known as (ranked) trees, and a comprehensive treatment of automata for such trees can be found in [GS97] and [CDG+07].

In the following, assume that multi-holed term contexts, $\mathsf{C}^m_{\mathsf{Term}}$ are defined over finite Υ and X. Let $\Omega = \Upsilon \cup (\mathrm{X} \times \{0\})$, and let Term_Ω, ranged over by $r, r', r_1, \ldots$, be the set of terms on Ω (that is, finite, ranked, ordered trees labelled from the ranked alphabet Ω). Note that $\mathsf{C}^m_{\mathsf{Term}} \subseteq \mathsf{Term}_\Omega$, since holes are nodes with rank 0, labelled from X.

4.2.1 Automata

Term automata generalise word automata in the following sense: whereas each label in a word is preceded by a single word, each label in a term is preceded by a number of terms corresponding to its rank (its children). Thus, just as a word automaton assigns states to a word based on the transition relation corresponding to the last label and the states that are assigned to the word preceding the label, a term automaton assigns states to a term based on the transition relation corresponding to the topmost label and the states that are assigned to the immediate children of that node. This form of automaton is known as a bottom-up of frontier-to-root automaton. (Top-down automata also exist, but I do not make use of them here.)

Definition 4.18 (ε-NFTA)**.** A *non-deterministic finite term automaton with ε-transitions* (abbreviated ε-NFTA) is a tuple $\mathcal{A} = (Q, \{f^a\}_{a \in \Omega \cup \{\varepsilon\}}, A)$ where:

- Q is the set of states, a finite set;
- for each $(a, n) \in \Omega$, $f^{(a,n)} \subseteq Q^n \times Q$ is the $n+1$-ary state transition relation for a;
- $f^\varepsilon \subseteq Q \times Q$ is the non-consuming state transition relation; and
- $A \subseteq Q$ is the set of accepting states.

To formally define the language recognised by an automaton, I make some auxiliary definitions. The definition of ε-closure for ε-NFTA is exactly as in Definition 4.3 for ε-NFA.

Definition 4.19 (Automaton-induced Mapping)**.** A term automaton $\mathcal{A}$ induces a function $[\![(\cdot)]\!]_{\mathcal{A}} : \mathsf{Term}_\Omega \to \mathcal{P}(Q)$ that maps terms $r \in \mathsf{Term}_\Omega$ to sets of states $[\![r]\!]_{\mathcal{A}} \subseteq Q$ according to the following definition:

$$[\![a(r_1, \ldots, r_n)]\!]_{\mathcal{A}} = \left\{ q \in Q \;\middle|\; \begin{array}{r} \text{there exist } q_1 \in [\![r_1]\!]_{\mathcal{A}}, \ldots, q_n \in [\![r_n]\!]_{\mathcal{A}} \text{ s.t.} \\ ((q_1, \ldots, q_n), q) \in (f^a \fatsemi \varepsilon\text{-closure}) \end{array} \right\}.$$

Definition 4.20 (Acceptance)**.** A term r is said to be *accepted* by automaton $\mathcal{A}$ if $[\![r]\!]_{\mathcal{A}} \cap A \neq \emptyset$. The *term language accepted by* $\mathcal{A}$, $\mathcal{L}_{\mathcal{A}}$, is the set of terms accepted by $\mathcal{A}$:

$$\mathcal{L}_{\mathcal{A}} = \{ r \in \mathsf{Term}_\Omega \mid [\![r]\!]_{\mathcal{A}} \cap A \neq \emptyset \}.$$

The class of languages that can be recognised by term automata is the class of *regular term languages*[3]. This class is closed under a number of operations; if $\mathcal{L}, \mathcal{L}', \mathcal{L}_1, \ldots, \mathcal{L}_n$ are regular term languages, then so are

- $\emptyset$, $\mathsf{C}^m_{\mathsf{Term}}$, Term_Ω,
- $\mathcal{L} \cup \mathcal{L}'$, $\mathcal{L} \cap \mathcal{L}'$, $\overline{\mathcal{L}} \stackrel{\text{def}}{=} \mathsf{Term}_\Omega \setminus \mathcal{L}$,
- $\{a(r_1, \ldots, r_n) \mid r_i \in \mathcal{L}_i\}$ for each $(a, n) \in \Omega$,
- $\mathcal{L} \circledcirc_x \mathcal{L}'$ for each $x \in \mathrm{X}$ — the set of terms obtained by replacing each occurrence of x in terms from $\mathcal{L}$ by (possibly distinct) terms from $\mathcal{L}'$, and
- $\mathcal{L} \mathbin{-\!\!\circledcirc^{\exists}_x} \mathcal{L}' \stackrel{\text{def}}{=} \{r \in \mathsf{Term}_\Omega \mid (\{r\} \circledcirc_x \mathcal{L}) \cap \mathcal{L}' \neq \emptyset\}$ for each $x \in \mathrm{X}$.

For details, consult [GS97].[4] These closure properties imply that it is possible to construct automata corresponding to most of the context logic connectives. In particular, $\circledcirc$ and $\mathbin{-\!\!\circledcirc^{\exists}}$ can be used to implement $\circ$ and $\mathbin{-\!\!\circ^{\exists}}$ respectively. However, for consistency with my presentation for sequences and unranked trees I present the constructions for $\oslash$, $\oslash\!\!-^{\exists}$ and $\mathbin{-\!\!\oslash^{\exists}}$ for term automata.

4.2.2 Constructions

Definition 4.21 ($\oslash$ Construction)**.** Given ε-NFTA

$$\mathcal{A}_1 = (Q_1, \{f_1^a\}_{a \in \Omega \cup \{\varepsilon\}}, A_1) \text{ and}$$
$$\mathcal{A}_1 = (Q_2, \{f_2^a\}_{a \in \Omega \cup \{\varepsilon\}}, A_2),$$

[3] In the literature, this is more commonly *regular tree languages*

[4] In [GS97], $\mathcal{L}_1 \circledcirc_x \mathcal{L}_2$ is denoted $\mathcal{L}_2 \cdot_x \mathcal{L}_1$ and called the x-product of $\mathcal{L}_2$ and $\mathcal{L}_1$, while $\mathcal{L}_1 \mathbin{-\!\!\circledcirc^{\exists}_x} \mathcal{L}_2$ is denoted $\mathcal{L}_2^{-x}\mathcal{L}_1$ and called the x-quotient of $\mathcal{L}_1$ by $\mathcal{L}_2$.

the ε-NFTA

$$\mathcal{A}_1 \oslash_x \mathcal{A}_2 = (Q, \{f^a\}_{a \in \Omega \cup \{\varepsilon\}}, A)$$

is defined as follows:

- $Q = (Q_1 \times \{0,1\}) \cup Q_2$;
- for $(a,m) \in \Omega$, $f^{(a,m)}$ is the smallest relation satisfying:
  - $(q_1', n) \in f^{(a,m)}((q_{1,1}, n_1), \ldots, (q_{1,m}, n_m))$ whenever $q_1' \in f_1^{(a,m)}(q_{1,1}, \ldots, q_{1,m})$ and $n = n_1 + \cdots + n_m$, and
  - $q_2' \in f^{(a,m)}(q_{2,1}, \ldots, q_{2,m})$ whenever $q_2' \in f_2^{(a,m)}(q_{2,1}, \ldots, q_{2,m})$;
- f^ε is the smallest relation satisfying:
  - $(q_1', n) \in f^\varepsilon((q_1, n))$ whenever $q_1' \in f_1^\varepsilon(q_1)$,
  - $q_2' \in f^\varepsilon(q_2)$ whenever $q_2' \in f_2^\varepsilon(q_2)$, and
  - $(q_1, 1) \in f^\varepsilon(q_2)$ whenever $q_1 \in f_1^{(x,0)}$ and $q_2 \in A_2$; and
- $A = A_1 \times \{1\}$.

Consider the automaton $\mathcal{A} = \mathcal{A}_1 \oslash_x \mathcal{A}_2$, where $\mathcal{A}_1$ and $\mathcal{A}_2$ are automata accepting languages $\mathcal{L}_1$ and $\mathcal{L}_2$ respectively. This construction of $\mathcal{A}$ is similar to the analogous construction for sequences. The main difference is that states from Q_2 are not paired with states from Q_1. The reason for this is that $f^{(x,0)}$ is a unary relation (*i.e.* a set) — the result of consuming an x-labelled node does not depend on the context in which it appears. In the sequence case, on the other hand, f^x is a binary relation — the result of consuming an x-labelled node depends on the state before the x was consumed.

On consuming part of a term, a state of $q_2 \in Q_2$ is possible if and only if that state is possible on consuming the same part of the term in $\mathcal{A}_2$. A state of $(q_1, 0)$ is possible if and only if the state q_1 is possible on consuming the same part of the term in $\mathcal{A}_1$. A state of $(q_1, 1)$ is possible if and only if there is a term r with $q_1 \in [\![r]\!]_{\mathcal{A}_1}$ and the consumed term belongs to the set $\{r\} \oslash_x \mathcal{L}_2$ — that is, the 1 indicates that exactly one occurrence of x in r has been substituted by a term accepted by $\mathcal{A}_2$. The 1 propagates up the term as it is consumed, and can only occur in one branch at each level. It is initially introduced by an ε-transition from a state $q_2 \in A_2$ to a state $(q_1, 1)$ where $q_1 \in f_1^{(x,0)}$ — that is, the subterm (which must belong to $\mathcal{L}_2$) is treated as an x in $\mathcal{A}_1$, but is tagged with a 1 to indicate that a substitution has occurred. A term is finally accepted

if it is the result of substituting one x in a term from $\mathcal{L}_1$ (*i.e.* that is accepted by $\mathcal{A}_1$) with a term from $\mathcal{L}_2$.

The above correctness argument is formalised in the following lemmata.

Lemma 53. *For all* $r \in \mathsf{Term}_\Omega$, $q_2 \in Q_2$,

$$q_2 \in [\![r]\!]_{\mathcal{A}} \iff q_2 \in [\![r]\!]_{\mathcal{A}_2}.$$

Proof. The proof is by induction of the structure of the term r.

Base case: $r = a$ for some $(a, 0) \in \Omega$.

$$\begin{aligned}
& q_2 \in [\![r]\!]_{\mathcal{A}} \\
\iff\ & \text{there exists } q_2' \in Q_2 \text{ s.t. } q_2' \in f^a \text{ and } q_2 \in \varepsilon\text{-}\mathsf{closure}(q_2') \\
\iff\ & \text{there exists } q_2' \in Q_2 \text{ s.t. } q_2' \in f_2^a \text{ and } q_2 \in \varepsilon\text{-}\mathsf{closure}_2(q_2') \\
\iff\ & q_2 \in [\![r]\!]_{\mathcal{A}_2}.
\end{aligned}$$

Inductive case: $r = a(r_1, \ldots, r_m)$ for some $(a, m) \in \Omega$ and $r_1, \ldots, r_m \in \mathsf{Term}_\Omega$.

$$\begin{aligned}
& q_2 \in [\![r]\!]_{\mathcal{A}} \\
\iff\ & \text{there exist } q_2', q_{2,1}, \ldots, q_{2,m} \in Q_2 \text{ s.t.} \\
& \quad q_2' \in f^a(q_{2,1}, \ldots, q_{2,m}) \text{ and} \\
& \qquad q_{2,1} \in [\![r_1]\!]_{\mathcal{A}} \text{ and } \ldots \text{ and } q_{2,m} \in [\![r_m]\!]_{\mathcal{A}} \text{ and} \\
& \qquad\qquad q_2 \in \varepsilon\text{-}\mathsf{closure}(q_2') \\
\text{(IH)} \iff\ & \text{there exist } q_2', q_{2,1}, \ldots, q_{2,m} \in Q_2 \text{ s.t.} \\
& \quad q_2' \in f^a(q_{2,1}, \ldots, q_{2,m}) \text{ and} \\
& \qquad q_{2,1} \in [\![r_1]\!]_{\mathcal{A}_2} \text{ and } \ldots \text{ and } q_{2,m} \in [\![r_m]\!]_{\mathcal{A}_2} \text{ and} \\
& \qquad\qquad q_2 \in \varepsilon\text{-}\mathsf{closure}(q_2') \\
\iff\ & \text{there exist } q_2', q_{2,1}, \ldots, q_{2,m} \in Q_2 \text{ s.t.} \\
& \quad q_2' \in f_2^a(q_{2,1}, \ldots, q_{2,m}) \text{ and} \\
& \qquad q_{2,1} \in [\![r_1]\!]_{\mathcal{A}_2} \text{ and } \ldots \text{ and } q_{2,m} \in [\![r_m]\!]_{\mathcal{A}_2} \text{ and} \\
& \qquad\qquad q_2 \in \varepsilon\text{-}\mathsf{closure}_2(q_2') \\
\iff\ & q_2 \in [\![r]\!]_{\mathcal{A}_2}.
\end{aligned}$$

□

Lemma 54. *For all* $r \in \mathsf{Term}_\Omega$, $q_1 \in Q_1$,

$$(q_1, 0) \in [\![r]\!]_{\mathcal{A}} \iff q_1 \in [\![r]\!]_{\mathcal{A}_1}.$$

The proof of this lemma is essentially the same as for Lemma 53, and so I omit the full details.

Lemma 55. *For all* $r \in \mathsf{Term}_\Omega$, $q_1 \in Q_1$,

$$(q_1, 1) \in [\![r]\!]_{\mathcal{A}}$$
$$\iff$$

there exist $r_1, r_2 \in \mathsf{Term}_\Omega$ *s.t.* $r \in r_1 \oslash_x r_2$ *and* $r_2 \in \mathcal{L}_2$ *and* $q_1 \in [\![r_1]\!]_{\mathcal{A}_1}$.

Proof. The proof is by induction on the structure of the term r.

Base case: $r = a$ for some $(a, 0) \in \Omega$.

$$\begin{aligned}
& (q_1, 1) \in [\![a]\!]_{\mathcal{A}} \\
\iff\ & \text{there exist } q_1' \in Q_1, q_2 \text{ and } q_2' \in Q_2 \text{ s.t.} \\
& \quad (q_1, 1) \in \varepsilon\text{-}\mathsf{closure}((q_1', 1)) \text{ and } (q_1', 1) \in f^\varepsilon(q_2) \text{ and} \\
& \quad\quad q_2 \in \varepsilon\text{-}\mathsf{closure}(q_2') \text{ and } q_2 \in f^a \\
\iff\ & \text{there exist } q_1' \in Q_1, q_2 \text{ and } q_2' \in Q_2 \text{ s.t.} \\
& \quad (q_1, 1) \in \varepsilon\text{-}\mathsf{closure}_1((q_1', 1)) \text{ and } q_1' \in f_1^{(x,0)} \text{ and} \\
& \quad\quad q_2 \in A_2 \text{ and } q_2 \in \varepsilon\text{-}\mathsf{closure}(q_2') \text{ and } q_2 \in f^a \\
\iff\ & q_1 \in [\![x]\!]_{\mathcal{A}_1} \text{ and } [\![a]\!]_{\mathcal{A}_2} \cap A_2 \neq \emptyset \\
\iff\ & \text{there exist } r_1, r_2 \in \mathsf{Term}_\Omega \text{ s.t.} \\
& \quad a \in r_1 \oslash_x r_2 \text{ and } r_2 \in \mathcal{L}_2 \text{ and } q_1 \in [\![r_1]\!]_{\mathcal{A}_1}.
\end{aligned}$$

Inductive case: $r = a(r^{(1)}, \ldots, r^{(m)})$ for some $(a, m) \in \Omega$ and $r^{(1)}, \ldots, r^{(m)} \in \mathsf{Term}_\Omega$. Consider the implication in each direction separately.

$\Longrightarrow$:

By definition $(q_1, 1) \in \varepsilon\text{-}\mathsf{closure}(q')$ with $q' \in f^a(q^{(1)}, \ldots, q^{(m)})$ for some $q' \in Q$, $q^{(i)} \in [\![r^{(i)}]\!]_{\mathcal{A}}$. From the definition of f^ε, either $q' = (q_1', 1)$ for some $q_1' \in Q_1$, or $q' = q_2$ for some $q_2 \in Q_2$.

In the first case, it must be that for exactly one k, $q^{(k)} = (q_{1,k}, 1)$ and for all $i \neq k$, $q^{(i)} = (q_{1,i}, 0)$. By the inductive hypothesis, there are $r_1', r_2 \in \mathsf{Term}_\Omega$ with $r^{(k)} \in r_1' \oslash_x r_2$, $r_2 \in \mathcal{L}_2$ and $q_{1,k} \in [\![r_1']\!]_{\mathcal{A}_1}$. By Lemma 54, for $i \neq k$, $q_{1,i} \in (\!|r^{(i)}|\!)_{\mathcal{A}_1}$. By definition, $q_1' \in f_1^a(q_{1,1}, \ldots, q_{1,m})$, and so

$$q_1' \in [\![a(r^{(1)}, \ldots, r^{(k-1)}, r_1', r^{(k+1)}, \ldots, r^{(m)})]\!]_{\mathcal{A}_1}.$$

Let

$$r_1 = a(r^{(1)}, \ldots, r^{(k-1)}, r_1', r^{(k+1)}, \ldots, r^{(m)})$$

and observe that $r \in r_1 \oslash_x r_2$. Further, since $(q_1', 1) \in \varepsilon\text{-}\mathsf{closure}((q_1', 1))$, $q_1 \in \varepsilon\text{-}\mathsf{closure}_1(q_1')$. Hence, $q_1 \in [\![r_1]\!]_{\mathcal{A}_1}$, as required.

In the second case, there mus be some $q_1' \in Q_1$ and $q_2' \in Q_2$ such that

$$(q_1, 1) \in \varepsilon\text{-}\mathsf{closure}((q_1', 1))$$
$$(q_1', 1) \in f^{\varepsilon}(q_2')$$
$$q_2' \in \varepsilon\text{-}\mathsf{closure}(q_2).$$

By the definition of f^{ε}, it follows that

$$q_1 \in \varepsilon\text{-}\mathsf{closure}_1(q_1')$$
$$q_1' \in f_1^{(x,0)}$$
$$q_2' \in A_2$$
$$q_2' \in \varepsilon\text{-}\mathsf{closure}_2(q_2).$$

Hence $q_1 \in [\![x]\!]_{\mathcal{A}_1}$. Further, since $q_2 \in [\![r]\!]_{\mathcal{A}_2}$ by Lemma 53, $q_2' \in [\![r]\!]_{\mathcal{A}_2} \cap A_2$. Thus $r \in \mathcal{L}_2$. Let $r_1 = x$ and $r_2 = r$, and observe that $r \in r_1 \oslash_x r_2$, as required.

$\Longleftarrow$:

Either $r_1 = x$ or $r_1 \neq x$; consider each case.

In the first case, $r = r_2 \in \mathcal{L}_2$ and so there is some $q_2 \in [\![r]\!]_{\mathcal{A}_2} \cap A_2$. By Lemma 53, $q_2 \in [\![r]\!]_{\mathcal{A}}$. Also, since $q_1 \in [\![x]\!]_{\mathcal{A}_1}$, there is some $q_1' \in Q_1$ with $q_1' \in f_1^{(x,0)}$ and $q_1 \in \varepsilon\text{-}\mathsf{closure}_1(q_1')$. Hence, by the definition of f^{ε}, $(q_1', 1) \in f^{\varepsilon}(q_2)$ and $(q_1, 1) \in \varepsilon\text{-}\mathsf{closure}((q_1', 1))$. Thus, $(q_1, 1) \in [\![r]\!]_{\mathcal{A}}$ as required.

In the second case, it must be that, for some $(a, m) \in \Omega$, $r_{1,1}, \ldots, r_{1,m}, r' \in \mathsf{Term}_{\Omega}$, and k with $1 \leq k \leq m$,

$$r_1 = a(r_{1,1}, \ldots, r_{1,m})$$
$$r = a(r_{1,1}, \ldots, r_{1,k-1}, r', r_{1,k+1}, \ldots, r_{1,m})$$
$$r' \in r_{1,k} \oslash_x r_2.$$

Since $q_1 \in [\![r_1]\!]_{\mathcal{A}_1}$ it follows that there are $q_1', q_{1,1}, \ldots, q_{1,m} \in Q_1$ with

$$q_1 \in \varepsilon\text{-}\mathsf{closure}_1(q_1')$$
$$q_1' \in f_1^a(q_{1,1}, \ldots, q_{1,m})$$
$$\text{for } 1 \leq i \leq m \quad q_{1,i} \in [\![r_{1,i}]\!]_{\mathcal{A}_1}.$$

Hence, by the inductive hypothesis, $(q_{1,k}, 1) \in [\![r']\!]_{\mathcal{A}}$. By Lemma 53, $(q_{1,i}, 0) \in [\![r_{1,i}]\!]_{\mathcal{A}}$ for $1 \leq i \leq m$. By definition,

$$(q_1', 1) \in f^a((q_{1,1}, 0), \ldots, (q_{1,k-1}, 0), (q_{1,k}, 1), (q_{1,k+1}, 0), \ldots, (q_{1,m}, 0))$$
$$(q_1, 1) \in \varepsilon\text{-}\mathsf{closure}(q_1', 1).$$

Hence, $(q_1, 1) \in [\![r]\!]_{\mathcal{A}}$, as required. □

Proposition 56 (Correctness of $\oslash$ Construction)**.** *The automaton* $\mathcal{A} = \mathcal{A}_1 \oslash_x \mathcal{A}_2$ *accepts the language* $\mathcal{L}_1 \oslash_x \mathcal{L}_2$*.*

Proof.

$$\begin{aligned} & r \in \mathcal{L}_1 \oslash_x \mathcal{L}_2 \\ \iff\ & \text{there exist } r_1, r_2 \in \mathsf{Term}_\Omega \text{ s.t. } r \in r_1 \oslash_x r_2 \\ & \qquad\qquad \text{and } r_1 \in \mathcal{L}_1 \text{ and } r_2 \in \mathcal{L}_2 \\ \text{(L. 55)} \iff\ & \text{there exists } q_1 \in A_1 \text{ s.t. } (q_1, 1) \in [\![w]\!]_{\mathcal{A}} \\ \iff\ & A \cap [\![r]\!]_{\mathcal{A}} \neq \emptyset. \end{aligned}$$

□

Definition 4.22 ($\oslash\!\!-^{\exists}$ Construction)**.** Given ε-NFTA

$$\begin{aligned} \mathcal{A}_1 &= (Q_1, \{f_1^a\}_{a \in \Omega \cup \{\varepsilon\}}, A_1) \text{ and} \\ \mathcal{A}_1 &= (Q_2, \{f_2^a\}_{a \in \Omega \cup \{\varepsilon\}}, A_2), \end{aligned}$$

the ε-NFTA

$$\mathcal{A}_1 \oslash\!\!-_x^{\exists} \mathcal{A}_2 = (Q, \{f^a\}_{a \in \Omega \cup \{\varepsilon\}}, A)$$

is defined as follows:

- $Q = Q_2$;
- for $a \in \Omega \cup \{\varepsilon\}$, $f^a = f_2^a$; and
- $q \in A$ if and only if there exists $r' \in \mathsf{Term}_\Omega$ s.t. $[\![r']\!]_{\hat{\mathcal{A}}_1 \times \hat{\mathcal{A}}_q} \cap (A_1 \times A_q) \neq \emptyset$, where
  - $\mathcal{A}_q = (Q_2 \times \{0,1\}, \{f_q^a\}_{a \in \Omega \cup \{\varepsilon\}}, A_q)$,
  - for $a \in \Omega \cup \{\varepsilon\}$, where a has arity m ($m = 1$ in the case where $a = \varepsilon$) f_q^a is the smallest relation satisfying
    - $(q_2', n) \in f_q^a((q_{2,1}, n_1), \ldots, (q_{2,m}, n_m))$ if $q_2' \in f_2^a(q_{2,1}, \ldots, q_{2,m})$ and $n = \sum_{i=1}^m n_i$, and
    - if $a = (x, 0)$, then $(q, 1) \in f_q^a$, and
  - $A_q = A_2 \times \{1\}$.

Consider the automaton $\mathcal{A} = \mathcal{A}_1 \oslash\!\!-_x^{\exists} \mathcal{A}_2$, where $\mathcal{A}_1$ and $\mathcal{A}_2$ are automata accepting languages $\mathcal{L}_1$ and $\mathcal{L}_2$ respectively. This automaton only differs from $\mathcal{A}_2$ in the accepting set. The accepting set is determined so that r is accepted

exactly when there exists some r' such that $r' \in \mathcal{L}_1$ and $(r' \oslash_x r) \cap \mathcal{L}_2 \neq \emptyset$. The first condition is established by $\mathcal{A}_1$ accepting r', while the second is established by $\mathcal{A}_q$ accepting r', for some $q \in [\![r]\!]_{\mathcal{A}} = [\![r]\!]_{\mathcal{A}_2}$ The automaton $\mathcal{A}_q$ is constructed to behave like $\mathcal{A}_2$, but with exactly one instance of x in any accepting run being treated as a term producing state q. For $q \in [\![r]\!]_{\mathcal{A}_2} = [\![r]\!]_{\mathcal{A}}$, the automaton $\mathcal{A}_q$ accepts a term r' only if $(r' \oslash_x r) \cap \mathcal{L}_2 \neq \emptyset$. Conversely, for a term r', if $(r' \oslash_x r) \cap \mathcal{L}_2 \neq \emptyset$ then there is some $q \in [\![r]\!]_{\mathcal{A}_2}$ such that $[\![r']\!]_{\mathcal{A}_q} \in A_q$. Thus, in order to determine whether $q \in A$ it is sufficient to consider whether any state in $A_1 \times A_q$ is reachable in the product pre-automaton $\hat{\mathcal{A}}_1 \times \hat{\mathcal{A}}_q$. (The pre-automaton product is defined for term automata in a similar fashion as for word automata.)

The above correctness argument is formalised in the following lemmata.

Lemma 57. *For all $r \in \mathsf{Term}_\Omega$, $q_2 \in Q = Q_2$,*

$$q \in [\![r]\!]_{\mathcal{A}} \iff q \in [\![r]\!]_{\mathcal{A}_2}.$$

Proof. By definition. □

For $q \in Q$, let $\mathcal{A}_q$ be given as in Definition 4.22.

Lemma 58. *For all $q \in Q$, for all $r \in \mathsf{Term}_\Omega$, $q_2 \in Q_2$,*

$$(q_2, 0) \in [\![r]\!]_{\mathcal{A}_q} \iff q_2 \in [\![r]\!]_{\mathcal{A}_2}.$$

The proof of this lemma is essentially the same as for Lemma 53.

Lemma 59. *For all $r, r_1 \in \mathsf{Term}_\Omega$, $q_2 \in Q_2$,*

$$\textit{there exists } q \in Q_2 \textit{ s.t. } q \in [\![r]\!]_{\mathcal{A}_2} \textit{ and } (q_2, 1) \in [\![r_1]\!]_{\mathcal{A}_q}$$
$$\iff$$
$$\textit{there exists } r_2 \in \mathsf{Term}_\Omega \textit{ s.t. } r_2 \in r_1 \oslash_x r \textit{ and } q_2 \in [\![r_2]\!]_{\mathcal{A}_2}.$$

Proof. Both directions are by induction on the structure of r_1.

$\Longrightarrow$:

Base case: $r_1 = a$. There must be some $q_2' \in Q_2$ with

$$(q_2, 1) \in \varepsilon\text{-}\mathsf{closure}_q((q_2', 1))$$
$$(q_2', 1) \in f_q^{(a,0)}.$$

The definition of $\mathcal{A}_q$ requires that $r_1 = a = x$, and $q_2' = q$. Let $r_2 = r$. Clearly, $r_2 \in r_1 \oslash_x r$. Also, by definition,

$$q_2' = q \in [\![r]\!]_{\mathcal{A}_2} = [\![r_2]\!]_{\mathcal{A}_2}$$
$$q_2 \in \varepsilon\text{-}\mathsf{closure}_2(q_2'),$$

and so $q_2 \in [\![r_2]\!]_{\mathcal{A}_2}$, as required.

Inductive case: $r_1 = a(r^{(1)}, \ldots, r^{(m)})$ for some $(a, m) \in \Omega$ and $r^{(1)}, \ldots, r^{(m)} \in \mathsf{Term}_\Omega$. Since $(q_2, 1) \in [\![r_1]\!]_{\mathcal{A}_q}$, there must be some $q_2', q_{2,1}, \ldots, q_{2,m} \in Q_2$ and some $1 \leq k \leq m$ with

$$\begin{gathered}\text{for } i \neq k \quad (q_{2,i}, 0) \in [\![r^{(i)}]\!]_{\mathcal{A}_q} \\ (q_{2,k}, 1) \in [\![r^{(k)}]\!]_{\mathcal{A}_q} \\ (q_2', 1) \in f_q^{(a,m)}((q_{2,1}, 0), \ldots, (q_{2,k-1}, 0), (q_{2,k}, 1), (q_{2,k+1}, 0), \ldots, (q_{2,m}, 0)) \\ (q_2, 1) \in \varepsilon\text{-}\mathsf{closure}_q((q_2', 1)).\end{gathered}$$

By the inductive hypothesis, there is some $r_2' \in \mathsf{Term}_\Omega$ such that $q_2' \in r^{(k)} \oslash_x r$ and $q_{2,k} \in [\![r_2']\!]_{\mathcal{A}_2}$. Let

$$r_2 = a(r^{(1)}, \ldots, r^{(k-1)}, r_2', r^{(k+1)}, \ldots, r^{(m)})$$

and observe that $r_2 \in r_1 \oslash_x r$. By Lemma 58, for each i, $q_{2,i} \in [\![r^{(i)}]\!]_{\mathcal{A}_2}$. By the definition of $\mathcal{A}_q$,

$$\begin{gathered}q_2' \in f_2^{(a,m)}(q_{2,1}, \ldots, q_{2,m}) \\ q_2 \in \varepsilon\text{-}\mathsf{closure}_2(q_2'),\end{gathered}$$

and so

$$q_2 \in [\![r_2]\!]_{\mathcal{A}_2}$$

as required.

$\Longleftarrow$:

Base case: $r_1 = a$. It must be that $r_1 = x$, since $r_2 \in r_1 \oslash_x r$. This means that $r_2 = r$. Let

$$q = q_2 \in [\![r_2]\!]_{\mathcal{A}_2} = [\![r]\!]_{\mathcal{A}_2}.$$

By definition,

$$(q_2, 1) = (q, 1) \in f_q^{(x,1)} \subseteq [\![r_1]\!]_{\mathcal{A}_q},$$

as required.

Inductive case: $r_1 = a(r^{(1)}, \ldots, r^{(m)})$ for some $(a, m) \in \Omega$ and $r^{(1)}, \ldots, r^{(m)} \in \mathsf{Term}_\Omega$. There must be some $1 \leq k \leq m$ and some $r_2' \in \mathsf{Term}_\Omega$ with

$$\begin{gathered}r_2' \in r^{(k)} \oslash_x r \\ r_2 = a(r^{(1)}, \ldots, r^{(k-1)}, r_2', r^{(k+1)}, \ldots, r^{(m)}).\end{gathered}$$

Since $q_2 \in [\![r_2]\!]_{\mathcal{A}_2}$, it follows there exist some $q_2', q_{2,1}, \ldots, q_{2,m}$ with

$$\text{for } i \neq k \quad q_{2,i} \in [\![r^{(i)}]\!]_{\mathcal{A}_2}$$
$$q_{2,k} \in [\![q_2']\!]_{\mathcal{A}_2}$$
$$q_2' \in f_2^{(a,m)}(q_{2,1}, \ldots, q_{2,m})$$
$$q_2 \in \varepsilon\text{-}\mathsf{closure}_2(q_2').$$

By the inductive hypothesis, there is some $q \in Q_2$ such that $q \in [\![r]\!]_{\mathcal{A}_2}$ and $(q_{2,k}, 1) \in [\![r^{(k)}]\!]_{\mathcal{A}_2}$. By Lemma 58, for each i, $(q_{2,i}, 0) \in [\![r^{(i)}]\!]_{\mathcal{A}_q}$. By the definition of $\mathcal{A}_q$,

$$(q_2', 1) \in f_q^{(a,m)}((q_{2,1}, 0), \ldots, (q_{2,k-1}, 0), (q_{2,k}, 1), (q_{2,k+1}, 0), \ldots, (q_{2,m}, 0))$$
$$(q_2, 1) \in \varepsilon\text{-}\mathsf{closure}_q((q_2', 1)),$$

and so

$$(q_2, 1) \in [\![r_1]\!]_{\mathcal{A}_q},$$

as required. □

Proposition 60 (Correctness of $\oslash\!\!-^{\exists}$ Construction)**.** *The automaton* $\mathcal{A} = \mathcal{A}_1 \oslash\!\!-_x^{\exists} \mathcal{A}_2$ *accepts the language* $\mathcal{L}_1 \oslash\!\!-_x^{\exists} \mathcal{L}_2$.

Proof.

$$\begin{aligned}
& r \in \mathcal{L}_1 \oslash\!\!-_x^{\exists} \mathcal{L}_2 \\
\iff\ & \text{there exist } r_1, r_2 \in \mathsf{Term}_\Omega \text{ s.t. } r_2 \in r_1 \oslash_x r \text{ and} \\
& \qquad r_1 \in \mathcal{L}_1 \text{ and } r_2 \in \mathcal{L}_2 \\
\iff\ & \text{there exist } r_1 \in \mathcal{L}_1, q_2 \in A_2, r_2 \in \mathsf{Term}_\Omega \text{ s.t.} \\
& \qquad r_2 \in r_1 \oslash_x r \text{ and } q_2 \in [\![r_2]\!]_{\mathcal{A}_2} \\
\text{(L. 59)}\ \iff\ & \text{there exist } r_1 \in \mathcal{L}_1, q_2 \in A_2, q \in Q_2 \text{ s.t.} \\
& \qquad q \in [\![r]\!]_{\mathcal{A}_2} \text{ and } (q_2, 10 \in [\![r_1]\!]_{\mathcal{A}_q} \\
\text{(L. 57)}\ \iff\ & \text{there exist } r_1 \in \mathcal{L}_1, q_2 \in A_2, q \in Q \text{ s.t.} \\
& \qquad q \in [\![r]\!]_{\mathcal{A}} \text{ and } (q_2, 10 \in [\![r_1]\!]_{\mathcal{A}_q} \\
\iff\ & \text{there exist } q \in [\![r]\!]_{\mathcal{A}}, r_1 \in \mathsf{Term}_\Omega, q_1 \in A_1, q_2 \in A_2 \text{ s.t.} \\
& \qquad q_1 \in [\![r_1]\!]_{\mathcal{A}_1} \text{ and } (q_2, 1) \in [\![r_1]\!]_{\mathcal{A}_q} \\
\iff\ & \text{there exist } q \in [\![r]\!]_{\mathcal{A}}, r_1 \in \mathsf{Term}_\Omega \text{ s.t.} \\
& \qquad [\![r]\!]_{\hat{\mathcal{A}}_1 \times \hat{\mathcal{A}}_2} \cap (A_1 \times A_2) \neq \emptyset \\
\iff\ & A \cap [\![r]\!]_{\mathcal{A}} \neq \emptyset.
\end{aligned}$$

□

Definition 4.23 ($-\!\!\oslash^{\exists}$ Construction)**.** Given ε-NFTA

$$\mathcal{A}_1 = (Q_1, \{f_1^a\}_{a \in \Omega \cup \{\varepsilon\}}, A_1) \text{ and}$$
$$\mathcal{A}_1 = (Q_2, \{f_2^a\}_{a \in \Omega \cup \{\varepsilon\}}, A_2),$$

the ε-NFTA

$$\mathcal{A}_1 \oslash\!\!-_x^{\exists} \mathcal{A}_2 = (Q, \{f^a\}_{a \in \Omega \cup \{\varepsilon\}}, A)$$

is defined as follows:

- $Q = Q_2 \times \{0, 1\}$;
- for $a \in \Omega \cup \{\varepsilon\}$, where a has arity m ($m = 1$ in the case where $a = \varepsilon$) f^a is the smallest relation satisfying
  - $(q_2', n) \in f^a((q_{2,1}, n_1), \ldots, (q_{2,m}, n_m))$ whenever $q_2' \in f_2^a(q_{2,1}, \ldots, q_{2,m})$ and $n = \sum_{i=1}^{m} n_i$, and
  - if $a = (x, 0)$, then $(q_2', 1) \in f^a$ whenever $q_2' \in [\![r]\!]_{\mathcal{A}_2}$ for any $r \in \mathcal{L}_1$; and
- $A = A_2 \times \{1\}$.

Consider the automaton $\mathcal{A} = \mathcal{A}_1 -\!\!\oslash_x^{\exists} \mathcal{A}_2$, where $\mathcal{A}_1$ and $\mathcal{A}_2$ are automata accepting languages $\mathcal{L}_1$ and $\mathcal{L}_2$ respectively. Once again, this construction is similar to the analogous construction for sequences, with the main difference stemming from the fact that $t(x, 0)$ is a unary relation. The construction also resembles that for $\oslash$, except that one x behaves like a term belonging to $\mathcal{L}_1$, rather than one term from $\mathcal{L}_2$ behaving like an x.

On consuming part of a term, a state $(q_2, 0)$ is possible if and only if q_2 is a possible state on running the automaton $\mathcal{A}_2$ on the same term. A state $(q_2, 1)$ is possible on consuming r if and only if there is some $r' \in \{r\} \oslash_x \mathcal{L}_2$ such that $q_2 \in [\![r']\!]_{\mathcal{A}_2}$. As before, the 1 component propagates up the term to ensure that exactly one x node is treated in this way.

In order for the construction to be effective, it must be possible to construct the set $Q' = \{q_2 \in Q_2 \mid \text{there exists } r \in \mathcal{L}_1 \text{ s.t. } r_1 \in [\![r]\!]_{\mathcal{A}_2}\}$. This can be done by considering the product pre-automaton $\hat{\mathcal{A}}_1 \times \hat{\mathcal{A}}_2$. Since the set of states is finite, the reachable states may be enumerated. Observe that $q_2 \in Q'$ if and only if there is some $q_1 \in A_1$ such that (q_1, q_2) is reachable in the product pre-automaton.

Lemma 61. *For all $r \in \mathsf{Term}_\Omega$, $q \in Q_2$,*

$$(q_2, 0) \in [\![r]\!]_{\mathcal{A}} \iff q_2 \in [\![r]\!]_{\mathcal{A}_2}.$$

The proof of this lemma is essentially the same as for Lemma 53, and so I omit the full details.

Lemma 62. *For all $r \in \mathsf{Term}_\Omega$, $q_2 \in Q_2$,*

$$(q_2, 1) \in [\![r]\!]_{\mathcal{A}}$$
$$\iff$$
there exist $r_1, r_2 \in \mathsf{Term}_\Omega$ s.t. $r_2 \in r \oslash_x r_1$ and $r_1 \in \mathcal{L}_1$ and $q_2 \in [\![r_2]\!]_{\mathcal{A}_2}$.

Proof. The proof is by induction on the structure of the term r.

Base case: $r = a$ for some $(a, 0) \in \Omega$.

$$\begin{aligned}
& (q_2, 1) \in [\![a]\!]_{\mathcal{A}} \\
\iff\ & \text{there exists } q_2' \in Q_2 \text{ s.t.} \\
& \quad (q_2', 1) \in t(a, 0) \text{ and } (q_2, 1) \in \varepsilon\text{-}\mathsf{closure}((q_2', 1)) \\
\iff\ & r = x \text{ and there exist } q_2' \in Q_2, r_1 \in \mathcal{L}_1 \text{ s.t.} \\
& \quad\quad q_2' \in [\![r_1]\!]_{\mathcal{A}_2} \text{ and } q_2 \in \varepsilon\text{-}\mathsf{closure}_2(q_2') \\
\iff\ & r = x \text{ and there exists } r_1 \in \mathcal{L}_1 \text{ s.t. } q_2 \in [\![r_1]\!]_{\mathcal{A}_2} \\
\iff\ & \text{there exist } r_1, r_2 \in \mathsf{Term}_\Omega \text{ s.t.} \\
& \quad\quad r_2 \in r \oslash_x r_1 \text{ and } r_1 \in \mathcal{L}_1 \text{ and } q_2 \in [\![r_2]\!]_{\mathcal{A}_2}.
\end{aligned}$$

Inductive case: $r = a(r^{(1)}, \ldots, r^{(m)})$ for some $(a, m) \in \Omega$ and $r^{(1)}, \ldots, r^{(m)} \in \mathsf{Term}_\Omega$, with $m \geq 1$. Consider the implication in each direction separately.

$\Longrightarrow$:

For some $q_2, q_{2,1}, \ldots, q_{2,m} \in Q_2$ and some $1 \leq k \leq m$, it must be the case that

$$\begin{gathered}
\text{for } i \neq k \quad (q_{2,i}, 0) \in [\![r^{(i)}]\!]_{\mathcal{A}} \\
(q_{2,k}, 1) \in [\![r^{(k)}]\!]_{\mathcal{A}} \\
(q_2', 1) \in f^{(a,m)}((q_{2,1}, 0), \ldots, (q_{2,k-1}, 0), (q_{2,k}, 1), (q_{2,k+1}, 0), \ldots, (q_{2,m}, 0)) \\
(q_2, 1) \in \varepsilon\text{-}\mathsf{closure}((q_2', 1)).
\end{gathered}$$

By Lemma 61,

$$\text{for } i \neq k \quad q_{2,i,} \in [\![r^{(i)}]\!]_{\mathcal{A}_2}.$$

By the inductive hypothesis, there exist $r_2', r_1 \in \mathsf{Term}_\Omega$ such that

$$\begin{gathered}
r_2' \in r^{(k)} \oslash_x r_1 \\
r_1 \in \mathcal{L}_1 \\
q_{2,k} \in [\![r_2']\!]_{\mathcal{A}_2}.
\end{gathered}$$

Let

$$r_2 = a(r^{(1)}, \ldots, r^{(k-1)}, r'_2, r^{(k+1)}, \ldots, r^{(m)})$$

and observe that

$$r_2 \in r \oslash_x r_1.$$

By the definition of $\mathcal{A}$,

$$q'_2 \in f_2^{(a,m)}(q_{2,1}, \ldots, q_{2,m})$$
$$q_2 \in \varepsilon\text{-}\mathsf{closure}_2(q'_2).$$

Hence

$$q_2 \in [\![a(r^{(1)}, \ldots, r^{(k-1)}, r'_2, r^{(k+1)}, \ldots, r^{(m)})]\!]_{\mathcal{A}_2} = [\![r_2]\!]_{\mathcal{A}_2}.$$

Thus, there exist $r_1, r_2 \in \mathsf{Term}_\Omega$ with $r_2 \in r \oslash_x r_1$, $r_2 \in \mathcal{L}_1$ and $q_2 \in [\![r_2]\!]_{\mathcal{A}_2}$, as required.

$\Longleftarrow$:

For some $1 \leq k \leq m$, and some $r'_2 \in \mathsf{Term}_\Omega$, it must be that

$$r_2 = a(r^{(1)}, \ldots, r^{(k-1)}, r'_2, r^{(k+1)}, \ldots, r^{(m)})$$
$$r'_2 \in r^{(k)} \oslash_x r_1.$$

Since $q_2 \in [\![r_2]\!]_{\mathcal{A}_2}$, it must be that, for some $q'_2, q_{2,1}, \ldots, q_{2,m} \in Q_2$,

$$\text{for } i \neq k \quad q_{2,i} \in [\![r^{(i)}]\!]_{\mathcal{A}_1}$$
$$q_{2,k} \in [\![r'_2]\!]_{\mathcal{A}_2}$$
$$q'_2 \in f_2^{(a,m)}(q_{2,1}, \ldots, q_{2,k-1}, q_{2,k}, q_{2,k+1}, \ldots, q_{2,m})$$
$$q_2 \in \varepsilon\text{-}\mathsf{closure}_2(q'_2).$$

By Lemma 61,

$$\text{for } i \neq k \quad (q_{2,i}, 0) \in [\![r^{(i)}]\!]_{\mathcal{A}}.$$

By the inductive hypothesis,

$$(q_{2,k}, 1) \in [\![r^{(k)}]\!]_{\mathcal{A}}.$$

By the definition of $\mathcal{A}$,

$$(q'_2, 1) \in f^{(a,m)}((q_{2,1}, 0), \ldots, (q_{2,k-1}, 0), (q_{2,k}, 1), (q_{2,k+1}, 0), \ldots, (q_{2,m}, 0))$$
$$(q_2, 1) \in \varepsilon\text{-}\mathsf{closure}(q'_2, 1).$$

Therefore,

$$(q_2, 1) \in [\![a(r^{(1)}, \ldots, r^{(m)})]\!]_{\mathcal{A}} = [\![r]\!]_{\mathcal{A}}$$

as required. □

Proposition 63 (Correctness of $-\!\!\oslash^{\exists}$ Construction)**.** *The automaton* $\mathcal{A} = \mathcal{A}_1 -\!\!\oslash^{\exists}_x \mathcal{A}_2$ *accepts the language* $\mathcal{L}_1 -\!\!\oslash^{\exists}_x \mathcal{L}_2$*.*

Proof.

$$
\begin{aligned}
& r \in \mathcal{L}_1 -\!\!\oslash^{\exists}_x \mathcal{L}_2 \\
\iff\ & \text{there exist } r_1, r_2 \text{ s.t. } r_2 \in r \oslash_x r_2 \text{ and } r_1 \in \mathcal{L}_1 \text{ and } r_2 \in \mathcal{L}_2 \\
\text{(L. 62)} \iff\ & \text{there exists } q_2 \in A_2 \text{ s.t. } (q_2, 1) \in [\![r]\!]_{\mathcal{A}} \\
\iff\ & A \cap [\![r]\!]_{\mathcal{A}} \neq \emptyset.
\end{aligned}
$$

□

4.2.3 Decidability

Given automaton constructions for the previously-described closure properties of regular term languages, together with the above automaton constructions, a formula $K \in \mathsf{K}^m_{\mathsf{Term}}$ and environment $\sigma \in \mathsf{LEnv}$ are encoded as an automaton $\mathcal{A}_{K,\sigma}$ as follows (where $x = \sigma\alpha$):

$$
\begin{aligned}
\mathcal{A}_{\mathbf{a}(K_1,\ldots,K_n),\sigma} &= \mathbf{a}(\mathcal{A}_{K_1,\sigma}, \ldots \mathcal{A}_{K_n,\sigma}) \cap \mathcal{A}_{\mathsf{C}^m_{\mathsf{Term}}} \\
\mathcal{A}_{\alpha,\sigma} &= \mathcal{A}_{\{x\}} \\
\mathcal{A}_{K_1 \circ_\alpha K_2,\sigma} &= (\mathcal{A}_{K_1,\sigma} \oslash_x \mathcal{A}_{K_2,\sigma}) \cap \mathcal{A}_{\mathsf{C}^m_{\mathsf{Term}}} \\
\mathcal{A}_{K_1 \circ\!\!-^{\exists}_\alpha K_2,\sigma} &= (\mathcal{A}_{K_1,\sigma} \oslash\!\!-^{\exists}_x \mathcal{A}_{K_2,\sigma}) \cap \mathcal{A}_{\mathsf{C}^m_{\mathsf{Term}}} \\
\mathcal{A}_{K_1 -\!\!\circ^{\exists}_\alpha K_2,\sigma} &= (\mathcal{A}_{K_1,\sigma} -\!\!\oslash^{\exists}_x \mathcal{A}_{K_2,\sigma}) \cap \mathcal{A}_{\mathsf{C}^m_{\mathsf{Term}}} \\
\mathcal{A}_{\mathrm{False},\sigma} &= \mathcal{A}_{\emptyset} \\
\mathcal{A}_{K_1 \to K_2,\sigma} &= (\overline{\mathcal{A}_{K_1,\sigma}} \cap \mathcal{A}_{\mathsf{C}^m_{\mathsf{Term}}}) \cup \mathcal{A}_{K_2,\sigma}.
\end{aligned}
$$

These constructions accept exactly the languages of term contexts that satisfy the corresponding formulae, and so the problems of model-checking and satisfiability are decidable.

Theorem 64. *Given sort* $\varsigma \in \mathsf{Sort}$*, quantifier-free formula* $K \in \mathsf{Formula}_\varsigma$*, environment* $\sigma \in \mathsf{LEnv}$*, and multi-holed term context* $c \in \mathsf{C}^m_{\mathsf{Term}}$ *with* $(c, \sigma) \in \mathsf{World}_\varsigma$*, it is decidable whether*

$$c, \sigma \models_\varsigma K.$$

Theorem 65. *Given sort* $\varsigma \in \mathsf{Sort}$*, quantifier-free formula* $K \in \mathsf{Formula}_\varsigma$*, and environment* $\sigma \in \mathsf{LEnv}$*, it is decidable whether there exists a multi-holed term context* $c \in \mathsf{C}^m_{\mathsf{Term}}$ *with* $(c, \sigma) \in \mathsf{World}_\varsigma$ *such that*

$$c, \sigma \models_\varsigma K.$$

Corollary 66. *Given sort* $\varsigma \in \mathsf{Sort}$ *and quantifier-free formula* $K \in \mathsf{Formula}_\varsigma$*, it is decidable whether there exists a pair of multi-holed term context* $c \in \mathsf{C}^m_{\mathsf{Term}}$ *and environment* $\sigma \in \mathsf{LEnv}$ *with* $(c, \sigma) \in \mathsf{World}_\varsigma$ *such that*

$$c, \sigma \models_\varsigma K.$$

4.3 Trees

The theory of automata can be adapted to (ranked) trees just as it can be adapted for terms. In [CDG+07], *hedge automata* are presented as such an adaptation. Here, I do not use hedge automata but an equivalent formalism that I term *forest automata*.

In the following, assume that multi-holed tree contexts, $\mathsf{C}^m_{\mathsf{Tree}}$, are defined over finite Σ and X. Let $\mathsf{Forest}_{\Sigma,\mathrm{X}}$, ranged over by $t, t', t_1, \ldots$, be the set of forests, which are defined in the same way as multi-holed tree contexts, but without the restriction that holes occur at most once. Note that $\mathsf{C}^m_{\mathsf{Tree}} \subseteq \mathsf{Forest}_{\Sigma,\mathrm{X}}$.

4.3.1 Automata

Forest automata generalise word automata: a run of an automaton consumes the forest from left to right, the new state on consuming a node depending on the previous state and the label of the node; however, the new state also depends on the state the automaton would be in having been run on the subforest beneath the node. ...

Definition 4.24 (ε-NFFA)**.** A *non-deterministic finite forest automaton with ε-transitions* (ε-NFFA) is a tuple $\mathcal{A} = (Q, e, \{f^a\}_{a \in \Sigma \cup \mathrm{X} \cup \{\varepsilon\}}, A)$ where:

- Q is the set of states, a finite set;
- $e \in Q$ is the initial state;
- for each $\mathbf{a} \in \Sigma$, $f^{\mathbf{a}} \subseteq (Q \times Q) \times Q$ is the state transition relation for $\mathbf{a}$;
- for each $x \in \mathrm{X}$, $f^x \subseteq Q \times Q$ is the state transition relation for x;
- $f^\varepsilon \subseteq Q \times Q$ is the non-consuming state transition relation; and
- $A \subseteq Q$ is the set of accepting states.

To formally define the language recognised by an automaton, I make some auxiliary definitions. The definition of ε-closure for ε-NFFA is exactly as in Definition 4.3 for ε-NFA.

Definition 4.25 (Automaton-induced Mappings)**.** An automaton $\mathcal{A}$ induces a function $[\![(\cdot)]\!]_{\mathcal{A}} : \mathsf{Forest}_{\Sigma,\mathrm{X}} \to \mathcal{P}(Q)$ the maps forests $t \in \mathsf{Forest}_{\Sigma,\mathrm{X}}$ to sets of states $[\![t]\!]_{\subseteq} Q$ according to the following definition:

$$
\begin{aligned}
[\![\varnothing]\!]_{\mathcal{A}} &= \varepsilon\text{-}\mathsf{closure}(e) \\
[\![t \otimes x]\!]_{\mathcal{A}} &= \{q \mid \text{there exists } q' \in [\![t]\!]_{\mathcal{A}} \text{ s.t. } (q', q) \in (f^x \fatsemi \varepsilon\text{-}\mathsf{closure})\} \\
[\![t_1 \otimes \mathbf{a}[t_2]]\!]_{\mathcal{A}} &= \left\{ q \;\middle|\; \begin{array}{r} \text{there exist } q_1 \in [\![t_1]\!]_{\mathcal{A}}, q_2 \in [\![t_2]\!]_{\mathcal{A}} \text{ s.t.} \\ ((q_1, q_2), q) \in (f^{\mathbf{a}} \fatsemi \varepsilon\text{-}\mathsf{closure}) \end{array} \right\}.
\end{aligned}
$$

An automaton $\mathcal{A}$ also induces a function $(\!|(\cdot)|\!)_{\mathcal{A}} : \mathsf{Forest}_{\Sigma,\mathrm{X}} \to \mathcal{P}(Q \times Q)$ that maps forests $t \in \mathsf{Forest}_{\Sigma,\mathrm{X}}$ to relations on states $(\!|t|\!)_{\mathcal{A}} \subseteq Q \times Q$ according to the following definition:

$$
\begin{aligned}
(\!|\varnothing|\!)_{\mathcal{A}} &= \varepsilon\text{-}\mathsf{closure} \\
[\![t \otimes x]\!]_{\mathcal{A}} &= (\!|t|\!)_{\mathcal{A}} \fatsemi f^x \fatsemi \varepsilon\text{-}\mathsf{closure} \\
[\![t_1 \otimes \mathbf{a}[t_2]]\!]_{\mathcal{A}} &= \left\{ (q, q') \;\middle|\; \begin{array}{r} \text{there exist } q_1 \in (\!|t_1|\!)_{\mathcal{A}}(q), q_2 \in [\![t_2]\!]_{\mathcal{A}} \text{ s.t.} \\ ((q_1, q_2), q') \in (f^{\mathbf{a}} \fatsemi \varepsilon\text{-}\mathsf{closure}) \end{array} \right\}.
\end{aligned}
$$

Remark. Forest automata can be viewed as a special case of term automata: node labels $\mathbf{a}$ are treated like binary term labels, hole labels x are treated like unary term labels, and the empty forest $\varnothing$ is treated like a nullary term label. This corresponds to the *previous sibling, last child* encoding of unranked trees as terms. Under this encoding, for example, the forest $\mathbf{a}[\mathbf{b}[\varnothing] \otimes x]$ is represented by the term $\mathbf{a}(\varnothing, x(\mathbf{b}(\varnothing, \varnothing)))$.

Definition 4.26 (Acceptance)**.** A forest t is said to be *accepted* by automaton $\mathcal{A}$ if $[\![t]\!]_{\mathcal{A}} \cap A \neq \emptyset$. The *language accepted by* $\mathcal{A}$, $\mathcal{L}_{\mathcal{A}}$, is the set of words accepted by $\mathcal{A}$:

$$\mathcal{L}_{\mathcal{A}} = \{t \in \mathsf{Forest}_{\Sigma,\mathrm{X}} \mid [\![t]\!]_{\mathcal{A}} \cap A \neq \emptyset\}.$$

The class of languages that can be recognised by forest automata is the class of *regular forest languages*. This class includes the empty language ($\emptyset$), all single-element languages, $\mathsf{Forest}_{\Sigma,\mathrm{X}}$ and $\mathsf{C}^m_{\mathsf{Tree}}$, and is closed under union, intersection, complementation (with respect to $\mathsf{Forest}_{\Sigma,\mathrm{X}}$) and concatenation.

4.3.2 Constructions

Non-deterministic linear substitution, and the existential duals of its adjoints, are defined for forests just as they were for words. Only nodes labelled from X are appropriate candidates for substitution, however. I now present the constructions for these connectives, which have features in common with

Definition 4.27 ($\oslash$ Construction)**.** Given ε-NFFA

$$\mathcal{A}_1 = (Q_1, e_1, \{f_1^a\}_{a \in \Sigma \cup \mathrm{X} \cup \{\varepsilon\}}, A_1) \text{ and}$$
$$\mathcal{A}_2 = (Q_2, e_2, \{f_2^a\}_{a \in \Sigma \cup \mathrm{X} \cup \{\varepsilon\}}, A_1),$$

the ε-NFFA

$$\mathcal{A}_1 \oslash_x \mathcal{A}_2 = (Q, e, \{f^a\}_{a \in \Sigma \cup \mathrm{X} \cup \{\varepsilon\}}, A)$$

is defined as follows:

- $Q = (Q_1 \times (Q_2 \cup \{0,1\})) \cup Q_2 \cup \{e\}$;
- e is fresh;
- for $\mathbf{a} \in \Sigma$, $f^{\mathbf{a}}$ is the smallest relation satisfying:
  - $(q_1'', n'') \in f^{\mathbf{a}}((q_1, n), (q_1', n'))$ whenever $q_1'' \in f_1^{\mathbf{a}}(q_1, q_1')$ and $n'' = n + n'$, with $n, n', n'' \in \{0,1\}$,
  - $(q_1, q_2'') \in f^{\mathbf{a}}((q_1, q_2), q_2')$ whenever $q_2'' \in f_2^{\mathbf{a}}(q_2, q_2')$ and for all $q_1 \in Q_1$, and
  - $f_2^{\mathbf{a}} \subseteq f^{\mathbf{a}}$;
- for $y \in \mathrm{X}$, f^y is the smallest relation satisfying:
  - $(q_1', n) \in f^y((q_1, n))$ whenever $q_1' \in f_1^y(q_1)$ and $n \in \{0,1\}$,
  - $(q_1, q_2') \in f^y((q_1, q_2))$ whenever $q_2' \in f_2^y(q_2)$ and $q_1 \in Q_1$, and
  - $f_2^y \subseteq f^y$;
- f^ε is the smallest relation satisfying:
  - $(q_1', n) \in f^\varepsilon((q_1, n))$ whenever $q_1' \in f_1^\varepsilon(q_1)$ and $n \in \{0,1\}$,
  - $(q_1, q_2') \in f^\varepsilon((q_1, q_2))$ whenever $q_2' \in f_2^\varepsilon(q_2)$ and $q_1 \in Q_1$,
  - $f_2^\varepsilon \subseteq f^\varepsilon$,
  - $(e_1, 0), e_2 \in f^\varepsilon(e)$,
  - $(q_1, e_2) \in f^\varepsilon((q_1, 0))$ for all $q_1 \in Q_1$, and
  - $(q_1', 1) \in f^\varepsilon((q_1, q_2))$ whenever $q_1' \in f_1^x(q_1)$ and $q_2 \in A_2$; and
- $A = A_1 \times \{1\}$.

Consider the automaton $\mathcal{A} = \mathcal{A}_1 \oslash_x \mathcal{A}_2$, where $\mathcal{A}_1$ and $\mathcal{A}_2$ are automata accepting languages $\mathcal{L}_1$ and $\mathcal{L}_2$ respectively. For a forest, t, each state $q \in [\![t]\!]_{\mathcal{A}}$ is of one of five types. If $q = e$ then $t = \varnothing$. If $q \in Q_2$ then $q \in [\![t]\!]_{\mathcal{A}_2}$. If $q = (q_1, 0)$ then $q_2 \in [\![t]\!]_{\mathcal{A}_1}$. If $q = (q_1, q_2)$ then $t = t_1 \otimes t_2$ such that $q_1 \in [\![t_1]\!]_{\mathcal{A}_1}$ and $q_2 \in [\![t_2]\!]_{\mathcal{A}_2}$. If $q = (q_1, 1)$ then $t = t_1 \oslash_x t_2$ such that $q_1 \in [\![t_1]\!]_{\mathcal{A}_1}$ and $t_2 \in \mathcal{L}_2$. The set of states $[\![t]\!]_{\mathcal{A}}$ is the most general satisfying these requirements, thereby ensuring that there is a $q \in [\![t]\!]_{\mathcal{A}} \cap A$ if and only if $t \in \mathcal{L}_1 \oslash_x \mathcal{L}_2$.

The correctness argument for the above construction is formalised in the following lemmata.

Lemma 67. *For all* $t \in \mathsf{Forest}_{\Sigma,\mathrm{X}}$,

$$e \in [\![t]\!]_{\mathcal{A}} \iff t = \varnothing.$$

Proof. By definition, $e \in [\![\varnothing]\!]_{\mathcal{A}}$. If $t \neq \varnothing$ then each $q \in [\![t]\!]_{\mathcal{A}}$ must be the result of some transition, but no transition results in the state e. □

Lemma 68. *For all* $t \in \mathsf{Forest}_{\Sigma,\mathrm{X}}$, $q_2 \in Q_2$,

$$q_2 \in [\![t]\!]_{\mathcal{A}} \iff q_2 \in [\![t]\!]_{\mathcal{A}_2}.$$

Proof. The proof is by induction on the structure of the forest t.

Base case: $t = \varnothing$. Observe that $e_2 \in [\![\varnothing]\!]_{\mathcal{A}}$, that $f_2^{\varepsilon} \subseteq f^{\varepsilon}$, and that if $q_2 \in f^{\varepsilon}(q_2')$ then $q' \in Q_2$ and $q_2 \in f_2^{\varepsilon}(q')$. Consequently, $q_2 \in [\![\varnothing]\!]_{\mathcal{A}}$ if and only if $q_2 \in [\![\varnothing]\!]_{\mathcal{A}_2}$.

Inductive case: $t = t' \otimes y$ for some $t' \in \mathsf{Tree}$ and $y \in \mathrm{X}$.

$$\begin{aligned}
& q_2 \in [\![t]\!]_{\mathcal{A}} \\
\iff\ & \text{there exists } q_2' \in Q \text{ s.t.} \\
& \qquad q_2 \in (f^y \fatsemi \varepsilon\text{-}\mathsf{closure})(q_2') \text{ and } q_2' \in [\![t']\!]_{\mathcal{A}} \\
\iff\ & \text{there exists } q_2' \in Q_2 \text{ s.t.} \\
& \qquad q_2 \in (f_2^y \fatsemi \varepsilon\text{-}\mathsf{closure}_2)(q_2') \text{ and } q_2' \in [\![t']\!]_{\mathcal{A}} \\
\text{(IH)}\ \iff\ & \text{there exists } q_2' \in Q_2 \text{ s.t.} \\
& \qquad q_2 \in (f_2^y \fatsemi \varepsilon\text{-}\mathsf{closure}_2)(q_2') \text{ and } q_2' \in [\![t']\!]_{\mathcal{A}_2} \\
\iff\ & q_2 \in [\![t' \otimes y]\!]_{\mathcal{A}_2} = [\![t]\!]_{\mathcal{A}_2}.
\end{aligned}$$

Inductive case: $t = t' \otimes \mathbf{a}[t'']$ for some $t', t'' \in \mathsf{Forest}_{\Sigma,\mathrm{X}}$ and $\mathbf{a} \in \Sigma$.

$$\begin{aligned}
& q_2 \in [\![t]\!]_{\mathcal{A}} \\
\iff\ & \text{there exist } q_2', q_2'' \in Q \text{ s.t. } q_2 \in (f^{\mathbf{a}} \fatsemi \varepsilon\text{-}\mathsf{closure})(q_2', q_2'') \text{ and} \\
& \qquad q_2' \in [\![t']\!]_{\mathcal{A}} \text{ and } q_2'' \in [\![t'']\!]_{\mathcal{A}} \\
\iff\ & \text{there exist } q_2', q_2'' \in Q_2 \text{ s.t. } q_2 \in (f_2^{\mathbf{a}} \fatsemi \varepsilon\text{-}\mathsf{closure}_2)(q_2', q_2'') \text{ and} \\
& \qquad q_2' \in [\![t']\!]_{\mathcal{A}} \text{ and } q_2'' \in [\![t'']\!]_{\mathcal{A}} \\
\text{(IH)}\quad \iff\ & \text{there exist } q_2', q_2'' \in Q_2 \text{ s.t. } q_2 \in (f_2^{\mathbf{a}} \fatsemi \varepsilon\text{-}\mathsf{closure}_2)(q_2', q_2'') \text{ and} \\
& \qquad q_2' \in [\![t']\!]_{\mathcal{A}_2} \text{ and } q_2'' \in [\![t'']\!]_{\mathcal{A}_2} \\
\iff\ & q_2 \in [\![t' \otimes \mathbf{a}[t'']]\!]_{\mathcal{A}_2} = [\![t]\!]_{\mathcal{A}_2}.
\end{aligned}$$

□

Lemma 69. *For all $t \in \mathsf{Forest}_{\Sigma,\mathrm{X}}$, $q_1 \in Q_1$,*

$$(q_1, 0) \in [\![t]\!]_{\mathcal{A}} \iff q_1 \in [\![t]\!]_{\mathcal{A}_1}.$$

Proof. The proof is by induction on the structure of the forest t.

Base case: $t = \varnothing$. Observe that $(e_1, 0) \in [\![\varnothing]\!]_{\mathcal{A}}$, and that, for all $q', q'' \in Q$ with $q' \in f^{\varepsilon}(q'')$, if either $q' \in Q_1 \times \{0\}$ or $q'' \in Q_1 \times \{0\}$ then there are $q_1', q_1'' \in Q_1$ such that $q' = (q_1', 0)$, $q'' = (q_1'', 0)$ and $q_1' \in f_1^{\varepsilon}(q_1'')$. Consequently, $(q_1, 0) \in [\![\varnothing]\!]_{\mathcal{A}}$ if and only if $q_1 \in [\![\varnothing]\!]_{\mathcal{A}_1}$.

Inductive case: $t = t' \otimes y$ for some $t' \in \mathsf{Tree}$ and $y \in \mathrm{X}$.

$$\begin{aligned}
& (q_1, 0) \in [\![t]\!]_{\mathcal{A}} \\
\iff\ & \text{there exists } q' \in Q \text{ s.t.} \\
& \qquad (q_1, 0) \in (f^y \fatsemi \varepsilon\text{-}\mathsf{closure})(q') \text{ and } q' \in [\![t']\!]_{\mathcal{A}} \\
\iff\ & \text{there exists } q_1' \in Q_1 \text{ s.t.} \\
& \qquad q_1 \in (f_1^y \fatsemi \varepsilon\text{-}\mathsf{closure}_1)(q_1') \text{ and } (q_1', 0) \in [\![t']\!]_{\mathcal{A}} \\
\text{(IH)}\quad \iff\ & \text{there exists } q_1' \in Q_1 \text{ s.t.} \\
& \qquad q_1 \in (f_1^y \fatsemi \varepsilon\text{-}\mathsf{closure}_1)(q_1') \text{ and } q_1' \in [\![t']\!]_{\mathcal{A}_1} \\
\iff\ & q_1 \in [\![t' \otimes y]\!]_{\mathcal{A}_1} = [\![t]\!]_{\mathcal{A}_1}.
\end{aligned}$$

Inductive case: $t = t' \otimes \mathbf{a}[t'']$ for some $t', t'' \in \mathsf{Forest}_{\Sigma,\mathrm{X}}$ and $\mathbf{a} \in \Sigma$.

$$\begin{aligned}
& (q_1, 0) \in [\![t]\!]_{\mathcal{A}} \\
\iff\ & \text{there exist } q', q'' \in Q \text{ s.t. } (q_1, 0) \in (f^{\mathbf{a}} \fatsemi \varepsilon\text{-}\mathsf{closure})(q', q'') \\
& \qquad \text{and } q' \in [\![t']\!]_{\mathcal{A}} \text{ and } q'' \in [\![t'']\!]_{\mathcal{A}} \\
\iff\ & \text{there exist } q_1', q_1'' \in Q_1 \text{ s.t. } q_1 \in (f_1^{\mathbf{a}} \fatsemi \varepsilon\text{-}\mathsf{closure}_1)(q_1', q_1'') \\
& \qquad \text{and } (q_1', 0) \in [\![t']\!]_{\mathcal{A}} \text{ and } (q_1'', 0) \in [\![t'']\!]_{\mathcal{A}} \\
\text{(IH)}\quad \iff\ & \text{there exist } q_1', q_1'' \in Q_1 \text{ s.t. } q_1 \in (f_1^{\mathbf{a}} \fatsemi \varepsilon\text{-}\mathsf{closure}_1)(q_1', q_1'') \\
& \qquad \text{and } q_1' \in [\![t']\!]_{\mathcal{A}_1} \text{ and } q_1'' \in [\![t'']\!]_{\mathcal{A}_1} \\
\iff\ & q_1 \in [\![t' \otimes \mathbf{a}[t'']]\!]_{\mathcal{A}_1} = [\![t]\!]_{\mathcal{A}_1}.
\end{aligned}$$

□

Lemma 70. *For all* $t \in \mathsf{Forest}_{\Sigma,\mathrm{X}}$, $q_1 \in Q_1$, $q_2 \in Q_2$,

$$(q_1, q_2) \in [\![t]\!]_{\mathcal{A}}$$
$$\iff$$
there exist $t_1, t_2 \in \mathsf{Forest}_{\Sigma,\mathrm{X}}$ *s.t.* $t = t_1 \otimes t_2$ *and* $q \in [\![t_1]\!]_{\mathcal{A}_1}$ *and* $q_2 \in [\![t_2]\!]_{\mathcal{A}_2}$.

Proof. Both directions are by induction on the structure of t.

$\Longrightarrow$:

Base case: $t = \varnothing$, and so $(q_1, q_2) \in \varepsilon\text{-}\mathsf{closure}(e)$. By the definition of $\mathcal{A}$, it follows that:

- $(q_1, q_2) \in \varepsilon\text{-}\mathsf{closure}((q_1, e_2))$, and hence $q_2 \in \varepsilon\text{-}\mathsf{closure}_2(e_2)$ and $q_2 \in [\![\varnothing]\!]_{\mathcal{A}_2}$;
- $(q_1, e_2) \in f^{\varepsilon}((q_1, 0))$; and
- $(q_1, 0) \in [\![\varnothing]\!]_{\mathcal{A}}$ and so $q_1 \in [\![\varnothing]\!]_{\mathcal{A}_2}$ (by Lemma 69).

Thus the choice of $t_1 = t_2 = \varnothing$ fulfils the requirements.

Inductive case: $t = t' \otimes y$ for some $t' \in \mathsf{Tree}$ and $y \in \mathrm{X}$. In this case, $(q_1, q_2) \in (f^y \fatsemi \varepsilon\text{-}\mathsf{closure})(q')$ for some $q' \in Q$. Either $q' = (q_1, q_2')$ for some $q_2' \in Q_2$, or $q' = (q_1', 0)$ for some $q_1' \in Q_1$.

If the former, it follows from the definition of $\mathcal{A}$ that $q_2 \in (f_2^y \fatsemi \varepsilon\text{-}\mathsf{closure}_2)(q_2')$. By the inductive hypothesis, there are t_1, t_2' with $t = t_1 \otimes t_2' \otimes y$, $q_1 \in [\![t_1]\!]_{\mathcal{A}_1}$ and $q_2' \in [\![t_2']\!]_{\mathcal{A}_2}$. Hence $q_2 \in [\![t_2' \otimes y]\!]_{\mathcal{A}_2}$, and so the choice of t_1 and $t_2 = t_2' \otimes y$ fulfils the requirements.

If the latter, it must be that:

- $(q_1, q_2) \in \varepsilon\text{-}\mathsf{closure}((q_1, e_2))$, and hence $q_2 \in \varepsilon\text{-}\mathsf{closure}_2(e_2)$ and $q_2 \in [\![\varnothing]\!]_{\mathcal{A}_2}$;
- $(q_1, e_2) \in f^{\varepsilon}((q_1, 0))$; and
- $(q_1, 0) \in [\![t]\!]_{\mathcal{A}}$ and so $q_1 \in [\![t]\!]_{\mathcal{A}_1}$ (by Lemma 69).

Therefore the choice of $t_1 = t$ and $t_2 = \varnothing$ fulfils the requirements.

Inductive case: $t = t' \otimes \mathbf{a}[t'']$ for some $t', t'' \in \mathsf{Forest}_{\Sigma,\mathrm{X}}$ and $\mathbf{a} \in \Sigma$. In this case, $(q_1, q_2) \in \varepsilon\text{-}\mathsf{closure}(q''')$ for some $q''' \in Q$ with $q''' \in f^{\mathbf{a}}(q', q'')$ for some $q', q'' \in Q$ with $q' \in [\![t']\!]_{\mathcal{A}}$ and $q'' \in [\![t'']\!]_{\mathcal{A}}$. Either $q''' = (q_1, q_2''')$ for some $q_2''' \in Q_2$, or $q''' = (q_1''', 0)$ for some $q_1''' \in Q_1$.

If the former, it follows from the definition of $\mathcal{A}$ that $q' = (q_1, q_2')$ and $q'' = q_2''$ for some $q_2', q_2'' \in Q_2$ with $q_2''' \in f_2^{\mathbf{a}}(q_2', q_2'')$. By the inductive hypothesis, there are $t_1, t_2' \in \mathsf{Forest}_{\Sigma,\mathrm{X}}$ with $t' = t_1 \otimes t_2'$, $q_1 \in [\![t_1]\!]_{\mathcal{A}_1}$ and $q_2' \in [\![t_2']\!]_{\mathcal{A}_2}$. Furthermore,

by Lemma 68, $q_2'' \in [\![t'']\!]_{\mathcal{A}_2}$ and so $q_2''' \in [\![t_2' \otimes \mathbf{a}[t'']]\!]_{\mathcal{A}_2}$. It must also be the case that $q_2 \in \varepsilon\text{-}\mathsf{closure}_2(q_2')$, and so $q_2 \in [\![t_2' \otimes \mathbf{a}[t'']]\!]_{\mathcal{A}_2}$. Therefore the choice of t_1 and $t_2' = t_2' \otimes \mathbf{a}[t'']$ fulfils the requirements.

If the latter, it must be that:

- $(q_1, q_2) \in \varepsilon\text{-}\mathsf{closure}((q_1, e_2))$, and hence $q_2 \in \varepsilon\text{-}\mathsf{closure}_2(e_2)$ and $q_2 \in [\![\varnothing]\!]_{\mathcal{A}_2}$;
- $(q_1, q_2) \in f^{\varepsilon}((q_1, 0))$; and
- $(q_1, 0) \in [\![t]\!]_{\mathcal{A}}$ and so $q_1 \in [\![t]\!]_{\mathcal{A}_1}$ (by Lemma 69).

Therefore the choice of $t_1 = t$ and $t_2 = \varnothing$ fulfils the requirements.

$\Longleftarrow$:

Base case: $t = \varnothing$. In this case, $t_1 = \varnothing$ and $t_2 = \varnothing$. By Lemma 69, $(q_1, 0) \in [\![\varnothing]\!]_{\mathcal{A}}$, and so, since $(q_1, e_2) \in f^{\varepsilon}((q_1, 0))$ by definition, $(q_1, e_2) \in [\![\varnothing]\!]_{\mathcal{A}}$. Furthermore, it must be that $q_2 \in \varepsilon\text{-}\mathsf{closure}_2(e_2)$ and hence $(q_1, q_2) \in [\![\varnothing]\!]_{\mathcal{A}}$ as required.

Inductive case: $t = t' \otimes y$ for some $t' \in \mathsf{Tree}$ and $y \in \mathrm{X}$. In this case, either $t_2 = t_2' \otimes y$ for some $t_2' \in \mathsf{Forest}_{\Sigma,\mathrm{X}}$, or $t_2 = \varnothing$ and $t_1 = t = t' \otimes y$.

If the former, $q_2 \in (f_2^y \fatsemi \varepsilon\text{-}\mathsf{closure}_2)(q_2')$ for some $q_2' \in Q_2$ with $q_2' \in [\![t_2']\!]_{\mathcal{A}_2}$. By the inductive hypothesis, $(q_1, q_2') \in [\![t_2 \otimes t_2']\!]_{\mathcal{A}}$. Therefore, by the definition of $\mathcal{A}$, $(q_1, q_2') \in [\![t_1 \otimes t_2' \otimes y]\!]_{\mathcal{A}} = [\![t]\!]_{\mathcal{A}}$ as required.

If the latter, $q_1 \in [\![t]\!]_{\mathcal{A}_1}$ and hence $(q_1, 0) \in [\![t]\!]_{\mathcal{A}}$ by Lemma 69. It follows then that $(q_1, e_2) \in [\![t]\!]_{\mathcal{A}}$. Furthermore, since $q_2 \in [\![\varnothing]\!]_{\mathcal{A}_2} = \varepsilon\text{-}\mathsf{closure}_2(e_2)$, it follows by the definition of $\mathcal{A}$ that $(q_1, q_2) \in [\![t]\!]_{\mathcal{A}}$, as required.

Inductive case: $t = t' \otimes \mathbf{a}[t'']$ for some $t', t'' \in \mathsf{Forest}_{\Sigma,\mathrm{X}}$ and $\mathbf{a} \in \Sigma$. In this case, either $t_2 = t_2' \otimes \mathbf{a}[t'']$ for some $t_2' \in \mathsf{Forest}_{\Sigma,\mathrm{X}}$, or $t_2 = \varnothing$ and $t_1 = t = t' \otimes \mathbf{a}[t'']$.

If the former, $q_2 \in \varepsilon\text{-}\mathsf{closure}_2(q_2''')$ for some $q_2'' \in Q_2$ with $q_2''' \in f_2^{\mathbf{a}}(q_2', q_2'')$ for some $q_2', q_2'' \in Q_2$ with $q_2' \in [\![t_2']\!]_{\mathcal{A}_2}$ and $q_2'' \in [\![t_2'']\!]_{\mathcal{A}_2}$. By the inductive hypothesis, $(q_1, q_2') \in [\![t_1 \otimes t_2']\!]_{\mathcal{A}} = [\![t']\!]_{\mathcal{A}}$. By Lemma 68, $q_2'' \in [\![t_2'']\!]_{\mathcal{A}}$. Hence, $(q_1, q_2''') \in f^{\mathbf{a}}((q_1, q_2'), q_2'') \subseteq [\![t' \otimes \mathbf{a}[t'']]\!]_{\mathcal{A}} = [\![t]\!]_{\mathcal{A}}$. Therefore, since by definition $(q_1, q_2) \in \varepsilon\text{-}\mathsf{closure}(q_1, q_2''')$, it follows that $q_1, q_2) \in [\![t]\!]_{\mathcal{A}}$, as required.

If the latter, $q_1 \in [\![t]\!]_{\mathcal{A}_1}$ and hence $(q_1, 0) \in [\![t]\!]_{\mathcal{A}}$ by Lemma 69. It follows then that $(q_1, e_2) \in [\![t]\!]_{\mathcal{A}}$. Furthermore, since $q_1 \in [\![\varnothing]\!]_{\mathcal{A}_2} = \varepsilon\text{-}\mathsf{closure}_2(e_2)$, it follows that $(q_1, q_2) \in [\![t]\!]_{\mathcal{A}}$, as required. □

Lemma 71. *For all $t \in \mathsf{Forest}_{\Sigma,\mathrm{X}}$, $q_1 \in Q_1$,*

$$(q_1, 1) \in [\![t]\!]_{\mathcal{A}}$$
$$\iff \textit{there exist } t_1, t_2 \textit{ s.t. } t \in t_1 \oslash_x t_2 \textit{ and } q_1 \in [\![t_1]\!]_{\mathcal{A}_1} \textit{ and } t_2 \in \mathcal{L}_2.$$

Proof. Both directions are by induction on the structure of t.

$\Longrightarrow$:

Base case: $t = \varnothing$. It must be the case that there are some $q_1' q_1'' \in Q_1$ and $q_2 \in Q_2$ with:

- $(q_1'', q_2) \in [\![\varnothing]\!]_{\mathcal{A}}$, and hence, by Lemma 70, $q_1'' \in [\![\varnothing]\!]_{\mathcal{A}_1}$ and $q_2 \in [\![\varnothing]\!]_{\mathcal{A}_2}$;
- $(q_1', 1) \in f^{\varepsilon}((q_1'', q_2))$, and hence $q_1' \in f_1^x(q_1'')$ and $q_2 \in A_2$, so $q_1' \in [\![x]\!]_{\mathcal{A}_2}$ and $\varnothing \in \mathcal{L}_2$; and
- $(q_1, 1) \in \varepsilon\text{-}\mathsf{closure}((q_1', 1))$, and hence $q_1 \in \varepsilon\text{-}\mathsf{closure}_1(q_1')$, so $q_1 \in [\![x]\!]_{\mathcal{A}_2}$.

Thus, choosing $t_1 = x$ and $t_2 = \varnothing$ fulfils the requirements, since $\varnothing \in x \oslash_x \varnothing$.

Inductive case: $t = t' \otimes y$ for some $t' \in \mathsf{Tree}$ and $y \in \mathrm{X}$. One of the following must be the case:

- $(q_1, 1) \in (f^y \fatsemi \varepsilon\text{-}\mathsf{closure})((q_1', 1))$ and $(q_1', 1) \in [\![t]\!]_{\mathcal{A}}$ for some $q_1' \in Q_1$; or
- $(q_1, 1) \in \varepsilon\text{-}\mathsf{closure}((q_1', 1))$ for some $q_1' \in Q_1$ with $(q_1', 1) \in f^{\varepsilon}((q_1'', q_2))$ for some $q_1'' \in Q_1$, $q_2 \in Q_2$ with $(q_1'', q_2) \in [\![t]\!]_{\mathcal{A}}$.

If the former, then, by the inductive hypothesis, there are t_1' and t_2' with $t' \in t_1' \oslash_x t_2$, $q_1' \in [\![t_1']\!]_{\mathcal{A}_1}$ and $t_2 \in \mathcal{L}_2$. By the definition of $\mathcal{A}$, $q_1 \in (f_1^y \fatsemi \varepsilon\text{-}\mathsf{closure}_1)(q_1')$ and so $q_1 \in [\![t_1' \otimes y]\!]_{\mathcal{A}_1}$. Since $(t_1' \oslash_x t_2) \otimes y \subseteq (t_1' \otimes y) \oslash_x t_2$, it follows that the choice of $t_1 = t_1' \otimes y$ and t_2 fulfils the requirements.

If the latter, then, by Lemma 70, $t = t_1' \otimes t_2$ with $q_1'' \in [\![t_1]\!]_{\mathcal{A}_1}$, $q_2 \in [\![t_2]\!]_{\mathcal{A}_2}$. Furthermore, by the definition of $\mathcal{A}$, $q_2 \in A_2$, and so $t_2 \in \mathcal{L}_2$. Also, $q_1' \in f_1^x(q_1'')$, so $t_1' \in [\![t_1' \otimes x]\!]_{\mathcal{A}_1}$. Moreover, $q_1 \in \varepsilon\text{-}\mathsf{closure}_1(q_1')$, so $q_1 \in [\![t_1' \otimes x]\!]_{\mathcal{A}_1}$. Since $t_1' \otimes t_2 \in (t_1' \otimes x) \oslash_x t_2$, it follows that the choice of $t_1 = t_1' \otimes x$ and t_2 fulfils the requirements.

Inductive case: $t = t' \otimes \mathbf{a}[t'']$ for some $t', t'' \in \mathsf{Forest}_{\Sigma,\mathrm{X}}$ and $\mathbf{a} \in \Sigma$. There must be some $q_1''' \in Q_1$ with $(q_1, 1) \in \varepsilon\text{-}\mathsf{closure}(q_1''', 1)$ such that one of the following holds:

- $(q_1''', 1) \in f^{\mathbf{a}}((q_1', 1), (q_1'', 0))$ for some $q_1', q_1'' \in Q_1$ with $(q_1', 1) \in [\![t']\!]_{\mathcal{A}}$ and $(q_1'', 0) \in [\![t'']\!]_{\mathcal{A}}$;
- $(q_1''', 1) \in f^{\mathbf{a}}((q_1', 0), (q_1'', 1))$ for some $q_1', q_1'' \in Q_1$ with $(q_1', 0) \in [\![t']\!]_{\mathcal{A}}$ and $(q_1'', 1) \in [\![t'']\!]_{\mathcal{A}}$; or
- $(q_1''', 1) \in f^{\mathbf{a}}((q_1', q_2))$ for some $q_1' \in Q_1$, $q_2 \in Q_2$ with $(q_1', q_2) \in [\![t]\!]_{\mathcal{A}}$.

In the first case, by the inductive hypothesis, there are $t_1', t_2 \in \mathsf{Forest}_{\Sigma,\mathrm{X}}$ with $t' \in t_1' \oslash_x t_2$, $q_1' \in [\![t_1']\!]_{\mathcal{A}_1}$ and $t_2 \in \mathcal{L}_2$. Furthermore, by Lemma 69,

$q_1'' \in [\![t'']\!]_{\mathcal{A}_1}$. By the definition of $\mathcal{A}$, $q_1''' \in f_1^{\mathbf{a}}(q_1', q_1'')$, and so $q_1''' \in [\![t_1' \otimes \mathbf{a}[t'']]\!]_{\mathcal{A}_1}$. Moreover, $q_1 \in \varepsilon\text{-}\mathsf{closure}_1(q_1''')$, so $q_1 \in [\![t_1' \otimes \mathbf{a}[t'']]\!]_{\mathcal{A}_1}$. Since $(t_1' \oslash_x t_2) \otimes \mathbf{a}[t''] \subseteq (t_1' \otimes \mathbf{a}[t'']) \oslash_x t_2$, it follows that the choice of $t_1 = t_1' \otimes \mathbf{a}[t'']$ and t_2 fulfils the requirements.

In the second case, by Lemma 69, $q_1' \in [\![t']\!]_{\mathcal{A}_1}$. Furthermore, by the inductive hypothesis, there are $t_1'', t_2 \in \mathsf{Forest}_{\Sigma,\mathrm{X}}$ with $t'' \in t_1'' \oslash_x t_2$, $q_1'' \in [\![t_1'']\!]_{\mathcal{A}_1}$ and $t_2 \in \mathcal{L}_2$. By the definition of $\mathcal{A}$, $q_1''' \in f_1^a(q_1', q_1'')$, and so $q_1''' \in [\![t' \otimes \mathbf{a}[t_1'']]\!]_{\mathcal{A}_1}$. Moreover, $q_1 \in \varepsilon\text{-}\mathsf{closure}_1(q_1''')$, so $q_1 \in [\![t_1 \otimes \mathbf{a}[t_1'']]\!]_{\mathcal{A}_1}$. Since $t' \otimes \mathbf{a}[t_1'' \oslash_x t_2] \subseteq (t' \otimes \mathbf{a}[t_1'']) \oslash_x t_2$, it follows that the choice of $t_1 = t' \otimes \mathbf{a}[t_1'']$ and t_2 fulfils the requirements.

In the third case, by Lemma 70, $t = t_1' \otimes t_2$ for some $t_1', t_2 \in \mathsf{Forest}_{\Sigma,\mathrm{X}}$ with $q_1' \in [\![t_1']\!]_{\mathcal{A}_1}$ and $q_2 \in [\![t_2]\!]_{\mathcal{A}_2}$. By the definition of $\mathcal{A}$, $q_2 \in A_2$, and so $t_2 \in \mathcal{L}_2$. Furthermore, $q_1''' \in f_1^x(q_1')$, so $q_1''' \in [\![t_1' \otimes x]\!]_{\mathcal{A}_1}$. Moreover, $q_1 \in \varepsilon\text{-}\mathsf{closure}_1(q_1''')$, and so $q_1 \in [\![t_1' \otimes x]\!]_{\mathcal{A}_1}$. Since $t_1' \otimes t_2 \in (t_1' \otimes x) \oslash_x t_2$, it follows that the choice of $t_1 = t_1' \otimes x$ and t_2 fulfils the requirements.

$\Longleftarrow$:

Base case: $t = \varnothing$. In this case, it must be that $t_1 = x$ and $t_2 = \varnothing$. Since $q_1 \in [\![x]\!]_{\mathcal{A}_1}$ it follows that $q_1 \in \varepsilon\text{-}\mathsf{closure}_1(q_1')$ for some $q_1' \in f_1^x(q_1'')$ for some $q_1'' \in [\![\varnothing]\!]_{\mathcal{A}_1}$. Further, since $\varnothing = t_2 \in \mathcal{L}_2$, there must be some $q_2 \in A_2$ with $q_2 \in [\![\varnothing]\!]_{\mathcal{A}_2}$. By Lemma 70, $(q_1'', q_2) \in [\![\varnothing]\!]_{\mathcal{A}}$. By the construction of $\mathcal{A}$, it follows that $(q_1', 1) \in f^{\varepsilon}(q_1'', q_2)$ and so $(q_1', 1) \in [\![\varnothing]\!]_{\mathcal{A}}$. Also, $(q_1, 1) \in \varepsilon\text{-}\mathsf{closure}(q_1', 1)$ and so $(q_1, 1) \in [\![\varnothing]\!]_{\mathcal{A}}$, as required.

Inductive case: $t = t' \otimes y$ for some $t' \in \mathsf{Tree}$ and $y \in \mathrm{X}$. One of the following must be the case:

- $t = t_1'' \otimes t_2$ and $t_1 = t_1'' \otimes x$ for some $t_1'' \in \mathsf{Forest}_{\Sigma,\mathrm{X}}$; or
- $t' \in t_1' \oslash_x$ and $t_1 = t_1' \otimes y$ for some $t_1' \in \mathsf{Forest}_{\Sigma,\mathrm{X}}$.

If the former, there is a $q_1'' \in [\![t_1'']\!]_{\mathcal{A}_1}$ such that $q_1 \in (f_1^x \fatsemi \varepsilon\text{-}\mathsf{closure}_1)(q_1'')$, and $q_2 \in [\![t]\!]_{\mathcal{A}_2} \cap A_2$. By Lemma 70, $(q_1'', q_2) \in [\![t_1'' \otimes t_2]\!]_{\mathcal{A}} = [\![t]\!]_{\mathcal{A}}$. From the definition of f^{ε}, it follows that $(q_1, 1) \in \varepsilon\text{-}\mathsf{closure}((q_1'', q_2))$ and so $(q_1, 1) \in [\![t]\!]_{\mathcal{A}}$ as required.

If the latter, there is a $q_1' \in [\![t_1']\!]_{\mathcal{A}_1}$ such that $q_1 \in (f_1^y \fatsemi \varepsilon\text{-}\mathsf{closure}_1)(q_1')$. By the inductive hypothesis, $(q_1', 1) \in [\![t']\!]_{\mathcal{A}}$. By the definition of $\mathcal{A}$, $(q_1, 1) \in (f^y \fatsemi \varepsilon\text{-}\mathsf{closure})((q_1', 1))$, and so $(q_1, 1) \in [\![t]\!]_{\mathcal{A}}$ as required.

Inductive case: $t = t' \otimes \mathbf{a}[t'']$ for some $t', t'' \in \mathsf{Forest}_{\Sigma,\mathrm{X}}$ and $\mathbf{a} \in \Sigma$. One of the following must hold:

- $t = t_1' \otimes t_2$ and $t_1 = t_1' \otimes x$ for some $t_1' \in \mathsf{Forest}_{\Sigma,\mathrm{X}}$;

- $t' \in t'_1 \oslash_x t_2$ and $t_1 = t'_1 \otimes \mathbf{a}[t'']$ for some $t'_1 \in \mathsf{Forest}_{\Sigma,\mathrm{X}}$; or
- $t'' \in t''_1 \oslash_x t_2$ and $t_1 = t' \otimes \mathbf{a}[t''_1]$ for some $t''_1 \in \mathsf{Forest}_{\Sigma,\mathrm{X}}$.

In the first case, there is a $q'_1 \in [\![t'_1]\!]_{\mathcal{A}_1}$ such that $q_1 \in (f_1^x \fatsemi \varepsilon\text{-}\mathsf{closure}_1)(q''_1)$, and $q_2 \in [\![t]\!]_{\mathcal{A}_2} \cap A_2$. By Lemma 70, $(q''_1, q_2) \in [\![t''_1 \otimes t_2]\!]_{\mathcal{A}} = [\![t]\!]_{\mathcal{A}}$. From the definition of f^ε, it follows that $(q_1, 1) \in \varepsilon\text{-}\mathsf{closure}((q''_1, q_2))$ and so $(q_1, 1) \in [\![t]\!]_{\mathcal{A}}$ as required.

In the second case, $q_1 \in \varepsilon\text{-}\mathsf{closure}_1(q'''_1)$ for some $q'''_1 \in f_1^{\mathbf{a}}(q'_1, q''_1)$ for some $q'_1 \in [\![t'_1]\!]_{\mathcal{A}_1}$ and $q''_1 \in [\![t'']\!]_{\mathcal{A}_1}$. By the inductive hypothesis, $(q'_1, 1) \in [\![t']\!]_{\mathcal{A}}$. By Lemma 69, $(q''_1, 0) \in [\![t'']\!]_{\mathcal{A}}$. By the definition of $\mathcal{A}$, $(q'''_1, 1) \in f^{\mathbf{a}}((q'_1, 1), (q''_1, 0))$, and so $(q'_1, 1) \in [\![t' \otimes \mathbf{a}[t'']]\!]_{\mathcal{A}} = [\![t]\!]_{\mathcal{A}}$. Furthermore, $(q_1, 1) \in \varepsilon\text{-}\mathsf{closure}(q'_1, 1)$ and so $(q_1, 1) \in [\![t]\!]_{\mathcal{A}}$, as required.

In the third case, $q_1 \in \varepsilon\text{-}\mathsf{closure}_1(q'''_1)$ for some $q'''_1 \in f_1^{\mathbf{a}}(q'_1, q''_1)$ for some $q'_1 \in [\![t']\!]_{\mathcal{A}_1}$ and $q''_1 \in [\![t''_1]\!]_{\mathcal{A}_1}$. By Lemma 69, $(q'_1, 0) \in [\![t']\!]_{\mathcal{A}}$. By the inductive hypothesis, $(q''_1, 1) \in [\![t'']\!]_{\mathcal{A}}$. By the definition of $\mathcal{A}$, $(q'''_1, 1) \in f^{\mathbf{a}}((q'_1, 0), (q''_1, 1))$, and so $(q'_1, 1) \in [\![t' \otimes \mathbf{a}[t'']]\!]_{\mathcal{A}} = [\![t]\!]_{\mathcal{A}}$. Furthermore, $(q_1, 1) \in \varepsilon\text{-}\mathsf{closure}(q'_1, 1)$ and so $(q_1, 1) \in [\![t]\!]_{\mathcal{A}}$, as required. □

Proposition 72 (Correctness of $\oslash$ Construction)**.** *The automaton $\mathcal{A} = \mathcal{A}_1 \oslash_x \mathcal{A}_2$ accepts the language $\mathcal{L}_1 \oslash_x \mathcal{L}_2$.*

Proof.

$$
\begin{aligned}
& t \in \mathcal{L}_1 \oslash_x \mathcal{L}_2 \\
\iff\ & \text{there exist } t_1, t_2 \in \mathsf{Forest}_{\Sigma,\mathrm{X}} \text{ s.t.} \\
& \qquad t \in t_1 \oslash_x t_2 \text{ and } t_1 \in \mathcal{L}_1 \text{ and } t_2 \in \mathcal{L}_2 \\
\text{(L. 71)} \iff\ & \text{there exists } q_1 \in A_1 \text{ s.t. } (q_1, 1) \in [\![t]\!]_{\mathcal{A}} \\
\iff\ & A \cap [\![t]\!]_{\mathcal{A}} \neq \varnothing.
\end{aligned}
$$

□

Definition 4.28 ($\oslash\!\!-^{\exists}$ Construction)**.** Given ε-NFFA

$$
\begin{aligned}
\mathcal{A}_1 &= (Q_1, e_1, \{f_1^a\}_{a \in \Sigma \cup \mathrm{X} \cup \{\varepsilon\}}, A_1) \text{ and} \\
\mathcal{A}_2 &= (Q_2, e_2, \{f_2^a\}_{a \in \Sigma \cup \mathrm{X} \cup \{\varepsilon\}}, A_1),
\end{aligned}
$$

the ε-NFFA

$$
\mathcal{A}_1 \oslash\!\!-_x^{\exists} \mathcal{A}_2 = (Q, e, \{f^a\}_{a \in \Sigma \cup \mathrm{X} \cup \{\varepsilon\}}, A)
$$

is defined as follows:

- $Q = \mathcal{P}(Q_2 \times Q_2)$;
- $e = \varepsilon\text{-}\mathsf{closure}_2$;
- for $\mathbf{a} \in \Sigma$,

$$f^{\mathbf{a}}(q, q') = \left\{ \left\{ (q_2, q_2') \;\middle|\; \begin{array}{l} q_2 \in Q_2 \text{ and} \\ \text{there exist } q_2'' \in q(q_2), \\ \qquad q_2''' \in q'(e_2) \text{ s.t.} \\ \qquad q_2' \in f_2^{\mathbf{a}}(q_2'', q_2''') \end{array} \right\} \fatsemi \varepsilon\text{-}\mathsf{closure}_2 \right\};$$

- for $y \in \mathrm{X}$, $f^y(q) = \{q \fatsemi f_2^y \fatsemi \varepsilon\text{-}\mathsf{closure}_2\}$;
- $f^\varepsilon = \emptyset$; and
- $q \in A$ if and only if there exists $t \in \mathsf{Forest}_{\Sigma,\mathrm{X}}$ s.t. $[\![t]\!]_{\hat{\mathcal{A}}_1 \times \hat{\mathcal{A}}_q} \cap A_1 \times A_q \neq \emptyset$, where
  - $\mathcal{A}_q = (Q_2 \times \{0,1\}, (e_2, 0), \{f_q^a\}_{a \in \Sigma \cup \mathrm{X} \cup \{\varepsilon\}\in,}, A_q)$,
  - for $\mathbf{a} \in \Sigma$,

$$f_q^{\mathbf{a}} = \{((q_2, n), (q_2', n'), (q_2'', n'')) \mid (q_2, q_2', q_2'') \in f_2^{\mathbf{a}} \text{ and } n + n' = n''\},$$

  - for $y \in \mathrm{X}$ with $y \neq x$,

$$f_q^y = \{((q_2, n), (q_2', n)) \mid (q_2, q_2') \in f_2^y \text{ and } n \in \{0,1\}\},$$

  - $f^x q = \begin{array}{r} \{((q_2, n), (q_2', n)) \mid (q_2, q_2') \in f_2^x \text{ and } n \in \{0,1\}\} \cup \\ \{((q_2, 0), (q_2', 1)) \mid (q_2, q_2') \in q\} \end{array}$,
  - $f_q^\varepsilon = \{((q_2, n), (q_2', n)) \mid (q_2, q_2') \in f_2^\varepsilon \text{ and } n \in \{0,1\}\}$, and
  - $A_q = A_2 \times \{1\}$.

Consider the automaton $\mathcal{A} = \mathcal{A}_1 \mathbin{\diamond\!\!-}_x^{\exists} \mathcal{A}_2$, where $\mathcal{A}_1$ and $\mathcal{A}_2$ are automata accepting languages $\mathcal{L}_1$ and $\mathcal{L}_2$ respectively. A state of $\mathcal{A}$ is a relation describing how $\mathcal{A}_2$ would behave on encountering the consumed forest when starting in an arbitrary state. That is, for any given forest t, $[\![t]\!]_{\mathcal{A}} = \{[\![t]\!]_{\mathcal{A}_2}\}$. The principle behind this construction is the same as in the sequence case.

The correctness argument for the above construction is formalised in the following lemmata.

Lemma 73. *For all $t \in \mathsf{Forest}_{\Sigma,\mathrm{X}}$,*

$$[\![t]\!]_{\mathcal{A}} = \{(\!|t|\!)_{\mathcal{A}_2}\}.$$

Proof. The proof is by induction on the structure of t. Note that, since $f^{\varepsilon} = \emptyset$, ε-closure is the identity relation on Q.

Base case: $t = \varnothing$.

$$\begin{aligned}
[\![\varnothing]\!]_{\mathcal{A}} &= \varepsilon\text{-closure}(e) \\
&= \varepsilon\text{-closure}(\varepsilon\text{-closure}_2) \\
&= \{\varepsilon\text{-closure}_2\} \\
&= \{(\!|\varnothing|\!)_{\mathcal{A}_2}\}\,.
\end{aligned}$$

Inductive case: $t = t' \otimes y$ for some $t' \in \mathsf{Tree}$ and $y \in \mathrm{X}$.

$$\begin{aligned}
[\![t' \otimes y]\!]_{\mathcal{A}} &= \{q \mid \text{there exists } q' \in [\![t']\!]_{\mathcal{A}} \text{ s.t. } q \in (ty \fatsemi \varepsilon\text{-closure})(q')\} \\
&= \{q \mid q \in (ty \fatsemi \varepsilon\text{-closure})((\!|t'|\!)_{\mathcal{A}_2})\} \\
&= ty((\!|t'|\!)_{\mathcal{A}_2}) \\
&= \{(\!|t'|\!)_{\mathcal{A}_2} \fatsemi f_2^y \fatsemi \varepsilon\text{-closure}_2\} \\
&= \{(\!|t' \otimes y|\!)_{\mathcal{A}_2}\}\,.
\end{aligned}$$

Inductive case: $t = t' \otimes \mathbf{a}[t'']$ for some $t', t'' \in \mathsf{Forest}_{\Sigma,\mathrm{X}}$ and $\mathbf{a} \in \Sigma$.

$$\begin{aligned}
[\![t' \otimes \mathbf{a}[t'']]\!]_{\mathcal{A}} &= \left\{ q \;\middle|\; \begin{array}{r} \text{there exist } q' \in [\![t']\!]_{\mathcal{A}}, q'' \in [\![t'']\!]_{\mathcal{A}} \text{ s.t.} \\ q \in (f^{\mathbf{a}} \fatsemi \varepsilon\text{-closure})(q', q'') \end{array} \right\} \\
&= \{q \mid q \in (f^{\mathbf{a}} \fatsemi \varepsilon\text{-closure})((\!|t'|\!)_{\mathcal{A}_2}, (\!|t''|\!)_{\mathcal{A}_2})\} \\
&= f^a((\!|t'|\!)_{\mathcal{A}_2}, (\!|t''|\!)_{\mathcal{A}_2}) \\
&= \left\{ \left\{ (q_2, q_2') \;\middle|\; \begin{array}{l} q_2 \in Q_2 \text{ and} \\ \text{there exist } q_2'' \in (\!|t'|\!)_{\mathcal{A}_2}(q_2), \\ \qquad q_2''' \in (\!|t''|\!)_{\mathcal{A}_2}(e_2) \text{ s.t.} \\ \qquad\quad q_2' \in f_2^{\mathbf{a}}(q_2'', q_2''') \end{array} \right\} \fatsemi \varepsilon\text{-closure}_2 \right\} \\
&= \left\{ \left\{ (q_2, q_2') \;\middle|\; \begin{array}{l} q_2 \in Q_2 \text{ and} \\ \text{there exist } q_2'' \in (\!|t'|\!)_{\mathcal{A}_2}(q_2), \\ \qquad\quad q_2''' \in [\![t'']\!]_{\mathcal{A}_2} \text{ s.t.} \\ q_2' \in (f_2^{\mathbf{a}} \fatsemi \varepsilon\text{-closure}_2)(q_2'', q_2''') \end{array} \right\} \right\} \\
&= \{(\!|t_1' \otimes \mathbf{a}[t'']|\!)_{\mathcal{A}_2}\}\,.
\end{aligned}$$

□

For $q \in Q$, let $\mathcal{A}_q$ be given as in Definition 4.28.

Lemma 74. *For all $q \in Q$, $q_2 \in Q_2$ and $t \in \mathsf{Forest}_{\Sigma,\mathrm{X}}$,*

$$(q_2, 0) \in [\![t]\!]_{\mathcal{A}_q} \iff q_2 \in [\![t]\!]_{\mathcal{A}_2}.$$

Proof. The proof is by induction on the structure of the forest t.

Base case: $t = \varnothing$.

$$\begin{aligned}(q_2, 0) \in [\![\varnothing]\!]_{\mathcal{A}_q} &\iff (q_2, 0) \in \varepsilon\text{-}\mathsf{closure}((e_2, 0)) \\ &\iff q_2 \in \varepsilon\text{-}\mathsf{closure}_2(e_2) \\ &\iff q_2 \in [\![\varnothing]\!]_{\mathcal{A}_2}.\end{aligned}$$

Inductive case: $t = t' \otimes y$ for some $t' \in \mathsf{Tree}$ and $y \in \mathrm{X}$.

$$\begin{aligned}& (q_2, 0) \in [\![t' \otimes y]\!]_{\mathcal{A}_q} \\ \iff\ & \text{there exists } q_2' \in Q_2 \text{ s.t. } (q_2, 0) \in (f^y \fatsemi \varepsilon\text{-}\mathsf{closure})((q_2', 0)) \\ & \qquad \text{and } (q_2' 0) \in [\![t']\!]_{\mathcal{A}_q} \\ \iff\ & \text{there exists } q_2' \in Q_2 \text{ s.t. } q_2 \in (f_2^y \fatsemi \varepsilon\text{-}\mathsf{closure}_2)(q_2') \\ & \qquad \text{and } q_2' \in [\![t']\!]_{\mathcal{A}_2} \\ \iff\ & q_2 \in [\![t' \otimes y]\!]_{\mathcal{A}_2}.\end{aligned}$$

Inductive case: $t = t' \otimes \mathbf{a}[t'']$ for some $t', t'' \in \mathsf{Forest}_{\Sigma,\mathrm{X}}$ and $\mathbf{a} \in \Sigma$.

$$\begin{aligned}& (q_2, 0) \in [\![t' \otimes \mathbf{a}[t'']]\!]_{\mathcal{A}_q} \\ \iff\ & \text{there exist } q_2', q_2'', q_2''' \in Q_2 \text{ s.t. } (q_2, 0) \in \varepsilon\text{-}\mathsf{closure}((q_2''', 0))) \text{ and} \\ & (q_2''', 0) \in f^{\mathbf{a}}((q_2', 0), (q_2'', 0)) \text{ and } (q_2', 0) \in [\![t']\!]_{\mathcal{A}_q} \text{ and } (q_2'', 0) \in [\![t'']\!]_{\mathcal{A}_q} \\ \iff\ & \text{there exist } q_2', q_2'', q_2''' \in Q_2 \text{ s.t. } q_2 \in \varepsilon\text{-}\mathsf{closure}_2(q_2''') \text{ and} \\ & \qquad q_2''' \in f_2^{\mathbf{a}}(q_2', q_2'') \text{ and } q_2' \in [\![t']\!]_{\mathcal{A}_2} \text{ and } q_2'' \in [\![t'']\!]_{\mathcal{A}_2} \\ \iff\ & q_2 \in [\![t' \otimes \mathbf{a}[t'']]\!]_{\mathcal{A}_2}.\end{aligned}$$

□

Lemma 75. *Suppose that $q = (\!|t|\!)_{\mathcal{A}_2}$ for some $t \in \mathsf{Forest}_{\Sigma,\mathrm{X}}$. Then for $t_1 \in \mathsf{Forest}_{\Sigma,\mathrm{X}}$, and $q_2 \in Q_2$,*

$$(q_2, 1) \in [\![t_1]\!]_{\mathcal{A}_q}$$
$$\iff$$
there exists $t_2 \in \mathsf{Forest}_{\Sigma,\mathrm{X}}$ s.t. $q_2 \in [\![t_2]\!]_{\mathcal{A}_2}$ and $t_2 \in t_1 \oslash_x t$.

Proof. The proof in both directions is by induction on the structure of the tree t_1.

$\Longrightarrow$:

Base case: $t_1 = \varnothing$. In this case, it is not possible that $(q_2, 1) \in [\![\varnothing]\!]_{\mathcal{A}_q}$ and so the implication holds trivially.

Inductive case: $t_1 = t_1' \otimes y$ for some $t_1' \in \mathsf{Tree}$ and $y \in \mathrm{X}$. In this case, $(q_2, 1) \in (f_q^x \fatsemi \varepsilon\text{-}\mathsf{closure}_q)(q_q)$ for some $q_q \in [\![t_1']\!]_{\mathcal{A}_q}$ with either:

- $q_q = (q_2' 1)$ for some $q_2' \in Q_2$; or
- $q_q = (q_2', 0)$ for some $q_2' \in Q_2$.

In the first case, since $(q_2', 1) \in [\![t_1']\!]_{\mathcal{A}_q}$, it follows that $q_2' \in [\![t_2']\!]_{\mathcal{A}_2}$ for some $t_2' \in t_1' \oslash_x t$, by the inductive hypothesis. From the definition of $\mathcal{A}_q$, it follows that $q_2 \in (f_2^y \fatsemi \varepsilon\text{-}\mathsf{closure}_2)(q_2')$ and hence $q_2 \in [\![t_2' \otimes y]\!]_{\mathcal{A}_2}$. Observe that $t_2' \otimes y \subseteq (t_1' \otimes y) \oslash_x t = t_1 \oslash_x t$, and so the choice $t_2 = t_2' \otimes y$ fulfils the requirements.

In the second case, $y = x$ and $(q_2, 1) \in \varepsilon\text{-}\mathsf{closure}_q((q_2'', 1))$ for some $q_2'' \in Q_2$ with $(q_2'', 1) \in f_q^x((q_2', 0))$. Since $(q_2', 0) \in [\![t_1']\!]_{\mathcal{A}_q}$, it follows that $q_2' \in [\![t_1']\!]_{\mathcal{A}_2}$, by Lemma 74. Furthermore, since $(q_2'', 1) \in f_q^x((q_2', 0))$, it follows that $q_2'' \in q(q_2') = (\!|t|\!)_{\mathcal{A}_2}(q_2')$ by Lemma 73, and so $q_2'' \in [\![t_1' \otimes t]\!]_{\mathcal{A}_2}$. Moreover, $q_2 \in \varepsilon\text{-}\mathsf{closure}_2(q_2'')$ and so $q_2 \in [\![t_1' \otimes t]\!]_{\mathcal{A}_2}$. Observe that $t_1' \otimes t \in (t_1' \otimes x) \oslash_x t = t_1 \oslash_x t$, and so the choice $t_2 = t_2' \otimes y$ fulfils the requirements.

Inductive case: $t_1 = t_1' \otimes \mathbf{a}[t_1'']$ for some $t_1', t_1'' \in \mathsf{Forest}_{\Sigma,\mathrm{X}}$ and $\mathbf{a} \in \Sigma$. In this case, $(q_2, 1) \in \varepsilon\text{-}\mathsf{closure}_q((q_2''', 1))$ for some $q_2''' \in Q_2$ with either:

- $(q_2''', 1) \in f_q^{\mathbf{a}}((q_2', 1), (q_2'', 0))$ for some $q_2', q_2'' \in Q_2$ with $(q_1', 1) \in [\![t_1']\!]_{\mathcal{A}_q}$ and $(q_2'', 0) \in [\![t_1'']\!]_{\mathcal{A}_q}$; or
- $(q_2''', 1) \in f_q^{\mathbf{a}}((q_2', 0), (q_2'', 1))$ for some $q_2', q_2'' \in Q_2$ with $(q_1', 0) \in [\![t_1']\!]_{\mathcal{A}_q}$ and $(q_2'', 1) \in [\![t_1'']\!]_{\mathcal{A}_q}$.

In the first case, by the inductive hypothesis, $q_2' \in [\![t_2']\!]_{\mathcal{A}_2}$ for some $t_2' \in t_1' \oslash_x t$. By Lemma 74, $q_2'' \in [\![t_1'']\!]_{\mathcal{A}_2}$. By the definition of $\mathcal{A}_q$, $q_2''' \in f_2^{\mathbf{a}}(q_2', q_2'')$ and so $q_2''' \in [\![t_2' \otimes \mathbf{a}[t_1'']]\!]_{\mathcal{A}_2}$. Furthermore, $q_2 \in \varepsilon\text{-}\mathsf{closure}_2(q_2''')$ and so $q_2 \in [\![t_2' \otimes \mathbf{a}[t_1'']]\!]_{\mathcal{A}_2}$. Observe that $t_2' \otimes \mathbf{a}[t_1''] \in (t_1' \oslash_x t) \otimes \mathbf{a}[t_1''] \subseteq (t_1' \otimes \mathbf{a}[t_1'']) \oslash_x t = t_1 \oslash_x t$, and so the choice $t_2 = t_2' \otimes \mathbf{a}[t_1'']$ fulfils the requirements.

In the second case, by Lemma 74, $q_2' \in [\![t_1']\!]_{\mathcal{A}_2}$. By the inductive hypothesis, $q_2'' \in [\![t_2'']\!]_{\mathcal{A}_2}$ for some $t_2'' \in t_1'' \oslash_x t$. By the definition of $\mathcal{A}_q$, $q_2''' \in f_2^{\mathbf{a}}(q_2', q_2'')$ and so $q_2''' \in [\![t_1' \otimes \mathbf{a}[t_2'']]\!]_{\mathcal{A}_2}$. Furthermore, $q_2 \in \varepsilon\text{-}\mathsf{closure}_2(q_2''')$ and so $q_2 \in [\![t_1' \otimes \mathbf{a}[t_2'']]\!]_{\mathcal{A}_2}$. Observe that $t_1' \otimes \mathbf{a}[t_2''] \in t_1' \otimes \mathbf{a}[t_1'' \oslash_x t] \subseteq (t_1' \otimes \mathbf{a}[t_1'']) \oslash_x t = t_1 \oslash_x t$, and so the choice $t_2 = t_1' \otimes \mathbf{a}[t_2'']$ fulfils the requirements.

$\Longleftarrow$:

Base case: $t_1 = \varnothing$. In this case, $t_1 \oslash_x t = \varnothing$ and so there is no $t_2 \in t_1 \oslash_x t$ and the implication holds trivially.

Inductive case: $t_1 = t_1' \otimes y$ for some $t_1' \in \mathsf{Tree}$ and $y \in \mathrm{X}$. In this case, either:

- $t_2 = t_2' \otimes y$ for some $t_2' \in t_1' \oslash_x t$; or

- $t_2 = t_1' \otimes t$ and $y = x$.

In the first case, it must be that $q_2 \in (f_2^y \fatsemi \varepsilon\text{-}\mathsf{closure}_2)(q_2')$ for some $q_2' \in [\![t_2']\!]_{\mathcal{A}_2}$. By the inductive hypothesis, $(q_2', 1) \in [\![t_1']\!]_{\mathcal{A}_q}$. By the construction of $\mathcal{A}_q$, it follows that $(q_2, 1) \in (f_q^y \fatsemi \varepsilon\text{-}\mathsf{closure}_q)((q_2', 1))$ and so $(q_2, 1) \in [\![t_1' \otimes y]\!]_{\mathcal{A}_q} = [\![t_1]\!]_{\mathcal{A}_q}$, as required.

In the second case, it must be that $q_2 \in (\!|t|\!)_{\mathcal{A}_2}(q_2') = q(q_2')$ for some $q_2' \in [\![t_1']\!]_{\mathcal{A}_2}$. By Lemma 74, $(q_2', 0) \in [\![t_1']\!]_{\mathcal{A}_q}$. Furthermore, by the construction of $\mathcal{A}_q$, it follows that $(q_2, 1) \in f_q^x((q_2', 0))$ and so $(q_2, 1) \in [\![t_1' \otimes x]\!]_{\mathcal{A}_q} = [\![t_1]\!]_{\mathcal{A}_q}$, as required.

Inductive case: $t_1 = t_1' \otimes \mathbf{a}[t_1'']$ for some $t_1', t_1'' \in \mathsf{Forest}_{\Sigma,\mathrm{X}}$ and $\mathbf{a} \in \Sigma$. In this case, either:

- $t_2 = t_2' \otimes \mathbf{a}[t_1'']$ for some $t_2' \in t_1' \oslash_x t$; or
- $t_2 = t_1' \otimes \mathbf{a}[t_2'']$ for some $t_2'' \in t_1'' \oslash_x t$.

In the first case, it must be that $q_2 \in \varepsilon\text{-}\mathsf{closure}_2(q_2''')$ for some $q_2''' \in f_2^{\mathbf{a}}(q_2', q_2'')$ for some $q_2' \in [\![t_2']\!]_{\mathcal{A}_2}$, $q_2'' \in [\![t_1'']\!]_{\mathcal{A}_2}$. By the inductive hypothesis, $(q_2', 1) \in [\![t_1']\!]_{\mathcal{A}_q}$. By Lemma 74, $(q_2'', 0) \in [\![t_1'']\!]_{\mathcal{A}_q}$. Furthermore, $(q_2''', 1) \in f_q^{\mathbf{a}}((q_2', 1), (q_2'', 0))$ and so $(q_2''', 1) \in [\![t_1' \otimes \mathbf{a}[t_1'']]\!]_{\mathcal{A}_q}$. Moreover, $(q_2, 1) \in \varepsilon\text{-}\mathsf{closure}_q(q_2''', 1)$ and so $(q_1, 1) \in [\![t_1' \otimes \mathbf{a}[t_1'']]\!]_{\mathcal{A}_q} = [\![t_1]\!]_{\mathcal{A}_q}$, as required.

In the second case, it must be that $q_2 \in \varepsilon\text{-}\mathsf{closure}_2(q_2''')$ for some $q_2''' \in f_2^{\mathbf{a}}(q_2', q_2'')$ for some $q_2' \in [\![t_1']\!]_{\mathcal{A}_2}$, $q_2'' \in [\![t_2'']\!]_{\mathcal{A}_2}$. By Lemma 74, $(q_2', 0) \in [\![t_1']\!]_{\mathcal{A}_q}$. By the inductive hypothesis, $(q_2'', 1) \in [\![t_1'']\!]_{\mathcal{A}_q}$. Also, $(q_2''', 1) \in f_q^{\mathbf{a}}((q_2', 0), (q_2'', 1))$ and so $(q_2''', 1) \in [\![t_1' \otimes \mathbf{a}[t_1'']]\!]_{\mathcal{A}_q}$. Moreover, $(q_2, 1) \in \varepsilon\text{-}\mathsf{closure}_q(q_2''', 1)$ and so $(q_1, 1) \in [\![t_1' \otimes \mathbf{a}[t_1'']]\!]_{\mathcal{A}_q} = [\![t_1]\!]_{\mathcal{A}_q}$, as required. □

Proposition 76 (Correctness of $\oslash\!-^{\exists}$ Construction)**.** *The automaton* $\mathcal{A} = \mathcal{A}_1 \oslash\!-_x^{\exists} \mathcal{A}_2$ *accepts the language* $\mathcal{L}_1 \oslash\!-_x^{\exists} \mathcal{L}_2$.

Proof.

$$
\begin{aligned}
& t \in \mathcal{L}_1 \mathbin{\oslash\!\!-}_x^{\exists} \mathcal{L}_2 \\
\iff\ & \text{there exist } t_1, t_2 \in \mathsf{Forest}_{\Sigma,\mathrm{X}} \text{ s.t. } t_1 \in \mathcal{L}_1 \text{ and} \\
& \qquad t_2 \in \mathcal{L}_2 \text{ and } t_2 \in t_1 \oslash_x t \\
\iff\ & \text{there exist } t_1, t_2 \in \mathsf{Forest}_{\Sigma,\mathrm{X}}, q_2 \in Q_2 \text{ s.t. } t_1 \in \mathcal{L}_1 \text{ and} \\
& \qquad q_2 \in [\![t_2]\!]_{\mathcal{A}_2} \text{ and } q_2 \in A_2 \text{ and } t_2 \in t_1 \oslash_x t \\
\iff\ & \text{there exists } q \in Q \text{ s.t. } q = (\!|t|\!)_{\mathcal{A}_2} \text{ and} \\
& \qquad \text{there exist } t_1, t_2 \in \mathsf{Forest}_{\Sigma,\mathrm{X}}, q_2 \in Q_2 \text{ s.t. } t_1 \in \mathcal{L}_1 \text{ and} \\
& \qquad q_2 \in [\![t_2]\!]_{\mathcal{A}_2} \text{ and } q_2 \in A_2 \text{ and } t_2 \in t_1 \oslash_x t \\
\text{(L. 75)}\ \iff\ & \text{there exists } q \in Q \text{ s.t. } q = (\!|t|\!)_{\mathcal{A}_2} \text{ and} \\
& \qquad \text{there exist } t_1 \in \mathsf{Forest}_{\Sigma,\mathrm{X}}, q_2 \in Q_2 \text{ s.t. } t_1 \in \mathcal{L}_1 \text{ and} \\
& \qquad (q_2, 1) \in [\![t_1]\!]_{\mathcal{A}_q} \text{ and } (q_2, 1) \in A_q \\
\iff\ & \text{there exists } q \in Q \text{ s.t. } q = (\!|t|\!)_{\mathcal{A}_2} \text{ and} \\
& \qquad \text{there exists } t_1 \in \mathsf{Forest}_{\Sigma,\mathrm{X}} \text{ s.t. } [\![t_1]\!]_{\mathcal{A}_1} \cap A_1 \neq \emptyset \text{ and} \\
& \qquad [\![t_1]\!]_{\mathcal{A}_q} \cap A_q \neq \emptyset \\
\iff\ & \text{there exists } q \in [\![t]\!]_{\mathcal{A}} \text{ s.t.} \\
& \qquad \text{there exists } t_1 \in \mathsf{Forest}_{\Sigma,\mathrm{X}} \text{ s.t. } [\![t_1]\!]_{\hat{\mathcal{A}}_1 \times \hat{\mathcal{A}}_q} \cap A_1 \times A_q \neq \emptyset \\
\iff\ & \text{there exists } q \in [\![t]\!]_{\mathcal{A}} \text{ s.t. } t \in A \\
\iff\ & [\![t]\!]_{\mathcal{A}} \cap A \neq \emptyset.
\end{aligned}
$$

□

Definition 4.29 ($-\!\!\oslash^{\exists}$ Construction)**.** Given ε-NFFA

$$
\begin{aligned}
\mathcal{A}_1 &= (Q_1, e_1, \{f_1^a\}_{a \in \Sigma \cup \mathrm{X} \cup \{\varepsilon\}}, A_1) \text{ and} \\
\mathcal{A}_2 &= (Q_2, e_2, \{f_2^a\}_{a \in \Sigma \cup \mathrm{X} \cup \{\varepsilon\}}, A_1),
\end{aligned}
$$

the ε-NFFA

$$
\mathcal{A}_1 \mathbin{-\!\!\oslash}_x^{\exists} \mathcal{A}_2 = (Q, e, \{f^a\}_{a \in \Sigma \cup \mathrm{X} \cup \{\varepsilon\}}, A)
$$

is defined as follows:

- $Q = Q_2 \times \{0, 1\}$;
- $e = (e_2, 0)$;
- for $\mathbf{a} \in \Sigma$, $f^{\mathbf{a}}$ is the smallest relation such that, $(q_2'', n'') \in f^{\mathbf{a}}((q_2, n), (q_2', n'))$ whenever $q'' \in f_2^{\mathbf{a}}(q, q')$ and $n'' = n + n'$;

- for $y \in \mathrm{X}$, f^y is the smallest relation such that
  - $(q_2', n) \in f^y((q_2, n))$ whenever $q_2' \in f_2^y(q_2)$ and $n \in \{0, 1\}$, and
  - if $y = x$ then $(q_2', 1) \in f^y((q_2, 0))$ whenever there is some $t \in \mathcal{L}_1$ such that $q_2' \in (\!|t|\!)_{\mathcal{A}_2}(q_2)$;
- f^ε is the smallest relation such that $(q_2', n) \in f^\varepsilon((q_2, n))$ whenever $q_2' \in f_2^\varepsilon(q_2)$ and $n \in \{0, 1\}$; and
- $A = A_2 \times \{1\}$.

Consider the automaton $\mathcal{A} = \mathcal{A}_1 \mathbin{-\!\!\oslash_x^\exists} \mathcal{A}_2$, where $\mathcal{A}_1$ and $\mathcal{A}_2$ are automata accepting languages $\mathcal{L}_1$ and $\mathcal{L}_2$ respectively. As in the sequence case, as state of the form $(q_2, 0)$ records that the automaton $\mathcal{A}_2$ would assign q_2 to the consumed forest, while a state of the form $(q_2, 1)$ records that $\mathcal{A}_2$ would assign q_2 to the result of substituting one instance of x by a forest from $\mathcal{L}_1$ in the consumed tree.

In order for this construction to be effective, it is necessary that the set $\{(\!|t_1|\!)_{\mathcal{A}_2} \mid t_1 \in \mathcal{L}_1\}$ be constructible. As in this sequence case, this is possible through reachability.

Lemma 77. *For all $t \in \mathsf{Forest}_{\Sigma,\mathrm{X}}$, $q_2 \in Q_2$,*

$$(q_2, 0) \in [\![t]\!]_{\mathcal{A}} \iff q_2 \in [\![t]\!]_{\mathcal{A}} \mathcal{A}_2.$$

The proof of this lemma is essentially the same as for Lemma 74 so I omit the details.

Lemma 78. *For all $t \in \mathsf{Forest}_{\Sigma,\mathrm{X}}$, $q_2 \in Q_2$,*

$$(q_2, 1) \in [\![t]\!]_{\mathcal{A}}$$
$$\iff$$
there exist $t_1, t_2 \in \mathsf{Forest}_{\Sigma,\mathrm{X}}$ s.t. $t_2 \in t \oslash_x t_1$ and $t_1 \in \mathcal{L}_1$ and $q_2 \in [\![t_2]\!]_{\mathcal{A}_2}$.

Proof. The proof in both directions is by induction on the structure of the forest t.

$\Longrightarrow$:

Base case: $t = \varnothing$. There are no $q_2', q_2'' \in Q_2$ such that $(q_2', 1) \in f^\varepsilon((q_2'', 0))$, and so it is not possible that $(q_2, 1) \in [\![t]\!]_{\mathcal{A}}$. Hence the implication holds vacuously in this case.

Inductive case: $t = t' \otimes y$ for some $t' \in \mathsf{Tree}$ and $y \in \mathrm{X}$. Assume that $(q_2, 1) \in [\![t]\!]_{\mathcal{A}}$. One of the following must apply:

- $(q_2, 1) \in (f^y \fatsemi \varepsilon\text{-}\mathsf{closure})((q_2', 1))$ for some $q_2' \in Q_2$ with $(q_2', 1) \in [\![t']\!]_{\mathcal{A}}\mathcal{A}$; or
- $y = y$ and $(q_2, 1) \in \varepsilon\text{-}\mathsf{closure}((q_2', 1))$ for some $q_2' \in Q_2$ with $(q_2', 1) \in f^x((q_2'', 0))$ for some $q_2' \in Q_2$ with $(q_2'', 0) \in [\![t']\!]_{\mathcal{A}}$.

In the first case, by the inductive hypothesis, there are $t_1, t_2' \in \mathsf{Forest}_{\Sigma,\mathrm{X}}$ with $t_2' \in t' \oslash_x t_1$, $t_1 \in \mathcal{L}_1$ and $q_2' \in [\![t_2']\!]_{\mathcal{A}_2}$. By the definition of $\mathcal{A}$, $q_2 \in (f_2^y \fatsemi \varepsilon\text{-}\mathsf{closure}_2)(q_2')$ and so $q_2 \in [\![t_2]\!]_{\mathcal{A}_2}$. Observe that $t_2' \otimes y \in (t' \otimes y) \oslash_x t_1 = t \oslash_x t_2$ and so the choice of t_1 and $t_2 = t_2' \otimes y$ fulfils the requirements.

In the second case, by Lemma 77, $q_2'' \in [\![t']\!]_{\mathcal{A}_2}$. By the definition of f^x, $q_2' \in (\!|t_1|\!)_{\mathcal{A}_2}(q_2'')$ for some $t_1 \in \mathcal{L}_1$. Hence, $t_2' \in [\![t' \otimes t_1]\!]_{\mathcal{A}_2}$. Furthermore, $q_2 \in \varepsilon\text{-}\mathsf{closure}_2(q_2')$ and so $q_2 \in [\![t' \otimes t_1]\!]_{\mathcal{A}_2}$. Let $t_2 = t' \otimes t_1$ and observe that $t_2 \in (t' \otimes x) \oslash_x t_1 = t \oslash_x t_1$. Hence t_1 and t_2 fulfil the requirements.

Inductive case: $t = t' \otimes \mathbf{a}[t'']$ for some $t', t'' \in \mathsf{Forest}_{\Sigma,\mathrm{X}}$ and $\mathbf{a} \in \Sigma$. Assume that $(q_2, 1) \in [\![t]\!]_{\mathcal{A}}$. It follows that $(q_2, 1) \in \varepsilon\text{-}\mathsf{closure}((q_2''', 1))$ for some $q_2''' \in Q_2$ with either:

- $(q_2''', 1) \in f^a((q_2', 1), (q_2'', 0))$ for some $q_2', q_2'' \in Q_2$ with $(q_2', 1) \in [\![t']\!]_{\mathcal{A}}$ and $(q_2', 0) \in [\![t'']\!]_{\mathcal{A}}$; or
- $(q_2''', 1) \in f^a((q_2', 0), (q_2'', 1))$ for some $q_2', q_2'' \in Q_2$ with $(q_2', 0) \in [\![t']\!]_{\mathcal{A}}$ and $(q_2', 1) \in [\![t'']\!]_{\mathcal{A}}$.

In the first case, by the inductive hypothesis, there are $t_1, t_2' \in \mathsf{Forest}_{\Sigma,\mathrm{X}}$ with $t_2' \in t' \oslash_x t_1$, $t_1 \in \mathcal{L}_1$ and $q_2' \in [\![t_2']\!]_{\mathcal{A}_2}$. By Lemma 77, $q_2'' \in [\![t'']\!]_{\mathcal{A}_2}$. Hence, by the definition of $f^{\mathbf{a}}$, it follows that $q_2''' \in f_2^{\mathbf{a}}(q_2', q_2'')$ and so $q_2''' \in [\![t_2' \otimes \mathbf{a}[t'']]\!]_{\mathcal{A}_2}$. Furthermore, $q_2 \in \varepsilon\text{-}\mathsf{closure}_2(q_2''')$ and so $q_2 \in [\![t_2' \otimes \mathbf{a}[t'']]\!]_{\mathcal{A}_2}$. Let $t_2 = t_2' \otimes \mathbf{a}[t'']$ and observe that $t_2 \in (t' \oslash_x t_1) \otimes \mathbf{a}[t''] \subseteq (t' \otimes \mathbf{a}[t'']) \oslash_x t_1 = t \oslash_x t_1$. Thus, t_1 and t_2 fulfil the requirements.

In the second case, by Lemma 77, $q_2' \in [\![t']\!]_{\mathcal{A}_2}$. By the inductive hypothesis, there are $t_1, t_2'' \in \mathsf{Forest}_{\Sigma,\mathrm{X}}$ with $t_2'' \in t'' \oslash_x t_1$, $t_1 \in \mathcal{L}_1$ and $q_2'' \in [\![t_2'']\!]_{\mathcal{A}_2}$. Hence, by the definition of $f^{\mathbf{a}}$, it follows that $q_2''' \in f_2^{\mathbf{a}}(q_2', q_2'')$ and so $q_2''' \in [\![t' \otimes \mathbf{a}[t_2'']]\!]_{\mathcal{A}_2}$. Furthermore, $q_2 \in \varepsilon\text{-}\mathsf{closure}_2(q_2''')$ and so $q_2 \in [\![t' \otimes \mathbf{a}[t_2'']]\!]_{\mathcal{A}_2}$. Let $t_2 = t' \otimes \mathbf{a}[t_2'']$ and observe that $t_2 \in t' \otimes \mathbf{a}[t'' \oslash_x t_1] \subseteq (t' \otimes \mathbf{a}[t'']) \oslash_x t_1 = t \oslash_x t_1$. Thus, t_1 and t_2 fulfil the requirements.

$\Longleftarrow$:

Base case: $t = \varnothing$. Since x does not appear in t, there can be no $t_1, t_2 \in \mathsf{Forest}_{\Sigma,\mathrm{X}}$ that fulfil the assumptions, and so the implication holds vacuously.

Inductive case: $t = t' \otimes y$ for some $t' \in \mathsf{Tree}$ and $y \in \mathrm{X}$. Assume that $t_1, t_2 \in \mathsf{Forest}_{\Sigma,\mathrm{X}}$ are such that $t_2 \in t \oslash_x t_1$, $t_1 \in \mathcal{L}_1$ and $q_2 \in [\![t_2]\!]_{\mathcal{A}_2}$. Either:

- $t_2 = t_2' \otimes y$ for some $t_2' \in t' \oslash_x t_1$; or

- $t_2 = t' \otimes t_1$ and $y = x$.

In the first case, $q_2 \in (f_2^y \fatsemi \varepsilon\text{-}\mathsf{closure}_2)(q_2')$ for some $q_2' \in Q_2$ with $q_2' \in [\![t_2']\!]_{\mathcal{A}_2}$. By the inductive hypothesis, $(q_2', 1) \in [\![t']\!]_{\mathcal{A}}$. By the definition of $\mathcal{A}$, $(q_2, 1) \in (f^y \fatsemi \varepsilon\text{-}\mathsf{closure})((q_2', 1))$ and so $(q_2, 1) \in [\![t']\!]_{\mathcal{A}}$ as required.

In the second case, $q_2 \in (\!|t_1|\!)_{\mathcal{A}_2}(q_2')$ for some $q_2' \in Q_2$ with $q_2' \in [\![t']\!]_{\mathcal{A}}\mathcal{A}_2$. By Lemma 77, $(q_2, 0) \in [\![t']\!]_{\mathcal{A}}$. By definition $(q_2, 1) \in f^x((q_2', 0))$ and so $(q_2, 1) \in [\![t' \otimes x]\!]_{\mathcal{A}} = [\![t]\!]_{\mathcal{A}}$, as required.

Inductive case: $t = t' \otimes \mathbf{a}[t'']$ for some $t', t'' \in \mathsf{Forest}_{\Sigma,\mathrm{X}}$ and $\mathbf{a} \in \Sigma$. Assume that $t_1, t_2 \in \mathsf{Forest}_{\Sigma,\mathrm{X}}$ are such that $t_2 \in t \oslash_x t_1$, $t_1 \in \mathcal{L}_1$ and $q_2 \in [\![t_2]\!]_{\mathcal{A}_2}$. Either:

- $t_2 = t_2' \otimes \mathbf{a}[t'']$ for some $t_2' \in t' \oslash_x t_1$; or
- $t_2 = t' \otimes \mathbf{a}[t_2'']$ for some $t_2'' \in t'' \oslash_x t_1$.

In the first case, $q_2 \in \varepsilon\text{-}\mathsf{closure}_2(q_2''')$ for some $q_2''' \in Q_2$ with $q_2''' \in f_2^{\mathbf{a}}(q_2', q_2'')$ for some $q_2', q_2'' \in Q_2$ with $q_2' \in [\![t_2']\!]_{\mathcal{A}_2}$ and $q_2'' \in [\![t'']\!]_{\mathcal{A}_2}$. By the inductive hypothesis, $(q_2', 1) \in [\![t']\!]_{\mathcal{A}}$. By lemma 77, $(q_2'', 0) \in [\![t'']\!]_{\mathcal{A}}$. By the definition of $\mathcal{A}$, $(q_2''', 1) \in f^{\mathbf{a}}((q_2', 1), (q_2'', 0))$ and so $(q_2''', 1) \in [\![t' \otimes \mathbf{a}[t'']]\!]_{\mathcal{A}} = [\![t]\!]_{\mathcal{A}}$. Furthermore, $(q_2, 1) \in \varepsilon\text{-}\mathsf{closure}((q_2''', 1))$ and so $(q_2, 1) \in [\![t]\!]_{\mathcal{A}}$, as required.

In the second case, $q_2 \in \varepsilon\text{-}\mathsf{closure}_2(q_2''')$ for some $q_2''' \in Q_2$ with $q_2''' \in f_2^{\mathbf{a}}(q_2', q_2'')$ for some $q_2', q_2'' \in Q_2$ with $q_2' \in [\![t']\!]_{\mathcal{A}_2}$ and $q_2'' \in [\![t_2'']\!]_{\mathcal{A}_2}$. By lemma 77, $(q_2', 0) \in [\![t']\!]_{\mathcal{A}}$. By the inductive hypothesis, $(q_2'', 1) \in [\![t'']\!]_{\mathcal{A}}$. By the definition of $\mathcal{A}$, $(q_2''', 1) \in f^{\mathbf{a}}((q_2', 0), (q_2'', 1))$ and so $(q_2''', 1) \in [\![t' \otimes \mathbf{a}[t'']]\!]_{\mathcal{A}} = [\![t]\!]_{\mathcal{A}}$. Furthermore, $(q_2, 1) \in \varepsilon\text{-}\mathsf{closure}((q_2''', 1))$ and so $(q_2, 1) \in [\![t]\!]_{\mathcal{A}}$, as required. □

Proposition 79 (Correctness of $-\!\oslash^{\exists}$ Construction)**.** *The automaton* $\mathcal{A} = \mathcal{A}_1 -\!\oslash_x^{\exists} \mathcal{A}_2$ *accepts the language* $\mathcal{L}_1 -\!\oslash_x^{\exists} \mathcal{L}_2$.

Proof.

$$
\begin{aligned}
& t \in \mathcal{L}_1 -\!\oslash_x^{\exists} \mathcal{L}_2 \\
\iff\ & \text{there exist } t_1, t_2 \in \mathsf{Forest}_{\Sigma,\mathrm{X}} \text{ s.t. } t_1 \in \mathcal{L}_1 \text{ and} \\
& \qquad\qquad t_2 \in \mathcal{L}_2 \text{ and } t_2 \in t \oslash_x t_1 \\
\iff\ & \text{there exists } q_2 \in A_2 \text{ s.t.} \\
& \quad \text{there exist } t_1, t_2 \in \mathsf{Forest}_{\Sigma,\mathrm{X}} \text{ s.t. } t_1 \in \mathcal{L}_1 \text{ and} \\
& \qquad\qquad t_2 \in \mathcal{L}_2 \text{ and } t_2 \in t \oslash_x t_1 \\
\text{(L. 78)} \iff\ & \text{there exists } q_2 \in A_2 \text{ s.t. } (q_2, 1) \in [\![t]\!]_{\mathcal{A}} \\
\iff\ & [\![t]\!]_{\mathcal{A}} \cap A \neq \emptyset.
\end{aligned}
$$

□

4.3.3 Decidability

Given automaton constructions for the previously-described closure properties of regular forest languages, together with the above automaton constructions, a formula $K \in \mathsf{K}^m_{\mathsf{Tree}}$ and environment $\sigma \in \mathsf{LEnv}$ are encoded as an automaton $\mathcal{A}_{K,\sigma}$ as follows (where $x = \sigma\alpha$):

$$\begin{aligned}
\mathcal{A}_{\mathbf{0},\sigma} &= \mathcal{A}_{\{\varnothing\}} \\
\mathcal{A}_{\mathbf{a}[K],\sigma} &= \mathcal{A}_{\{\mathbf{a}[y]\}} \oslash_y \mathcal{A}_{K,\sigma} \\
\mathcal{A}_{K_1 \otimes K_2,\sigma} &= (\mathcal{A}_{K_1,\sigma} \otimes \mathcal{A}_{K_2,\sigma}) \cap \mathcal{A}_{\mathsf{C}^m_{\mathsf{Tree}}} \\
\mathcal{A}_{\alpha,\sigma} &= \mathcal{A}_{\{x\}} \\
\mathcal{A}_{K_1 \circ_\alpha K_2,\sigma} &= (\mathcal{A}_{K_1,\sigma} \oslash_x \mathcal{A}_{K_2,\sigma}) \cap \mathcal{A}_{\mathsf{C}^m_{\mathsf{Tree}}} \\
\mathcal{A}_{K_1 \circ\!\!-^{\exists}_{\alpha} K_2,\sigma} &= \left(\mathcal{A}_{K_1,\sigma} \oslash\!\!-^{\exists}_{x} \mathcal{A}_{K_2,\sigma}\right) \cap \mathcal{A}_{\mathsf{C}^m_{\mathsf{Tree}}} \\
\mathcal{A}_{K_1 -\!\!\circ^{\exists}_{\alpha} K_2,\sigma} &= \left(\mathcal{A}_{K_1,\sigma} -\!\!\oslash^{\exists}_{x} \mathcal{A}_{K_2,\sigma}\right) \cap \mathcal{A}_{\mathsf{C}^m_{\mathsf{Tree}}} \\
\mathcal{A}_{\mathrm{False},\sigma} &= \mathcal{A}_{\emptyset} \\
\mathcal{A}_{K_1 \to K_2,\sigma} &= \left(\overline{\mathcal{A}_{K_1,\sigma}} \cap \mathcal{A}_{\mathsf{C}^m_{\mathsf{Tree}}}\right) \cup \mathcal{A}_{K_2,\sigma}.
\end{aligned}$$

These constructions accept exactly the languages of tree contexts that satisfy the corresponding formulae, and so the problems of model-checking and satisfiability are decidable.

Theorem 80. *Given sort $\varsigma \in \mathsf{Sort}$, quantifier-free formula $K \in \mathsf{Formula}_\varsigma$, environment $\sigma \in \mathsf{LEnv}$, and multi-holed tree context $c \in \mathsf{C}^m_{\mathsf{Tree}}$ with $(c, \sigma) \in \mathsf{World}_\varsigma$, it is decidable whether*

$$c, \sigma \models_\varsigma K.$$

Theorem 81. *Given sort $\varsigma \in \mathsf{Sort}$, quantifier-free formula $K \in \mathsf{Formula}_\varsigma$, and environment $\sigma \in \mathsf{LEnv}$, it is decidable whether there exists a multi-holed tree context $c \in \mathsf{C}^m_{\mathsf{Tree}}$ with $(c, \sigma) \in \mathsf{World}_\varsigma$ such that*

$$c, \sigma \models_\varsigma K.$$

Corollary 82. *Given sort $\varsigma \in \mathsf{Sort}$ and quantifier-free formula $K \in \mathsf{Formula}_\varsigma$, it is decidable whether there exists a pair of multi-holed tree context $c \in \mathsf{C}^m_{\mathsf{Tree}}$ and environment $\sigma \in \mathsf{LEnv}$ with $(c, \sigma) \in \mathsf{World}_\varsigma$ such that*

$$c, \sigma \models_\varsigma K.$$

4.4 Infinite Alphabets

So far in this chapter, the alphabets Σ and X have been constrained to be finite, since finite automata necessitate finite alphabets. However, in Chapter 2, the

alphabets were assumed to be infinite. It is possible to extend the decision procedures given in this chapter to the setting of infinite alphabets with some minor technical manipulation. The reason that this is possible is that formulae and environments only refer to finite subsets of Σ and X, and are indifferent to all other labels. For instance, a sequences formula that does not mention $\mathbf{a}$ or $\mathbf{b}$ cannot distinguish between the sequences $\mathbf{a} \cdot \mathbf{a}$, $\mathbf{a} \cdot \mathbf{b}$, $\mathbf{b} \cdot \mathbf{a}$ and $\mathbf{b} \cdot \mathbf{b}$ — either all four satisfy the formula or none of them does. Hence, it is sufficient to use a single label to stand for all of the labels which do not occur in the formula or environment.

In the following, assume that Σ and X are infinite, and, for any particular formula and environment under consideration, let $\hat{\Sigma}$ and $\hat{\mathrm{X}}$ be the finite subsets of Σ and X respectively that are referred to.

In order to deal with holes labels that do not occur in $\hat{\mathrm{X}}$, let $\star_{\mathrm{X}}$ be some fresh hole label (with respect to $\hat{\mathrm{X}}$). From the perspective of automaton the automaton constructions, $\star_{\mathrm{X}}$ is treated exactly like any other hole label, with the exception that it is not restricted to occurring linearly. That is, the automaton construction for $\mathcal{A}_{\mathsf{C}^m_{\mathsf{Seq}}}$ (or $\mathcal{A}_{\mathsf{C}^m_{\mathsf{Tree}}}$ or $\mathcal{A}_{\mathsf{C}^m_{\mathsf{Term}}}$) is adapted to enforce linearity on the holes in $\hat{\mathrm{X}}$ but to on $\star_{\mathrm{X}}$.

If a context satisfies a formula with respect to an environment, then, when all of the holes in the context that are not labelled from $\hat{\mathrm{X}}$ are relabelled to $\star_{\mathrm{X}}$, then the resulting pseudo-context will be accepted by the corresponding automaton. Conversely, any pseudo-context that is accepted by the automaton corresponding to a given formula and environment can be rewritten to a context by replacing each instance of $\star_{\mathrm{X}}$ by a fresh hole label from $\mathrm{X} \setminus \hat{\mathrm{X}}$.

A similar approach applies to dealing with node-labels that do not occur in $\hat{\Sigma}$ — they are rewritten to representations in terms of a finite alphabet. For sequences and trees, this means simply introducing a new label, $\star_\Sigma$, fresh with respect to $\hat{\Sigma}$, to which all labels in a given context that do not occur in $\hat{\Sigma}$ are rewritten.

In the case of terms there is an additional conundrum: the ranked alphabet Υ can contain labels of infinitely many different ranks. Clearly, it would not be viable to introduce a new label of each of these ranks to the finite automaton constructions. Instead, labels of arbitrary rank are encoded using just two fresh ranked labels: $\star^0_\Upsilon$ of rank 0, and $\star^2_\Upsilon$ of rank 2. A nullary label is encoded as $\star^0_\Upsilon$, a unary label as $\star^2_\Upsilon((\cdot), \star^0_\Upsilon)$, and a binary label as $\star^2_\Upsilon((\cdot), (\cdot))$. Labels of higher ranks are encoded by repeated nesting of $\star^2_\Upsilon$. This encoding preserves the satisfaction of the formula.

4.5 Quantification

Deciding satisfiability for logics with quantification presents issues that cannot be dealt with by automata directly. Instead, I invoke the quantifier normalisation results from §3.5 to transform formulae into normal forms for which model-checking and satisfiability can be reduced easily to the quantifier-free case. The results in this section apply to each of the context logic models which are decidable without quantification: in particular, sequences, terms and trees. Note that the automata-based procedures can easily by extended to handle the connectives $\circledcirc$ and $\mapsto$ that are introduced in the quantifier normalisation process.

Theorem 83 (Decidability of Model-Checking)**.** *Given sort* $\varsigma \in \mathsf{Sort}$*, formula* $K \in \mathsf{Formula}_\varsigma$*, environment* $\sigma \in \mathsf{LEnv}$*, and multi-holed context* $c \in \mathsf{C}^m$ *with* $(c, \sigma) \in \mathsf{World}_\varsigma$*, it is decidable whether*

$$c, \sigma \models_\varsigma K. \tag{4.3}$$

Proof. The decision procedure is first to apply Lemma 33 to convert K into the form $И\alpha_1, \ldots, \alpha_n. K'$ for some quantifier-free K'. Let $x_1, \ldots, x_n \in \mathrm{X}$ be distinct hole labels that are all fresh with respect to σ. Now determine if

$$c, \sigma[\alpha_1 \mapsto x_1] \cdots [\alpha_n \mapsto x_n] \models_{\varsigma'} K' \tag{4.4}$$

where $\varsigma' = \mathsf{c}(\phi \cup \{\alpha_1, \ldots, \alpha_n\})$ and $\varsigma = \mathsf{c}\phi$. By the definition of the satisfaction relation for $И$, if (4.4) holds then so does (4.3). Conversely, by the universal characterisation of $И$ (Lemma 4), if (4.3) holds then so does (4.4). □

The following lemma is a straightforward consequence of the definition of the satisfaction relation for $И$ and the hole substitution property (Property 2.34).

Lemma 84. *For a given sort* $\varsigma = \mathsf{c}\phi \in \mathsf{Sort}$*, formula* $K \in \mathsf{Formula}_\varsigma$*, environment* $\sigma \in \mathsf{LEnv}$*, and hole variable* $\alpha \in \Theta$ *with* $\alpha \notin \phi = \operatorname{dom}\sigma$*,*

$$\begin{aligned}
&\textit{there exists } c \in \mathsf{C} \textit{ s.t. } c, \sigma \models_\varsigma И\alpha. K \\
\iff &\textit{there exists } x \in \mathrm{X} \textit{ s.t. } x \sharp \sigma \textit{ and} \\
&\quad \textit{there exists } c \in \mathsf{C} \textit{ s.t. } c, \sigma[\alpha \mapsto x] \models_{\mathsf{c}(\phi \cup \{\alpha\})} K \wedge \circledcirc \\
\iff &\textit{for all } x \in \mathrm{X}, x \sharp \sigma \implies \\
&\quad \textit{there exists } c \in \mathsf{C} \textit{ s.t. } c, \sigma[\alpha \mapsto x] \models_{\mathsf{c}(\phi \cup \{\alpha\})} K \wedge \circledcirc.
\end{aligned}$$

Theorem 85 (Decidability of Satisfiability)**.** *Given sort* $\varsigma \in \mathsf{Sort}$*, formula* $K \in \mathsf{Formula}_\varsigma$*, and environment* $\sigma \in \mathsf{LEnv}$*, it is decidable whether there exists a*

multi-holed context $c \in \mathsf{C}^m$ *with* $(c, \sigma) \in \mathsf{World}_\varsigma$ *such that*

$$c, \sigma \models_\varsigma K.$$

Proof. The decision procedure is first to apply Lemma 33 to convert K into the form $\mathsf{M}\alpha_1, \ldots, \alpha_n. K'$ for some quantifier-free K'. Let $x_1, \ldots, x_n \in \mathrm{X}$ be distinct hole labels that are all fresh with respect to σ. Now determine if there exists $c \in \mathsf{C}^m$ such that

$$c, \sigma[\alpha_1 \mapsto x_1] \cdots [\alpha_n \mapsto x_n] \models_{\varsigma'} K'$$

where $\varsigma' = \mathsf{c}(\phi \cup \{\alpha_1, \ldots, \alpha_n\})$ and $\varsigma = \mathsf{c}\phi$. By Lemma 84, this is the case if and only if there exists a $c \in \mathsf{C}$ such that $c, \sigma, \models_\varsigma K$. □

Part II

Reasoning about Programs

Chapter 5

Local Reasoning

In this chapter, I define a simple imperative programming language, parameterised by its basic commands. The programs are interpreted over different domains depending on the intended semantics of the basic commands. Each domain is a context algebra, and the basic commands are interpreted as *local actions*, and so a sound local Hoare logic can be defined for the language.

I assume a fixed set of program variables Var. Program variables will be interpreted over the set of values Val, that at least includes the integers. Hence, I assume a syntax for value expressions that includes basic arithmetic operators and comparisons, as well as variables and elementary Boolean operators. However, I leave the actual definition of expression syntax open-ended, so that it can be extended to allow for other values than simply integers. In practice, when no additional expression constructions are required I will implicitly work with the minimal expression definitions meeting the assumptions.

Assumption 5.1 (Expression Syntax)**.** Assume a set of *value expressions* Expr, ranged over by $E, E_1, \ldots$, such that, for all $E_1, E_2 \in \mathsf{Expr}$,

$$\mathbb{Z} \subseteq \mathsf{Expr}$$
$$\mathsf{Var} \subseteq \mathsf{Expr}$$
$$E_1 + E_2 \in \mathsf{Expr}$$
$$E_1 - E_2 \in \mathsf{Expr}$$
$$E_1 * E_2 \in \mathsf{Expr}.$$

Assume a set of *Boolean expressions* BExp, ranged over by $B, B_1, \ldots$, such

that, for all $E_1, E_2 \in \mathsf{Expr}$ and $B_1, B_2 \in \mathsf{BExp}$,

$$E_1 = E_2 \in \mathsf{BExp}$$
$$E_1 < E_2 \in \mathsf{BExp}$$
$$\mathtt{false} \in \mathsf{BExp}$$
$$B_1 \Rightarrow B_2 \in \mathsf{BExp}.$$

Remark. It is not necessary for every expression to be meaningful when evaluated in an arbitrary context. For example, subtracting a string-valued variable from an integer may very well be undefined. This is captured by the semantics of expressions being partial functions.

I also assume a set of basic commands Cmd, ranged over by φ. The choice of these basic commands depends on the domain over which the language is to be used. For instance, to work with the heap, commands for allocation, mutation, lookup and disposal of heap cells would be necessary, whereas to work with a set, commands for inserting or removing elements would be appropriate.

Definition 5.1 (Programming Language Syntax)**.** Given a set of basic commands Cmd, ranged over by φ, the language $\mathcal{L}_{\mathsf{Cmd}}$, ranged over by $\mathbb{C}, \mathbb{C}_1, \ldots$, is defined as follows:

$$\begin{aligned}\mathbb{C} ::= {} & \varphi \mid \mathtt{skip} \mid \boldsymbol{x} := E \mid \mathbb{C}_1; \mathbb{C}_2 \\ & \mid \mathtt{if}\ B\ \mathtt{then}\ \mathbb{C}_1\ \mathtt{else}\ \mathbb{C}_2 \mid \mathtt{while}\ B\ \mathtt{do}\ \mathbb{C} \\ & \mid \mathtt{procs}\ \overrightarrow{\boldsymbol{r}_1} := \mathtt{f}_1(\overrightarrow{\boldsymbol{x}_1})\ \{\mathbb{C}_1\}, \ldots, \overrightarrow{\boldsymbol{r}_k} := \mathtt{f}_k(\overrightarrow{\boldsymbol{x}_k})\ \{\mathbb{C}_k\}\ \mathtt{in}\ \mathbb{C} \\ & \mid \mathtt{call}\ \boldsymbol{r}_1, \ldots, \boldsymbol{r}_i := \mathtt{f}(E_1, \ldots, E_j) \mid \mathtt{local}\ \boldsymbol{x}\ \mathtt{in}\ \mathbb{C}\end{aligned}$$

where $\boldsymbol{x}, \boldsymbol{r}_1, \ldots \in \mathsf{Var}$ range over program variables, $\overrightarrow{\boldsymbol{x}_i}, \overrightarrow{\boldsymbol{r}_i} \in \mathsf{Var}^*$ range over vectors of program variables, $E, E_1, \ldots \in \mathsf{Expr}$ range over expressions, $B \in \mathsf{BExp}$ ranges over Boolean expressions, and $\mathtt{f}, \mathtt{f}_1, \ldots \in \mathsf{PName}$ range over procedure names. The names $\mathtt{f}_1, \ldots, \mathtt{f}_k$ of procedures defined in a single $\mathtt{procs} - \mathtt{in}$ block are required to be pairwise distinct. The parameter and return variables are required to be pairwise distinct within each procedure definition.

5.1 Semantics

I assume a set of values, Val, with $\mathbb{Z} \subseteq \mathsf{Val}$. These values may be held by program variables and are the results of evaluating variable expressions (when defined). At a given point execution of a program, the current valuation of

the accessible program variables is called the variable scope, which is formally defined below.

Definition 5.2 (Scope)**.** The set of *variable scopes* Scope, ranged over by $\rho, \rho', \rho_1, \ldots,$ is the set of finite partial functions $\rho : \mathsf{Var} \rightharpoonup_{\text{fin}} \mathsf{Val}$ from variable names to values.

Since expressions were defined in an open-ended fashion, their semantics must also be open-ended.

Assumption 5.2 (Expression Semantics)**.** Assume a semantics of value expressions $\mathcal{E}[\![(\cdot)]\!] : \mathsf{Expr} \to (\mathsf{Scope} \rightharpoonup \mathsf{Val})$, which satisfies the following equations:

$$\begin{aligned}
\mathcal{E}[\![n]\!]\,\rho &= n \\
\mathcal{E}[\![\boldsymbol{x}]\!]\,\rho &= \rho\boldsymbol{x} \\
\mathcal{E}[\![E_1 \texttt{ + } E_2]\!]\,\rho &= \mathcal{E}[\![E_1]\!]\,\rho + \mathcal{E}[\![E_2]\!]\,\rho \\
\mathcal{E}[\![E_1 \texttt{ - } E_2]\!]\,\rho &= \mathcal{E}[\![E_1]\!]\,\rho - \mathcal{E}[\![E_2]\!]\,\rho \\
\mathcal{E}[\![E_1 \texttt{ * } E_2]\!]\,\rho &= \mathcal{E}[\![E_1]\!]\,\rho \times \mathcal{E}[\![E_2]\!]\,\rho
\end{aligned}$$

for all $\rho \in \mathsf{Scope}$, $n \in \mathbb{Z}$, $\boldsymbol{x} \in \mathsf{Var}$, and all $E_1, E_2 \in \mathsf{Expr}$ with $\mathcal{E}[\![E_1]\!], \mathcal{E}[\![E_2]\!] \in \mathbb{Z}$.

Assume a semantics of Boolean expressions $\mathcal{B}[\![(\cdot)]\!] : \mathsf{Expr} \to (\mathsf{Scope} \rightharpoonup \mathsf{Bool})$, where $\mathsf{Bool} = \{\mathbf{T}, \mathbf{F}\}$, which satisfies the following equations:

$$\mathcal{B}[\![E_1 \texttt{ = } E_2]\!]\,\rho = \begin{cases} \text{undefined} & \text{if } \mathcal{E}[\![E_1]\!]\,\rho \text{ or } \mathcal{E}[\![E_2]\!]\,\rho \text{ is undefined} \\ \mathbf{T} & \text{if } \mathcal{E}[\![E_1]\!]\,\rho = \mathcal{E}[\![E_2]\!]\,\rho \\ \mathbf{F} & \text{otherwise} \end{cases}$$

$$\mathcal{B}[\![E_1 \texttt{ < } E_2]\!]\,\rho = \begin{cases} \text{undefined} & \text{if } \mathcal{E}[\![E_1]\!]\,\rho \text{ or } \mathcal{E}[\![E_2]\!]\,\rho \text{ is undefined} \\ \mathbf{T} & \text{if } \mathcal{E}[\![E_1]\!]\,\rho <_{\mathbb{Z}} \mathcal{E}[\![E_2]\!]\,\rho \\ \mathbf{F} & \text{if } \mathcal{E}[\![E_1]\!]\,\rho \geq_{\mathbb{Z}} \mathcal{E}[\![E_2]\!] \end{cases}$$

$$\mathcal{B}[\![\texttt{false}]\!]\,\rho = \mathbf{F}$$

$$\mathcal{B}[\![B_1 \texttt{ => } B_2]\!]\,\rho = \begin{cases} \text{undefined} & \text{if } \mathcal{B}[\![B_1]\!]\,\rho \text{ or } \mathcal{B}[\![B_2]\!]\,\rho \text{ is undefined} \\ \mathbf{T} & \text{if } \mathcal{B}[\![B_1]\!]\,\rho = \mathbf{T} \implies \mathcal{B}[\![E_2]\!]\,\rho = \mathbf{T} \\ \mathbf{F} & \text{otherwise.} \end{cases}$$

5.1.1 Operational Semantics

Definition 5.3 (Procedure Definition Environment)**.** The set of *procedure definition environments*, PDef, ranged over by $\mu, \mu', \mu_1, \ldots,$ is defined as $\mathsf{PDef} =$

$\mathsf{PName} \rightharpoonup_{\text{fin}} (\mathsf{Var}^* \times \mathcal{L}_{\mathsf{Cmd}} \times \mathsf{Var}^*)$, the set of finite partial functions from procedure names to triples of a list of input variables, a program and a list of output variables. *Procedure definition stacks*, PDef^*, ranged over by $\gamma, \gamma', \gamma_1, \ldots$, are finite sequences of procedure definition environment. The operation of looking-up a procedure in a procedure definition stack, lookup : $\mathsf{PName} \times (\mathsf{PDef}^*) \rightharpoonup (\mathsf{Var}^* \times \mathcal{L}_{\mathsf{Cmd}} \times \mathsf{Var}^*) \times (\mathsf{PDef}^*)$, is defined as follows:

$$\text{lookup}(\mathtt{f}, \mu \cdot \gamma) = \begin{cases} (\mu\mathtt{f}, \mu \cdot \gamma) & \text{if } \mathtt{f} \in \text{dom}\, \mu \\ \text{lookup}(\mathtt{f}, \gamma) & \text{otherwise.} \end{cases}$$

Procedure definition stacks allow procedures to be re-defined when a procedure with the same name is in the current procedure scope. The lookup operation not only returns the definition of a procedure, but also the procedure definition stack that should be used in executing the procedure. This procedure definition stack consists of the procedure definitions that were in scope at the point when the procedure was defined, as well as the contemporaneous procedure definitions. This last point permits the definition of (mutually) recursive procedures.

As well as the variable scope, program state also includes a data store, which is operated on by the basic commands of Cmd. I therefore assume a set Store of *data stores*, ranged over by $\chi, \chi', \chi_1, \ldots$. The actual choice of Store will depend on the semantics to be given to the basic commands of Cmd, since only these commands directly manipulate the store. The set of *program states* $\mathsf{State} = \mathsf{Scope} \times \mathsf{Store}$ is the set of pairs of variable scopes and data stores.

Assumption 5.3 (Semantics of Basic Commands)**.** Assume an semantic interpretation function for basic commands,

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

Assume furthermore that for each $\varphi \in \mathsf{Cmd}$, $\mathcal{C}[\![\varphi]\!]$ preserves the domain of scopes. That is, for all $\rho, \rho' \in \mathsf{Scope}$ if $\rho' \in \mathcal{C}[\![\varphi]\!]\,\rho$ then $\text{dom}\,\rho = \text{dom}\,\rho'$.

The result of a successful execution of a program is always a program state. However, not every execution is necessarily successful; an execution that, for instance, tries to assign to a variable that is not in scope is considered to fail. Such executions are called *faulting* executions, and the (fresh) symbol $\lightning$ is used to denote their results. The set of *outcomes* $\mathsf{Outcome} = \mathsf{State} \cup \{\lightning\}$ is the set of program states plus the faulting outcome $\lightning$.

Remark. Note that not every program necessarily terminates from a given initial state; programs may also diverge or loop forever. However, I am chiefly concerned with terminating executions here, and so such executions are ignored by the semantics.

I am now able to define a big-step operational semantics for programs, given by judgements of the form $\mathbb{C}, \gamma, \rho, \chi \rightsquigarrow o$, denoting that, when run in the context of procedure definition stack γ, variable scope ρ and data store χ, the program $\mathbb{C}$ may result in outcome o.

Definition 5.4 (Operational Semantics)**.** The big-step relation

$$\rightsquigarrow : (\mathcal{L}_{\mathsf{Cmd}} \times (\mathsf{PDef}^*) \times \mathsf{Scope} \times \mathsf{Store}) \times \mathsf{Outcome}$$

is defined by the rules given in Figures 5.1 and 5.2.

Notation. The notation $\mathbb{C}, \mu, \rho, \chi \not\rightsquigarrow o$ denotes that there is no derivation of $\mathbb{C}, \mu, \rho, \chi, \rightsquigarrow o$.

5.1.2 Context Algebras Revisited

Fundamental to the concept of local reasoning is the view of state as being a resource. So far, the state space, particularly the space of data stores, has been treated rather opaquely with regard to its structure; to view state as a resource, the space must be blessed with additional structure. In particular, I will choose it to be a (compositional) context algebra (Definition 2.24).

Assuming that data stores form a context algebra, program states also form a context algebra by the following two results, the proofs of which are trivial.

Proposition 86 (Variable Scope Context Algebra)**.** *Let* $\mathcal{A}_{\mathsf{Scope}} = (\mathsf{Scope}, \mathsf{Scope}, *, *, \{\varnothing\})$ *where* $*$ *is the union of partial functions with disjoint domains and* $\varnothing$ *is the scope with the empty domain.* $\mathcal{A}_{\mathsf{Scope}}$ *is a context algebra: the* variable scope context algebra.

Proposition 87 (Direct Product of Context Algebras)**.** *Let* $\mathcal{A}_1 = (\mathsf{D}_1, \mathsf{C}_1, \circ_1, \bullet_1, \mathbf{I}_1)$ *and* $\mathcal{A}_2 = (\mathsf{D}_2, \mathsf{C}_2, \circ_2, \bullet_2, \mathbf{I}_2)$ *be context algebras. Then their* direct product $\mathcal{A}_1 \times \mathcal{A}_2 = (\mathsf{D}_1 \times \mathsf{D}_2, \mathsf{C}_1 \times \mathsf{C}_2, \circ_1 \times \circ_2, \bullet_1 \times \bullet_2, \mathbf{I}_1 \times \mathbf{I}_2)$ *is also a context algebra.*[1]

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

$$\frac{(\rho', \chi') \in \mathcal{C}\,[\![\varphi]\!]\,(\rho, \chi)}{\varphi, \gamma, \rho, \chi \rightsquigarrow \rho', \chi'} \qquad \frac{}{\texttt{skip}, \gamma, \rho, \chi \rightsquigarrow \rho, \chi}$$

$$\frac{\mathbb{C}_1, \gamma, \rho, \chi \rightsquigarrow \rho', \chi' \quad \mathbb{C}_2, \gamma, \rho', \chi' \rightsquigarrow \rho'', \chi''}{\mathbb{C}_1; \mathbb{C}_2, \gamma, \rho, \chi \rightsquigarrow \rho'', \chi''}$$

$$\frac{\mathcal{E}\,[\![E]\!]\,(\rho[\boldsymbol{x} \mapsto v]) = v'}{\boldsymbol{x} := E, \gamma, \rho[\boldsymbol{x} \mapsto v], \chi \rightsquigarrow \rho[\boldsymbol{x} \mapsto v'], \chi}$$

$$\frac{\mathcal{B}\,[\![B]\!]\,\rho = b \quad \mathbb{C}_b, \gamma, \rho, \chi \rightsquigarrow \rho', \chi'}{\texttt{if } B \texttt{ then } \mathbb{C}_{\mathbf{T}} \texttt{ else } \mathbb{C}_{\mathbf{F}}, \gamma, \rho, \chi \rightsquigarrow \rho', \chi'}$$

$$\frac{\mathcal{B}\,[\![B]\!]\,\rho = \mathbf{T} \quad \mathbb{C}; \texttt{while } B \texttt{ do } \mathbb{C}, \gamma, \rho, \chi \rightsquigarrow \rho', \chi'}{\texttt{while } B \texttt{ do } \mathbb{C}, \gamma, \rho, \chi \rightsquigarrow \rho', \chi'}$$

$$\frac{\mathcal{B}\,[\![B]\!]\,\rho = \mathbf{F}}{\texttt{while } B \texttt{ do } \mathbb{C}, \gamma, \rho, \chi \rightsquigarrow \rho, \chi}$$

$$\frac{\mathbb{C}, [\mathtt{f}_1 \mapsto (\overrightarrow{\boldsymbol{x}_1}, \mathbb{C}_1, \overrightarrow{\boldsymbol{r}_1}), \ldots, \mathtt{f}_k \mapsto (\overrightarrow{\boldsymbol{x}_k}, \mathbb{C}_k, \overrightarrow{\boldsymbol{r}_k})] \cdot \gamma, \rho, \chi \rightsquigarrow \rho', \chi'}{\texttt{procs } \overrightarrow{\boldsymbol{r}_1} := \mathtt{f}_1(\overrightarrow{\boldsymbol{x}_1})\ \{\mathbb{C}_1\}, \ldots, \overrightarrow{\boldsymbol{r}_k} := \mathtt{f}_k(\overrightarrow{\boldsymbol{x}_k})\ \{\mathbb{C}_k\}\ \texttt{in } \mathbb{C}, \gamma, \rho, \chi \rightsquigarrow \rho', \chi'}$$

$$\frac{\begin{array}{c}\text{lookup}(\mathtt{f}, \gamma) = (((\boldsymbol{x}_1, \ldots, \boldsymbol{x}_j), \mathbb{C}, (\boldsymbol{r}_1, \ldots, \boldsymbol{r}_i)), \gamma') \\ v_1 = \mathcal{E}\,[\![E_1]\!]\,\rho \quad \cdots \quad v_j = \mathcal{E}\,[\![E_j]\!]\,\rho \\ \mathbb{C}, \gamma', \emptyset[\boldsymbol{r}_1 \mapsto w_1] \ldots [\boldsymbol{r}_i \mapsto w_i][\boldsymbol{x}_1 \mapsto v_1] \ldots [\boldsymbol{x}_j \mapsto v_j], \chi \rightsquigarrow \rho', \chi' \\ \boldsymbol{y}_1, \ldots, \boldsymbol{y}_i \in \text{dom}\,\rho \quad \rho[\boldsymbol{y}_1 \mapsto \rho'\boldsymbol{r}_1] \ldots [\boldsymbol{y}_i \mapsto \rho'\boldsymbol{r}_i] = \rho''\end{array}}{\texttt{call } \boldsymbol{y}_1, \ldots, \boldsymbol{y}_i := \mathtt{f}(E_1, \ldots, E_j), \gamma, \rho, \chi \rightsquigarrow \rho'', \chi'}$$

$$\frac{\boldsymbol{x} \notin \text{dom}\,\rho \quad \boldsymbol{x} \notin \text{dom}\,\rho' \quad \mathbb{C}, \gamma, \rho[\boldsymbol{x} \mapsto v], \chi \rightsquigarrow \rho'[\boldsymbol{x} \mapsto w], \chi'}{\texttt{local } \boldsymbol{x} \texttt{ in } \mathbb{C}, \gamma, \rho, \chi \rightsquigarrow \rho', \chi'}$$

$$\frac{\mathbb{C}, \gamma, \rho[\boldsymbol{x} \mapsto v], \chi \rightsquigarrow \rho'[\boldsymbol{x} \mapsto w], \chi'}{\texttt{local } \boldsymbol{x} \texttt{ in } \mathbb{C}, \gamma, \rho[\boldsymbol{x} \mapsto u], \chi \rightsquigarrow \rho'[\boldsymbol{x} \mapsto u], \chi'}$$

Figure 5.1: Operational semantics rules for $\mathcal{L}_{\mathsf{Cmd}}$ (non-faulting cases)

$$\frac{\mathcal{C}[\![\varphi]\!](\rho,\chi) \text{ undefined}}{\varphi,\gamma,\rho,\chi \rightsquigarrow \lightning} \qquad \frac{\mathbb{C}_1,\gamma,\rho,\chi \rightsquigarrow \rho',\chi' \quad \mathbb{C}_2,\gamma,\rho',\chi' \rightsquigarrow \lightning}{\mathbb{C}_1;\mathbb{C}_2,\gamma,\rho,\chi \rightsquigarrow \lightning}$$

$$\frac{\mathbb{C}_1,\gamma,\rho,\chi \rightsquigarrow \lightning}{\mathbb{C}_1;\mathbb{C}_2,\gamma,\rho,\chi \rightsquigarrow \lightning} \qquad \frac{\mathcal{E}[\![E]\!]\,\rho \text{ undefined}}{\boldsymbol{x} := E,\gamma,\rho,\chi \rightsquigarrow \lightning} \qquad \frac{\boldsymbol{x} \notin \operatorname{dom}\rho}{\boldsymbol{x} := E,\gamma,\rho,\chi \rightsquigarrow \lightning}$$

$$\frac{\mathcal{B}[\![B]\!]\,\rho = b \quad \mathbb{C}_b,\gamma,\rho,\chi \rightsquigarrow \lightning}{\texttt{if } B \texttt{ then } \mathbb{C}_{\mathbf{T}} \texttt{ else } \mathbb{C}_{\mathbf{F}},\gamma,\rho,\chi \rightsquigarrow \lightning} \qquad \frac{\mathcal{B}[\![B]\!]\,\rho \text{ undefined}}{\texttt{if } B \texttt{ then } \mathbb{C}_{\mathbf{T}} \texttt{ else } \mathbb{C}_{\mathbf{F}},\gamma,\rho,\chi \rightsquigarrow \lightning}$$

$$\frac{\mathcal{B}[\![B]\!]\,\rho = \mathbf{T} \quad \mathbb{C};\texttt{while } B \texttt{ do } \mathbb{C},\gamma,\rho,\chi \rightsquigarrow \lightning}{\texttt{while } B \texttt{ do } \mathbb{C},\gamma,\rho,\chi \rightsquigarrow \lightning} \qquad \frac{\mathcal{B}[\![B]\!]\,\rho \text{ undefined}}{\texttt{while } B \texttt{ do } \mathbb{C},\gamma,\rho,\chi \rightsquigarrow \lightning}$$

$$\frac{\mathbb{C},[\mathtt{f}_1 \mapsto (\overrightarrow{\boldsymbol{x}_1},\mathbb{C}_1,\overrightarrow{\boldsymbol{r}_1}),\ldots,\mathtt{f}_k \mapsto (\overrightarrow{\boldsymbol{x}_k},\mathbb{C}_k,\overrightarrow{\boldsymbol{r}_k})]\cdot\gamma,\rho,\chi \rightsquigarrow \lightning}{\texttt{procs } \overrightarrow{\boldsymbol{r}_1} := \mathtt{f}_1(\overrightarrow{\boldsymbol{x}_1})\ \{\mathbb{C}_1\},\ldots,\overrightarrow{\boldsymbol{r}_k} := \mathtt{f}_k(\overrightarrow{\boldsymbol{x}_k})\ \{\mathbb{C}_k\}\ \texttt{in } \mathbb{C},\gamma,\rho,\chi \rightsquigarrow \lightning}$$

$$\frac{\operatorname{lookup}(\mathtt{f},\gamma) \text{ undefined or } \operatorname{lookup}(\mathtt{f},\gamma) \notin ((\mathsf{Var}^j \times \mathcal{L}_{\mathsf{Cmd}} \times \mathsf{Var}^i) \times \mathsf{PDef}^*)}{\texttt{call } \boldsymbol{y}_1,\ldots,\boldsymbol{y}_i := \mathtt{f}(E_1,\ldots,E_j),\gamma,\rho,\chi \rightsquigarrow \lightning}$$

$$\frac{1 \le k \le j \quad \mathcal{E}[\![E_k]\!]\,\rho \text{ undefined}}{\texttt{call } \boldsymbol{y}_1,\ldots,\boldsymbol{y}_i := \mathtt{f}(E_1,\ldots,E_j),\gamma,\rho,\chi \rightsquigarrow \lightning}$$

$$\frac{\begin{array}{c}\operatorname{lookup}(\mathtt{f},\gamma) = (((\boldsymbol{x}_1,\ldots,\boldsymbol{x}_j),\mathbb{C},(\boldsymbol{r}_1,\ldots,\boldsymbol{r}_i)),\gamma') \\ v_1 = \mathcal{E}[\![E_1]\!]\,\rho \quad \cdots \quad v_j = \mathcal{E}[\![E_j]\!]\,\rho \\ \mathbb{C},\gamma',\emptyset[\boldsymbol{r}_1 \mapsto w_1]\ldots[\boldsymbol{r}_i \mapsto w_i][\boldsymbol{x}_1 \mapsto v_1]\ldots[\boldsymbol{x}_j \mapsto v_j],\chi \rightsquigarrow \lightning\end{array}}{\texttt{call } \boldsymbol{y}_1,\ldots,\boldsymbol{y}_i := \mathtt{f}(E_1,\ldots,E_j),\gamma,\rho,\chi \rightsquigarrow \lightning}$$

$$\frac{1 \le k \le i \quad \boldsymbol{y}_k \notin \operatorname{dom}\rho}{\texttt{call } \boldsymbol{y}_1,\ldots,\boldsymbol{y}_i := \mathtt{f}(E_1,\ldots,E_j),\gamma,\rho,\chi \rightsquigarrow \lightning}$$

$$\frac{\mathbb{C},\gamma,\rho[\boldsymbol{x} \mapsto v],\chi \rightsquigarrow \lightning}{\texttt{local } \boldsymbol{x} \texttt{ in } \mathbb{C},\gamma,\rho,\chi \rightsquigarrow \lightning}$$

Figure 5.2: Operational semantics rules for $\mathcal{L}_{\mathsf{Cmd}}$ (faulting cases)

Special Context Algebras

It is occasionally necessary or useful to consider context algebras with additional properties or structure. I introduce two special forms of context algebra here: left-cancellative context algebras and context algebras with zero.

Definition 5.5 (Left-Cancellative Context Algebra)**.** A *left-cancellative context algebra* $\mathcal{A} = (\mathsf{D}, \mathsf{C}, \circ, \bullet, \mathbf{I})$ is a context algebra with the additional property that, for all $c_1, c_2, c \in \mathsf{C}$, $c_1 \circ c = c_2 \circ c$ only if $c_1 = c_2$.

Left-cancellativitity is a common property for context algebras that represent structured data. Indeed, every context algebra considered in this thesis is left-cancellative. I frequently abbreviate "left-cancellative" to simply "cancellative", although this terminology is technically inaccurate, since right-cancellativity does not have a significant role.

Definition 5.6 (Context Algebra with Zero)**.** A *context algebra with zero* $\mathcal{A} = (\mathsf{D}, \mathsf{C}, \circ, \bullet, \mathbf{I}, \mathbf{0})$ is a context algebra $(\mathsf{D}, \mathsf{C}, \circ, \bullet, \mathbf{I})$ together with a distinguished set of abstract data structures $\mathbf{0} \subseteq \mathsf{D}$ such that the relation

$$\{(c, s) \mid \text{there exists } o \in \mathbf{0} \text{ s.t. } c \bullet o = s\} \subseteq \mathsf{C} \times \mathsf{D}$$

is a total surjective function.

Many of the context algebras considered in this thesis can be viewed as context algebras with zero: for trees and sequences, take $\mathbf{0} = \{\varnothing\}$; for heaps, take $\mathbf{0} = \{\text{emp}\}$; for variable stores, take $\mathbf{0} = \{\varnothing\}$; and for $\mathcal{A}_1 \times \mathcal{A}_2$, where $\mathcal{A}_1$ and $\mathcal{A}_2$ are context algebras with zeros $\mathbf{0}_1$ and $\mathbf{0}_2$ respectively, take $\mathbf{0} = \mathbf{0}_1 \times \mathbf{0}_2$. Terms are a notable exception for which it is not typically possible to define a zero.

5.1.3 Axiomatic Semantics

I define an axiomatic semantics for $\mathcal{L}_{\mathsf{Cmd}}$ as a program logic based on local Hoare reasoning. This semantics treats the space of program states as a context algebra $\mathcal{A}_{\mathsf{State}} = (\mathsf{State}, \mathsf{C}_{\mathsf{State}}, \circ, \bullet, \mathbf{I})$; this is justified by the following assumption, taking $\mathcal{A}_{\mathsf{State}} = \mathcal{A}_{\mathsf{Scope}} \times \mathcal{A}_{\mathsf{Store}}$.

Assumption 5.4 (Data Store Context Algebra)**.** Assume a context algebra $\mathcal{A}_{\mathsf{Store}} = (\mathsf{Store}, \mathsf{C}_{\mathsf{Store}}, \circ_{\mathsf{Store}}, \bullet_{\mathsf{Store}}, \mathbf{I}_{\mathsf{Store}})$, based on the (previously-assumed) set of data stores, Store.

Hoare logic judgements make assertions about program state. For simplicity, I use semantic predicates as assertions, rather than using logical formulae; this reflects the practice of [COY07].

Definition 5.7 (Predicate)**.** The set of *state predicates* $\mathcal{P}(\mathsf{State})$, ranged over by $P, Q, R, P', P_1, \ldots$, is simply the set of sets of states. The set of *state-context predicates* $\mathcal{P}(\mathsf{C}_{\mathsf{State}})$, ranged over by $K, K', K_1, \ldots$, is simply the set sets of state contexts.

Definition 5.8 (Procedure Specification Environment)**.** A *procedure specification* $\mathtt{f} : \mathsf{P} \rightarrowtail \mathsf{Q}$ comprises

- a procedure name $\mathtt{f} \in \mathsf{PName}$,
- a parametrised precondition $\mathsf{P} : \mathsf{Val}^n \to \mathcal{P}(\mathsf{State})$, and
- a parametrised postcondition $\mathsf{Q} : \mathsf{Val}^m \to \mathcal{P}(\mathsf{State})$,

for some n, m.

A *procedure specification environment* is a set of procedure specifications. The metavariables $\Gamma, \Gamma', \ldots$ range over procedure specification environments.

Notation. In proof judgements, Γ, Γ' stands for the union $\Gamma \cup \Gamma'$.

Definition 5.9 (Predicate-Valued Semantics of Boolean Expressions)**.** The *predicate-valued semantics of Boolean expressions*, $\mathcal{P}[\![(\cdot)]\!] : \mathsf{BExp} \to \mathcal{P}(\mathsf{State})$, is defined in terms of their truth-valued semantics as follows:

$$\mathcal{P}[\![B]\!] = \{(\rho, \chi) \mid \mathcal{B}[\![B]\!]\,\rho = \mathbf{T}\}\,.$$

Definition 5.10 (Safety Predicates)**.** Given a value expression, $E \in \mathsf{Expr}$, the *expression safety predicate for* E is defined as follows:

$$\mathit{vsafe}(E) = \{(\rho, \chi) \mid \mathcal{E}[\![E]\!]\,\rho \text{ is defined}\}\,.$$

Similarly, given a Boolean expression, $B \in \mathsf{BExp}$, the *expression safety predicate for* B is defined as follows:

$$\mathit{bsafe}(B) = \{(\rho, \chi) \mid \mathcal{B}[\![B]\!]\,\rho \text{ is defined}\}\,.$$

Assumption 5.5 (Axioms for Basic Commands)**.** Assume a set of axioms for the basic commands,

$$\mathrm{Ax}[\![(\cdot)]\!] : \mathsf{Cmd} \to \mathcal{P}(\mathcal{P}(\mathsf{State}) \times \mathcal{P}(\mathsf{State})).$$

$$\frac{(P,Q) \in \mathrm{Ax}\,[\![\varphi]\!]}{\Gamma \vdash \{P\}\ \varphi\ \{Q\}}\ \textsc{Axiom} \qquad \frac{\Gamma \vdash \{P\}\ \mathbb{C}\ \{Q\}}{\Gamma \vdash \{K \bullet P\}\ \mathbb{C}\ \{K \bullet Q\}}\ \textsc{Frame}$$

$$\frac{P \subseteq P' \quad \Gamma \vdash \{P'\}\ \mathbb{C}\ \{Q'\} \quad Q' \subseteq Q}{\Gamma \vdash \{P\}\ \mathbb{C}\ \{Q\}}\ \textsc{Cons}$$

$$\frac{\text{for all } i \in I,\ \Gamma \vdash \{P_i\}\ \mathbb{C}\ \{Q_i\}}{\Gamma \vdash \left\{\bigvee_{i\in I} P_i\right\}\ \mathbb{C}\ \left\{\bigvee_{i\in I} Q_i\right\}}\ \textsc{Disj}$$

$$\frac{}{\Gamma \vdash \{P\}\ \texttt{skip}\ \{P\}}\ \textsc{Skip}$$

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

$$\frac{P \subseteq \mathit{bsafe}(B) \quad \Gamma \vdash \{P \wedge \mathcal{P}\,[\![B]\!]\}\ \mathbb{C}_1\ \{Q\} \quad \Gamma \vdash \{P \wedge \neg\mathcal{P}\,[\![B]\!]\}\ \mathbb{C}_2\ \{Q\}}{\Gamma \vdash \{P\}\ \texttt{if}\ B\ \texttt{then}\ \mathbb{C}_1\ \texttt{else}\ \mathbb{C}_2\ \{Q\}}\ \textsc{If}$$

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

$$\frac{(\boldsymbol{x} \Rightarrow v * \rho, \chi) \in \mathit{vsafe}(E)}{\Gamma \vdash \{\{(\boldsymbol{x} \Rightarrow v * \rho, \chi)\}\}\ \boldsymbol{x} := E\ \{\{(\boldsymbol{x} \Rightarrow \mathcal{E}\,[\![E]\!]\,(\boldsymbol{x} \Rightarrow v * \rho) * \rho, \chi)\}\}}\ \textsc{Assgn}$$

$$\frac{P \wedge \mathit{vsafe}(\boldsymbol{x}) \equiv \emptyset \quad \Gamma \vdash \begin{array}{c}\{(\boldsymbol{x} \Rightarrow - \times \mathbf{I}_{\mathsf{Store}}) \bullet P\} \\ \mathbb{C} \\ \{(\boldsymbol{x} \Rightarrow - \times \mathbf{I}_{\mathsf{Store}}) \bullet Q\}\end{array}}{\Gamma \vdash \{P\}\ \texttt{local}\ \boldsymbol{x}\ \texttt{in}\ \mathbb{C}\ \{Q\}}\ \textsc{Local}$$

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

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

$$\frac{\Gamma \vdash \{P\}\ \mathbb{C}\ \{Q\}}{\Gamma, \Gamma' \vdash \{P\}\ \mathbb{C}\ \{Q\}}\ \textsc{PWk}$$

Figure 5.3: Local Hoare logic rules for $\mathcal{L}_{\mathsf{Cmd}}$

5.2 Soundness

Definition 5.11 (Semantic Triples)**.**

$$\begin{aligned}
\gamma \models \{P\}\ \mathbb{C}\ \{Q\} \iff & \text{ for all } (\rho,\chi) \in P, o \in \mathsf{Outcome},\ \mathbb{C},\gamma,\rho,\chi \rightsquigarrow o \implies \\
& \quad o \neq \lightning \text{ and there exist } (\rho',\chi') \in Q \text{ s.t. } o = (\rho',\chi') \\
\gamma \models \Gamma \iff & \text{ for all } (\mathtt{f} : \mathsf{P} \rightarrowtail \mathsf{Q}) \in \Gamma, \\
& \quad \text{there exist } \overrightarrow{\boldsymbol{x}}, \overrightarrow{\boldsymbol{r}} \in \mathsf{Var}^*, \mathbb{C} \in \mathcal{L}_{\mathsf{Cmd}}, \gamma' \in \mathsf{PDef}^* \text{ s.t.} \\
& \quad\quad ((\overrightarrow{\boldsymbol{x}}, \mathbb{C}, \overrightarrow{\boldsymbol{r}}), \gamma') = \mathrm{lookup}(\mathtt{f}, \gamma) \text{ and} \\
& \quad\quad \gamma' \models^{\mathbb{C}} \begin{array}{l} \{\exists \overrightarrow{v}.\, \{\overrightarrow{\boldsymbol{x}} \Rightarrow \overrightarrow{v} * \overrightarrow{\boldsymbol{r}} \Rightarrow -\} \times \mathsf{P}(\overrightarrow{v})\} \\ \{\exists \overrightarrow{w}.\, \{\overrightarrow{\boldsymbol{x}} \Rightarrow - * \overrightarrow{\boldsymbol{r}} \Rightarrow \overrightarrow{w}\} \times \mathsf{Q}(\overrightarrow{w})\} \end{array} \\
\Gamma \models \{P\}\ \mathbb{C}\ \{Q\} \iff & \text{ for all } \gamma \in \mathsf{PDef}^*,\ \gamma \models \Gamma \implies \gamma \models \{P\}\ \mathbb{C}\ \{Q\}
\end{aligned}$$

Assumption 5.6 (Axiom Soundness)**.** For all $\varphi \in \mathsf{Cmd}$, for all $(P,Q) \in \mathrm{Ax}\,[\![\varphi]\!]$ and for all $w \in P$, $\mathcal{C}\,[\![\varphi]\!]\,(\rho,\chi)$ is defined and $\mathcal{C}\,[\![\varphi]\!]\,(\rho,\chi) \subseteq Q$.

Assumption 5.7 (Primitive Locality)**.** For all $\varphi \in \mathsf{Cmd}$, $s, s' \in \mathsf{State}$ and $c \in \mathsf{C}_{\mathsf{State}}$, if $\mathcal{C}\,[\![\varphi]\!]\,(s)$ is defined then $\mathcal{C}\,[\![\varphi]\!]\,(c \bullet s)$ is also defined and for every $s' \in \mathcal{C}\,[\![\varphi]\!]\,(c \bullet s)$ there is a $s'' \in \mathcal{C}\,[\![\varphi]\!]\,(s)$ with $s' = c \bullet s''$.

Assumption 5.8 (Expression Locality)**.** For all value expressions $E \in \mathsf{Expr}$, and all scopes $\rho, \rho' \in \mathsf{Scope}$ with $\mathcal{E}\,[\![E]\!]\,\rho$ and $\rho * \rho'$ both defined, $\mathcal{E}\,[\![E]\!]\,(\rho * \rho') = \mathcal{E}\,[\![E]\!]\,\rho$. Similarly, for all Boolean expressions $B \in \mathsf{BExp}$, and all scopes $\rho, \rho' \in \mathsf{Scope}$ with $\mathcal{B}\,[\![B]\!]\,\rho$ and $\rho * \rho'$ both defined, $\mathcal{B}\,[\![B]\!]\,(\rho * \rho') = \mathcal{B}\,[\![B]\!]\,\rho$.

The above assumption is simple to establish for the semantics of basic arithmetic and Boolean expressions. Indeed, typically the only expression constructor that is not indifferent to the variable store is variable lookup.

Lemma 88 (Operational Locality)**.** *For all* $\mathbb{C} \in \mathcal{L}_{\mathsf{Cmd}}$, $\gamma \in \mathsf{PDef}^*$, $s \in \mathsf{State}$, $c \in \mathsf{C}_{\mathsf{State}}$, $o \in \mathsf{Outcome}$, *if*

$$\mathbb{C}, \gamma, s \not\rightsquigarrow \lightning \tag{5.1}$$

$$\mathbb{C}, \gamma, c \bullet s \rightsquigarrow o \tag{5.2}$$

then $o \neq \lightning$ *and* $o = c \bullet s'$ *for some* s' *with*

$$\mathbb{C}, \gamma, s \rightsquigarrow s'. \tag{5.3}$$

Proof. The proof is by induction on the structure of the derivation of (5.2). Where necessary, assume $s = (\rho,\chi)$ and $c = (c_\rho, c_\chi)$.

$\mathbb{C} = \varphi \in \mathsf{Cmd}$ cases:

If $o = \lightning$ then $\mathcal{C}[\![\varphi]\!](c \bullet s)$ is undefined and so $\mathcal{C}[\![\varphi]\!](s)$ is also undefined (by Primitive Locality), which contradicts (5.1). Otherwise, Primitive Locality means that $\mathcal{C}[\![\varphi]\!]\, s = s'$ for some s' such that $\mathcal{C}[\![\varphi]\!](c \bullet s) = o = c \bullet s'$, and hence (5.3).

$\mathbb{C} = \texttt{skip}$ case:

This case is trivial, since it must be that $o = c \bullet s$.

$\mathbb{C} = \mathbb{C}_1; \mathbb{C}_2$ cases:

The derivation of (5.2) must have a subderivation

$$\mathbb{C}_1, \gamma, c \bullet s \rightsquigarrow o'$$

for some $o' \in \mathsf{Outcome}$. By (5.1) and the operational semantics,

$$\mathbb{C}_1, \gamma, s \not\rightsquigarrow \lightning$$

and so by the inductive hypothesis, $o' \neq \lightning$ and $o' = c \bullet s''$ for some s'' with

$$\mathbb{C}_1, \gamma, s \rightsquigarrow s''. \tag{5.4}$$

The condition that $o' \neq \lightning$ rules out the derivation for (5.2) in which $\mathbb{C}_1$ faults. The remaining cases for the derivation of (5.2) both have a subderivation of the form

$$\mathbb{C}_2, \gamma, c \bullet s'' \rightsquigarrow o.$$

By (5.1), (5.4) and the operational semantics

$$\mathbb{C}_2, \gamma, s'' \not\rightsquigarrow \lightning$$

and so by the inductive hypothesis, $o \neq \lightning$ and $o = c \bullet s'$ for some s' with

$$\mathbb{C}_2, \gamma, s'' \rightsquigarrow s'.$$

Finally, (5.3) holds by the operational semantics.

$\mathbb{C} = \boldsymbol{x} := E$ cases:

Suppose that $o = \lightning$. Thus must mean that either $\mathcal{E}[\![E]\!](c_\rho * \rho)$ is undefined or $\boldsymbol{x} \notin \mathrm{dom}(c_\rho * \rho)$. In the first case, Expression Locality implies that $\mathcal{E}[\![E]\!]\, \rho$ is also undefined, which contradicts (5.1); in the second case, it must be that $\boldsymbol{x} \notin \mathrm{dom}\, \rho$, which also contradicts (5.1). Thus, it cannot be that $o = \lightning$. The remaining case requires that $\boldsymbol{x} \in \mathrm{dom}(c_\rho * \rho)$ and $\mathcal{E}[\![E]\!](c_\rho * \rho) = v'$ for some v' with $o = ((c_\rho * \rho)[\boldsymbol{x} \mapsto v'], c_\chi \bullet \chi)$. By (5.1), $\boldsymbol{x} \in \mathrm{dom}\, \rho$ and $\mathcal{E}[\![E]\!]\, \rho = v'$ by Expression Locality. Consequently, $o = c \bullet (\rho[\boldsymbol{x} \mapsto v'], \chi)$ and, by the operational semantics,

$$\mathbb{C}, \gamma, s \rightsquigarrow \rho[\boldsymbol{x} \mapsto v'], \chi$$

as required.

$\mathbb{C} = \texttt{if } B \texttt{ then } \mathbb{C}_{\mathbf{T}} \texttt{ else } \mathbb{C}_{\mathbf{F}}$ cases:
Suppose that $\mathcal{B}[\![B]\!](c_\rho * \rho)$ is undefined. By Expression Locality, $\mathcal{B}[\![B]\!]\rho$ must also be undefined, which contradicts (5.1). Hence, $\mathcal{B}[\![B]\!](c_\rho * \rho) = b$ for some $b \in \{\mathbf{T}, \mathbf{F}\}$, and the derivation of (5.2) has a subderivation

$$\mathbb{C}_b, \gamma, c \bullet s \rightsquigarrow o.$$

By (5.1) and Expression Locality, $\mathcal{B}[\![B]\!]\rho = b$. Furthermore, (5.1) gives that

$$\mathbb{C}_b, \gamma, s \not\rightsquigarrow \lightning .$$

By the inductive hypothesis, it must therefore be that $o \neq \lightning$ and $o = c \bullet s'$ for some s' with

$$\mathbb{C}_b, \gamma, s \rightsquigarrow s'.$$

Consequently, (5.3) holds by the operational semantics.

$\mathbb{C} = \texttt{while } B \texttt{ do } \mathbb{C}'$ cases:
Suppose that $\mathcal{B}[\![B]\!](c_\rho * \rho)$ is undefined. By Expression Locality, $\mathcal{B}[\![B]\!]\rho$ must also be undefined, which contradicts (5.1).

Suppose instead that $\mathcal{B}[\![B]\!](c_\rho * \rho) = \mathbf{F}$. The derivation of (5.2) requires that $o = c \bullet s$. By (5.1) and Expression Locality, $\mathcal{B}[\![B]\!]\rho = \mathbf{F}$ also, and the operational semantics gives (5.3).

Suppose finally that $\mathcal{B}[\![B]\!](c_\rho * \rho) = \mathbf{T}$. The derivation of (5.2) must have a subderivation

$$\mathbb{C}'; \mathbb{C}, \gamma, c \bullet s \rightsquigarrow o.$$

By (5.1) and Expression Locality, $\mathcal{B}[\![B]\!]\rho = \mathbf{T}$ and furthermore

$$\mathbb{C}'; \mathbb{C}, \gamma, s \not\rightsquigarrow o.$$

By the inductive hypothesis, it must therefore be that $o \neq \lightning$ and $o = c \bullet s'$ for some s' with

$$\mathbb{C}'; \mathbb{C}, \gamma, s \rightsquigarrow s'.$$

Consequently, (5.3) holds by the operational semantics.

$\mathbb{C} = \texttt{procs } \overrightarrow{r_1} := \mathtt{f}_1(\overrightarrow{x_1})\ \{\mathbb{C}_1\}, \ldots, \overrightarrow{r_k} := \mathtt{f}_k(\overrightarrow{x_k})\ \{\mathbb{C}_k\} \texttt{ in } \mathbb{C}'$ cases:
The derivation of (5.2) must have a subderivation

$$\mathbb{C}', \gamma', c \bullet s \rightsquigarrow o$$

where $\gamma' = [\mathtt{f}_1 \mapsto (\overrightarrow{x_1}, \mathbb{C}_1, \overrightarrow{r_1}), \ldots, \mathtt{f}_k \mapsto (\overrightarrow{x_k}, \mathbb{C}_k, \overrightarrow{r_k})] \cdot \gamma$. By (5.1), it must be that

$$\mathbb{C}', \gamma', s \not\rightsquigarrow \lightning$$

and so, by the inductive hypothesis, $o = c \bullet s'$ for some s' with

$$\mathbb{C}', \gamma', s \rightsquigarrow s'.$$

Consequently, (5.3) holds by the operational semantics.

$\mathbb{C} = \texttt{call } \boldsymbol{y}_1, \ldots, \boldsymbol{y}_i := \texttt{f}(E_1, \ldots, E_j)$ cases:
There are five possible derivations for (5.2). Suppose that $o = \lightning$ on account of $\text{lookup}(\texttt{f}, \gamma)$; this would mean that the program would also fault on s, violating (5.1). Suppose that $o = \lightning$ because $\mathcal{E}[\![E_k]\!](c_\rho * \rho)$ is undefined for some $1 \leq k \leq j$; Expression Locality would mean that $\mathcal{E}[\![E_k]\!]\rho$ would also be undefined, and so the program would fault on s, again violating (5.1). Suppose that $o = \lightning$ because $\boldsymbol{y}_k \notin \operatorname{dom} c_\rho * \rho$ for some $1 \leq k \leq i$; this would mean that $\boldsymbol{y} \notin \operatorname{dom} \rho$, again violating (5.1).

The remaining two possible derivations require a subderivation

$$\mathbb{C}, \gamma', \emptyset[\boldsymbol{r}_1 \mapsto w_1] \ldots [\boldsymbol{r}_i \mapsto w_i][\boldsymbol{x}_1 \mapsto v_1] \ldots [\boldsymbol{x}_j \mapsto v_j], c_\chi \bullet \chi \rightsquigarrow o'$$

where

$$\begin{gathered}\text{lookup}(\texttt{f}, \gamma) = (((\boldsymbol{x}_1, \ldots, \boldsymbol{x}_j), \mathbb{C}', (\boldsymbol{r}_1, \ldots, \boldsymbol{r}_i)), \gamma') \\ \text{for all } 1 \leq k \leq j,\ v_k = \mathcal{E}[\![E_k]\!](c_\rho * \rho).\end{gathered}$$

By (5.1) and Expression locality, it must be that $v_k = \mathcal{E}[\![E_k]\!]\rho$, for each $1 \leq k \leq j$. Also by (5.1), it must be that

$$\mathbb{C}, \gamma', \emptyset[\boldsymbol{r}_1 \mapsto w_1] \ldots [\boldsymbol{r}_i \mapsto w_i][\boldsymbol{x}_1 \mapsto v_1] \ldots [\boldsymbol{x}_j \mapsto v_j], \chi \not\rightsquigarrow \lightning.$$

Hence, by the inductive hypothesis, $o' \neq \lightning$ and $o' = (\emptyset, c_\chi) \bullet (\rho'', \chi')$ for some ρ'', χ' with

$$\mathbb{C}, \gamma', \emptyset[\boldsymbol{r}_1 \mapsto w_1] \ldots [\boldsymbol{r}_i \mapsto w_i][\boldsymbol{x}_1 \mapsto v_1] \ldots [\boldsymbol{x}_j \mapsto v_j], \chi \rightsquigarrow \rho'', \chi'.$$

Let $\rho' = \rho[\boldsymbol{y}_1 \mapsto \rho''\boldsymbol{r}_1] \ldots [\boldsymbol{y}_i \mapsto \rho''\boldsymbol{r}_i]$ and observe that by (5.1) and the operational semantics,

$$\mathbb{C}, \gamma, \rho, \chi \rightsquigarrow \rho', \chi'.$$

Moreover,

$$\begin{aligned}(c_\rho * \rho)[\boldsymbol{y}_1 \mapsto \rho''\boldsymbol{r}_1] \ldots [\boldsymbol{y}_i \mapsto \rho''\boldsymbol{r}_i] &= c_\rho * (\rho[\boldsymbol{y}_1 \mapsto \rho''\boldsymbol{r}_1] \ldots [\boldsymbol{y}_i \mapsto \rho''\boldsymbol{r}_i]) \\ &= c_\rho * \rho'.\end{aligned}$$

Hence, by the operational semantics $o = (c_\rho * \rho', c_\chi \bullet \chi') = c \bullet (\rho', \chi')$, as required.

$\mathbb{C} = \texttt{local } \boldsymbol{x} \texttt{ in } \mathbb{C}'$ cases:
The derivation of (5.2) must have a subderivation

$$\mathbb{C}', \gamma, (c_\rho * \rho)[\boldsymbol{x} \mapsto v], c_\chi \bullet \chi \rightsquigarrow o'.$$

For some c'_ρ with $\boldsymbol{x} \notin \operatorname{dom} c'_\rho$, either $c_\rho = c'_\rho$ or $c_\rho = c'_\rho[\boldsymbol{x} \mapsto v']$ for some v'. Note that $(c_\rho * \rho)[\boldsymbol{x} \mapsto v] = c'_\rho * (\rho[\boldsymbol{x} \mapsto v])$. By (5.1),

$$\mathbb{C}', \gamma, \rho[\boldsymbol{x} \mapsto v], \chi \not\rightsquigarrow \lightning .$$

Hence, by the inductive hypothesis, $o' \neq \lightning$ and $o' = (c'_\rho, c_\chi) \bullet (\rho'', \chi')$ for some ρ'', χ' with

$$\mathbb{C}', \gamma, \rho[\boldsymbol{x} \mapsto v], \chi \rightsquigarrow \rho'', \chi'.$$

This rules out the possibility that $o = \lightning$. The operational semantics rules require then that $\boldsymbol{x} \in \operatorname{dom}(c'_\rho * \rho'')$, and so it must be that $\boldsymbol{x} \in \operatorname{dom} \rho''$. Let $w = \rho''\boldsymbol{x}$. Let ρ' be such that $\boldsymbol{x} \notin \operatorname{dom} \rho'$ and $\rho'' = \rho'[\boldsymbol{x} \mapsto w]$. Now

$$\begin{aligned} c'_\rho * \rho'' &= c'_\rho * (\rho'[\boldsymbol{x} \mapsto w]) \\ &= (c'_\rho * \rho')[\boldsymbol{x} \mapsto w] \\ &= (c_\rho * \rho')[\boldsymbol{x} \mapsto w] \end{aligned}$$

If $\boldsymbol{x} \notin \operatorname{dom}(c_\rho * \rho)$ then it must be that $o = (c_\rho * \rho', c_\chi \bullet \chi') = c \bullet (\rho', \chi')$. By the operational semantics,

$$\mathbb{C}, \gamma, \rho, \chi \rightsquigarrow \rho', \chi'$$

as required.

Otherwise, either $\boldsymbol{x} \in \operatorname{dom} c_\rho$ or $\boldsymbol{x} \in \operatorname{dom} \rho$ (the two cases being mutually exclusive). If the former, then it must be that $o = ((c_\rho * \rho')[\boldsymbol{x} \mapsto c_\rho \boldsymbol{x}], c_\chi \bullet \chi') = c \bullet (\rho', \chi')$. As before, the operational semantics establishes (5.3) for $s' = (\rho', \chi')$. If the latter, then it must be that $o = ((c_\rho * \rho')[\boldsymbol{x} \mapsto \rho\boldsymbol{x}], c_\chi \bullet \chi') = c \bullet (\rho'[\boldsymbol{x} \mapsto \rho\boldsymbol{x}], \chi')$. By the operational semantics,

$$\mathbb{C}, \gamma, \rho, \chi \rightsquigarrow \rho'[\boldsymbol{x} \mapsto \rho\boldsymbol{x}], \chi'$$

as required. □

Theorem 89 (Soundness)**.**

$$\Gamma \vdash \{P\}\ \mathbb{C}\ \{Q\} \implies \Gamma \models \{P\}\ \mathbb{C}\ \{Q\}\,.$$

Proof. The proof is by induction on the structure of the derivation of $\Gamma \vdash \{P\}\ \mathbb{C}\ \{Q\}$. Consider each case for the last rule applied.

AXIOM case:
In this case, $\mathbb{C} = \varphi$ for some $\varphi \in \mathsf{Cmd}$ and $(P, Q) \in \mathrm{Ax}[\![\varphi]\!]$. Suppose that $\gamma \models \Gamma$, that $(\rho, \chi) \in P$ and that $o \in \mathsf{Outcome}$ is such that $\varphi, \gamma, \rho, \chi \rightsquigarrow o$. If $o = \lightning$ then the operational semantics requires that $\mathcal{C}[\![\varphi]\!](\rho, \chi)$ is undefined, which violates the assumption of Axiom Soundness. Thus, $o = (\rho', \chi')$ for some $(\rho', \chi') \in \mathcal{C}[\![\varphi]\!](\rho, \chi)$. Axiom Soundness implies that $(\rho', \chi') \in Q$, as required.

FRAME case:
In this case, $P = K \bullet P'$ and $Q = K \bullet Q'$ for some K, P', Q' with

$$\Gamma \models \{P'\}\ \mathbb{C}\ \{Q'\}$$

by the inductive hypothesis. Suppose that $\gamma \models \Gamma$ and that $s \in K \bullet P'$. Now it must be that $s = c \bullet s'$ for some $c \in K$ and $s' \in P'$. Furthermore, $\mathbb{C}, \gamma, s' \not\rightsquigarrow \lightning$. Suppose that o is such that $\mathbb{C}, \gamma, c \bullet s' \rightsquigarrow o$. By Operational Locality, it must be that $o \neq \lightning$ and $o = c \bullet s''$ for some s'' with $\mathbb{C}, \gamma, s' \rightsquigarrow s''$. By the assumption, it must be that $s'' \in Q'$, and hence $o = c \bullet s'' \in K \bullet Q'$, as required.

CONS case:
In this case, $P \subseteq P'$ and $Q' \subseteq Q'$ for some P', Q' with

$$\Gamma \models \{P'\}\ \mathbb{C}\ \{Q'\}$$

by the inductive hypothesis. Suppose that $\gamma \models \Gamma$ and that $s \in P$. It must be that $s \in P'$ also, and so for all o with $\mathbb{C}, \gamma, s \rightsquigarrow o$, $o \neq \lightning$ and $o = s' \in Q'$. Since $Q' \subseteq Q$, $s' \in Q$, as required.

DISJ case:
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

$$\Gamma \models \{P_i\}\ \mathbb{C}\ \{Q_i\}$$

for each $i \in I$, by the inductive hypothesis. Suppose that $s \in P$. Then $s \in P_i$ for some $i \in I$. Hence, for all o with $\mathbb{C}, \gamma, s \rightsquigarrow o$, $o \neq \lightning$ and $o = s' \in Q_i \subseteq Q$, as required.

IF case:
In this case, $\mathbb{C} = \texttt{if}\ B\ \texttt{then}\ \mathbb{C}_{\mathbf{T}}\ \texttt{else}\ \mathbb{C}_{\mathbf{F}}$ for some $B \in \mathsf{BExp}$ and $\mathbb{C}_{\mathbf{T}}, \mathbb{C}_{\mathbf{F}} \in \mathcal{L}_{\mathsf{Cmd}}$, $P \subseteq \mathit{bsafe}(B)$, and, by the inductive hypothesis,

$$\Gamma \models \{P \wedge \mathcal{P}[\![B]\!]\}\ \mathbb{C}_{\mathbf{T}}\ \{Q\}$$
$$\Gamma \models \{P \wedge \neg\mathcal{P}[\![B]\!]\}\ \mathbb{C}_{\mathbf{F}}\ \{Q\}\,.$$

4 Suppose that $s = (\rho, \chi) \in P$. Since $P \subseteq \mathit{bsafe}(B)$, $b = \mathcal{B}[\![B]\!]\,\rho$ is defined. Suppose that $\mathbb{C}, \gamma, s \rightsquigarrow o$ for some $\gamma \models \Gamma$ and o. Then the operational semantics

requires that $\mathbb{C}_b, \gamma, s \rightsquigarrow o$ also. By the semantic triples, $o \neq \lightning$ and $o \in Q$, as required.

WHILE case:
In this case, $\mathbb{C} = \texttt{while}\ B\ \texttt{do}\ \mathbb{C}'$ for some $B \in \mathsf{BExp}$ and $\mathbb{C}' \in \mathcal{L}_{\mathsf{Cmd}}$, $Q = P \wedge \neg \mathcal{P}[\![B]\!]$, $P \subseteq \mathit{bsafe}(B)$, and, by the inductive hypothesis,

$$\Gamma \models \{P \wedge \mathcal{P}[\![B]\!]\}\ \mathbb{C}'\ \{P\}\,.$$

Fix some $\gamma \in \mathsf{PDef}^*$ such that $\gamma \models \Gamma$. I claim that for all $s = (\rho, \chi) \in P$ and $o \in \mathsf{Outcome}$ with $\mathbb{C}, \gamma, s \rightsquigarrow o$, it is the case that $o \neq \lightning$ and $o \in Q$. This claim immediately establishes that the required semantic triple holds. The proof of the claim is by induction on the structure of the operational semantic derivation.

Since $P \subseteq \mathit{bsafe}(B)$, either $\mathcal{B}[\![B]\!]\,\rho = \mathbf{T}$ or $\mathcal{B}[\![B]\!]\,\rho = \mathbf{F}$. If the former then $\mathbb{C}'; \mathbb{C}, \gamma, s \rightsquigarrow o$, and hence either $\mathbb{C}', \gamma, s \rightsquigarrow \lightning$ (in which case $o = \lightning$) or $\mathbb{C}', \gamma, s \rightsquigarrow s'$ and $\mathbb{C}, \gamma, s' \rightsquigarrow o$ for some s'. By the assumption that $\Gamma \models \{P \wedge \mathcal{P}[\![B]\!]\}\ \mathbb{C}'\ \{P\}$, since $s \in P \wedge \mathcal{P}[\![B]\!]$, the faulting case cannot apply and $s' \in P$. Applying the inductive hypothesis, we can conclude that $o \neq \lightning$ and $o \in Q$, as required. If, on the other hand, $\mathcal{B}[\![B]\!]\,\rho = \mathbf{F}$ then it must be that $o = s \in P \wedge \neg \mathcal{P}[\![B]\!] = Q$, as required.

ASSGN case:
In this case, $\mathbb{C} = \boldsymbol{x} := E$ for some $\boldsymbol{x} \in \mathsf{Var}$ and $E \in \mathsf{Expr}$, $P = \{(\boldsymbol{x} \Rightarrow v * \rho, \chi)\} \subseteq \mathit{vsafe}(E)$ for some $v \in \mathsf{Val}$, $\rho \in \mathsf{Scope}$ and $\chi \in \mathsf{Store}$, and

$$Q = \{(\boldsymbol{x} \Rightarrow \mathcal{E}[\![E]\!]\,(\boldsymbol{x} \Rightarrow v * \rho) * \rho, \chi)\}\,.$$

Fix some $\gamma \in \mathsf{PDef}^*$ such that $\gamma \models \Gamma$. If $s \in P$ then $s = (\boldsymbol{x} \Rightarrow v * \rho, \chi)$. Let $o \in \mathsf{Outcome}$ be such that $\boldsymbol{x} := E, \gamma, \boldsymbol{x} \Rightarrow v * \rho, \chi \rightsquigarrow o$. By the definition of *vsafe*, there is some $v' \in \mathsf{Val}$ with $\mathcal{E}[\![E]\!]\,(\rho[\boldsymbol{x} \mapsto v]) = v'$. By the operational semantics, this means that $o \neq \lightning$, and $o = (\rho[\boldsymbol{x} \mapsto v], \chi) \in Q$, as required.

LOCAL case:
In this case, $\mathbb{C} = \texttt{local}\ \boldsymbol{x}\ \texttt{in}\ \mathbb{C}'$ for some $\boldsymbol{x} \in \mathsf{Var}$ and $\mathbb{C}' \in \mathcal{L}_{\mathsf{Cmd}}$, $P \wedge \mathit{vsafe}(\boldsymbol{x}) \equiv \emptyset$, and, by the inductive hypothesis,

$$\Gamma \models \{(\boldsymbol{x} \Rightarrow - \times \mathbf{I}_{\mathsf{Store}}) \bullet P\}\ \mathbb{C}'\ \{(\boldsymbol{x} \Rightarrow - \times \mathbf{I}_{\mathsf{Store}}) \bullet Q\}\,.$$

Fix some $\gamma \in \mathsf{PDef}^*$ such that $\gamma \models \Gamma$, some $(\rho, \chi) \in P$, and $o \in \mathsf{Outcome}$ with $\mathbb{C}, \gamma, \rho, \chi \rightsquigarrow o$. By the definition of *vsafe*, $(\rho[\boldsymbol{x} \mapsto v], \chi) \in (\boldsymbol{x} \Rightarrow - \times \mathbf{I}_{\mathsf{Store}}) \bullet P$ for every $v \in \mathsf{Val}$, and $\boldsymbol{x} \notin \operatorname{dom} \rho$. Therefore $o \neq \lightning$ and $o = (\rho', \chi')$ for some $\rho' \in \mathsf{Scope}$, $\chi' \in \mathsf{Store}$ with $\boldsymbol{x} \notin \operatorname{dom} \rho'$ and

$$\mathbb{C}', \gamma, \rho[\boldsymbol{x} \mapsto v], \chi \rightsquigarrow \rho'[\boldsymbol{x} \mapsto w], \chi'$$

for some w. It must be that $(\rho'[\boldsymbol{x} \mapsto w], \chi') \in (\boldsymbol{x} \Rightarrow - \times \mathbf{I}_{\mathsf{Store}}) \bullet Q$, and so $(\rho', \chi') \in Q$, as required.

PDEF case:

In this case $\mathbb{C} = \mathtt{procs}\ \overrightarrow{\boldsymbol{r}_1} := \mathtt{f}_1(\overrightarrow{\boldsymbol{x}_1})\ \{\mathbb{C}_1\}, \ldots, \overrightarrow{\boldsymbol{r}_k} := \mathtt{f}_k(\overrightarrow{\boldsymbol{x}_k})\ \{\mathbb{C}_k\}\ \mathtt{in}\ \mathbb{C}'$, Γ makes no reference to any $\mathtt{f}_i$, and, for some Γ' that refers only to the $\mathtt{f}_i$ procedures,

$$\Gamma, \Gamma' \models \{P\}\ \mathbb{C}\ \{Q\}$$

$$\text{for all } (\mathtt{f}_i : \mathsf{P} \rightarrowtail \mathsf{Q}) \in \Gamma,\ \Gamma, \Gamma' \models \begin{array}{c} \{\exists \overrightarrow{v}.\, \overrightarrow{\boldsymbol{x}_i} \Rightarrow \overrightarrow{v} * \overrightarrow{\boldsymbol{r}_i} \Rightarrow - \times \mathsf{P}(\overrightarrow{v})\} \\ \mathbb{C}_i \\ \{\exists \overrightarrow{w}.\, \overrightarrow{\boldsymbol{x}_i} \Rightarrow - * \overrightarrow{\boldsymbol{r}_i} \Rightarrow \overrightarrow{w} \times \mathsf{Q}(\overrightarrow{w})\} \end{array}$$

by the inductive hypothesis. Fix some $\gamma \in \mathsf{PDef}^*$ with $\gamma \models \Gamma$, and suppose that $s \in P$ and $o \in \mathsf{Outcome}$ are such that $\mathbb{C}, \gamma, s \rightsquigarrow o$. By the operational semantics, it must be that

$$\mathbb{C}', [\mathtt{f}_1 \mapsto (\overrightarrow{\boldsymbol{x}_1}, \mathbb{C}_1, \overrightarrow{\boldsymbol{r}_1}), \ldots, \mathtt{f}_k \mapsto (\overrightarrow{\boldsymbol{x}_k}, \mathbb{C}_k, \overrightarrow{\boldsymbol{r}_k})] \cdot \gamma, s \rightsquigarrow o.$$

Now it must be that $[\mathtt{f}_1 \mapsto (\overrightarrow{\boldsymbol{x}_1}, \mathbb{C}_1, \overrightarrow{\boldsymbol{r}_1}), \ldots, \mathtt{f}_k \mapsto (\overrightarrow{\boldsymbol{x}_k}, \mathbb{C}_k, \overrightarrow{\boldsymbol{r}_k})] \cdot \gamma \models \Gamma, \Gamma'$, by the semantic triples for the procedure bodies and the fact that $\gamma \models \Gamma$. Hence, by the semantic triple for $\mathbb{C}'$, $o \neq \lightning$ and $o = q' \in Q$, as required.

PCALL case:

In this case, for some $(\mathtt{f} : \mathsf{P} \rightarrowtail \mathsf{Q}) \in \Gamma$, $\mathbb{C} = \mathtt{call}\ \overrightarrow{\boldsymbol{r}} := \mathtt{f}(\overrightarrow{E})$, $P = \{(\overrightarrow{\boldsymbol{r}} \Rightarrow \overrightarrow{v} * \rho')\} \times \mathsf{P}\left(\mathcal{E}\left[\!\left[\overrightarrow{E}\right]\!\right](\overrightarrow{\boldsymbol{r}} \Rightarrow \overrightarrow{v} * \rho')\right)$ and $Q = \exists \overrightarrow{w}.\, \{(\overrightarrow{\boldsymbol{r}} \Rightarrow \overrightarrow{w} * \rho')\} \times \mathsf{Q}(\overrightarrow{w})$, with $\{(\overrightarrow{\boldsymbol{r}} \Rightarrow \overrightarrow{v} * \rho)\} \times \mathsf{Store} \subseteq \mathit{vsafe}(\overrightarrow{E})$. Fix $\gamma \in \mathsf{PDef}^*$ with $\gamma \models \Gamma$, and suppose that $(\rho, \chi) \in P$ and $o \in \mathsf{Outcome}$ are such that $\mathbb{C}, \gamma, \rho, \chi \rightsquigarrow o$. It must be that, for some $\overrightarrow{\boldsymbol{x}}, \overrightarrow{\boldsymbol{y}} \in \mathsf{Var}^*$, $\mathbb{C}' \in \mathcal{L}_{\mathsf{Cmd}}$ and $\gamma' \in \mathsf{PDef}^*$ with $((\overrightarrow{\boldsymbol{x}}, \mathbb{C}', \overrightarrow{\boldsymbol{y}}), \gamma') = \mathrm{lookup}(\mathtt{f}, \gamma)$,

$$\gamma' \models \begin{array}{c} \{\exists \overrightarrow{u}.\, \{\overrightarrow{\boldsymbol{x}} \Rightarrow \overrightarrow{u} * \overrightarrow{\boldsymbol{y}} \Rightarrow -\} \times \mathsf{P}(\overrightarrow{u})\} \\ \mathbb{C}' \\ \{\exists \overrightarrow{w}.\, \{\overrightarrow{\boldsymbol{x}} \Rightarrow - * \overrightarrow{\boldsymbol{y}} \Rightarrow \overrightarrow{w}\} \times \mathsf{Q}(\overrightarrow{w})\} \end{array}. \tag{5.5}$$

Since $\mathrm{lookup}(\mathtt{f}, \gamma)$ is defined and has the correct type (enforced by the types of P and Q) the first faulting case does not apply. By the *vsafe* condition, it must be that $\overrightarrow{u} = \mathcal{E}\left[\!\left[\overrightarrow{E}\right]\!\right](\overrightarrow{\boldsymbol{r}} \Rightarrow \overrightarrow{v} * \rho')$ is defined, and so the second faulting case does not apply either. If the third faulting case applied, then, for some $\overrightarrow{w}$

$$\mathbb{C}', \gamma', \overrightarrow{\boldsymbol{x}} \Rightarrow \overrightarrow{u} * \overrightarrow{\boldsymbol{y}} \Rightarrow \overrightarrow{w}, \chi \rightsquigarrow \lightning.$$

Yet, since $\chi \in \mathsf{P}(\overrightarrow{u})$, this would violate (5.5), and so the third faulting case does not apply. The fourth and final faulting case is ruled out by the fact that P stipulates that each return variable in $\overrightarrow{\boldsymbol{r}}$ must be in $\mathrm{dom}\,\rho$. This leaves only the

successful case, which requires that $o = \rho'', \chi'$ for some $\rho'' \in \mathsf{Scope}$, $\chi' \in \mathsf{Store}$ with $\rho'' = \rho[\overrightarrow{\boldsymbol{r}} \mapsto \rho' \overrightarrow{\boldsymbol{y}}]$ for some ρ' with

$$\mathbb{C}', \gamma', \overrightarrow{\boldsymbol{x}} \Rightarrow \overrightarrow{u} * \overrightarrow{\boldsymbol{y}} \Rightarrow \overrightarrow{w}, \chi \rightsquigarrow \rho', \chi'.$$

By (5.5), it must be that $\chi' \in \mathsf{Q}\,(\rho' \overrightarrow{\boldsymbol{y}})$ and so $(\rho'', \chi') \in Q$, as required.

PWk case:

In this case, $\Gamma = \Gamma_1, \Gamma_2$ for some Γ_1, Γ_2 with $\Gamma_1 \models \{P\}\ \mathbb{C}\ \{Q\}$ by the inductive hypothesis. Suppose that $\gamma \in \mathsf{PDef}^*$ with $\gamma \models \Gamma$. Then $\gamma \models \Gamma_1$ also, and so $\gamma \models \{P\}\ \mathbb{C}\ \{Q\}$, as required. □

Chapter 6

Locality Refinement

In this chapter, I show how local reasoning about an abstract module can be justified by local reasoning about the module's implementation. Essentially, this amounts to showing that the module implementation *refines* the specification implicit in the abstract local reasoning. I present two techniques for establishing this refinement: *locality-preserving* and *locality-breaking* translations.

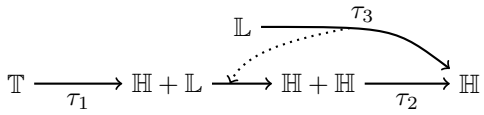


Figure 6.1: Module translations

The development is motivated by examples. In particular, I demonstrate a stepwise refinement of an abstract tree module ($\mathbb{T}$) first to an implementation that uses the heap ($\mathbb{H}$) and an abstract list module ($\mathbb{L}$) and then, by implementing the list module itself in the heap, to an implementation that is ultimately based on the heap module. Figure 6.1 illustrates this development.

The translation τ_1 from the tree module $\mathbb{T}$ to the combined heap-and-list module $\mathbb{H} + \mathbb{L}$ considered as the motivating example of a locality-preserving translation in §6.3, and its soundness is formally established in §6.3.2. The translation τ_3 from the list module $\mathbb{T}$ to the heap module $\mathbb{L}$ is considered as the motivating example of a locality-breaking translation in §6.4, and its soundness is formally established in §6.4.2. The modularity of the translation technique means that this translation can be lifted to a sound translation from the combined heap-and-list module $\mathbb{H} + \mathbb{L}$ to the double-heap module $\mathbb{H} + \mathbb{H}$ which preserves the heap module. The final step of the development, translation τ_2 from the double-heap $\mathbb{H} + \mathbb{H}$ to the single heap $\mathbb{H}$ is another, albeit simple,

example of a locality-preserving translation, considered in §6.3.3.

Before I embark on tackling this refinement, I formalise the concept of a module for the purposes of this chapter (§6.1), and then the concept of a sound module translation (§6.2). I then introduce the two techniques and establish their soundness in general, illustrating their application with the examples described above. In §6.5 I consider a number of technical points arising from the work.

Collaboration and Contribution

The work presented in this chapter was undertaken in collaboration with Wheelhouse and Gardner, and published in [DYGW10a]. An extended technical report [DYGW10b] contains additional proof details. The two techniques developed from ideas suggested by Uri Zarfaty and Mohammad Raza. My own particular contribution was in formalising and establishing the soundness of the techniques, which was undertaken in close collaboration with Wheelhouse.

6.1 Abstract Modules

In Chapter 5, I introduced a programming language, $\mathcal{L}_{\mathsf{Cmd}}$, that was parametrised by a set of basic commands, Cmd. For this language, I gave an operational semantics, $\rightsquigarrow$, that was parametrised by a state space, Store, and a denotational semantics for the basic commands, $\mathcal{C}[\![(\cdot)]\!]$. I also gave an axiomatic semantics, $\vdash$, that was parametrised by a context algebra, $(\mathsf{Store}, \mathsf{C}_{\mathsf{Store}}, \bullet, \mathbf{I}_{\mathsf{Store}})$, and a set of axioms for the basic commands, $\mathrm{Ax}[\![(\cdot)]\!]$.[1] The axiomatic semantics was shown to be sound with respect to the operational semantics under certain conditions relating the parameters.

The purpose of parametrising the language in this way was to apply it to different levels of abstraction. A number of different abstractions are considered in this chapter, including heaps (the standard model used in separation logic [IO01, Rey02]), trees (the original motivation for context logic [CGZ05]) and lists. Many more abstractions are possible, such as the heap-with-free-set model of Raza and Gardner [RG09], the DOM model of Gardner *et al.* [GSWZ08] and the segment logic model of Gardner and Wheelhouse [GW09].[2] These levels

[1] The syntax and semantics of expressions were also parameters, however, I will assume a fixed choice for these.

[2] As presented, the last two need some manipulation to fit the context algebra mould used here. The DOM model comprises different types of contexts and data structures; a context algebra may be formed from these by taking their (disjoint) unions to give a single set of contexts and data structures, albeit discriminated by application and composition. The

of abstraction are captured by the notion of an *abstract module*, which constitutes the parameters necessary to determine a programming language and its axiomatic semantics. (The operational semantics at arbitrary levels of abstraction is not of interest here, and so the semantics of the basic commands are not included in the definition of an abstract module.)

Definition 6.1 (Abstract Module)**.** An *abstract module*

$$\mathbb{A} = (\mathsf{Cmd}_{\mathbb{A}}, \mathcal{A}_{\mathbb{A}}, \mathrm{Ax}\,[\![(\cdot)]\!]_{\mathbb{A}})$$

consists of:

- a set of basic commands, $\mathsf{Cmd}_{\mathbb{A}}$;
- a context algebra $\mathcal{A}_{\mathbb{A}} = (\mathsf{D}_{\mathbb{A}}, \mathsf{C}_{\mathbb{A}}, \circ_{\mathbb{A}}, \bullet_{\mathbb{A}}, \mathbf{I}_{\mathbb{A}})$; and
- an axiomatisation for the basic commands

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

  where $\mathsf{State}_{\mathbb{A}} = \mathsf{Scope} \times \mathsf{D}_{\mathbb{A}}$.

Notation. The language determined by the abstract module $\mathbb{A}$ is denoted $\mathcal{L}_{\mathbb{A}}$ (this is in fact $\mathcal{L}_{\mathsf{Cmd}_{\mathbb{A}}}$). The axiomatic semantic judgement determined by the abstract module $\mathbb{A}$ is denoted $\vdash_{\mathbb{A}}$. When the abstract module $\mathbb{A}$ can be inferred from context, the subscript $_{\mathbb{A}}$ may be dropped.

6.1.1 Heap Module

The abstract heap module $\mathbb{H} \stackrel{\text{def}}{=} (\mathsf{Cmd}_{\mathbb{H}}, \mathcal{A}_{\mathbb{H}}, \mathrm{Ax}\,[\![(\cdot)]\!]_{\mathbb{H}})$ should be familiar to aficionados of separation logic; its commands consist of heap allocation, disposal, mutation and lookup. Heaps are modelled as (effectively) finite partial functions from heap addresses (Addr) to values (Val), as in §2.3.2. The address set is assumed to be the positive integers, *i.e.* $\mathsf{Addr} = \mathbb{Z}^+$, and contained within the value set, *i.e.* $\mathsf{Addr} \subseteq \mathsf{Val}$. 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).

Definition 6.2 (Heap Update Commands)**.** The set of *heap update commands* $\mathsf{Cmd}_{\mathbb{H}}$, ranged over by φ, is defined as follows:

$$\varphi ::= \boldsymbol{x} := \mathtt{alloc}(E) \;\mid\; \mathtt{dispose}(E_1, E_2) \;\mid\; [E_1] := E_2 \;\mid\; \boldsymbol{x} := [E]$$

segment logic model includes a separation-logic-style $*$ and a restriction operator; contexts in this sense may be defined as a segment to be adjoined by $*$ plus a set of labels to restrict.

where $\boldsymbol{x} \in \mathsf{Var}$ ranges over variables and $E, E_1, E_2 \in \mathsf{Expr}$ range over value expressions.

Definition 6.3 (Heap Context Algebra)**.** The *heap context algebra* is $\mathcal{A}_{\mathbb{H}} \stackrel{\text{def}}{=} (\mathsf{Heap}, \mathsf{Heap}, *, *, \{\mathrm{emp}\})$ as given in Definition 2.37.

Definition 6.4 (Heap Axiomatisation)**.** The *heap axiomatisation*, $\mathrm{Ax}\,[\![(\cdot)]\!]_{\mathbb{H}} : \mathsf{Cmd}_{\mathbb{H}} \to \mathcal{P}(\mathcal{P}(\mathsf{Scope} \times \mathsf{Heap}) \times \mathcal{P}(\mathsf{Scope} \times \mathsf{Heap}))$ is defined as follows:

$$\mathrm{Ax}\,[\![\boldsymbol{x} := \mathtt{alloc}(E)]\!]_{\mathbb{H}} \stackrel{\text{def}}{=} \left\{ \left(\begin{array}{l} (\boldsymbol{x} \Rrightarrow v * \rho \times \mathrm{emp} \wedge w \geq 1)\,, \\ \left(\begin{array}{l} \exists a.\, \boldsymbol{x} \Rrightarrow a * \rho \times \\ \quad a \mapsto - * \cdots * (a + w) \mapsto - \end{array} \right) \end{array} \right) \;\middle|\; w = \mathcal{E}\,[\![E]\!]\,(\boldsymbol{x} \Rrightarrow v * \rho) \right\}$$

$$\mathrm{Ax}\,[\![\mathtt{dispose}(E_1, E_2)]\!]_{\mathbb{H}} \stackrel{\text{def}}{=} \left\{ \big((\rho \times a \mapsto - * \cdots * (a + v) \mapsto -)\,, (\rho \times \mathrm{emp}) \big) \;\middle|\; \begin{array}{r} a = \mathcal{E}\,[\![E_1]\!]\,\rho \text{ and} \\ v = \mathcal{E}\,[\![E_2]\!]\,\rho \end{array} \right\}$$

$$\mathrm{Ax}\,[\![[E_1] := E_2]\!]_{\mathbb{H}} \stackrel{\text{def}}{=} \left\{ \big((\rho \times a \mapsto -)\,, (\rho \times a \mapsto v) \big) \;\middle|\; a = \mathcal{E}\,[\![E_1]\!]\,\rho \text{ and } v = \mathcal{E}\,[\![E_2]\!]\,\rho \right\}$$

$$\mathrm{Ax}\,[\![\boldsymbol{x} := [E]]\!]_{\mathbb{H}} \stackrel{\text{def}}{=} \left\{ \big((\boldsymbol{x} \Rrightarrow v * \rho \times a \mapsto w)\,, (\boldsymbol{x} \Rrightarrow w * \rho \times a \mapsto w) \big) \;\middle|\; a = \mathcal{E}\,[\![E]\!]\,\rho \right\}$$

6.1.2 Tree Module

The abstract tree module $\mathbb{T} \stackrel{\text{def}}{=} (\mathsf{Cmd}_{\mathbb{T}}, \mathcal{A}_{\mathbb{T}}, \mathrm{Ax}\,[\![(\cdot)]\!]_{\mathbb{T}})$ should be familiar to adherents of context logic; its commands consist of node-relative traversal, node creation and subtree deletion. The tree model consists of *uniquely-labelled* trees, which resemble those considered in §2.1.1 except that each label may only occur once in any given tree or context. This is so that nodes in a tree are uniquely addressable by their labels. It is therefore assumed that the set of tree labels, Σ, is contained with in the value set, Val, *i.e.* $\Sigma \subseteq \mathsf{Val}$. It is also assumed that $\mathsf{Addr} \subseteq \Sigma$, in order that heap addresses can be used to implement node addresses.

Definition 6.5 (Tree Update Commands)**.** The set of *tree update commands* $\mathsf{Cmd}_{\mathbb{T}}$, ranged over by φ, is defined as follows:

$$\begin{aligned}\varphi ::= {} & \boldsymbol{x} := \mathtt{getUp}(E) \mid \boldsymbol{x} := \mathtt{getLeft}(E) \mid \boldsymbol{x} := \mathtt{getRight}(E) \\ & \mid \boldsymbol{x} := \mathtt{getFirst}(E) \mid \boldsymbol{x} := \mathtt{getLast}(E) \\ & \mid \mathtt{newNodeAfter}(E) \mid \mathtt{deleteTree}(E)\end{aligned}$$

where $\boldsymbol{x} \in \mathsf{Var}$ ranges over variables and $E \in \mathsf{Expr}$ ranges over value expressions.

Definition 6.6 (Uniquely-Labelled Tree Context Algebra)**.** The set of *uniquely-labelled trees* UTree, ranged over by $t, t_1, \ldots$, and the set of *uniquely-labelled tree contexts* $\mathsf{C}_{\mathsf{UTree}}$, ranged over by $c, c_1, \ldots$, are defined inductively as follows:

$$\begin{aligned}t ::= {} & \varnothing \mid \mathbf{a}[t] \mid t_1 \otimes t_2 \\ c ::= {} & \_ \mid \mathbf{a}[c] \mid c \otimes t \mid t \otimes c\end{aligned}$$

where $\otimes$ is considered to be associative and to have $\varnothing$ as its identity, and each label $\mathbf{a} \in \Sigma$ may occur *at most once* in any uniquely-labelled tree or context.

Context application $\bullet : \mathsf{C}_{\mathsf{UTree}} \times \mathsf{UTree} \rightharpoonup \mathsf{UTree}$ and composition $\circ : \mathsf{C}_{\mathsf{UTree}} \times \mathsf{C}_{\mathsf{UTree}} \rightharpoonup \mathsf{C}_{\mathsf{UTree}}$ are defined in terms of substitution as follows:

$$\begin{aligned}c \bullet t &\stackrel{\text{def}}{=} c[t/\_] \text{ provided that } c[t/\_] \in \mathsf{UTree} \\ c \circ c' &\stackrel{\text{def}}{=} c[c'/\_] \text{ provided that } c[c'/\_] \in \mathsf{C}_{\mathsf{UTree}}.\end{aligned}$$

The *(uniquely-labelled) tree context algebra* is $\mathcal{A}_{\mathbb{T}} \stackrel{\text{def}}{=} (\mathsf{UTree}, \mathsf{C}_{\mathsf{UTree}}, \circ, \bullet, \{\_\})$.

Definition 6.7 (Tree Axiomatisation)**.** The *tree axiomatisation*, $\mathrm{Ax}\,[\![(\cdot)]\!]_{\mathbb{T}} : \mathsf{Cmd}_{\mathbb{T}} \to \mathcal{P}(\mathcal{P}(\mathsf{Scope} \times \mathsf{UTree}) \times \mathcal{P}(\mathsf{Scope} \times \mathsf{UTree}))$ is defined as follows:

$$\mathrm{Ax}\,[\![\boldsymbol{x} := \mathtt{getUp}(E)]\!]_{\mathbb{T}} \stackrel{\text{def}}{=}$$
$$\left\{ \left(\begin{array}{c} (\boldsymbol{x} \Rrightarrow v * \rho \times \mathbf{a}[t_1 \otimes \mathbf{b}[t_2] \otimes t_3]), \\ (\boldsymbol{x} \Rrightarrow \mathbf{a} * \rho \times \mathbf{a}[t_1 \otimes \mathbf{b}[t_2] \otimes t_3]) \end{array} \right) \;\middle|\; \mathcal{E}\,[\![E]\!]\,(\boldsymbol{x} \Rrightarrow v * \rho) = \mathbf{b} \right\}$$

$$\mathrm{Ax}\,[\![\boldsymbol{x} := \mathtt{getLeft}(E)]\!]_{\mathbb{T}} \stackrel{\text{def}}{=}$$
$$\left\{ \left(\begin{array}{c} (\boldsymbol{x} \Rrightarrow v * \rho \times \mathbf{a}[t_1] \otimes \mathbf{b}[t_2]), \\ (\boldsymbol{x} \Rrightarrow \mathbf{a} * \rho \times \mathbf{a}[t_1] \otimes \mathbf{b}[t_2]) \end{array} \right) \;\middle|\; \mathcal{E}\,[\![E]\!]\,(\boldsymbol{x} \Rrightarrow v * \rho) = \mathbf{b} \right\} \cup$$
$$\left\{ \left(\begin{array}{c} (\boldsymbol{x} \Rrightarrow v * \rho \times \mathbf{a}[\mathbf{b}[t_1] \otimes t_2]), \\ (\boldsymbol{x} \Rrightarrow \boldsymbol{nil} * \rho \times \mathbf{a}[\mathbf{b}[t_1] \otimes t_2]) \end{array} \right) \;\middle|\; \mathcal{E}\,[\![E]\!]\,(\boldsymbol{x} \Rrightarrow v * \rho) = \mathbf{b} \right\}$$

$$\mathrm{Ax}\,[\![\boldsymbol{x} := \texttt{getRight}(E)]\!]_{\mathbb{T}} \stackrel{\text{def}}{=}$$
$$\left\{ \left(\begin{array}{c} (\boldsymbol{x} \Rightarrow v * \rho \times \mathbf{b}[t_1] \otimes \mathbf{a}[t_2])\,, \\ (\boldsymbol{x} \Rightarrow \mathbf{a} * \rho \times \mathbf{b}[t_1] \otimes \mathbf{a}[t_2]) \end{array} \right) \;\middle|\; \mathcal{E}\,[\![E]\!]\,(\boldsymbol{x} \Rightarrow v * \rho) = \mathbf{b} \right\} \cup$$
$$\left\{ \left(\begin{array}{c} (\boldsymbol{x} \Rightarrow v * \rho \times \mathbf{a}[t_1 \otimes \mathbf{b}[t_2]])\,, \\ (\boldsymbol{x} \Rightarrow \boldsymbol{nil} * \rho \times \mathbf{a}[t_1 \otimes \mathbf{b}[t_2]]) \end{array} \right) \;\middle|\; \mathcal{E}\,[\![E]\!]\,(\boldsymbol{x} \Rightarrow v * \rho) = \mathbf{b} \right\}$$

$$\mathrm{Ax}\,[\![\boldsymbol{x} := \texttt{getFirst}(E)]\!]_{\mathbb{T}} \stackrel{\text{def}}{=}$$
$$\left\{ \left(\begin{array}{c} (\boldsymbol{x} \Rightarrow v * \rho \times \mathbf{a}[\mathbf{b}[t_1] \otimes t_2])\,, \\ (\boldsymbol{x} \Rightarrow \mathbf{b} * \rho \times \mathbf{a}[\mathbf{b}[t_1] \otimes t_2]) \end{array} \right) \;\middle|\; \mathcal{E}\,[\![E]\!]\,(\boldsymbol{x} \Rightarrow v * \rho) = \mathbf{a} \right\} \cup$$
$$\left\{ \left(\begin{array}{c} (\boldsymbol{x} \Rightarrow v * \rho \times \mathbf{a}[\mathbf{0}])\,, \\ (\boldsymbol{x} \Rightarrow \boldsymbol{nil} * \rho \times \mathbf{a}[\mathbf{0}]) \end{array} \right) \;\middle|\; \mathcal{E}\,[\![E]\!]\,(\boldsymbol{x} \Rightarrow v * \rho) = \mathbf{a} \right\}$$

$$\mathrm{Ax}\,[\![\boldsymbol{x} := \texttt{getLast}(E)]\!]_{\mathbb{T}} \stackrel{\text{def}}{=}$$
$$\left\{ \left(\begin{array}{c} (\boldsymbol{x} \Rightarrow v * \rho \times \mathbf{a}[t_1 \otimes \mathbf{b}[t_2]])\,, \\ (\boldsymbol{x} \Rightarrow \mathbf{b} * \rho \times \mathbf{a}[t_1 \otimes \mathbf{b}[t_2]]) \end{array} \right) \;\middle|\; \mathcal{E}\,[\![E]\!]\,(\boldsymbol{x} \Rightarrow v * \rho) = \mathbf{a} \right\} \cup$$
$$\left\{ \left(\begin{array}{c} (\boldsymbol{x} \Rightarrow v * \rho \times \mathbf{a}[\mathbf{0}])\,, \\ (\boldsymbol{x} \Rightarrow \boldsymbol{nil} * \rho \times \mathbf{a}[\mathbf{0}]) \end{array} \right) \;\middle|\; \mathcal{E}\,[\![E]\!]\,(\boldsymbol{x} \Rightarrow v * \rho) = \mathbf{a} \right\}$$

$$\mathrm{Ax}\,[\![\texttt{newNodeAfter}(E)]\!]_{\mathbb{T}} \stackrel{\text{def}}{=}$$
$$\left\{ \left((\rho \times \mathbf{a}[t])\,, (\exists \mathbf{b}.\, \rho \times \mathbf{a}[t] \times \mathbf{b}[\mathbf{0}]) \right) \;\middle|\; \mathcal{E}\,[\![E]\!]\,(\boldsymbol{x} \Rightarrow v * \rho) = \mathbf{a} \right\}$$

$$\mathrm{Ax}\,[\![\texttt{deleteTree}(E)]\!]_{\mathbb{T}} \stackrel{\text{def}}{=} \left\{ \left((\rho \times \mathbf{a}[t])\,, (\rho \times \mathbf{0}) \right) \;\middle|\; \mathcal{E}\,[\![E]\!]\,(\boldsymbol{x} \Rightarrow v * \rho) = \mathbf{a} \right\}$$

6.1.3 List Module

This list module $\mathbb{L} \stackrel{\text{def}}{=} (\mathsf{Cmd}_{\mathbb{L}}, \mathcal{A}_{\mathbb{L}}, \mathrm{Ax}\,[\![(\cdot)]\!]_{\mathbb{L}})$ is an example of a somewhat more exotic abstract module. The module provides an addressable set of lists of unique elements, called 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.

Definition 6.8 (List Update Commands)**.** The set of *list commands* $\mathsf{Cmd}_{\mathbb{L}}$,

ranged over by φ, is defined as follows:

$$\begin{aligned}\varphi ::= {} & \boldsymbol{x} := E.\texttt{getHead}() \mid \boldsymbol{x} := E.\texttt{getTail}() \\ & \mid \boldsymbol{x} := E_1.\texttt{getNext}(E_2) \mid \boldsymbol{x} := E_1.\texttt{getPrev}(E_2) \\ & \mid \boldsymbol{x} := E.\texttt{pop}() \mid E_1.\texttt{push}(E_2) \\ & \mid E_1.\texttt{remove}(E_2) \mid E_1.\texttt{insert}(E_2, E_3) \\ & \mid \boldsymbol{x} := \texttt{newList}() \mid \texttt{deleteList}(E)\end{aligned}$$

where $\boldsymbol{x} \in \mathsf{Var}$ ranges over variables and $E, E_1, \ldots \in \mathsf{Expr}$ range over value expressions.

List stores are a more complicated example of a context algebra than we have previously seen. They 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.

Each list in the store is a finite sequence of values, each of which occurs only once in the list. As with sequences, lists may be extended by applying contexts, as, for instance, $v_1 \cdot \_ \cdot v_2 \bullet w_1 \cdot w_2 = v_1 \cdot w_1 \cdot w_2 \cdot v_2$. However, lists may also be *completed*, which means they cannot be extended by applying a context. Completed lists are indicated by square brackets, and the result of applying a context is undefined. For example, $v_1 \cdot \_ \cdot v_2 \bullet [w_1 \cdot w_2]$ is undefined.

It is necessary to deal with complete lists in order to specify a number of the update and lookup commands on lists. For example, `getFirst` returns the first item in a list; given the partial list $v_1 \cdot v_2$, it is not clear that v_1 is the first element of the list — indeed, in the context $w \cdot \_$ it is certainly not the first element — however, given the completed list $[v_1 \cdot v_2]$ it is completely certain that v_1 is the first element.

Remark. It is quite possible to define list stores to allow the list to be open at one end but not the other. For example, $[v_1$ would allow additional list elements to be added by context application to the right of v_1, but not to the left. I have chosen not to take this approach in order to avoid further complicating an already-complex definition.

Definition 6.9 (List Stores and Contexts)**.** The set of *lists* Lst, ranged over by $l, l_1, \ldots$, the set of *list contexts* $\mathsf{C}_{\mathsf{Lst}}$, ranged over by $lc, lc_1, \ldots$, the set of *list stores* LStore, ranged over by $ls, ls_1, \ldots$, and the set of *list store contexts* $\mathsf{C}_{\mathsf{LStore}}$,

ranged over by $lsc, lsc_1, \ldots$ are defined inductively as follows:

$$l ::= \varnothing \mid v \mid l_1 \cdot l_2 \qquad ls ::= \mathrm{emp} \mid a \Mapsto l \mid a \Mapsto [l] \mid ls_1 * ls_2$$
$$lc ::= \_ \mid lc \cdot l \mid l \cdot lc \qquad lsc ::= ls \mid a \Mapsto lc \mid a \Mapsto [lc] \mid lsc_1 * lsc_2.$$

where values $v \in \mathsf{Val}$ are taken to occur uniquely in each list or list context, addresses $a \in \mathsf{Addr}$ are taken to occur uniquely in each list store or list store context, $\cdot$ is taken to be associative with identity $\varnothing$, and $*$ is taken to be associative and commutative with identity emp.

Context application is defined to allow list stores to be split in the same fashion as heaps, for instance

$$a_1 \Mapsto v_1 \cdot v_2 \cdot v_3 * a_2 \Mapsto w_1 \cdot v_1 = a_1 \Mapsto v_1 \cdot v_2 \cdot v_3 \bullet a_2 \Mapsto w_1 \cdot v_1,$$

but also to allow list stores to be split within the individual lists themselves, as in

$$\begin{aligned} & a_1 \Mapsto v_1 \cdot v_2 \cdot v_3 * a_2 \Mapsto w_1 \cdot v_1 \\ &= a_1 \Mapsto v_1 \cdot \_ \cdot v_3 \bullet (a_1 \Mapsto v_2 * a_2 \Mapsto w_1 \cdot v_1) \\ &= (a_1 \Mapsto v_1 \cdot \_ * a_2 \Mapsto \_ \cdot v_1) \bullet (a_1 \Mapsto v_2 \cdot v_3 * a_2 \Mapsto w_1). \end{aligned}$$

Completed lists cannot be extended by application, but they can be split, as in

$$a_2 \Mapsto [w_1 \cdot w_2] = a_2 \Mapsto [w_1 \cdot \_] \bullet a_2 \Mapsto w.$$

Definition 6.10 (Application and Composition)**.** The application of list store contexts to list stores $\bullet : \mathsf{C}_{\mathsf{LStore}} \times \mathsf{LStore} \rightharpoonup \mathsf{LStore}$ is defined inductively by:

$$\begin{aligned} \mathrm{emp} \bullet ls &\stackrel{\mathrm{def}}{=} ls \\ (lsc * a \Mapsto l) \bullet ls &\stackrel{\mathrm{def}}{=} (lsc \bullet ls) * a \Mapsto l \\ (lsc * a \Mapsto [l]) \bullet ls &\stackrel{\mathrm{def}}{=} (lsc \bullet ls) * a \Mapsto [l] \\ (lsc * a \Mapsto lc) \bullet (ls * a \Mapsto l) &\stackrel{\mathrm{def}}{=} (lsc \bullet ls) * a \Mapsto lc[l/\_] \\ (lsc * a \Mapsto [lc]) \bullet (ls * a \Mapsto l) &\stackrel{\mathrm{def}}{=} (lsc \bullet ls) * a \Mapsto [lc[l/\_]] \end{aligned}$$

where $lc[l/\_]$ denotes the substitution of l for the context hole in lc. The result of the application is undefined when either the right-hand side is badly formed or no case applies.

The composition of list store contexts $\circ : \mathsf{C}_{\mathsf{LStore}} \times \mathsf{C}_{\mathsf{LStore}} \rightharpoonup \mathsf{C}_{\mathsf{LStore}}$ is defined

inductively by:

$$\begin{aligned}
\mathrm{emp} \circ lsc' &= lsc' \\
(lsc * a \Mapsto l) \circ lsc' &= (lsc \circ lsc') * a \Mapsto l \\
(lsc * a \Mapsto [l]) \circ lsc' &= (lsc \circ lsc') * a \Mapsto [l] \\
(lsc * a \Mapsto lc) \circ (lsc' * a \Mapsto l) &= (lsc \circ lsc') * a \Mapsto lc[l/_] \\
(lsc * a \Mapsto lc) \circ (lsc' * a \Mapsto lc') &= (lsc \circ lsc') * a \Mapsto lc[lc'/_] \\
(lsc * a \Mapsto [lc]) \circ (lsc' * a \Mapsto l) &= (lsc \circ lsc') * a \Mapsto [lc[l/_]] \\
(lsc * a \Mapsto [lc]) \circ (lsc' * a \Mapsto lc') &= (lsc \circ lsc') * a \Mapsto \left[lc[lc'/_]\right] .
\end{aligned}$$

Definition 6.11 (List Context Algebra)**.** The *list-store context algebra* $\mathcal{A}_{\mathbb{L}} \stackrel{\text{def}}{=} (\mathsf{LStore}, \mathsf{C}_{\mathsf{LStore}}, \circ, \bullet, \{\mathrm{emp}\})$ is given by the above definitions.

Definition 6.12 (List Axiomatisation)**.** The *list axiomatisation*, $\mathrm{Ax}\,[\![(\cdot)]\!]_{\mathbb{L}} : \mathsf{Cmd}_{\mathbb{L}} \to \mathcal{P}(\mathcal{P}(\mathsf{Scope} \times \mathsf{LStore}) \times \mathcal{P}(\mathsf{Scope} \times \mathsf{LStore}))$ is defined as follows:

$$\begin{aligned}
&\mathrm{Ax}\,[\![\boldsymbol{x} := E.\mathtt{getHead}()]\!]_{\mathbb{L}} \stackrel{\text{def}}{=} \\
&\qquad \left\{ \left(\begin{array}{l} (\boldsymbol{x} \Rrightarrow v * \rho \times a \Mapsto [w \cdot l]) , \\ (\boldsymbol{x} \Rrightarrow w * \rho \times a \Mapsto [w \cdot l]) \end{array} \right) \;\middle|\; a = \mathcal{E}\,[\![E]\!]\,(\boldsymbol{x} \Rrightarrow v * \rho) \right\} \cup \\
&\qquad\qquad \left\{ \left(\begin{array}{l} (\boldsymbol{x} \Rrightarrow v * \rho \times a \Mapsto [\varnothing]) , \\ (\boldsymbol{x} \Rrightarrow \boldsymbol{nil} * \rho \times a \Mapsto [\varnothing]) \end{array} \right) \;\middle|\; a = \mathcal{E}\,[\![E]\!]\,(\boldsymbol{x} \Rrightarrow v * \rho) \right\}
\end{aligned}$$

$$\begin{aligned}
&\mathrm{Ax}\,[\![\boldsymbol{x} := E.\mathtt{getTail}()]\!]_{\mathbb{L}} \stackrel{\text{def}}{=} \\
&\qquad \left\{ \left(\begin{array}{l} (\boldsymbol{x} \Rrightarrow v * \rho \times a \Mapsto [l \cdot w]) , \\ (\boldsymbol{x} \Rrightarrow w * \rho \times a \Mapsto [l \cdot w]) \end{array} \right) \;\middle|\; a = \mathcal{E}\,[\![E]\!]\,(\boldsymbol{x} \Rrightarrow v * \rho) \right\} \cup \\
&\qquad\qquad \left\{ \left(\begin{array}{l} (\boldsymbol{x} \Rrightarrow v * \rho \times a \Mapsto [\varnothing]) , \\ (\boldsymbol{x} \Rrightarrow \boldsymbol{nil} * \rho \times a \Mapsto [\varnothing]) \end{array} \right) \;\middle|\; a = \mathcal{E}\,[\![E]\!]\,(\boldsymbol{x} \Rrightarrow v * \rho) \right\}
\end{aligned}$$

$$\begin{aligned}
&\mathrm{Ax}\,[\![\boldsymbol{x} := E_1.\mathtt{getNext}(E_2)]\!]_{\mathbb{L}} \stackrel{\text{def}}{=} \\
&\quad \left\{ \left(\begin{array}{l} (\boldsymbol{x} \Rrightarrow v * \rho \times a \Mapsto w \cdot u) , \\ (\boldsymbol{x} \Rrightarrow u * \rho \times a \Mapsto w \cdot u) \end{array} \right) \;\middle|\; \begin{array}{r} a = \mathcal{E}\,[\![E_1]\!]\,(\boldsymbol{x} \Rrightarrow v * \rho) \text{ and} \\ w = \mathcal{E}\,[\![E_2]\!]\,(\boldsymbol{x} \Rrightarrow v * \rho) \end{array} \right\} \cup \\
&\quad\quad \left\{ \left(\begin{array}{l} (\boldsymbol{x} \Rrightarrow v * \rho \times a \Mapsto [l \cdot w]) , \\ (\boldsymbol{x} \Rrightarrow \boldsymbol{nil} * \rho \times a \Mapsto [l \cdot w]) \end{array} \right) \;\middle|\; \begin{array}{r} a = \mathcal{E}\,[\![E_1]\!]\,(\boldsymbol{x} \Rrightarrow v * \rho) \text{ and} \\ w = \mathcal{E}\,[\![E_2]\!]\,(\boldsymbol{x} \Rrightarrow v * \rho) \end{array} \right\}
\end{aligned}$$

$$\mathrm{Ax}\,[\![\boldsymbol{x} := E_1.\mathtt{getPrev}(E_2)]\!]_{\mathbb{L}} \stackrel{\text{def}}{=}$$
$$\left\{ \left(\begin{array}{l} (\boldsymbol{x} \Rightarrow v * \rho \times a \Mapsto u \cdot w)\,, \\ (\boldsymbol{x} \Rightarrow u * \rho \times a \Mapsto u \cdot w) \end{array} \right) \;\middle|\; \begin{array}{r} a = \mathcal{E}\,[\![E_1]\!]\,(\boldsymbol{x} \Rightarrow v * \rho) \text{ and} \\ w = \mathcal{E}\,[\![E_2]\!]\,(\boldsymbol{x} \Rightarrow v * \rho) \end{array} \right\} \cup$$
$$\left\{ \left(\begin{array}{l} (\boldsymbol{x} \Rightarrow v * \rho \times a \Mapsto [w \cdot l])\,, \\ (\boldsymbol{x} \Rightarrow \boldsymbol{nil} * \rho \times a \Mapsto [w \cdot l]) \end{array} \right) \;\middle|\; \begin{array}{r} a = \mathcal{E}\,[\![E_1]\!]\,(\boldsymbol{x} \Rightarrow v * \rho) \text{ and} \\ w = \mathcal{E}\,[\![E_2]\!]\,(\boldsymbol{x} \Rightarrow v * \rho) \end{array} \right\}$$

$$\mathrm{Ax}\,[\![\boldsymbol{x} := E.\mathtt{pop}()]\!]_{\mathbb{L}} \stackrel{\text{def}}{=}$$
$$\left\{ \left(\begin{array}{l} (\boldsymbol{x} \Rightarrow v * \rho \times a \Mapsto [w \cdot l])\,, \\ (\boldsymbol{x} \Rightarrow w * \rho \times a \Mapsto [l]) \end{array} \right) \;\middle|\; a = \mathcal{E}\,[\![E]\!]\,(\boldsymbol{x} \Rightarrow v * \rho) \right\}$$

$$\mathrm{Ax}\,[\![E_1.\mathtt{push}(E_2)]\!]_{\mathbb{L}} \stackrel{\text{def}}{=}$$
$$\left\{ \big((\rho \times a \Mapsto [l] \wedge v \notin l)\,, (\rho \times a \Mapsto [v \cdot l]) \big) \;\middle|\; \begin{array}{r} a = \mathcal{E}\,[\![E_1]\!]\,\rho \text{ and} \\ v = \mathcal{E}\,[\![E_2]\!]\,\rho \end{array} \right\}$$

$$\mathrm{Ax}\,[\![E_1.\mathtt{remove}(E_2)]\!]_{\mathbb{L}} \stackrel{\text{def}}{=}$$
$$\left\{ \big((\rho \times a \Mapsto v)\,, (\rho \times a \Mapsto \varnothing) \big) \;\middle|\; a = \mathcal{E}\,[\![E_1]\!]\,\rho \text{ and } v = \mathcal{E}\,[\![E_2]\!]\,\rho \right\}$$

$$\mathrm{Ax}\,[\![E_1.\mathtt{insert}(E_2, E_3)]\!]_{\mathbb{L}} \stackrel{\text{def}}{=}$$
$$\left\{ \left(\begin{array}{r} (\rho \times a \Mapsto [l_1 \cdot v \cdot l_2] \wedge v \notin l_1 \cdot v \cdot l_2)\,, \\ (\rho \times a \Mapsto [l_1 \cdot v \cdot w \cdot l_2]) \end{array} \right) \;\middle|\; \begin{array}{r} a = \mathcal{E}\,[\![E_1]\!]\,\rho \text{ and} \\ v = \mathcal{E}\,[\![E_2]\!]\,\rho \text{ and} \\ w = \mathcal{E}\,[\![E_3]\!]\,\rho \end{array} \right\}$$

$$\mathrm{Ax}\,[\![\boldsymbol{x} := \mathtt{newList}()]\!]_{\mathbb{L}} \stackrel{\text{def}}{=} \big\{ \big((\boldsymbol{x} \Rightarrow - \times \mathrm{emp})\,, (\exists a.\, \boldsymbol{x} \Rightarrow a \times a \Mapsto [\varnothing]) \big) \big\}$$

$$\mathrm{Ax}\,[\![\mathtt{deleteList}(E)]\!]_{\mathbb{L}} \stackrel{\text{def}}{=} \big\{ \big((\rho \times a \Mapsto [l])\,, (\rho \times \mathrm{emp}) \big) \;\big|\; a = \mathcal{E}\,[\![E]\!]\,\rho \big\}$$

6.1.4 Combining Abstract Modules

It is useful to have a mechanism for combining modules. The most intuitive approach to combining two abstract modules is to allow arbitrary interleavings of their commands, whilst interpreting the commands over the product of their data-store context algebras. Information is thus shared between modules through the common variable store.

Definition 6.13 (Abstract Module Combination)**.** Given abstract modules $\mathbb{A}_1 = (\mathsf{Cmd}_{\mathbb{A}_1}, \mathcal{A}_{\mathbb{A}_1}, \mathrm{Ax}\,[\![(\cdot)]\!]_{\mathbb{A}_1})$ and $\mathbb{A}_2 = (\mathsf{Cmd}_{\mathbb{A}_2}, \mathcal{A}_{\mathbb{A}_2}, \mathrm{Ax}\,[\![(\cdot)]\!]_{\mathbb{A}_2})$, their combination

$$\mathbb{A}_1 + \mathbb{A}_2 \stackrel{\text{def}}{=} (\mathsf{Cmd}_{\mathbb{A}_1} \oplus \mathsf{Cmd}_{\mathbb{A}_2}, \mathcal{A}_{\mathbb{A}_1} \times \mathcal{A}_{\mathbb{A}_2}, \mathrm{Ax}\,[\![(\cdot)]\!]_{\mathbb{A}_1 + \mathbb{A}_2})$$

is an abstract module, where

- $\mathsf{Cmd}_{\mathbb{A}_1} \oplus \mathsf{Cmd}_{\mathbb{A}_2} \stackrel{\text{def}}{=} (\mathsf{Cmd}_{\mathbb{A}_1} \times \{1\}) \cup (\mathsf{Cmd}_{\mathbb{A}_2} \times \{2\})$ is the discriminated union of the command sets,
- $\mathcal{A}_{\mathbb{A}_1} \times \mathcal{A}_{\mathbb{A}_2}$ is the direct product of the context algebras, and
- $\mathrm{Ax}\,[\![(\cdot)]\!]_{\mathbb{A}_1+\mathbb{A}_2} : \mathsf{Cmd}_{\mathbb{A}_1} \oplus \mathsf{Cmd}_{\mathbb{A}_2} \to \mathcal{P}(\mathcal{P}(\mathsf{Scope} \times \mathsf{D}_{\mathbb{A}_1} \times \mathsf{D}_{\mathbb{A}_2}) \times \mathcal{P}(\mathsf{Scope} \times \mathsf{D}_{\mathbb{A}_1} \times \mathsf{D}_{\mathbb{A}_2}))$ is defined as

  $$\mathrm{Ax}\,[\![(\varphi,1)]\!]_{\mathbb{A}_1+\mathbb{A}_2} \stackrel{\text{def}}{=} \left\{(\pi_1(P,\chi_2), \pi_1(Q,\chi_2)) \;\middle|\; (P,Q) \in \mathrm{Ax}\,[\![\varphi]\!]_{\mathbb{A}_1} \text{ and } \chi_2 \in \mathsf{D}_{\mathbb{A}_2}\right\}$$
  $$\mathrm{Ax}\,[\![(\varphi,2)]\!]_{\mathbb{A}_1+\mathbb{A}_2} \stackrel{\text{def}}{=} \left\{(\pi_2(P,\chi_1), \pi_2(Q,\chi_1)) \;\middle|\; (P,Q) \in \mathrm{Ax}\,[\![\varphi]\!]_{\mathbb{A}_2} \text{ and } \chi_1 \in \mathsf{D}_{\mathbb{A}_1}\right\}$$

  where

  $$\pi_1(P,\chi_2) \stackrel{\text{def}}{=} \{(\rho,\chi_1,\chi_2) \mid (\rho,\chi_1) \in P\}$$
  $$\pi_2(P,\chi_1) \stackrel{\text{def}}{=} \{(\rho,\chi_1,\chi_2) \mid (\rho,\chi_2) \in P\}\,.$$

Notation. When the command sets $\mathsf{Cmd}_{\mathbb{A}_1}$ and $\mathsf{Cmd}_{\mathbb{A}_2}$ are disjoint, I will drop the tags when referring to the commands in the combined abstract module. When the tags are necessary, I will indicate them with an appropriately place subscript.

Remark. When the abstract modules have context algebras with zeros, the axiom sets can be simplified by taking the unchanged component from the zero of the relevant context algebra. All other cases can be achieved by frame. This direction was taken in [DYGW10a].

6.2 Module Translations

I now define what it means to correctly implement one module in terms of another, using *module translations*.

Definition 6.14 (Module Translation)**.** A module translation $\tau : \mathbb{A} \to \mathbb{B}$ from abstract module $\mathbb{A}$ to abstract module $\mathbb{B}$ comprises:

- an *abstraction relation* $\alpha_\tau \subseteq \mathsf{D}_{\mathbb{B}} \times \mathsf{D}_{\mathbb{A}}$, and
- a *substitutive implementation function* $[\![(\cdot)]\!]_\tau : \mathcal{L}_{\mathbb{A}} \to \mathcal{L}_{\mathbb{B}}$ which uniformly substitutes each basic command of $\mathsf{Cmd}_{\mathbb{A}}$ with a call to a procedure written in $\mathcal{L}_{\mathbb{B}}$.

Notation. The abstraction relation α_τ is lifted to a *predicate translation* $[\![(\cdot)]\!]_\tau : \mathcal{P}(\mathsf{State}_\mathbb{A}) \to \mathcal{P}(\mathsf{State}_\mathbb{B})$ as follows:

$$[\![P]\!] = \{(\rho, \chi_\mathbb{B}) \mid \text{there exists } \chi_\mathbb{A} \text{ s.t. } (\rho, \chi_\mathbb{A}) \in P \text{ and } \chi_\mathbb{B}\ \alpha_\tau\ \chi_\mathbb{A}\}\,.$$

When the translation τ is implicit from context, the subscripts on the abstraction relation, implementation function and predicate translation may be dropped.

In the context of a 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. (Of course, there is no reason why a module should not be abstract with respect to one translation and concrete with respect to another, or even both the abstract and concrete module with respect to a single translation.)

Definition 6.15 (Sound Module Translation)**.** A module translation $\tau : \mathbb{A} \to \mathbb{B}$ is said to be sound, if for all $P, Q \in \mathcal{P}(\mathsf{State}_\mathbb{A})$ and $\mathbb{C} \in \mathcal{L}_\mathbb{A}$,

$$\vdash_\mathbb{A} \{P\}\ \mathbb{C}\ \{Q\} \quad \Longrightarrow \quad \vdash_\mathbb{A} \{[\![P]\!]_\tau\}\ [\![\mathbb{C}]\!]_\tau\ \{[\![Q]\!]_\tau\}\,.$$

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 I have elected to work with partial correctness Hoare triples, 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. I 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 [IO01, Rey02, CGZ05].

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`; such an implementation is 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 does not hold vacuously.

A more stringent approach would be to require the abstraction relation to be surjective, and therefore every satisfiable predicate to be satisfiable under translation. This condition is, however, not met by one of the implementations considered here.

In this chapter, I present two techniques for constructing sound module translations. *Locality-preserving translations*, discussed in §6.3, closely preserve the structure of the abstract module's context algebra through the translation, which leads to an elegant inductive proof transformation from the abstract to the concrete. In particular, application at the abstract level corresponds to application at the concrete level, and so the abstract frame rule is transformed to the concrete frame rule.

Locality-breaking translations, discussed in §6.4, on the other hand do not necessarily preserve the structure of the abstract module's context algebra. Even so, certain properties of the proof theory can be used to simplify the problem of establishing the soundness of such a translation.

6.2.1 Modularity

An important property of module translations is that they are composable: given translations $\tau_1 : \mathbb{A}_1 \to \mathbb{A}_2$ and $\tau_2 : \mathbb{A}_2 \to \mathbb{A}_3$, the translation $\tau_2 \circ \tau_1 : \mathbb{A}_1 \to \mathbb{A}_3$ can be defined in the natural fashion. If its constituent translations are sound then so is the composition. Therefore, it is possible to construct module translations stepwise.

A translation $\tau : \mathbb{A}_1 \to \mathbb{A}_2$ can be naturally lifted to a 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 translation, we might also expect $\tau + \mathbb{B}$ to be sound, however, it is not obvious that this is the case in general. The techniques for constructing sound translations in this chapter do, however, admit such a lifting, since they transform high-level proofs to low-level proofs in a fashion that can preserve any additional module component. Thus, these techniques are modular, since translations for independent modules may be effectively combined.

6.3 Locality-Preserving Translations

Sometimes, there is a close correspondence between the locality exhibited by a high-level module and the locality of the low-level module on top of which it is implemented. In this section, I expand on this intuition and formalise the concept of a *locality-preserving translation*. In §6.3.1, I establish that locality-preserving translations give sound module translations, and in §6.3.2 and §6.3.3 I give locality-preserving translations $\tau_1 : \mathbb{T} \to \mathbb{H} + \mathbb{L}$ and $\tau_2 : \mathbb{H} + \mathbb{H} \to \mathbb{H}$.

So what exactly does it mean for there to be a correspondence between locality at the high level and locality at the low level? Consider Figure 6.2, which depicts a typical tree from the tree module $\mathbb{T}$ (a), together with possible

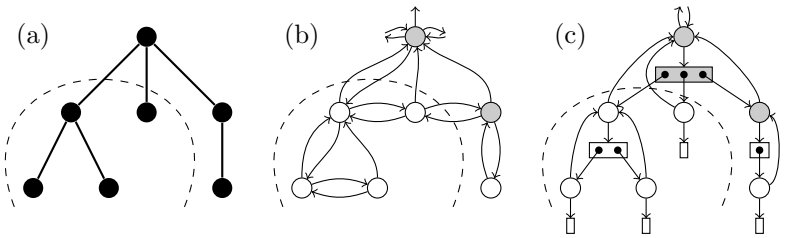


Figure 6.2: An abstract tree from $\mathbb{T}$ (a), and representations of the tree in $\mathbb{H}$ (b), and in $\mathbb{H} + \mathbb{L}$ (c).

representations of that tree in the heap module $\mathbb{H}$ (b), and in the combined heap-and-list module $\mathbb{H} + \mathbb{L}$ (c).

In (b), each tree node is represented by a memory block comprising four pointer fields (depicted by a circle with outgoing arrows) which record the addresses of the memory blocks representing the left sibling, parent, right sibling and first child. Where there is no such node (for example, when a node has no children) the pointer field holds the ***nil*** value (depicted by the absence of an arrow).

In (c), each tree node is represented by a list of pointers to the node's children (depicted by a box with dots for each value in the list) and a memory block that comprises a pointer to the node's parent and a pointer to the list of children.

Already, we can get some impression of why these representations can be used to preserve locality between the levels of abstraction, since any portion of the abstract tree can be identified with a portion of the tree's low-level representation that represents it. In and of itself, this is not enough for locality to be preserved, since locality pertains to the footprints of the module operations — that is, the resource involved in performing the operation. Thus, it is necessary to consider whether the implementations of commands only operate on the representation of the footprint of the abstract command they implement.

As an example, consider the implementation of $n := \texttt{getUp}(m)$ using a heap representation like the one described above. The most intuitive implementation would be simply to look up the node's parent pointer and return the result: $n := [m.\texttt{parent}]$.[3] The abstract footprint of `getUp` includes the entire tree at the parent node; the concrete footprint simply consists of the heap cell at address $m.\texttt{parent}$. Since the concrete footprint is contained within the representation

[3]The syntax $m.\texttt{parent}$ is a notational convenience. Assuming that the parent pointer is held in the second cell of memory block representing the node, this simply corresponds to $m + 1$.

of the abstract footprint, it is reasonable to say that locality is preserved by the implementation.

For contrast, consider an implementation of $n := \texttt{getUp}(m)$ that, rather than simply looking up the parent pointer, traverses the entire tree from the root, searching for a node that has m as its child. The footprint of this implementation is clearly not in general contained within the representation of the abstract footprint, and locality is not preserved by such an implementation.

To formalise the intuition behind a locality-preserving implementation, it is necessary to relate abstract data structures and contexts to their concrete representations. The representation of abstract data structure χ is given by the concrete predicate $\langle\!\langle \chi \rangle\!\rangle$, and the representation of abstract context c is given by the concrete predicate $\langle\!\langle c \rangle\!\rangle$. Applying a frame at the abstract level should correspond to applying the representation of that frame at the concrete level. To ensure that this is possible, context application should be preserved by representation, *i.e.* $\langle\!\langle c \bullet \chi \rangle\!\rangle \equiv \langle\!\langle c \rangle\!\rangle \bullet \langle\!\langle \chi \rangle\!\rangle$. This principle consists in the *application preservation property*.

Consider once more Figure 6.2. The dashed line in (a) depicts a splitting of the abstract tree into a context and a subtree; the dashed lines in (b) and (c) depict corresponding splittings of the representation of the tree. A key issue arises in establishing that application is preserved by these representations: while an abstract tree is agnostic to the context in which it resides, and an abstract context is agnostic to the tree that fills its context hole (and, indeed, the outer context in which it resides), the same is not true of their representations. In particular, in (b) the indicated subtree "knows" that there is no subtree directly to its left, the address of its parent node and the address of the first node to its right; similarly, the indicated subtree "knows" the address of the subtree's left-most and right-most nodes at its root level. If the subtree representation were to be put in the representation of a different context without updating this "knowledge" then the pointers would not join up correctly and application preservation would be violated.

This "knowledge" that context and subdata representations have about each other is called the *interface*. The representations must be parametrised by interfaces to ensure that context and data representations correctly mesh. Thus, application preservation is expressed as $\langle\!\langle c \bullet \chi \rangle\!\rangle^I \equiv \exists I'.\, \langle\!\langle c \rangle\!\rangle^I_{I'} \bullet \langle\!\langle \chi \rangle\!\rangle^{I'}$. Note that a context representation actually has two interfaces: one between it and the subdata in its hole and one between it and its own surrounding context.

The interface is divided into two parts: the knowledge that a context representation needs about the data to be put in its hole is called the *in-interface*,

while the knowledge that a data representation needs about its surrounding context is called the *out-interface*. In Figure 6.2 (a), the in-interface consists of the addresses of the left- and right-most nodes at the top level of the subtree, while the out-interface consists of the addresses of the nodes immediately left, above and to the right of the context hole. In Figure 6.2 (b), the in-interface consists of the addresses of all of the nodes at the top level of the subtree, while the out-interface consists of the address of the parent node of the context hole. Typically, as in these examples, the interfaces are a collection of pointers, which are identified in the diagrams by the arrows that cross the dividing line between context and subdata — the direction of the arrow determining whether it belongs to the in- or out-interface.

Having established an application-preserving representation, a high-level proof can be transformed to a low-level proof by simply replacing the high-level predicates with their low-level representations, provided that we can prove that the axioms hold under representation for the implementations of the module commands. In many cases, such as the simple `getUp` implementation described above, this is not difficult to achieve. However, in certain cases it transpires that the low-level footprint is a little bit larger than the representation of the high-level footprint.

Consider, for example, the operation of disposing the subtree indicated in Figure 6.2. At the high level, it is clear that the footprint comprises only the subtree that is to be deleted. In both implementations, something more than the representation of this is required: for the heap implementation, the pointers from the context into the deleted subtree must be updated; for the heap-and-list implementation, the pointers to the subtree's top-level nodes must be removed from their parent's child list.

How should we deal with the spanner that such commands throw into the works? One solution is simply to decide that the representation chosen was unsuitable and look for a better one. This approach is justified by the principle that the representation of a command's footprint should include the footprint of the command's implementation. But if we are willing to sacrifice this principle are we able to repair our existing approach?

In fact we can, by introducing the concept of *crust* — a predicate that corresponds to the minimal extra footprint, taken from the surrounding context, required by the command implementations. In Figure 6.2, the crust for the indicated subtree is depicted by the shaded portions of the representation. For the heap implementation, this is the nodes (in the example, the parent and right sibling) that have pointers into the subtree; for the heap-and-list implementa-

tion, this is the parent node, its child list and its other children.[4] In general, the crust is represented by the predicate $⋒_I^F$, parametrised by interface I and some additional parameter F that together fully determine it.

When a high-level proof is to be transformed to a low-level proof, the high-level predicates are replaced with their low-level representations plus the corresponding crust. However, the proof transformation now faces the problem that a high-level frame cannot simply be replaced with its representation plus the outer crust — the frame would duplicate parts of the data that had previously been included in the crust. This *inner crust* therefore has to be removed from the frame before it is applied.

For example, consider once again disposing the subtree indicated in Figure 6.2. In the heap implementation, the command operates on the representation of the subtree (as indicated by the dashed line) plus the crust (as indicated by the shaded nodes). If the context is added by frame, at the low level what is added is the representation of the crust *minus* the inner crust (the shaded nodes).

The *crust inclusion property*, which (in part) states that the representation of a context together with its outer crust contains the inner crust, establishes that the required frame exists. A little more is needed of the crust inclusion property, however, since the operations we are considering actually alter the crust. Yet the change made to the crust cannot materially affect the context — in fact, it only changes its in-interface. The crust inclusion property therefore establishes that not only can that inner crust be removed from the combination of a context representation and its outer crust, but that when it is replaced by another inner crust, differing from the original in only the in-interface, the result is also the combination of a representation of the same context and its outer crust, but with the (inner) in-interface of the context representation updated accordingly. Together with application preservation, this establishes that when a frame is applied it will mesh correctly with both the pre- and postcondition.

Finally, in order to complete the proof transformation, it is sufficient to establish that the implementations of the basic commands satisfy the representations of their specifications, with the addition of crust. Loosely:

$$\vdash \left\{\exists in.\, ⋒_{in,out}^{F} \bullet \langle\!\langle P \rangle\!\rangle^{in,out}\right\} [\![\varphi]\!] \left\{\exists in.\, ⋒_{in,out}^{F} \bullet \langle\!\langle Q \rangle\!\rangle^{in,out}\right\}$$

for every $(P, Q) \in \mathrm{Ax}\,[\![\varphi]\!]$. This is the *axiom correctness property.*

[4]It is probably not clear why the children of the parent node are part of the crust. The reason is that the implementation of `newNodeAfter` inserts a new node in the parent's child list, which must therefore be unique in the list — knowing that the other values in the list are addresses of disjoint portions of the heap from the new node is used to establish this.

Having fleshed out the intuition behind locality-preserving translations, I now introduce their formal definition. I first define the concept of *pre-locality-preserving translations*, which have the appropriate form, and then restrict locality-preserving translations to being those that exhibit the properties of application preservation, crust inclusion and axiom correctness.

Definition 6.16 (Pre-Locality-Preserving Translation)**.** A *pre-locality-preserving translation* $\tau : \mathbb{A} \to \mathbb{B}$ comprises:

- a set of *in-interfaces* $\mathcal{I}_{\text{in}}$ and a set of *out-interfaces* $\mathcal{I}_{\text{out}}$, whose Cartesian product constitutes the set of *interfaces* $\mathcal{I} = \mathcal{I}_{\text{in}} \times \mathcal{I}_{\text{out}}$;
- a *data representation function* $\langle\!\langle (\cdot) \rangle\!\rangle^{(\cdot)} : \mathsf{D}_{\mathbb{A}} \times \mathcal{I} \to \mathcal{P}(\mathsf{D}_{\mathbb{B}})$;
- a *context representation function* $\langle\!\langle (\cdot) \rangle\!\rangle^{(\cdot)}_{(\cdot)} : \mathsf{C}_{\mathbb{A}} \times \mathcal{I} \times \mathcal{I} \to \mathcal{P}(\mathsf{C}_{\mathbb{B}})$;
- a set of *crust parameters* $\mathcal{F}$;
- a crust predicate $\Subset^F_I \in \mathcal{P}(\mathsf{D}_{\mathbb{B}})$, parametrised by interface $I \in \mathcal{I}$ and crust parameter $F \in \mathcal{F}$; and
- a *substitutive implementation function* $[\![(\cdot)]\!]_\tau : \mathcal{L}_{\mathbb{A}} \to \mathcal{L}_{\mathbb{B}}$.

Note that a pre-locality-preserving translation defines a module translation, for any given $out \in \mathcal{I}_{\text{out}}$ and $F \in \mathcal{F}$ by taking the abstraction relation α to be

$$\alpha = \left\{ (\chi_{\mathbb{B}}, \chi_{\mathbb{A}}) \;\middle|\; \chi_{\mathbb{B}} \in \exists in.\, \Subset^F_{in,out} \bullet \langle\!\langle \chi_{\mathbb{A}} \rangle\!\rangle^{in,out} \right\}.$$

Frequently, there will be a natural choice of *out* and F, but in general any choice is permissible.

Definition 6.17 (Intermediate Translation Functions)**.** Given a pre-locality-preserving translation, the *intermediate data-predicate translation function*

$$(\!|(\cdot)|\!)^{(\cdot)} : \mathcal{P}(\mathsf{State}_{\mathbb{A}}) \times (\mathcal{I}_{\text{out}} \times \mathcal{F}) \to \mathcal{P}(\mathsf{State}_{\mathbb{B}})$$

is defined as:

$$(\!|P|\!)^{out,F} = \bigvee_{(\rho,\chi_{\mathbb{A}}) \in P} \{\rho\} \times \left(\exists in.\, \Subset^F_{in,out} \bullet \langle\!\langle \chi_{\mathbb{A}} \rangle\!\rangle^{in,out}\right).$$

The *intermediate context-predicate translation function*

$$(\!|(\cdot)|\!)^{(\cdot)}_{(\cdot)} : \mathcal{P}(\mathsf{C}_{\mathsf{State}_{\mathbb{A}}}) \times (\mathcal{I}_{\text{out}} \times \mathcal{F}) \times (\mathcal{I}_{\text{out}} \times \mathcal{F}) \to \mathcal{P}(\mathsf{C}_{\mathsf{State}_{\mathbb{B}}})$$

is defined as:

$$(\!|K|\!)^{out,F}_{out',F'} = \bigvee_{(\rho,c_{\mathbb{A}}) \in K} \{\rho\} \times \left(\forall in'.\, \Subset^F_{in',out'} \multimap \left(\exists in.\, \Subset^F_{in,out} \circ \langle\!\langle c_{\mathbb{A}} \rangle\!\rangle^{in,out}_{in',out'}\right)\right).$$

For a module translation defined by a pre-locality-preserving translation with respect to $out \in \mathcal{I}_{\text{out}}$ and $F \in \mathcal{F}$, the predicate translation can be expressed simply in terms of the intermediate data-predicate translation: $[\![P]\!] = (\!|P|\!)^{out,F}$.

Pre-locality-preserving translations to not embody the intuition of what it means for a module translation to (soundly) preserve locality. This is reserved for locality-preserving translations, which require the following three properties.

Property 6.18 (Application Preservation)**.** *Context application is preserved by the representation functions. That is, for all $c \in \mathsf{C}_{\mathbb{A}}$, $\chi \in \mathsf{D}_{\mathbb{A}}$ and $I \in \mathcal{I}$,*

$$\langle\!\langle c \bullet_{\mathbb{A}} \chi \rangle\!\rangle^{I} \equiv \exists I'.\, \langle\!\langle c \rangle\!\rangle^{I}_{I'} \bullet_{\mathbb{B}} \langle\!\langle \chi \rangle\!\rangle^{I'}.$$

Property 6.19 (Crust Inclusion)**.** *For all $out, out' \in \mathcal{I}_{\text{out}}$, $F \in \mathcal{F}$ and $c \in \mathsf{C}_{\mathbb{A}}$, there exist $K \in \mathcal{P}(\mathsf{C}_{\mathbb{B}})$ and $F' \in \mathcal{F}$ such that, for all $in \in \mathcal{I}_{\text{in}}$,*

$$\left(\exists in'.\, \Cap{}{}{F}_{in',out'} \circ \langle\!\langle c \rangle\!\rangle^{in',out'}_{in,out}\right) \equiv K \circ \Cap{}{}{F'}_{in,out}.$$

Property 6.20 (Axiom Correctness)**.** *For all $\varphi \in \mathsf{Cmd}_{\mathbb{A}}$, $(P, Q) \in \mathrm{Ax}\,[\![\varphi]\!]_{\mathbb{A}}$, $out \in \mathcal{I}_{\text{out}}$ and $F \in \mathcal{F}$,*

$$\vdash_{\mathbb{B}} \left\{(\!|P|\!)^{out,F}\right\}\ [\![\varphi]\!]\ \left\{(\!|Q|\!)^{out,F}\right\}.$$

Definition 6.21 (Locality Preserving Translation)**.** A *locality-preserving translation* is a pre-locality-preserving translation that satisfies Properties 6.18, 6.19 and 6.20.

Theorem 90 (Soundness of Locality-Preserving Translations)**.** *A locality-preserving translation is a sound module translation (for any fixed choice of $out \in \mathcal{I}_{\text{out}}$ and $F \in \mathcal{F}$).*

6.3.1 Proof of Soundness of Locality-Preserving Translations

In this section, assume that $\tau : \mathbb{A} \to \mathbb{B}$ is a locality preserving translation. The following proposition is sufficient to establish that τ gives rise to a sound module translation.

Proposition 91. *For all $out \in \mathcal{I}_{\text{out}}$ and $F \in \mathcal{F}$,and for all $P, K \in \mathcal{P}(\mathsf{State}_{\mathbb{A}})$ and $\mathbb{C} \in \mathcal{L}_{\mathbb{A}}$,*

$$\Gamma \vdash_{\mathbb{A}} \{P\}\ \mathbb{C}\ \{Q\} \implies [\![\Gamma]\!] \vdash_{\mathbb{B}} \left\{(\!|P|\!)^{out,F}\right\}\ \mathbb{C}\ \left\{(\!|Q|\!)^{out,F}\right\},$$

where

$$[\![\Gamma]\!] = \left\{ \mathtt{f} : (\!|P'|\!)^{out',F'} \rightarrowtail (\!|Q'|\!)^{out',F'} \;\middle|\; \begin{array}{l} (\mathtt{f} : P' \rightarrowtail Q') \in \Gamma \text{ and} \\ out' \in \mathcal{I}_{\text{out}} \text{ and } F' \in \mathcal{F} \end{array} \right\}.$$

Before embarking on the proof of this proposition, two auxiliary lemmata are required. The following lemma gives and alternative characterisation of the crust inclusion property.

Lemma 92 (Crust Inclusion II)**.** *For all $K \in \mathcal{P}(\mathsf{C}_{\mathbb{A}})$, $in \in \mathcal{I}_{\text{in}}$, $out, out' \in \mathcal{I}_{\text{out}}$ and $F \in \mathcal{F}$,*

$$\left(\exists in'.\, \Cap{F}{in',out'} \circ \langle\!\langle K \rangle\!\rangle^{in',out'}_{in,out}\right) \subseteq \exists F'.\, \left(\forall in''.\, \Cap{F'}{in'',out} \multimap \left(\exists in'.\, \Cap{F}{in',out'} \circ \langle\!\langle K \rangle\!\rangle^{in',out'}_{in'',out}\right)\right) \circ \Cap{F'}{in,out}.$$

Note that the converse of this property is trivially true, hence the entailment holds in both directions.

Proof. Fix arbitrary $K \in \mathcal{P}(\mathsf{C}_{\mathbb{A}})$, $in \in \mathcal{I}_{\text{in}}$, $out, out' \in \mathcal{I}_{\text{out}}$ and $F \in \mathcal{F}$. Fix $c' \in \mathsf{C}_{\mathbb{A}}$ with

$$c' \in \left(\exists in'.\, \Cap{F}{in',out'} \circ \langle\!\langle K \rangle\!\rangle^{in',out'}_{in,out}\right) \equiv \bigvee_{c \in K} \left(\exists in'.\, \Cap{F}{in',out'} \circ \langle\!\langle c \rangle\!\rangle^{in',out'}_{in,out}\right).$$

There exists $c'' \in K$ such that

$$c' \in \left(\exists in'.\, \Cap{F}{in',out'} \circ \langle\!\langle c'' \rangle\!\rangle^{in',out'}_{in,out}\right).$$

By the Crust Inclusion Property, there exist $K' \in \mathcal{P}(\mathsf{C}_{\mathbb{B}})$ and $F' \in \mathcal{F}$ such that, for all $in'' \in \mathcal{I}_{\text{in}}$,

$$\left(\exists in'.\, \Cap{F}{in',out'} \circ \langle\!\langle c'' \rangle\!\rangle^{in',out'}_{in'',out}\right) \equiv K' \circ \Cap{F'}{in'',out}. \tag{6.1}$$

Hence, $c' \in K' \circ \Cap{F'}{in,out}$, and so there are $c_1 \in K'$ and $c_2 \in \Cap{F'}{in,out}$ with $c' = c_1 \circ c_2$. Fix $in'' \in \mathcal{I}_{\text{in}}$ and $c_2' \in \Cap{F'}{in'',out}$. Since $c_1 \circ \circ_2 \in K' \circ \Cap{F'}{in'',out}$, it follows by (6.1) that

$$c_1 \circ c_2' \in \left(\exists in'.\, \Cap{F}{in',out'} \circ \langle\!\langle c'' \rangle\!\rangle^{in',out'}_{in'',out}\right) \subseteq \exists in'.\, \Cap{F}{in',out'} \circ \langle\!\langle K \rangle\!\rangle^{in',out'}_{in'',out}.$$

The choice of c_2' was arbitrary, and so

$$\text{for all } c_2', c'' \in \mathsf{C}_{\mathbb{B}},\ c_2' \in \Cap{F}{in'',out} \text{ and } c'' = c_1 \circ c_2' \implies c'' \in \exists in'.\, \Cap{F}{in',out'} \bullet \langle\!\langle K \rangle\!\rangle^{in',out'}_{in'',out}.$$

Hence

$$c_1 \in \Cap{F}{in'',out} \multimap \left(\exists in'.\, \Cap{F}{in',out'} \bullet \langle\!\langle K \rangle\!\rangle^{in',out'}_{in'',out}\right)$$

and since the choice of in'' was arbitrary,

$$c_1 \in \forall in''.\, \Cap{F}{in'',out} \multimap \left(\exists in'.\, \Cap{F}{in',out'} \bullet \langle\!\langle K \rangle\!\rangle^{in',out'}_{in'',out}\right)$$

Since $c' = c_1 \circ c_2$,

$$c' \in \exists F'.\, \left(\forall in''.\, \Cap{}^{F}_{in'',out} \multimap \left(\exists in'.\, \Cap{}^{F}_{in',out'} \bullet \langle\!\langle K \rangle\!\rangle^{in',out'}_{in'',out}\right)\right) \circ \Cap{}^{F'}_{in,out}.$$

Since the choice of c' was arbitrary, it follows that

$$\left(\exists in'.\, \Cap{}^{F}_{in',out'} \circ \langle\!\langle K \rangle\!\rangle^{in',out'}_{in,out}\right) \subseteq \exists F'.\, \left(\forall in''.\, \Cap{}^{F'}_{in'',out} \multimap \left(\exists in'.\, \Cap{}^{F}_{in',out'} \circ \langle\!\langle K \rangle\!\rangle^{in',out'}_{in'',out}\right)\right) \circ \Cap{}^{F'}_{in,out}$$

as required. □

The proof of Proposition 91 uses the intermediate translation functions. Key to establishing the soundness of locality preserving translations is that these functions preserve locality context application, which is established by the following lemma.

Lemma 93 (Application Preservation II)**.** *For all* $K \in \mathcal{P}(\mathsf{C}_{\mathsf{State}_{\mathbb{A}}})$*, all* $P \in \mathcal{P}(\mathsf{State}_{\mathbb{A}})$*,* $out \in \mathcal{I}_{\text{out}}$ *and* $F \in \mathcal{F}$*,*

$$(\!|K \bullet_{\mathbb{A}} P|\!)^{out,F} \equiv \exists out', F'.\, (\!|K|\!)^{out,F}_{out',F'} \bullet_{\mathbb{B}} (\!|P|\!)^{out',F'}.$$

Proof.

$$(\!|K \bullet P|\!)^{out,F}$$

$$\equiv \bigvee_{\substack{(\rho',c_{\mathbb{A}})\in K \\ (\rho,\chi_{\mathbb{A}})\in P}} \{\rho' * \rho\} \times \left(\exists in.\, \Cap{}^{F}_{in,out} \bullet \langle\!\langle c_{\mathbb{A}} \bullet \chi_{\mathbb{A}} \rangle\!\rangle^{in,out}\right)$$

(Property 6.18)

$$\equiv \bigvee_{\substack{(\rho',c_{\mathbb{A}})\in K \\ (\rho,\chi_{\mathbb{A}})\in P}} \{\rho' * \rho\} \times \left(\exists in.\, \Cap{}^{F}_{in,out} \bullet \exists in', out'.\, \langle\!\langle c_{\mathbb{A}} \rangle\!\rangle^{in,out}_{in',out'} \bullet \langle\!\langle \chi_{\mathbb{A}} \rangle\!\rangle^{in',out'}\right)$$

$$\equiv \bigvee_{\substack{(\rho',c_{\mathbb{A}})\in K \\ (\rho,\chi_{\mathbb{A}})\in P}} \{\rho' * \rho\} \times \left(\exists in', out'.\, \exists in.\, \Cap{}^{F}_{in,out} \circ \langle\!\langle c_{\mathbb{A}} \rangle\!\rangle^{in,out}_{in',out'} \bullet \langle\!\langle \chi_{\mathbb{A}} \rangle\!\rangle^{in',out'}\right)$$

(Lemma 92)

$$\equiv \bigvee_{\substack{(\rho',c_{\mathbb{A}})\in K\\(\rho,\chi_{\mathbb{A}})\in P}} \{\rho' * \rho\} \times \left(\begin{array}{l} \exists in', out'.\, \exists F'. \\ \left(\forall in''.\, \Cap{}{}{F'}{in'',out'} \multimap \left(\exists in.\, \Cap{}{}{F}{in,out} \circ \langle\!\langle c_{\mathbb{A}} \rangle\!\rangle^{in,out}_{in'',out'}\right)\right) \\ \qquad\qquad \circ \Cap{}{}{F'}{in',out'} \bullet \langle\!\langle \chi_{\mathbb{A}} \rangle\!\rangle^{in',out'} \end{array} \right)$$

$$\equiv \exists out', F'.$$

$$\left(\bigvee_{(\rho',c_{\mathbb{A}})\in K} \{\rho'\} \times \left(\forall in''.\, \Cap{}{}{F'}{in'',out'} \multimap \left(\exists in.\, \Cap{}{}{F}{in,out} \circ \langle\!\langle c_{\mathbb{A}} \rangle\!\rangle^{in,out}_{in'',out'}\right)\right)\right)$$

$$\bullet \left(\bigvee_{(\rho,\chi_{\mathbb{A}})\in P} \{\rho\} \times \left(\exists in'.\, \Cap{}{}{F'}{in',out'} \bullet \langle\!\langle \chi_{\mathbb{A}} \rangle\!\rangle^{in',out'}\right)\right)$$

$$\equiv \exists out', F'.\, (\!|K|\!)^{out,F}_{out',F'} \bullet (\!|P|\!)^{out',F'}.$$

□

The proof of Proposition 91 inductively transforms a proof in $\mathbb{A}$ into a proof in $\mathbb{B}$.

Proof. The proof is by induction on the structure of the proof of $\vdash_{\mathbb{A}} \{P\}\ \mathbb{C}\ \{Q\}$, considering the cases for the last rule applied in the proof. Assume as the inductive hypothesis that the translated premises have proofs in $\mathbb{B}$. I show how to derive a proof of the translated conclusions from these translated premises. (I omit the procedure specification environment when it plays no role in the derivation.)

Fix arbitrary $out \in \mathcal{I}_{\text{out}}$, $F \in \mathcal{F}$.

Axiom case:

This case is immediate by the Axiom Correctness Property (Property 6.20).

Frame case:

$$\dfrac{\dfrac{\dfrac{\text{for all } out', F',\ \left\{(\!|P|\!)^{out',F'}\right\}\ \mathbb{C}\ \left\{(\!|Q|\!)^{out',F'}\right\}}{\text{for all } out', F',\ \begin{array}{c}\left\{(\!|K|\!)^{out,F}_{out',F'} \bullet (\!|P|\!)^{out',F'}\right\}\\ \mathbb{C}\\ \left\{(\!|K|\!)^{out,F}_{out',F'} \bullet (\!|Q|\!)^{out',F'}\right\}\end{array}}\ \textsc{Frame}}{\begin{array}{c}\left\{\exists out', F'.\, (\!|K|\!)^{out,F}_{out',F'} \bullet (\!|P|\!)^{out',F'}\right\}\\ \mathbb{C}\\ \left\{\exists out', F'.\, (\!|K|\!)^{out,F}_{out',F'} \bullet (\!|Q|\!)^{out',F'}\right\}\end{array}}\ \textsc{Disj}}{\left\{(\!|K \bullet P|\!)^{out,F}\right\}\ \mathbb{C}\ \left\{(\!|K \bullet Q|\!)^{out,F}\right\}}\ \text{Lemma 93}$$

Cons case:

$$\dfrac{\dfrac{P \subseteq P'}{(\!|P|\!)^{out,F} \subseteq (\!|P'|\!)^{out,F}} \quad \begin{array}{c}\left\{(\!|P'|\!)^{out,F}\right\}\\ \mathbb{C} \\ \left\{(\!|Q'|\!)^{out}\,F\right\}\end{array} \quad \dfrac{Q' \subseteq Q}{(\!|Q'|\!)^{out,F} \subseteq (\!|Q|\!)^{out,F}}}{\left\{(\!|P|\!)^{out,F}\right\}\ \mathbb{C}\ \left\{(\!|Q|\!)^{out}\,F\right\}}\ \textsc{Cons}$$

Disj case:

$$\dfrac{\dfrac{\text{for all } i \in I,\ \left\{(\!|P_i|\!)^{out,F}\right\}\ \mathbb{C}\ \left\{(\!|Q_i|\!)^{out,F}\right\}}{\left\{\bigvee_{i\in I} (\!|P_i|\!)^{out,F}\right\}\ \mathbb{C}\ \left\{\bigvee_{i\in I} (\!|Q_i|\!)^{out,F}\right\}}\ \textsc{Disj}}{\left\{(\!|\bigvee_{i\in I} P_i|\!)^{out,F}\right\}\ \mathbb{C}\ \left\{(\!|\bigvee_{i\in I} Q_i|\!)^{out,F}\right\}}$$

PDef case:

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}\text{for all } (\mathtt{f}_i : \mathsf{P} \rightarrowtail \mathsf{Q}) \in \Gamma,\\ out' \in \mathcal{I}_{\text{out}}, F' \in \mathcal{F},\end{array}\ [\![\Gamma', \Gamma]\!] \vdash_{\mathbb{B}} \begin{array}{c}\left\{\left(\!\left|\begin{array}{r}\exists \overrightarrow{v}.\,\{\overrightarrow{\boldsymbol{x}_i} \Rightarrow \overrightarrow{v} * \overrightarrow{\boldsymbol{r}_i} \Rightarrow -\}\\ \times \mathsf{P}(\overrightarrow{v})\end{array}\right|\!\right)^{out',F'}\right\}\\ \mathbb{C}_i\\ \left\{\left(\!\left|\begin{array}{r}\exists \overrightarrow{w}.\,\{\overrightarrow{\boldsymbol{x}_i} \Rightarrow - * \overrightarrow{\boldsymbol{r}_i} \Rightarrow \overrightarrow{w}\}\\ \times \mathsf{Q}(\overrightarrow{w})\end{array}\right|\!\right)^{out',F'}\right\}\end{array}}{\begin{array}{c}\text{for all } (\mathtt{f}_i : \mathsf{P} \rightarrowtail \mathsf{Q}) \in \Gamma,\\ out' \in \mathcal{I}_{\text{out}}, F' \in \mathcal{F},\end{array}\ [\![\Gamma', \Gamma]\!] \vdash_{\mathbb{B}} \begin{array}{c}\left\{\begin{array}{r}\exists \overrightarrow{v}.\,\{\overrightarrow{\boldsymbol{x}_i} \Rightarrow \overrightarrow{v} * \overrightarrow{\boldsymbol{r}_i} \Rightarrow -\}\\ \times (\!|\mathsf{P}(\overrightarrow{v})|\!)^{out',F'}\end{array}\right\}\\ \mathbb{C}_i\\ \left\{\begin{array}{r}\exists \overrightarrow{w}.\,\{\overrightarrow{\boldsymbol{x}_i} \Rightarrow - * \overrightarrow{\boldsymbol{r}_i} \Rightarrow \overrightarrow{w}\}\\ \times (\!|\mathsf{Q}(\overrightarrow{w})|\!)^{out',F'}\end{array}\right\}\end{array}}}{\text{for all } (\mathtt{f}_i : \mathsf{P} \rightarrowtail \mathsf{Q}) \in [\![\Gamma]\!],\ [\![\Gamma', \Gamma]\!] \vdash_{\mathbb{B}} \begin{array}{c}\{\exists \overrightarrow{v}.\,\{\overrightarrow{\boldsymbol{x}_i} \Rightarrow \overrightarrow{v} * \overrightarrow{\boldsymbol{r}_i} \Rightarrow -\} \times \mathsf{P}(\overrightarrow{v})\}\\ \mathbb{C}_i\\ \{\exists \overrightarrow{w}.\,\{\overrightarrow{\boldsymbol{x}_i} \Rightarrow - * \overrightarrow{\boldsymbol{r}_i} \Rightarrow \overrightarrow{w}\} \times \mathsf{Q}(\overrightarrow{w})\}\end{array}}}{(\star)}$$

$$\dfrac{(\star) \quad [\![\Gamma', \Gamma]\!] \vdash_{\mathbb{B}} \left\{(\!|P|\!)^{out,F}\right\}\ \mathbb{C}\ \left\{(\!|Q|\!)^{out,F}\right\}}{\begin{array}{c}\left\{(\!|P|\!)^{out,F}\right\}\\ [\![\Gamma']\!] \vdash_{\mathbb{B}} \mathtt{procs}\ \overrightarrow{\boldsymbol{r}_1} := \mathtt{f}_1(\overrightarrow{\boldsymbol{x}_1})\ \{\mathbb{C}_1\}, \ldots, \overrightarrow{\boldsymbol{r}_k} := \mathtt{f}_k(\overrightarrow{\boldsymbol{x}_k})\ \{\mathbb{C}_k\}\ \mathtt{in}\ \mathbb{C}\\ \left\{(\!|R|\!)^{out,F}\right\}\end{array}}\ \textsc{PDef}$$

The two further premises of the PDef rule, which are not shown in the above derivation, are easily dispatched since $[\![(\cdot)]\!]$ preserves the procedure names in a procedure specification environment.

The cases for the remaining rules follow by the point-wise and variable-preserving nature of the translation functions. □

This completes the proof of Theorem 90.

6.3.2 Module Translation: $\tau_1 : \mathbb{T} \to \mathbb{H} + \mathbb{L}$

In this section, I give the formal definition of a locality-preserving translation τ_1 from the tree module into the combination of the heap and list modules.

Notation. A number of notational conventions are used to simplify predicates over $\mathcal{A}_{\mathbb{H}} + \mathcal{A}_{\mathbb{L}}$. Pure heap predicates are implicitly lifted to the combined domain with the list-store component emp; list-store predicates are lifted similarly. The operator $*$ on the combined domain is the product of the $*$ operators from each of the two domains.

Definition 6.22 ($\tau_1 : \mathbb{T} \to \mathbb{H} + \mathbb{L}$)**.** The pre-locality-preserving translation $\tau_1 : \mathbb{T} \to \mathbb{H} + \mathbb{L}$ is constructed as follows:

- the in-interfaces $\mathcal{I}_{\text{in}} = (\mathsf{Addr})^*$ are sequences of addresses;
- the out-interfaces $\mathcal{I}_{\text{out}} = \mathsf{Addr}$ are addresses;
- the data representation function is defined inductively by

$$\begin{aligned}
\langle\!\langle \varnothing \rangle\!\rangle^{in,out} &\stackrel{\text{def}}{=} \text{emp} \wedge in = \varnothing \\
\langle\!\langle \mathbf{a}[t] \rangle\!\rangle^{in,out} &\stackrel{\text{def}}{=} in = \mathbf{a} \wedge \exists a, l.\, \mathbf{a} \mapsto out, a * a \Rrightarrow [l] * \langle\!\langle t \rangle\!\rangle^{l,\mathbf{a}} \\
\langle\!\langle t_1 \otimes t_2 \rangle\!\rangle^{in,out} &\stackrel{\text{def}}{=} \exists l_1, l_2.\, (in = l_1 \cdot l_2) \wedge \langle\!\langle t_1 \rangle\!\rangle^{l_1,out} * \langle\!\langle t_2 \rangle\!\rangle^{l_2,out};
\end{aligned}$$

- the context representation function is defined inductively by

$$\begin{aligned}
\langle\!\langle - \rangle\!\rangle^{in,out}_{in',out'} &\stackrel{\text{def}}{=} \text{emp} \wedge (in = in') \wedge (out = out') \\
\langle\!\langle \mathbf{a}[c] \rangle\!\rangle^{in,out}_{I'} &\stackrel{\text{def}}{=} in = \mathbf{a} \wedge \exists a, l.\, \mathbf{a} \mapsto out, a * a \Rrightarrow [l] * \langle\!\langle c \rangle\!\rangle^{l,\mathbf{a}}_{I'} \\
\langle\!\langle c \otimes t \rangle\!\rangle^{in,out}_{I'} &\stackrel{\text{def}}{=} \exists l_1, l_2.\, (in = l_1 \cdot l_2) \wedge \langle\!\langle c \rangle\!\rangle^{l_1,out}_{I'} * \langle\!\langle t \rangle\!\rangle^{l_2,out} \\
\langle\!\langle t \otimes c \rangle\!\rangle^{in,out}_{I'} &\stackrel{\text{def}}{=} \exists l_1, l_2.\, (in = l_1 \cdot l_2) \wedge \langle\!\langle t \rangle\!\rangle^{l_1,out} * \langle\!\langle c \rangle\!\rangle^{l_2,out}_{I'};
\end{aligned}$$

- the crust parameters $\mathcal{F} = (\mathsf{Addr})^* \times (\mathsf{Addr})^* \times \mathsf{Addr}$ are tuples of two sequences of addresses and a further address;
- the crust predicate is defined as

$$\Cap^{l_1,l_2,a}_{in,out} \stackrel{\text{def}}{=} \exists a'.\, out \mapsto a, a' * a' \Rrightarrow [l_1 \cdot in \cdot l_2] * \prod_{\mathbf{a} \in l_1 \cdot l_2}^{*} \mathbf{a} \mapsto out; \text{ and}$$

- the substitutive implementation function is given by replacing each tree-module command with a call to the correspondingly-named procedure given in Figure 6.3.

Theorem 94 (Soundness of τ_1)**.** *The pre-locality-preserving translation τ_1 is a locality-preserving translation.*

$$E.\texttt{parent} \stackrel{\text{def}}{=} E$$

$$E.\texttt{children} \stackrel{\text{def}}{=} E + 1$$

$$n := \texttt{newNode} \stackrel{\text{def}}{=} n := \texttt{alloc}(2)$$

$$\texttt{disposeNode}(E) \stackrel{\text{def}}{=} \texttt{dispose}(E, 2)$$

```
m := getUp(n) {
  local x in
    m := [n.parent];
    x := [m.parent];
    if x = nil then
      m := nil
}

m := getLast(n) {
  local x in
    x := [n.children];
    m := x.getTail()
}

newNodeAfter(n) {
  local x, y, z, w in
    x := [n.parent];
    z := [x.children];
    y := newNode();
    [y.parent] := x;
    z.insert(n, y);
    w := newList();
    [y.children] := w
}

m := getFirst(n) {
  local x in
    x := [n.children];
    m := x.getHead()
}

m := getRight(n) {
  local x, y in
    x := [n.parent];
    y := [x.children];
    m := y.getNext(n)
}

m := getLeft(n) {
  local x, y in
    x := [n.parent];
    y := [x.children];
    m := y.getPrev(n)
}

deleteTree(n) {
  local x, y, z in
    x := [n.parent];
    y := [x.children];
    y.remove(n);
    y := [n.children];
    z := y.getHead();
    while z ≠ nil do
      call deleteTree(z);
      z := y.getHead()  ;
    deleteList(y);
    disposeNode(n)
}
```

Figure 6.3: Procedures for the implementation of list-based trees

Soundness of $\tau_1 : \mathbb{T} \to \mathbb{H} + \mathbb{L}$

Lemma 95 (τ_1 Application Preservation)**.** *For all $c \in \mathsf{C}_{\mathsf{UTree}}$, $t \in \mathsf{UTree}$ and $I \in \mathcal{I}$,*

$$\langle\!\langle c \bullet t \rangle\!\rangle^I \equiv \exists I'.\, \langle\!\langle c \rangle\!\rangle^I_{I'} \bullet \langle\!\langle t \rangle\!\rangle^{I'}.$$

Proof. The proof is by induction on the structure of the context c, assuming some fixed $t \in \mathsf{UTree}$.

$c = \_$ case:

$$\begin{aligned}\exists I'.\, \langle\!\langle \_ \rangle\!\rangle^I_{I'} \bullet \langle\!\langle t \rangle\!\rangle^{I'} &\equiv \exists I'.\, I = I' \wedge \langle\!\langle t \rangle\!\rangle^{I'} \\ &\equiv \langle\!\langle \chi \rangle\!\rangle^I \\ &\equiv \langle\!\langle \_ \bullet t \rangle\!\rangle^I.\end{aligned}$$

$c = \mathbf{a}[c']$ case:

$$\begin{aligned}&\exists I'.\, \langle\!\langle \mathbf{a}[c'] \rangle\!\rangle^{in,out}_{I'} \bullet \langle\!\langle t \rangle\!\rangle^{I'} \\ &\qquad\equiv \exists I'.\, \left(in = \mathbf{a} \wedge \exists a, l.\, \mathbf{a} \mapsto out, a * a \Rrightarrow [l] * \langle\!\langle c' \rangle\!\rangle^{l,\mathbf{a}}_{I'}\right) \bullet \langle\!\langle t \rangle\!\rangle^{I'} \\ &\qquad\equiv in = \mathbf{a} \wedge \exists a, l.\, \mathbf{a} \mapsto out, a * a \Rrightarrow [l] * \left(\exists I'.\, \langle\!\langle c' \rangle\!\rangle^{l,\mathbf{a}}_{I'} \bullet \langle\!\langle t \rangle\!\rangle^{I'}\right) \\ &\qquad\equiv in = \mathbf{a} \wedge \exists a, l.\, \mathbf{a} \mapsto out, a * a \Rrightarrow [l] * \langle\!\langle c' \bullet t \rangle\!\rangle^{l,\mathbf{a}} \\ &\qquad\equiv \langle\!\langle \mathbf{a}[c' \bullet t] \rangle\!\rangle^{in,out} \\ &\qquad\equiv \langle\!\langle \mathbf{a}[c'] \bullet t \rangle\!\rangle^{in,out}.\end{aligned}$$

$c = c' \otimes t'$ case:

$$\begin{aligned}&\exists I'.\, \langle\!\langle c' \otimes t' \rangle\!\rangle^{in,out}_{I'} \bullet \langle\!\langle t \rangle\!\rangle^{I'} \\ &\qquad\equiv \exists I'.\, \left(\exists l_1, l_2.\, (in = l_1 \cdot l_2) \wedge \langle\!\langle c' \rangle\!\rangle^{l_1,out}_{I'} * \langle\!\langle t' \rangle\!\rangle^{l_2,out}\right) \bullet \langle\!\langle t \rangle\!\rangle^{I'} \\ &\qquad\equiv \exists l_1, l_2.\, (in = l_1 \cdot l_2) \wedge \left(\exists I'.\, \langle\!\langle c' \rangle\!\rangle^{l_1,out}_{I'} \bullet \langle\!\langle t \rangle\!\rangle^{I'}\right) * \langle\!\langle t' \rangle\!\rangle^{l_2,out} \\ &\qquad\equiv \exists l_1, l_2.\, (in = l_1 \cdot l_2) \wedge \langle\!\langle c' \bullet t \rangle\!\rangle^{l_1,out} * \langle\!\langle t' \rangle\!\rangle^{l_2,out} \\ &\qquad\equiv \langle\!\langle (c' \bullet t) \otimes t' \rangle\!\rangle^{in,out} \\ &\qquad\equiv \langle\!\langle (c' \otimes t') \bullet t \rangle\!\rangle^{in,out}.\end{aligned}$$

The case where $c = t' \otimes c'$ follows the same pattern as the $c' \otimes t'$ case. □

Lemma 96 (τ_1 Crust Inclusion)**.** *For all $out, out' \in \mathcal{I}_{\text{out}}$, $F \in \mathcal{F}$ and $c \in \mathsf{C}_{\mathsf{UTree}}$, there exist $K \in \mathcal{P}(\mathsf{C}_{\mathbb{H}+\mathbb{L}})$ and $F' \in \mathcal{F}$ such that, for all $in \in \mathcal{I}_{\text{in}}$,*

$$\left(\exists in'.\, \Cap^F_{in',out'} \circ \langle\!\langle c \rangle\!\rangle^{in',out'}_{in,out}\right) \equiv K \circ \Cap^{F'}_{in,out}.$$

Proof. The proof is by induction on the structure of the context c, assuming some fixed $out, out' \in \mathcal{I}_{\text{out}}$ and $F \in \mathcal{F}$.

$c = \_$ case:

Choose $K = \text{emp}$ if $out' = out$ and $K = \text{False}$ otherwise; choose $F' = F$. If $out' \neq out$ then both sides of the equation are equivalent to False; otherwise, observe that

$$\begin{aligned}\exists in'.\, \Cap{}{}{F}_{in',out'} \circ \langle\!\langle - \rangle\!\rangle^{in',out'}_{in,out} &\equiv \Cap{}{}{F}_{in,out} \\ &\equiv K \circ \Cap{}{}{F}_{in,out}.\end{aligned}$$

$c = \mathbf{a}[c']$ case:

By the inductive hypothesis, there exist K' and F' such that, for all in,

$$\exists l.\, \Cap{}{}{\varnothing,\varnothing,out'}_{l,\mathbf{a}} \circ \langle\!\langle c' \rangle\!\rangle^{l,\mathbf{a}}_{in,out} \equiv K' \circ \Cap{}{}{F'}_{in,out}.$$

Choose $K = \Cap{}{}{F}_{\mathbf{a},out'} \circ K'$; choose F' as given above. Observe that

$$\begin{aligned}&\exists in'.\, \Cap{}{}{F}_{in',out'} \circ \langle\!\langle \mathbf{a}[c'] \rangle\!\rangle^{in',out'}_{in,out} \\ &\quad\equiv \exists in'.\, \Cap{}{}{F}_{in',out'} \circ \left(in' = \mathbf{a} \wedge \exists a, l.\, \mathbf{a} \mapsto out', a * a \Rightarrow [l] * \langle\!\langle c' \rangle\!\rangle^{l,\mathbf{a}}_{in,out}\right) \\ &\quad\equiv \Cap{}{}{F}_{\mathbf{a},out'} \circ \exists l.\, \Cap{}{}{\varnothing,\varnothing,out'}_{l,\mathbf{a}} \circ \langle\!\langle c' \rangle\!\rangle^{l,\mathbf{a}}_{in,out} \\ &\quad\equiv \Cap{}{}{F}_{\mathbf{a},out'} \circ K' \circ \Cap{}{}{F'}_{in,out} \\ &\quad\equiv K \circ \Cap{}{}{F'}_{in,out}.\end{aligned}$$

$c = c' \otimes t'$ case:

Observe that there is exactly one choice of $l_2 \in (\mathsf{Addr})^*$ such that $\langle\!\langle t' \rangle\!\rangle^{l_2,out'} \not\equiv \text{false}$. Let $\hat{l}_2$ be that choice. Observe also that there exists some K' such that

$$\langle\!\langle t' \rangle\!\rangle^{\hat{l}_2,out'} \equiv K' * \prod_{\mathbf{a} \in \hat{l}_2}^{*} \mathbf{a} \mapsto out'.$$

Let $(l_1', l_2', a') = F$. By the inductive hypothesis, there exist K'' and F' such that, for all in,

$$\exists l_1.\, \Cap{}{}{l_1', \hat{l}_2 \cdot l_2', a'}_{l_1,out'} \circ \langle\!\langle c' \rangle\!\rangle^{l_1,out'}_{in,out} \equiv K'' \circ \Cap{}{}{F'}_{in,out}.$$

Choose $K = K' \circ K''$; choose F' as given above. Observe that

$$
\begin{aligned}
&\exists in'.\, \Cap^{l_1', l_2', a'}_{in', out'} \circ \langle\!\langle c' \otimes t' \rangle\!\rangle^{in', out'}_{in, out} \\
&\qquad \equiv \exists in'. \left(\exists a.\, out' \mapsto a', a * a \Mapsto [l_1' \cdot in' \cdot l_2'] * \prod^{*}_{\mathbf{a} \in l_1' \cdot l_2'} \mathbf{a} \mapsto out' \right) \\
&\qquad\qquad\qquad\qquad \circ \exists l_1.\, in' = l_1 \cdot \hat{l}_2 \wedge \langle\!\langle c' \rangle\!\rangle^{l_1, out'}_{in, out} * \langle\!\langle t' \rangle\!\rangle^{\hat{l}_2, out'} \\
&\qquad \equiv \exists l_1. \left(\exists a.\, out' \mapsto a', a * a \Mapsto \left[l_1' \cdot l_1 \cdot \hat{l}_2 \cdot l_2' \right] * \prod^{*}_{\mathbf{a} \in l_1' \cdot l_2'} \mathbf{a} \mapsto out' \right) \\
&\qquad\qquad\qquad\qquad \circ \langle\!\langle c' \rangle\!\rangle^{l_1, out'}_{in, out} * K' * \prod^{*}_{\mathbf{a} \in \hat{l}_2} \mathbf{a} \mapsto out' \\
&\qquad \equiv K' \circ \exists l_1.\, \Cap^{l_1', \hat{l}_2 \cdot l_2', a'}_{l_1, out'} \circ \langle\!\langle c' \rangle\!\rangle^{l_1, out'}_{in, out} \\
&\qquad \equiv K' \circ K'' \circ \Cap^{F'}_{in, out} \\
&\qquad \equiv K \circ \Cap^{F'}_{in, out}.
\end{aligned}
$$

The case where $c = t' \otimes c'$ follows the same pattern as the $c' \otimes t'$ case. □

Lemma 97 (τ_1 Axiom Correctness)**.** *For all* $\varphi \in \mathsf{Cmd}_{\mathbb{L}}$*,* $(P, Q) \in \mathrm{Ax}\,[\![\varphi]\!]_{\mathbb{L}}$*,* $out \in \mathcal{I}_{\mathrm{out}}$ *and* $F \in \mathcal{F}$*,*

$$\vdash_{\mathbb{B}} \left\{ (\!|P|\!)^{out, F} \right\} \; [\![\varphi]\!] \; \left\{ (\!|Q|\!)^{out, F} \right\}.$$

I omit the full proof details here, which are due to Wheelhouse and can be found in [DYGW10b]. Figure 6.4 shows one of the simpler proof cases involved in establishing axiom correctness, namely for the `getUp` axiom.

This completes the proof of Theorem 94.

6.3.3 Module Translation: $\tau_2 : \mathbb{H} + \mathbb{H} \to \mathbb{H}$

Another example of a locality-preserving translation is the natural implementation of a pair of heap modules $\mathbb{H} + \mathbb{H}$ with a single heap $\mathbb{H}$ that simply treats the two heaps as (disjoint) portions of the same heap.

Definition 6.23 ($\tau_2 : \mathbb{H} + \mathbb{H} \to \mathbb{H}$)**.** The pre-locality-preserving translation $\tau_2 : \mathbb{H} + \mathbb{H} \to \mathbb{H}$ is constructed as follows:

- the interface sets are both unit sets,[5] *i.e.* $\mathcal{I}_{\mathrm{in}} = \{1\} = \mathcal{I}_{\mathrm{out}}$, and hence they can be ignored;

[5]It makes no difference which set, as long as it only has one element.

$$
\begin{array}{l}
m := \texttt{getUp}(n)\ \{ \\
\quad \left\{m \Rrightarrow - * n \Rrightarrow \mathbf{b} \times \exists in.\, \Cap^F_{in,out} \bullet \langle\!\langle \mathbf{a}[t_1 \otimes \mathbf{b}[t_2] \otimes t_3] \rangle\!\rangle^{in,out}\right\} \\
\quad \left\{\begin{array}{l} m \Rrightarrow - * n \Rrightarrow \mathbf{b} \times \Cap^F_{\mathbf{a},out} \bullet \exists a, l_1, l_3.\, \mathbf{a} \mapsto out, a * a \Mapsto [l_1 \cdot \mathbf{b} \cdot l_3] \\ \qquad * \langle\!\langle t_1 \rangle\!\rangle^{l_1,\mathbf{a}} * \exists a', l.\, \mathbf{b} \mapsto \mathbf{a}, a' * a' \Mapsto [l_2] * \langle\!\langle t_2 \rangle\!\rangle^{l_2,\mathbf{b}} * \langle\!\langle t_3 \rangle\!\rangle^{l_3,\mathbf{a}} \end{array}\right\} \\
\quad \{m \Rrightarrow - * n \Rrightarrow \mathbf{b} \times \mathbf{a} \mapsto out * \mathbf{b} \mapsto \mathbf{a}\} \\
\quad \texttt{local}\ x\ \texttt{in} \\
\qquad \{m \Rrightarrow - * n \Rrightarrow \mathbf{b} * x \Rrightarrow - \times \mathbf{a} \mapsto out * \mathbf{b} \mapsto \mathbf{a}\} \\
\qquad m := [n.\texttt{parent}]\,; \\
\qquad \{m \Rrightarrow \mathbf{a} * n \Rrightarrow \mathbf{b} * x \Rrightarrow - \times \mathbf{a} \mapsto out * \mathbf{b} \mapsto \mathbf{a}\} \\
\qquad x := [m.\texttt{parent}]\,; \\
\qquad \{m \Rrightarrow \mathbf{a} * n \Rrightarrow \mathbf{b} * x \Rrightarrow out \times \mathbf{a} \mapsto out * \mathbf{b} \mapsto \mathbf{a}\} \\
\qquad \texttt{if}\ x = \mathit{nil}\ \texttt{then} \\
\qquad\quad m := \mathit{nil} \\
\qquad \{m \Rrightarrow \mathbf{a} * n \Rrightarrow \mathbf{b} * x \Rrightarrow out \times \mathbf{a} \mapsto out * \mathbf{b} \mapsto \mathbf{a}\} \\
\quad \{m \Rrightarrow \mathbf{a} * n \Rrightarrow \mathbf{b} \times \mathbf{a} \mapsto out * \mathbf{b} \mapsto \mathbf{a}\} \\
\quad \left\{\begin{array}{l} m \Rrightarrow \mathbf{a} * n \Rrightarrow \mathbf{b} \times \Cap^F_{\mathbf{a},out} \bullet \exists a, l_1, l_3.\, \mathbf{a} \mapsto out, a * a \Mapsto [l_1 \cdot \mathbf{b} \cdot l_3] \\ \qquad * \langle\!\langle t_1 \rangle\!\rangle^{l_1,\mathbf{a}} * \exists a', l.\, \mathbf{b} \mapsto \mathbf{a}, a' * a' \Mapsto [l_2] * \langle\!\langle t_2 \rangle\!\rangle^{l_2,\mathbf{b}} * \langle\!\langle t_3 \rangle\!\rangle^{l_3,\mathbf{a}} \end{array}\right\} \\
\quad \left\{m \Rrightarrow \mathbf{a} * n \Rrightarrow \mathbf{b} \times \exists in.\, \Cap^F_{in,out} \bullet \langle\!\langle \mathbf{a}[t_1 \otimes \mathbf{b}[t_2] \otimes t_3] \rangle\!\rangle^{in,out}\right\} \\
\}
\end{array}
$$

Figure 6.4: Proof outline for `getUp` implementation

- the representation function (which is the same for both data and contexts) is defined as
$$\langle\!\langle (h_1, h_2) \rangle\!\rangle = \{h_1\} * \{h_2\}\,;$$
- the crust parameter set is also the unit set and the crust predicate is defined as $\Cap = \mathrm{emp}$; and
- the implementation function is given by replacing the commands for both heaps with their detagged versions, for example
$$[\![n := \texttt{alloc}_1(E)]\!] = n := \texttt{alloc}(E) = [\![n := \texttt{alloc}_2(E)]\!]\,.$$

Theorem 98 (Soundness of τ_2)**.** *The pre-locality-preserving translation τ_2 is a locality-preserving translation.*

This theorem is trivial: application preservation holds by properties of $*$; crust inclusion holds since the crust is simply emp; and axiom correctness holds because the axioms of $\mathbb{H}+\mathbb{H}$ are almost directly translated to those of $\mathbb{H}$, modulo a frame.

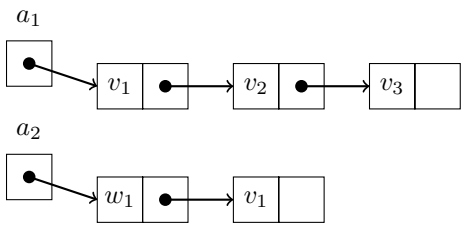


Figure 6.5: Representation of the list store $a_1 \Mapsto [v_1 \cdot v_2 \cdot v_3] * a_2 \Mapsto [w_1 \cdot v_1]$ as singly linked lists in the heap

6.4 Locality-Breaking Translations

There is not always a close correspondence between the locality exhibited by a high-level module and the locality of the low-level module on which it is implemented. In this section, I reduce the burden of proof for a sound module translation in such cases by introducing *locality-breaking translations*. In §6.4.1, I establish that locality-breaking translations give sound module translations, and in §6.4.2 I give a locality-breaking translation $\tau_3 : \mathbb{L} \to \mathbb{H}$.

Our motivating example of an module translation that does not preserve locality is an implementation of the list module (as defined in §6.1.3) that represents each abstract list with a singly-linked list in the heap. An example of a list store represented in this way is depicted in Figure 6.5. Consider the operation of removing the value v_3 from the list at address a_1. At the abstract level, the footprint of the operation is simply $a_1 \Mapsto v_3$, but in the implementation the list at a_1 must be traversed all the way to the node labelled v_3 — in this case, this is the entire list.

To apply the locality-preserving translation approach, an arbitrary amount of additional datastructure would have to be included in the crust. This seems to defeat the purpose of the locality-preserving approach in the fist place, so perhaps a more direct approach is called for.

In order to prove the soundness of a module translation, it is necessary to demonstrate a transformation from high-level proofs about programs that use an abstract module to low-level proofs of those programs using the concrete module implementation. Inductively transforming proofs is, intuitively, a good strategy because the high-level rules are matched by the low-level rules. Since the definition of predicate translations preserves conjunctions, disjunctions and entailments, and also the variable scope, the majority of the proof rules can be inductively transformed to their low-level counterparts directly. The two exceptions to this are Frame and Axiom. For locality-preserving translations,

FRAME was dealt with by the fact that a locality-preserving translation preserves context application, and AXIOM was dealt with by the fact that the implementations of the basic commands satisfy the axioms under such a translation.

When it is not appropriate to consider a translation that preserves locality, it would be useful if it were only necessary to consider proofs which do not use the frame rule, or, at least, only use it in a limited fashion. Intuitively, of course this should be possible — the purpose of the frame rule is to factor out parts of the state that do not play a role (and hence, do not change) in part of the program under consideration. Thus, it should be possible to transform a proof to one in which the frame rule is only applied to basic statements (*i.e.* basic commands and assignment) by factoring in the state earlier in the proof (*i.e.* at the leaves of the proof). This intuition is formalised in the following lemma.

Lemma 99 (Frame-Free Derivations)**.** *Let $\mathbb{A}$ be an abstract module. If there is a proof derivation of $\vdash_{\mathbb{A}} \{P\}\ \mathbb{C}\ \{Q\}$ then there is also a derivation that only uses the frame rule in the following ways:*

$$\dfrac{\dfrac{}{\Gamma \vdash_{\mathbb{A}} \{P'\}\ \mathbb{C}'\ \{Q'\}}\ (\dagger)}{\Gamma \vdash_{\mathbb{A}} \{K \bullet P'\}\ \mathbb{C}'\ \{K \bullet Q'\}}\ \textsc{Frame} \tag{6.2}$$

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash_{\mathbb{A}} \{P'\}\ \mathbb{C}'\ \{Q'\}}}{\Gamma \vdash_{\mathbb{A}} \{(K_{\mathsf{Scope}} \times \mathbf{I}_{\mathbb{A}}) \bullet P'\}\ \mathbb{C}'\ \{(K_{\mathsf{Scope}} \times \mathbf{I}_{\mathbb{A}}) \bullet Q'\}}\ \textsc{Frame} \tag{6.3}$$

where (†) *is either* AXIOM *or* ASSGN*.*

Consider a translation $\tau : \mathbb{A} \to \mathbb{B}$. Lemma 99 implies that it is only necessary to provide proofs of $\vdash_{\mathbb{B}} \{[\![P]\!]_\tau\}\ [\![\mathbb{C}]\!]_\tau\ \{[\![Q]\!]_\tau\}$ when there is a proof of $\vdash_{\mathbb{A}} \{P\}\ \mathbb{C}\ \{Q\}$ having the prescribed form. Provided that there is are proofs that the implementation of each command in $\mathsf{Cmd}_{\mathbb{A}}$ satisfies the translation of its axioms under every possible frame, the proof in $\mathbb{A}$ can be transformed to a proof in $\mathbb{B}$ by straightforward induction. Considerations can be reduced to only *singleton* frames by considering an arbitrary frame as a disjunction of singletons and applying the DISJ rule. Considerations can be reduced even further to singleton frames that have no variable component, since the variable scope component can be added by FRAME at the low-level. This condition is formalised in the definition of a locality-breaking translation.

Definition 6.24 (Locality-Breaking Translation)**.** A *locality-breaking translation* $\tau : \mathbb{A} \to \mathbb{B}$ is a module translation having the property that, for all $c \in \mathsf{C}_{\mathbb{A}}$,

$\varphi \in \mathsf{Cmd}_{\mathbb{A}}$, and $(P, Q) \in \textsc{Ax}\,[\![\varphi]\!]_{\mathbb{A}}$ there is a derivation of

$$\vdash_{\mathbb{B}} \{[\![\{(\varnothing, c)\} \bullet P]\!]_\tau\}\ [\![\varphi]\!]_\tau\ \{[\![\{(\varnothing, c)\} \bullet Q]\!]_\tau\}\,.$$

Theorem 100 (Soundness of Locality-Breaking Translations)**.** *A locality-breaking translation is a sound module translation.*

6.4.1 Proof of Soundness of Locality-Breaking Translations

Proof of Lemma 99. The result is a special case of the more general result, that if there is a derivation of $\Gamma \vdash_{\mathbb{A}} \{P\}\ \mathbb{C}\ \{Q\}$ then there is a derivation of $F(\Gamma) \vdash_{\mathbb{A}} \{P\}\ \mathbb{C}\ \{Q\}$ with the required property, where

$$F(\Gamma) = \{\mathtt{f} : K \bullet \mathsf{P} \rightarrowtail K \bullet \mathsf{Q} \mid K \in \mathcal{P}(\mathsf{C}_{\mathbb{A}}) \text{ and } (\mathtt{f} : \mathsf{P} \rightarrowtail \mathsf{Q}) \in \Gamma\}\,.$$

Note that $\Gamma \subseteq F(\Gamma) = F(F(\Gamma))$. Since procedure specifications are only relevant to the PDef and PCall rules, I will omit them when considering other rules.

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 Frame or PDef 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.

Consider the case where the last rule of the derivation is Frame:

$$\dfrac{\dfrac{\vdots}{\{P'\}\ \mathbb{C}\ \{Q'\}}\ (\ddagger)}{\{K \bullet P'\}\ \mathbb{C}\ \{K \bullet Q'\}}\ \textsc{Frame}$$

By applying the disjunction rule, this can be reduced to the case of singleton frames $\{c\}$, transforming the derivation as follows:

$$\dfrac{\text{for all } c \in K, \quad \dfrac{\dfrac{\vdots}{\{P'\}\ \mathbb{C}\ \{Q'\}}\ (\ddagger)}{\{\{c\} \bullet P'\}\ \mathbb{C}\ \{\{c\} \bullet Q'\}}\ \textsc{Frame}}{\{K \bullet P'\}\ \mathbb{C}\ \{K \bullet Q'\}}\ \textsc{Disj}$$

Now consider cases for (‡), the last rule applied before Frame.

If the rule is Cons then, since $P \subseteq P'$ implies that $\{c\} \bullet P \subseteq \{c\} \bullet P$, the application of the Frame can be moved earlier in the derivation, transforming it as follows:

$$\dfrac{\begin{array}{ll}\{c\} \bullet P' \subseteq \{c\} \bullet P'' & \dfrac{\{P''\}\ \mathbb{C}\ \{Q''\}}{\{\{c\} \bullet P''\}\ \mathbb{C}\ \{\{c\} \bullet Q''\}}\ \textsc{Frame} \\ \{c\} \bullet Q' \subseteq \{c\} \bullet Q'' & \end{array}}{\{\{c\} \bullet P'\}\ \mathbb{C}\ \{\{c\} \bullet Q'\}}\ \textsc{Cons}$$

The application of the frame rule can then by removed by the inductive hypothesis.

If the rule is Disj then, since $\bullet$ right-distributes over $\vee$, the derivation can be transformed as follows:

$$\dfrac{\text{for all } i \in I, \quad \dfrac{\dfrac{\vdots}{\{P_i\}\ \mathbb{C}\ \{Q_i\}}}{\{\{c\} \bullet P_i\}\ \mathbb{C}\ \{\{c\} \bullet Q_i\}}\ \textsc{Frame}}{\{\{c\} \bullet \bigvee_{i\in I} P_i\}\ \mathbb{C}\ \{\{c\} \bullet \bigvee_{i\in I} Q_i\}}\ \textsc{Disj}$$

If the rule is Local then it is possible that the frame c includes a program variable with the same name as one that is scoped 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 scope and store components, that is, for some $\rho \in \mathsf{Scope}$ and $c_{\mathbb{A}} \in \mathsf{C}_{\mathbb{A}}$, $c = (\rho, c_{\mathbb{A}})$ and so $\{c\} = (\{\rho\} \times \mathbf{I}_{\mathbb{A}}) \circ \{(\varnothing, c_{\mathbb{A}})\}$. Hence the derivation can be transformed as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\{(\boldsymbol{x} \Rrightarrow - \times \mathbf{I}_{\mathbb{A}}) \bullet P'\}\ \mathbb{C}'\ \{(\boldsymbol{x} \Rrightarrow - \times \mathbf{I}_{\mathbb{A}}) \bullet Q'\}}\ \textsc{Frame}}{\{(\boldsymbol{x} \Rrightarrow - \times \{c_{\mathbb{A}}\}) \bullet P'\}\ \mathbb{C}'\ \{(\boldsymbol{x} \Rrightarrow - \times \{c_{\mathbb{A}}\}) \bullet Q'\}}\ \textsc{Local}}{\{\{(\varnothing, c_{\mathbb{A}})\} \bullet P'\}\ \texttt{local}\ \boldsymbol{x}\ \texttt{in}\ \mathbb{C}'\ \{\{(\varnothing, c_{\mathbb{A}})\} \bullet Q'\}}\ \textsc{Frame}}{\{\{c\} \bullet P'\}\ \texttt{local}\ \boldsymbol{x}\ \texttt{in}\ \mathbb{C}'\ \{\{c\} \bullet Q'\}}\ \textsc{Frame}$$

The side condition for the Local rule, that $(\{(\varnothing, c_{\mathbb{A}})\} \bullet P') \wedge \mathit{vsafe}(\boldsymbol{x}) \equiv \emptyset$, follows from the original side-condition that $P' \wedge \mathit{vsafe}(\boldsymbol{x}) \equiv \emptyset$. The applications of the Frame rule are now either of the form of (6.3) or can be removed by the inductive hypothesis.

If the rule is PCall then it is again necessary to split the framing context into its components, that is, some $\rho \in \mathsf{Scope}$ and $c_{\mathbb{A}} \in \mathsf{C}_{\mathbb{A}}$ with $c = (\rho, c_{\mathbb{A}})$. The PCall rule uses some $\mathtt{f} : \mathsf{P} \rightarrowtail \mathsf{Q} \in \Gamma$. By definition, $\mathtt{f} : \{c_{\mathbb{A}}\} \bullet \mathsf{P} \rightarrowtail \{c_{\mathbb{A}}\} \bullet \mathsf{Q} \in F(\Gamma)$. Hence, the derivation can be transformed as follows:

$$\dfrac{\dfrac{\{(\overrightarrow{\boldsymbol{r}} \Rrightarrow \overrightarrow{w} * \rho')\} \times \mathsf{D}_{\mathbb{A}} \subseteq \mathit{vsafe}(\overrightarrow{E})}{F(\Gamma) \vdash_{\mathbb{A}} \left\{(\overrightarrow{\boldsymbol{r}} \Rrightarrow \overrightarrow{v} * \rho') \times \left(\{(c_{\mathbb{A}})\} \bullet \mathsf{P}\left(\mathcal{E}\left[\!\left[\overrightarrow{E}\right]\!\right](\overrightarrow{\boldsymbol{r}} \Rrightarrow \overrightarrow{v} * \rho')\right)\right)\right\}\ \mathtt{call}\ \overrightarrow{\boldsymbol{r}} := \mathtt{f}(\overrightarrow{E})}\ \textsc{PCall}}{F(\Gamma) \vdash_{\mathbb{A}} \left\{\{(\rho, c_{\mathbb{A}})\} \bullet \left((\overrightarrow{\boldsymbol{r}} \Rrightarrow \overrightarrow{v} * \rho') \times \mathsf{P}\left(\mathcal{E}\left[\!\left[\overrightarrow{E}\right]\!\right](\overrightarrow{\boldsymbol{r}} \Rrightarrow \overrightarrow{v} * \rho')\right)\right)\right\}\ \mathtt{call}\ \overrightarrow{\boldsymbol{r}} := \mathtt{f}(\overrightarrow{E})}\ \textsc{Frame}$$
$$\{\exists \overrightarrow{w}.\, \{(\overrightarrow{\boldsymbol{r}} \Rrightarrow \overrightarrow{w} * \rho')\} \times (\{(c_{\mathbb{A}})\} \bullet \mathsf{Q}(\overrightarrow{w}))\}$$
$$\{\{(\rho, c_{\mathbb{A}})\} \bullet (\exists \overrightarrow{w}.\, \{(\overrightarrow{\boldsymbol{r}} \Rrightarrow \overrightarrow{w} * \rho')\} \times \mathsf{Q}(\overrightarrow{w}))\}$$

The application of the frame rule is now of the form of (6.3), with the frame being $\{\rho\} \times \mathbf{I}_{\mathbb{A}}$.

The remaining cases for the penultimate rule are straightforward.

Consider the case when PDEF is the last rule applied:

$$\dfrac{\dfrac{\text{for all } (\mathtt{f}_i : \mathsf{P} \rightarrowtail \mathsf{Q}) \in \Gamma, \quad \Gamma', \Gamma \vdash_{\mathbb{A}} \begin{array}{c}\vdots \\ \{\exists \overrightarrow{v}.\, \overrightarrow{\boldsymbol{x}_i} \Rightarrow \overrightarrow{v} * \overrightarrow{\boldsymbol{r}_i} \Rightarrow - \times \mathsf{P}(\overrightarrow{v})\} \\ \mathbb{C}_i \\ \{\exists \overrightarrow{w}.\, \overrightarrow{\boldsymbol{x}_i} \Rightarrow - * \overrightarrow{\boldsymbol{r}_i} \Rightarrow \overrightarrow{w} \times \mathsf{Q}(\overrightarrow{w})\}\end{array}}{(\star)}}{\Gamma' \vdash_{\mathbb{A}} \{P\}\ \mathtt{procs}\ \overrightarrow{r_1} := \mathtt{f}_1(\overrightarrow{x_1})\ \{\mathbb{C}_1\}, \ldots, \overrightarrow{r_k} := \mathtt{f}_k(\overrightarrow{x_k})\ \{\mathbb{C}_k\}\ \mathtt{in}\ \mathbb{C}'\ \{Q\}}\ \textsc{PDef}$$

$$(\star) \quad \dfrac{\vdots}{\Gamma', \Gamma \vdash_{\mathbb{A}} \{P\}\ \mathbb{C}'\ \{Q\}}$$

The derivations for the function bodies can be extended by applying the frame rule to give:

$$\dfrac{\text{for all } K \in \mathcal{P}(\mathsf{C}_{\mathbb{A}}),\ (\mathtt{f}_i : \mathsf{P} \rightarrowtail \mathsf{Q}) \in \Gamma, \quad \Gamma', \Gamma \vdash_{\mathbb{A}} \begin{array}{c}\vdots \\ \{\exists \overrightarrow{v}.\, \overrightarrow{\boldsymbol{x}_i} \Rightarrow \overrightarrow{v} * \overrightarrow{\boldsymbol{r}_i} \Rightarrow - \times \mathsf{P}(\overrightarrow{v})\} \\ \mathbb{C}_i \\ \{\exists \overrightarrow{w}.\, \overrightarrow{\boldsymbol{x}_i} \Rightarrow - * \overrightarrow{\boldsymbol{r}_i} \Rightarrow \overrightarrow{w} \times \mathsf{Q}(\overrightarrow{w})\}\end{array}}{\Gamma', \Gamma \vdash_{\mathbb{A}} \begin{array}{c}\vdots \\ \{\exists \overrightarrow{v}.\, \overrightarrow{\boldsymbol{x}_i} \Rightarrow \overrightarrow{v} * \overrightarrow{\boldsymbol{r}_i} \Rightarrow - \times (K \bullet \mathsf{P}(\overrightarrow{v}))\} \\ \mathbb{C}_i \\ \{\exists \overrightarrow{w}.\, \overrightarrow{\boldsymbol{x}_i} \Rightarrow - * \overrightarrow{\boldsymbol{r}_i} \Rightarrow \overrightarrow{w} \times (K \bullet \mathsf{Q}(\overrightarrow{w}))\}\end{array}}\ \textsc{Frame}$$

These derivations and the derivation of the premise $\Gamma', \Gamma \vdash_{\mathbb{A}} \{P\}\ \mathbb{C}'\ \{Q\}$ can be transformed by the inductive hypothesis so that they only use the frame rule in the required manner and use the procedure environment $F(\Gamma', \Gamma) = F(\Gamma'), F(\Gamma)$. These derivations can then be recombined to give the required derivation as follows:

$$\dfrac{\dfrac{\text{for all } (\mathtt{f}_i : \mathsf{P} \rightarrowtail \mathsf{Q}) \in F(\Gamma), \quad F(\Gamma', \Gamma) \vdash_{\mathbb{A}} \begin{array}{c}\vdots \\ \{\exists \overrightarrow{v}.\, \overrightarrow{\boldsymbol{x}_i} \Rightarrow \overrightarrow{v} * \overrightarrow{\boldsymbol{r}_i} \Rightarrow - \times \mathsf{P}(\overrightarrow{v})\} \\ \mathbb{C}_i \\ \{\exists \overrightarrow{w}.\, \overrightarrow{\boldsymbol{x}_i} \Rightarrow - * \overrightarrow{\boldsymbol{r}_i} \Rightarrow \overrightarrow{w} \times \mathsf{Q}(\overrightarrow{w})\}\end{array}}{(\star)}}{\begin{array}{c}\{P\} \\ F(\Gamma') \vdash_{\mathbb{A}} \mathtt{procs}\ \overrightarrow{r_1} := \mathtt{f}_1(\overrightarrow{x_1})\ \{\mathbb{C}_1\}, \ldots, \overrightarrow{r_k} := \mathtt{f}_k(\overrightarrow{x_k})\ \{\mathbb{C}_k\}\ \mathtt{in}\ \mathbb{C}' \\ \{Q\}\end{array}}\ \textsc{PDef}$$

$$(\star) \quad \dfrac{\vdots}{F(\Gamma', \Gamma) \vdash_{\mathbb{A}} \{P\}\ \mathbb{C}'\ \{Q\}}$$

The two further conditions on the PDEF rule, not included above, hold for the transformed derivation because F preserves the names of the functions in procedure specifications. □

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 $\vdash_{\mathbb{A}} \{P\}\ \mathbb{C}\ \{Q\}$ there is a derivation of $\vdash_{\mathbb{B}} \{[\![P]\!]_\tau\}\ [\![\mathbb{C}]\!]_\tau\ \{[\![Q]\!]_\tau\}$. First, transform the high-level derivation using Lemma 99 into a frame-free derivation. Now transform this derivation to the required low-level derivation by replacing each subderivation of the form

$$\dfrac{\dfrac{}{\Gamma \vdash_{\mathbb{A}} \{P'\}\ \varphi\ \{Q'\}}\ \textsc{Axiom}}{\Gamma \vdash_{\mathbb{A}} \{K \bullet P'\}\ \varphi\ \{K \bullet Q'\}}\ \textsc{Frame}$$

with the derivation

$$\dfrac{\text{for all } (\rho, c_{\mathbb{A}}) \in K, \quad \dfrac{\dfrac{(\star)}{\vdash_{\mathbb{B}} \{[\![\{(\varnothing, c_{\mathbb{A}}) \bullet P'\}]\!]\}\ [\![\varphi]\!]\ \{[\![\{(\varnothing, c_{\mathbb{A}}) \bullet Q'\}]\!]\}}\ \textsc{Frame}}{\vdash_{\mathbb{B}} \{[\![\{(\rho, c_{\mathbb{A}}) \bullet P'\}]\!]\}\ [\![\varphi]\!]\ \{[\![\{(\rho, c_{\mathbb{A}}) \bullet Q'\}]\!]\}}\ \textsc{Frame}}{\vdash_{\mathbb{B}} \{[\![K \bullet P']\!]\}\ [\![\varphi]\!]\ \{[\![K \bullet Q']\!]\}}\ \textsc{Disj}$$

where $(\star)$ stands for the framed derivation provided by the locality-breaking translation, and replacing all other rules with their low-level equivalents.

This completes the proof of Theorem 100.

6.4.2 Module Translation: $\tau_3 : \mathbb{L} \to \mathbb{H}$

I return to the example of an implementation of the list-store module with singly-linked lists in the heap to illustrate a locality-breaking translation.

Definition 6.25 ($\tau_3 : \mathbb{L} \to \mathbb{H}$)**.** The module translation $\tau_3 : \mathbb{L} \to \mathbb{H}$ is constructed as follows:

- the abstraction relation $\alpha_{\tau_3} \subseteq \mathsf{Heap} \times \mathsf{LStore}$ is defined by

$$h\ \alpha_{\tau_3}\ ls \iff h \in (\!|ls|\!)$$

where $(\!|(\cdot)|\!) : \mathsf{LStore} \to \mathcal{P}(\mathsf{Heap})$ is defined inductively as follows:

$$\begin{aligned}
(\!|\mathrm{emp}|\!) &\stackrel{\text{def}}{=} \mathrm{emp} \\
(\!|a \mapsto l * ls|\!) &\stackrel{\text{def}}{=} \mathrm{false} \\
(\!|a \mapsto [l] * ls|\!) &\stackrel{\text{def}}{=} \exists x.\, a \mapsto x * \langle\!\langle l \rangle\!\rangle^{(x, \mathbf{nil})} * (\!|ls|\!)
\end{aligned}$$

where

$$\begin{aligned}
\langle\!\langle \varnothing \rangle\!\rangle^{(x,y)} &\stackrel{\text{def}}{=} (x = y) \wedge \mathrm{emp} \\
\langle\!\langle v \rangle\!\rangle^{(x,y)} &\stackrel{\text{def}}{=} x \mapsto v, y \\
\langle\!\langle l_1 \cdot l_2 \rangle\!\rangle^{(x,y)} &\stackrel{\text{def}}{=} \exists z.\, \langle\!\langle l_1 \rangle\!\rangle^{(x,z)} * \langle\!\langle l_2 \rangle\!\rangle^{(z,y)}; \text{ and}
\end{aligned}$$

$E.\texttt{value} \stackrel{\text{def}}{=} E$

$E.\texttt{next} \stackrel{\text{def}}{=} E + 1$

$x := \texttt{newNode}() \stackrel{\text{def}}{=} x := \texttt{alloc}(2)$

$x := \texttt{newRoot}() \stackrel{\text{def}}{=} x := \texttt{alloc}(1)$

$\texttt{disposeNode}(x) \stackrel{\text{def}}{=} \texttt{dispose}(x, 2)$

$\texttt{disposeRoot}(x) \stackrel{\text{def}}{=} \texttt{dispose}(x, 1)$

```
v := getHead(i) {
  local x in
    x := [i];
    if x = nil then
      v := x
    else
      v := [x.value]
}

v := getTail(i) {
  local x, y in
    x := [i];
    if x = nil then
      v := x
    else
      y := [x.next];
      while y ≠ nil do
        x := y;
        y := [x.next]   ;
      v := [x.value]
}

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 = nil then
      v := x
    else
      v := [x.value]
}
```

```
v := getPrev(i, w) {
  local x, y in
    x := [i];
    v := [x.value];
    if v = w then
      v := nil
    else
      while v ≠ w do
        y := x;
        x := [y.next];
        v := [x.value]   ;
      v := [y.value]
}

v := pop(i) {
  local x, y in
    x := [i];
    if x = nil then
      v := x
    else
      y := [x.next];
      [i] := y;
      v := [x.value];
      disposeNode(x)
}

push(i, v) {
  local X, y in
    x := newNode();
    [x.value] := v;
    y := [i];
    [x.next] := y;
    [i] := x
}
```

Figure 6.6: Linked-list-based list store implementation

```
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)
}

i := newList() {
  i := newRoot();
  [i] := nil
}
```

```
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
}

deleteList(i) {
  local x, y in
    x := [i];
    while x ≠ nil do
      y := x;
      x := [y.next];
      disposeNode(y)   ;
    disposeRoot(i)
}
```

Figure 6.7: Linked-list-based list store implementation

- the substitutive implementation function is given by replacing each list-module command with a call tot he correspondingly-named procedure given in Figures 6.6 and 6.7.

Note that the abstraction relation is not surjective, since incomplete lists do not have heap representations. The intuition behind this approach is that incomplete lists are purely a useful means to the ultimate end of reasoning about complete lists. Typically, clients of the list module will work with complete lists — only complete lists may be created or deleted, for a start — and so restricting their specifications to only describe stores of complete lists should be no significant problem. Of course, it is perfectly acceptable to use assertions and specifications that refer to incomplete lists within client proofs; the transformation of this proof to a low-level proof will complete all of these lists by making use of Lemma 99.

The fact that incomplete lists do not have representations can be seen as an advantage from the point of view of establishing that τ_3 is a locality-breaking translation. This is since it is only necessary to prove that the framed axioms

$$
\begin{array}{l}
\left\{\exists p.\, \boldsymbol{i} \Rrightarrow a * \boldsymbol{w} \Rrightarrow w * \boldsymbol{v} \Rrightarrow - * \boldsymbol{x} \Rrightarrow - \times a \mapsto p * \langle\!\langle l_1 \cdot w \rangle\!\rangle^{(p,y)}\right\} \\
\boldsymbol{x} := [\boldsymbol{i}]\,; \\
\left\{\exists p.\, \boldsymbol{i} \Rrightarrow a * \boldsymbol{w} \Rrightarrow w * \boldsymbol{v} \Rrightarrow - * \boldsymbol{x} \Rrightarrow p \times a \mapsto p * \langle\!\langle l_1 \cdot w \rangle\!\rangle^{(p,y)}\right\} \\
\left\{\begin{array}{l} \exists p.\, \boldsymbol{w} \Rrightarrow w * \boldsymbol{v} \Rrightarrow - * \boldsymbol{x} \Rrightarrow p \times \\ \left((l_1 = \varnothing \wedge p \mapsto w, y) \vee (\exists v, l_1', q.\, l_1 = v \cdot l_1'' \wedge p \mapsto w, q * \langle\!\langle l_1'' \cdot w \rangle\!\rangle^{(q,y)})\right) \end{array}\right\} \\
\boldsymbol{v} := [\boldsymbol{x}.\mathtt{value}]\,; \\
\left\{\begin{array}{l} \exists v, x, l_1', l_1'', q.\, \boldsymbol{w} \Rrightarrow w * \boldsymbol{v} \Rrightarrow v * \boldsymbol{x} \Rrightarrow x \times \\ l_1 \cdot w = l_1' \cdot v \cdot l_1'' \wedge \langle\!\langle l_1' \rangle\!\rangle^{(p,x)} * x \mapsto v, q * \langle\!\langle l_1'' \rangle\!\rangle^{(q,y)} \end{array}\right\} \\
\mathtt{while}\ \boldsymbol{v} \neq \boldsymbol{w}\ \mathtt{do} \\
\quad \left\{\begin{array}{l} \exists v, v', x, x', l_1', l_1'', q.\, \boldsymbol{w} \Rrightarrow w * \boldsymbol{v} \Rrightarrow v * \boldsymbol{x} \Rrightarrow x \times \\ l_1 \cdot w = l_1' \cdot v \cdot v' \cdot l_1'' \wedge \langle\!\langle l_1' \rangle\!\rangle^{(p,x)} * x \mapsto v, x' * x' \mapsto v', q * \langle\!\langle l_1'' \rangle\!\rangle^{(q,y)} \end{array}\right\} \\
\quad \boldsymbol{x} := [\boldsymbol{x}.\mathtt{next}]\,; \\
\quad \boldsymbol{v} := [\boldsymbol{x}.\mathtt{value}] \\
\quad \left\{\begin{array}{l} \exists v, v', x, x', l_1', l_1'', q.\, \boldsymbol{w} \Rrightarrow w * \boldsymbol{v} \Rrightarrow v' * \boldsymbol{x} \Rrightarrow x' \times \\ l_1 \cdot w = l_1' \cdot v \cdot v' \cdot l_1'' \wedge \langle\!\langle l_1' \rangle\!\rangle^{(p,x)} * x \mapsto v, x' * x' \mapsto v', q * \langle\!\langle l_1'' \rangle\!\rangle^{(q,y)} \end{array}\right\} \\
\quad \left\{\begin{array}{l} \exists v, x, l_1', l_1'', q.\, \boldsymbol{w} \Rrightarrow w * \boldsymbol{v} \Rrightarrow v * \boldsymbol{x} \Rrightarrow x \times \\ l_1 \cdot w = l_1' \cdot v \cdot l_1'' \wedge \langle\!\langle l_1' \rangle\!\rangle^{(p,x)} * x \mapsto v, q * \langle\!\langle l_1'' \rangle\!\rangle^{(q,y)} \end{array}\right\}\ ; \\
\left\{\exists x.\, \boldsymbol{w} \Rrightarrow w * \boldsymbol{v} \Rrightarrow w * \boldsymbol{x} \Rrightarrow x \times \langle\!\langle l_1 \rangle\!\rangle^{(p,x)} * x \mapsto w, y\right\} \\
\boldsymbol{x} := [\boldsymbol{x}.\mathtt{next}] \\
\left\{\boldsymbol{w} \Rrightarrow w * \boldsymbol{v} \Rrightarrow w * \boldsymbol{x} \Rrightarrow y \times \langle\!\langle l_1 \cdot w \rangle\!\rangle^{(p,y)}\right\} \\
\left\{\exists p.\, \boldsymbol{i} \Rrightarrow a * \boldsymbol{w} \Rrightarrow w * \boldsymbol{v} \Rrightarrow w * \boldsymbol{x} \Rrightarrow y \times a \mapsto p * \langle\!\langle l_1 \cdot w \rangle\!\rangle^{(p,y)}\right\}
\end{array}
$$

Figure 6.8: Proof outline for `getNext` implementation (common part)

hold under the translation for frames that complete all of the lists in the precondition — in all other cases, the precondition is false, and so the triple holds trivially.

Theorem 101 (Soundness of τ_3)**.** *The module translation τ_3 is a locality-breaking translation.*

Again, I omit the full details of this proof, due to Wheelhouse, which can be found in [DYGW10b], but give a particular case: `getNext`. Recall the axiom for `getNext`:

$$
\begin{array}{l}
\mathrm{Ax}\, [\![\boldsymbol{x} := E_1.\mathtt{getNext}(E_2)]\!]_{\mathbb{L}} \stackrel{\text{def}}{=} \\
\quad \left\{ \left(\begin{array}{l} (\boldsymbol{x} \Rrightarrow v * \rho \times a \Mapsto w \cdot u)\,, \\ (\boldsymbol{x} \Rrightarrow u * \rho \times a \Mapsto w \cdot u) \end{array} \right) \;\middle|\; \begin{array}{r} a = \mathcal{E}\,[\![E_1]\!]\,(\boldsymbol{x} \Rrightarrow v * \rho) \text{ and} \\ w = \mathcal{E}\,[\![E_2]\!]\,(\boldsymbol{x} \Rrightarrow v * \rho) \end{array} \right\} \cup \\
\quad \left\{ \left(\begin{array}{l} (\boldsymbol{x} \Rrightarrow v * \rho \times a \Mapsto [l \cdot w])\,, \\ (\boldsymbol{x} \Rrightarrow \boldsymbol{nil} * \rho \times a \Mapsto [l \cdot w]) \end{array} \right) \;\middle|\; \begin{array}{r} a = \mathcal{E}\,[\![E_1]\!]\,(\boldsymbol{x} \Rrightarrow v * \rho) \text{ and} \\ w = \mathcal{E}\,[\![E_2]\!]\,(\boldsymbol{x} \Rrightarrow v * \rho) \end{array} \right\}.
\end{array}
$$

Fix arbitrary $a \in \mathsf{Addr}$, $w, u \in \mathsf{Val}$, $l \in \mathsf{Val}^*$ and $c \in \mathsf{C}_{\mathsf{LStore}}$. It is sufficient to

$$
\begin{array}{l}
\boldsymbol{v} := \texttt{getNext}(\boldsymbol{i}, w)\ \{ \\
\quad \{[\![\boldsymbol{i} \Rrightarrow a * w \Rrightarrow w * \boldsymbol{v} \Rrightarrow - \times a \Mapsto [l_1 \cdot w \cdot u \cdot l_2] * ls]\!]\} \\
\quad \left\{ \begin{array}{l} \exists y, z, p.\ \boldsymbol{i} \Rrightarrow a * w \Rrightarrow w * \boldsymbol{v} \Rrightarrow - \times \\ \qquad a \mapsto p * \langle\!\langle l_1 \cdot w \rangle\!\rangle^{(p,y)} * y \mapsto u, z * \langle\!\langle l_2 \rangle\!\rangle^{(z,\boldsymbol{nil})} * (\!| ls |\!) \end{array} \right\} \\
\quad \texttt{local } \boldsymbol{x} \texttt{ in} \\
\qquad \{\exists p.\ \boldsymbol{i} \Rrightarrow a * w \Rrightarrow w * \boldsymbol{v} \Rrightarrow - * \boldsymbol{x} \Rrightarrow - \times a \mapsto p * \langle\!\langle l_1 \cdot w \rangle\!\rangle^{(p,y)} * y \mapsto u\} \\
\qquad\qquad\qquad\qquad \text{(see Figure 6.8)} \\
\qquad \{\exists p.\ \boldsymbol{i} \Rrightarrow a * w \Rrightarrow w * \boldsymbol{v} \Rrightarrow w * \boldsymbol{x} \Rrightarrow y \times a \mapsto p * \langle\!\langle l_1 \cdot w \rangle\!\rangle^{(p,y)} * y \mapsto u\} \\
\qquad \{\boldsymbol{v} \Rrightarrow w * \boldsymbol{x} \Rrightarrow y \times y \mapsto u\} \\
\qquad \texttt{if } \boldsymbol{x} = \boldsymbol{nil} \texttt{ then} \\
\qquad\quad \boldsymbol{v} := \boldsymbol{x} \\
\qquad \texttt{else} \\
\qquad\quad \boldsymbol{v} := [\boldsymbol{x}.\texttt{value}] \\
\qquad \{\boldsymbol{v} \Rrightarrow u * \boldsymbol{x} \Rrightarrow y \times y \mapsto u\} \\
\quad \left\{ \begin{array}{l} \exists y, z, p.\ \boldsymbol{i} \Rrightarrow a * w \Rrightarrow w * \boldsymbol{v} \Rrightarrow u \times \\ \qquad a \mapsto p * \langle\!\langle l_1 \cdot w \rangle\!\rangle^{(p,y)} * y \mapsto u, z * \langle\!\langle l_2 \rangle\!\rangle^{(z,\boldsymbol{nil})} * (\!| ls |\!) \end{array} \right\} \\
\quad \{[\![\boldsymbol{i} \Rrightarrow a * w \Rrightarrow w * \boldsymbol{v} \Rrightarrow u \times a \Mapsto [l_1 \cdot w \cdot u \cdot l_2] * ls]\!]\} \\
\}
\end{array}
$$

Figure 6.9: Proof outline for `getNext` implementation (success case)

$$
\begin{array}{l}
\boldsymbol{v} := \texttt{getNext}(\boldsymbol{i}, w)\ \{ \\
\quad \{[\![\boldsymbol{i} \Rrightarrow a * w \Rrightarrow w * \boldsymbol{v} \Rrightarrow - \times a \Mapsto [l \cdot w] * ls]\!]\} \\
\quad \left\{ \begin{array}{l} \exists p.\ \boldsymbol{i} \Rrightarrow a * w \Rrightarrow w * \boldsymbol{v} \Rrightarrow - \times \\ \qquad a \mapsto p * \langle\!\langle l \cdot w \rangle\!\rangle^{(p,\boldsymbol{nil})} * (\!| ls |\!) \end{array} \right\} \\
\quad \texttt{local } \boldsymbol{x} \texttt{ in} \\
\qquad \{\exists p.\ \boldsymbol{i} \Rrightarrow a * w \Rrightarrow w * \boldsymbol{v} \Rrightarrow - * \boldsymbol{x} \Rrightarrow - \times a \mapsto p * \langle\!\langle l \cdot w \rangle\!\rangle^{(p,\boldsymbol{nil})}\} \\
\qquad\qquad\qquad\qquad \text{(see Figure 6.8)} \\
\qquad \{\exists p.\ \boldsymbol{i} \Rrightarrow a * w \Rrightarrow w * \boldsymbol{v} \Rrightarrow w * \boldsymbol{x} \Rrightarrow y \times a \mapsto p * \langle\!\langle l \cdot w \rangle\!\rangle^{(p,\boldsymbol{nil})}\} \\
\qquad \{\boldsymbol{v} \Rrightarrow w * \boldsymbol{x} \Rrightarrow \boldsymbol{nil} \times \text{emp}\} \\
\qquad \texttt{if } \boldsymbol{x} = \boldsymbol{nil} \texttt{ then} \\
\qquad\quad \boldsymbol{v} := \boldsymbol{x} \\
\qquad \texttt{else} \\
\qquad\quad \boldsymbol{v} := [\boldsymbol{x}.\texttt{value}] \\
\qquad \{\boldsymbol{v} \Rrightarrow \boldsymbol{nil} * \boldsymbol{x} \Rrightarrow \boldsymbol{nil} \times \text{emp}\} \\
\quad \left\{ \begin{array}{l} \exists p.\ \boldsymbol{i} \Rrightarrow a * w \Rrightarrow w * \boldsymbol{v} \Rrightarrow u \times \\ \qquad a \mapsto p * \langle\!\langle l \cdot w \rangle\!\rangle^{(p,\boldsymbol{nil})} * (\!| ls |\!) \end{array} \right\} \\
\quad \{[\![\boldsymbol{i} \Rrightarrow a * w \Rrightarrow w * \boldsymbol{v} \Rrightarrow u \times a \Mapsto [l \cdot w] * ls]\!]\} \\
\}
\end{array}
$$

Figure 6.10: Proof outline for `getNext` implementation (failure case)

establish that the procedure body of `getNext` meets the following specifications:

$$\begin{array}{l}\{[\![\boldsymbol{i} \Rightarrow a * \boldsymbol{w} \Rightarrow w * \boldsymbol{v} \Rightarrow \_ \times (\{c\} \bullet a \Mapsto w \cdot u)]\!]\} \\ \{[\![\boldsymbol{i} \Rightarrow a * \boldsymbol{w} \Rightarrow w * \boldsymbol{v} \Rightarrow u \times (\{c\} \bullet a \Mapsto w \cdot u)]\!]\}\end{array} \tag{6.4}$$

$$\begin{array}{l}\{[\![\boldsymbol{i} \Rightarrow a * \boldsymbol{w} \Rightarrow w * \boldsymbol{v} \Rightarrow \_ \times (\{c\} \bullet a \Mapsto [l \cdot w])]\!]\} \\ \{[\![\boldsymbol{i} \Rightarrow a * \boldsymbol{w} \Rightarrow w * \boldsymbol{v} \Rightarrow \boldsymbol{nil} \times (\{c\} \bullet a \Mapsto [l \cdot w])]\!]\}\end{array}. \tag{6.5}$$

Either specification (6.4) holds trivially, since the precondition is equivalent to false, or $c = a \Mapsto [l_1 \cdot \_ \cdot l_2] * ls$ for some $l_1, l_2 \in \mathsf{Lst}$ with w and u not in either l_1 or l_2, and some $ls \in \mathsf{LStore}$. A proof outline for this case is given in Figure 6.9. Similarly, either specification (6.5) holds trivially or $c = ls$ for some $ls \in \mathsf{LStore}$. A proof outline for this case is given in Figure 6.10.

6.5 Commentary

6.5.1 On the Conjunction Rule

Notably absent from the Hoare logic rules used here has been the conjunction rule:

$$\frac{I \neq \emptyset \quad \text{for all } i \in I,\ \Gamma \vdash \{P_i\}\ \mathbb{C}\ \{Q_i\}}{\Gamma \vdash \{\bigwedge_{i \in I} P_i\}\ \mathbb{C}\ \{\bigwedge_{i \in I} Q_i\}}\ \textsc{Conj}.$$

This raises two important questions: is the omission justifiable, and what are the consequences for the module-translation theory of including Conj?

Justification for Omission

If we take the axiomatic semantics to be the normative specification of the programming language's semantics, then including Conj potentially changes the semantics by making it possible to derive more triples. However, typically this is not the case: Conj is *admissible* in the Hoare logic.

If the conjunction rule were not admissible, then there must be some Γ, $\mathbb{C}$, P, Q_1, Q_2 such that $\Gamma \vdash \{P\}\ \mathbb{C}\ \{Q_1\}$ and $\Gamma \vdash \{P\}\ \mathbb{C}\ \{Q_2\}$ are derivable but $\Gamma \vdash \{Q\}\ \mathbb{C}\ \{Q_1 \wedge Q_2\}$ is not derivable (without Conj). That is, for some command $\mathbb{C}$ and predicate P, Conj can be used to derive a stronger postcondition for P under $\mathbb{C}$.

Of course, it is perfectly possible to define axioms to satisfy this condition. Consider, for example, a command on the double-heap model ($\mathcal{A}_{\mathbb{H}} \times \mathcal{A}_{\mathbb{H}}$) that

allocates a single cell in either of the two heaps, specified as follows:

$$\mathrm{Ax}\,[\![\boldsymbol{x} := \texttt{allocEither}()]\!] \stackrel{\text{def}}{=} \left\{ \begin{array}{c} ((\boldsymbol{x} \Rightarrow - \times \mathrm{emp} \times \mathrm{emp}), (\exists a.\, \boldsymbol{x} \Rightarrow a \times a \mapsto - \times \mathrm{emp})), \\ ((\boldsymbol{x} \Rightarrow - \times \mathrm{emp} \times \mathrm{emp}), (\exists a.\, \boldsymbol{x} \Rightarrow a \times \mathrm{emp} \times a \mapsto -)) \end{array} \right\}$$

Note that the choice of heap in which the cell is allocated is not at the discretion of the implementation, but rather at the discretion of the prover — the 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 seems paradoxical, since the program must somehow correctly guess the prover's choice.

Remark. One way of resolving this is that, from the program's perspective, the two cases are actually *the same* and the distinction is only a logical abstraction. Consider, for example, the representation of a double heap in a single heap, used in §6.3.3.

The conjunction rule is not compatible with angelic nondeterminism — if the prover can make two different choices then the program satisfies their intersection, as illustrated by the following derivation:

$$\dfrac{\dfrac{}{\begin{array}{c}\{\boldsymbol{x} \Rightarrow - \times \mathrm{emp} \times \mathrm{emp}\} \\ \boldsymbol{x} := \texttt{allocEither}() \\ \{\exists a.\, \boldsymbol{x} \Rightarrow a \times a \mapsto - \times \mathrm{emp}\}\end{array}}\,\textsc{Axiom} \quad \dfrac{}{\begin{array}{c}\{\boldsymbol{x} \Rightarrow - \times \mathrm{emp} \times \mathrm{emp}\} \\ \boldsymbol{x} := \texttt{allocEither}() \\ \{\exists a.\, \boldsymbol{x} \Rightarrow a \times \mathrm{emp} \times a \mapsto -\}\end{array}}\,\textsc{Axiom}}{\{\boldsymbol{x} \Rightarrow - \times \mathrm{emp} \times \mathrm{emp}\}\ \boldsymbol{x} := \texttt{allocEither}()\ \{\mathrm{false}\}}\,\textsc{Conj}$$

Using Conj, we are able to conclude that `allocEither` must diverge. Without Conj, we cannot draw the same conclusion.

Note, however, that none of the commands from the modules that I have introduced exhibits angelic nondeterminism. Together, the following two conditions on command $\varphi \in \mathsf{Cmd}$ are sufficient to establish that it does not exhibit angelic nondeterminism:

- for all $(P, Q), (P', Q') \in \mathrm{Ax}\,[\![\varphi]\!]$ with $(P, Q) \neq (P', Q')$, $P \wedge P' \equiv \emptyset$; and
- the predicate $\bigvee \{P \mid (P, Q) \in \mathrm{Ax}\,[\![\varphi]\!]\}$ is precise[6].

These conditions are not difficult to check and hold for all of the modules I have so far described. They imply that at most one axiom describes the behaviour

[6]Predicate P is *precise* if, for every $s \in \mathsf{State}$ there is at most one $c \in \mathsf{C}_{\mathsf{State}}$ and $s' \in P$ such that $s = c \bullet s'$.

of the command from any given state. Hence, the conjunction rule cannot be used to derive a stronger post condition for any of the basic commands.

None of the program constructions introduces angelic nondeterminism[7] and so Conj is admissible for all of the modules considered here. Thus, since nothing is gained by the inclusion of the conjunction rule, its omission is justified.

Consequences of Inclusion

Assume that the axiomatic semantics is extended with Conj — what additional conditions are required for locality-preserving and locality-breaking translations to be sound?

For locality-preserving translations, a case can be added to the proof of Proposition 91 that deals with the Conj rule in the same fashion as the Disj rule, provided that $(\!|(\cdot)|\!)^{(\cdot)}$ distributes over conjunctions. Together, the following two conditions are sufficient to establish this:

- for all $\chi, \chi' \in \mathsf{D}$ with $\chi \neq \chi'$, and all $I \in \mathcal{I}$, $\langle\!\langle \chi \rangle\!\rangle^I \wedge \langle\!\langle \chi' \rangle\!\rangle^I \equiv \emptyset$; and
- for all $out \in \mathcal{I}_{\text{out}}$ and $F \in \mathcal{F}$, the predicate $\bigvee \{ \Cap^F_{in,out} \mid in \in \mathcal{I}_{\text{in}} \}$ is precise.

Remark. It is not a coincidence that these conditions are similar to those that prevent a command from behaving angelically: in both cases, the conditions are constraining the predicate transformers corresponding to the abstraction relation or command to being *conjunctive*.

Note that the translation $\tau_2 : \mathbb{H} + \mathbb{H} \to \mathbb{H}$ does not satisfy the first of these properties, since $\langle\!\langle (1 \mapsto 0, \text{emp}) \rangle\!\rangle = \{1 \mapsto 0\} = \langle\!\langle (\text{emp}, 1 \mapsto 0) \rangle\!\rangle$. Suppose that `allocEither` were added to $\mathbb{H} + \mathbb{H}$. Then, without Conj, it could be soundly implemented in $\mathbb{H}$ by allocating a single cell and returning its address. With Conj, however, this does not work; the implementation must diverge.

For locality-breaking translations, a case can be added to the proof of Lemma 99 to deal with pushing Frame over Conj, in a similar fashion to Disj, provided that the context algebra $\mathcal{A}_{\mathbb{A}}$ is left-cancellative (Definition 5.5). Left-cancellativity ensures that $\{c\} \bullet \bigwedge_{i \in I} P_i \equiv \bigwedge_{i \in I} \{c\} \bullet P_i$. It is also necessary for the predicate translation $[\![(\cdot)]\!]$ to distribute over conjunction; 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).

[7] Proving this statement is rather involved, especially for recursion and looping, for which the proof would typically invoke Tarski's fixedpoint theorem and therefore require a proof of monotonicity of programs.

6.5.2 On Crust

In §6.3, I insinuated that the need for crust came about from having chosen the wrong representation function in the first place. However, it is perhaps not necessarily clear that there is a choice for the representation function that avoids using crust at all. In fact, such a choice is embodied in a key element of the soundness proof, namely the intermediate translation functions (Definition 6.17).

The intermediate translation functions essentially add the outer crust to representations of data structures and contexts while removing the inner crust from representations of contexts. Because of the crust inclusion and application preservation properties of the original representations, this modified representation also preserves application (Lemma 93). (Note that the interface set for this modified representation is $\mathcal{I}_{\text{out}} \times \mathcal{F}$, which can all be considered as the out part.) This modified representation incorporates all of the footprint required at the low level by its very construction, and so no further crust is required.

6.5.3 On Locality-Preserving versus Locality-Breaking

Despite the fact that their names imply a black-and-white distinction between locality-preserving and locality-breaking translations, there is no clear distinction between where the two techniques are applicable.

As an example, consider the implementation of the list module from §6.4.2. I proved that this implementation gave a sound module translation using the locality-breaking technique, since some of the basic operations have a low-level footprint that incorporates an arbitrarily large portion of the linked list. However, it is quite reasonable to identify portions of the low-level state — individual nodes in a linked list — that correspond to portions of the high level state — individual elements of the corresponding list. In fact, the locality-preserving approach can be applied to this implementation, by treating the portion of the linked list that leads up to the nodes of interest as crust.

On the other hand, consider the implementation of the tree module from §6.3.2. In this case, the crust could be arbitrarily large — accounting for all of the siblings of the top level of the tree. This suggests that the locality-breaking approach might have been prudent.

Clearly, there is a significant overlap in the applicability of the two approaches, but are there any cases in which one approach is applicable but the other is not? The answer is, in fact, "no" — a locality-preserving translation can be used to construct a locality-breaking translation and vice versa.

From -preserving to -breaking is simple. Assume that $\tau : \mathbb{A} \to \mathbb{B}$ is a locality-preserving translation. For all $c \in \mathsf{C}_{\mathbb{A}}$, $\varphi \in \mathsf{Cmd}_{\mathbb{A}}$ and $(P, Q) \in \mathrm{Ax}\,[\![\varphi]\!]_{\mathbb{A}}$ there is a derivation of $\vdash_{\mathbb{A}} \{\{\varnothing, c\} \bullet P\}\ \varphi\ \{\{\varnothing, c\} \bullet Q\}$, simply using Axiom followed by Frame. By the soundness of locality-preserving translations, there must therefore also be a derivation of $\vdash_{\mathbb{B}} \{[\![\{\varnothing, c\} \bullet P]\!]_\tau\}\ [\![\varphi]\!]_\tau\ \{[\![\{\varnothing, c\} \bullet Q]\!]_\tau\}$. Thus τ defines a locality-breaking translation.

From -breaking to -preserving is slightly trickier. Assume that $\tau : \mathbb{A} \to \mathbb{B}$ is a locality-breaking translation. For the interface sets, choose $\mathcal{I}_{\mathrm{in}} = \{1\}$ and $\mathcal{I}_{\mathrm{in}} = \mathsf{C}_{\mathbb{A}}$. Define the representation functions as follows:

$$\langle\!\langle \chi_{\mathbb{A}} \rangle\!\rangle^{c} = \{\chi_{\mathbb{B}} \mid \chi_{\mathbb{B}}\ \alpha_\tau\ (c \bullet \chi_{\mathbb{A}})\}$$

$$\langle\!\langle c \rangle\!\rangle^{c_1}_{c_2} = \begin{cases} \mathbf{I}_{\mathbb{B}} & \text{if } c_2 = c_1 \bullet c \\ \emptyset & \text{otherwise.} \end{cases}$$

Finally, choose $\mathcal{F} = \{1\}$ and define $\Subset_c = \mathbf{I}_{\mathbb{B}}$. Application preservation holds by construction, as does crust inclusion. Axiom correctness simply reduces to the criterion that τ is a locality-breaking translation, *i.e.* that the axioms, with each singleton frame, hold under translation. Thus τ defines a locality-preserving translation.

6.5.4 On Abstract Predicates

One way of viewing the translation functions is as abstract predicates; that is, $[\![P]\!]$ is an abstract predicate parametrised by P. However, viewing this as a completely abstract entity does not confer abstract local reasoning. Exposing such axioms as $[\![P]\!] \vee [\![Q]\!] \leftrightarrow [\![P \vee Q]\!]$ is a start — it allows the low-level disjunction rule to implement its high-level counterpart — and is possible with some formulations of abstract predicates [DYDG+10]. However, abstract predicates do not currently provide a mechanism for exporting meta-theorems such as the soundness of the abstract frame rule. That is to say, there is no way to expose the fact that if $\{[\![P]\!]\}\ \mathbb{C}\ \{[\![Q]\!]\}$ holds then so does $\{[\![K \bullet P]\!]\}\ \mathbb{C}\ \{[\![K \bullet P]\!]\}$. This work suggests that including such a mechanism could be a valuable addition to the abstract predicate methodology.

6.5.5 On Data Refinement

The locality refinement techniques presented here can be viewed as instances of the data refinement technique known as *downward simulation* (or L-simulation) [dE99]. An implementation downward simulates a specification if there is some translation from the abstract model to the concrete model such that

the possible results of translating from an abstract state to a concrete state and then performing the implementation of an operation are contained within the possible results of performing the abstractly specified operation followed by translating the result to a concrete state. One key difference with the approach presented here is that downward simulation is typically considered with regard to denotational rather than axiomatic semantics. By working with axiomatic semantics, locality is embedded into the abstract specification of modules.

6.5.6 On Concurrency

Extending the results presented here to a concurrent setting is not entirely trivial. A separation logic-style parallel rule is only appropriate when the model has a commutative $*$-like operator; while the heap and list modules have such an operator, the tree model does not. (Segment logic [GW09] is a promising approach to introducing a commutative $*$ to structured models such as trees.)

Assume that we have a sound module translation from an abstract module to a concrete one, and that both models include a commutative $*$ connective, such that, for every state s there is some context c such that, for all states s', $s * s' = c \bullet c'$. If the implementations of the module operations are linearisable [HW90] — that is, they are observationally equivalent to atomic operations — then the separation logic parallel rule should be sound at the abstract level. That is, if

$$\{[\![P_1]\!]\}\ [\![\mathbb{C}_1]\!]\ \{[\![Q_1]\!]\}$$
$$\{[\![P_2]\!]\}\ [\![\mathbb{C}_2]\!]\ \{[\![Q_2]\!]\}$$

then

$$\{[\![P_1 * P_2]\!]\}\ [\![\mathbb{C}_1 \parallel \mathbb{C}_2]\!]\ \{[\![Q_1 * Q_2]\!]\}\,.$$

Intuitively, this is because the operations of the threads are effectively interleaved, and so the state of the other thread at each point can be viewed as a frame.

For locality-preserving translations it is possible that a more direct approach can be taking to establishing the soundness of the abstract parallel rule, especially if the translation can be shown to preserve the $*$ connective. This principle is embodied in the concurrent abstract predicate methodology presented in Chapter 7. Where $*$ is incompletely preserved it is likely that the crust will play an important role, representing resource that may be shared between threads, and therefore must be manipulated in a thread-safe manner.

Chapter 7

Concurrent Abstract Predicates

In this chapter, I present a program logic that allows fine-grained abstraction in the presence of sharing, by introducing *concurrent abstract predicates* (CAP). These predicates present a fiction of disjointness; that is, they can be used *as if* each predicate represents a disjoint resource, whereas in fact resources are shared between predicates. For example, given a set implemented by a linked list, abstract predicates can be used to assert "the set contains 5, which I control" and "the set contains 6, which I control". Both predicates assert properties about the same shared structure, and both can be used at the same time by separate threads: for example, the two elements can be concurrently removed from the set.

Concurrent abstract predicates capture information about the permitted changes to the shared structure. In the case of the set predicates, each predicate gives the thread full control over a particular element of the set. Only the thread owning the predicate can remove this element. This control is implemented using permission resources [DFPV09]. The permissions encapsulated in a predicate must ensure that the predicate is stable: that is, immune from any interference that the environment could have permission to perform. Predicates are therefore able to specify independent properties about the data, even though the data are shared.

With the CAP program logic, a module implementation can be verified against a high-level specification expressed using concurrent abstract predicates. Clients of the module can then be verified purely in terms of this high-level specification, without reference to the module's implementation. I demonstrate this

by presenting two implementations of a lock module satisfying the same abstract lock specification, and using this specification to build two implementations of a concurrent set satisfying the same abstract set specification. At each level, the reasoning treats the lower level entirely abstractly, avoiding dealing directly with the implementation details. Hence, concurrent abstract predicates provide the necessary abstraction for compositional reasoning about concurrent systems.

This chapter begins with an informal development of the CAP system using the example of a lock module, considering two different implementations (§7.1). In §7.2, I demonstrate the compositionality of the system by building two implementations of an abstract set specification on top of the abstract lock module. In §7.3, I formalise the CAP system and prove that it is sound with respect to an operational semantics.

Collaboration and Contribution

The work presented in this chapter, which was first presented in [DYDG$^+$10], was undertaken in collaboration with Dodds, Gardner, Parkinson and Vafeiadis. My key contribution was the observation that the permission models of Dodds *et al.* [DFPV09] could be used to realise abstract disjoint specifications. I also contributed to the technical development, in particular suggesting the method by which shared regions can be dynamically created and disposed.

7.1 Informal Development

I develop the core idea of this chapter: to define abstract specifications for concurrent modules and prove that concrete module implementations satisfy these specifications. In this section, I use the motivating example of a lock module, one of the simplest examples of concurrent resource sharing. I define an abstract specification for a lock (§7.1.1) and show two different implementations that satisfy the specification (§7.1.2 and §7.1.4).

7.1.1 Lock Specification

A lock module is a simple mechanism for ensuring that concurrent threads do not attempt to access a protected resource simultaneously. When a thread wants to access the resource protected by lock l it calls the function `lock`(l). The function does not return until it has acquired the lock on behalf of the thread — it may have to wait if another thread already holds the lock. Having acquired the lock, the thread is able to access the protected resource without

interference (assuming that all other threads obey the locking discipline). Once its operation on the protected resource is complete, the thread calls $\texttt{unlock}(l)$ to allow other threads the ability to access the resource. A lock module also typically has a mechanism for creating new locks, such as $\texttt{makelock}(n)$, which creates a lock (that is initially locked) and also allocates a memory block of length n.

The lock module functions have the following specification:

$$\begin{array}{rcl} \{\mathsf{isLock}(\boldsymbol{l})\} & \texttt{lock}(\boldsymbol{l}) & \{\mathsf{isLock}(\boldsymbol{l}) * \mathsf{Locked}(\boldsymbol{l})\} \\ \{\mathsf{Locked}(\boldsymbol{l})\} & \texttt{unlock}(\boldsymbol{l}) & \{\mathrm{emp}\} \\ \{\mathrm{emp}\} & \texttt{makelock}(\boldsymbol{n}) & \left\{ \begin{array}{l} \exists l.\, \mathrm{ret} = l \wedge \mathsf{isLock}(l) * \mathsf{Locked}(l) \\ \quad * \, l \mapsto - * \cdots * (l + \boldsymbol{n} - 1) \mapsto - \end{array} \right\}. \end{array}$$

This specification, which is presented by the lock module to its clients, is abstract and so independent of the underlying implementation.[1] The assertions $\mathsf{isLock}(l)$ and $\mathsf{Locked}(l)$ are *abstract predicates*: their exact meaning is hidden from the client and depends on the module implementation. The predicate $\mathsf{isLock}(l)$ asserts l identifies a lock which the thread may acquire, while $\mathsf{Locked}(l)$ asserts that the thread currently holds the lock identified by l. The underlying state model is a separation algebra, so the separating conjunction $P * Q$ is used to assert that the state can be split disjointly into two parts, one satisfying P and the other satisfying Q.

Although the concrete interpretations of abstract predicates are hidden from clients of the module, the clients need some additional knowledge about the predicates in order to use them effectively. This information is embodied in *abstract predicate axioms*, which also form part of the module specification. For the lock module, the following two axioms are provided:

$$\mathsf{isLock}(l) \leftrightarrow \mathsf{isLock}(l) * \mathsf{isLock}(l)$$
$$\mathsf{Locked}(l) * \mathsf{Locked}(l) \leftrightarrow \mathrm{false}.$$

The first axiom allows the client to share freely the knowledge that l is a lock. When a thread branches in two, its resources are split disjointly between the two threads. This axiom allows both threads to have the knowledge that l is a lock, which is essential if they are to use the lock for synchronisation.

[1] This specification resembles those used in the work of Gotsman *et al.* [GBC+07] and Hobor *et al.* [HAN08] on dynamically-allocated locks.

Remark. Here, I do not track how $\mathsf{isLock}(l)$ is split. To allow for lock l to be disposed, it would be necessary to keep track of how its isLock predicate is split, for instance with fractional permissions [Boy03], to be sure that no other thread was expecting to use l as a lock when the time came to dispose it.

The second axiom captures the fact that holding a lock (by having the $\mathsf{Locked}(l)$ predicate) is exclusive. Using this axiom, together with the specification for `lock`, the following triple can be inferred:

$$\{\mathsf{isLock}(l)\}\ \texttt{lock}(l);\texttt{lock}(l)\ \{\text{false}\}\,.$$

Under the partial correctness interpretation of triples, this implies that attempting to acquire a lock twice in succession will not terminate. This is the expected behaviour of a (non-reentrant) lock.[2]

7.1.2 A Compare-and-Swap Lock Implementation

Consider the following lock module implementation based on an atomic compare-and-swap operation.

```
lock(l) {
  local b in
    b := false;
    while ¬b do
      ⟨b := CAS(l − 1, 0, 1)⟩
}
unlock(l) {
  ⟨[l − 1] := 0⟩
}
```

```
l := makelock(n) {
  local x in
    x := alloc(n + 1);
    [x] := 1;
    l := x + 1
}
```

Angle brackets are used to denote atomic statements. Atomic operations appear to occur instantaneously. Many instruction sets support primitive atomic operations including read, write and compare-and-swap. The compare-and-swap instruction $\texttt{CAS}(a, v_1, v_1)$ compares the value stored at heap address a with value v_1 and replaces it with v_2 if the two are equal. The return value indicates whether the swap took place.

The intuition behind the algorithm is that the lock for the block at address l is stored at address $l - 1$; a value of 0 indicates that the lock is not held,

[2] Reentrant locks allow a thread holding a lock to successfully acquire a lock that is already held. The specification is necessarily different to that for ordinary locks.

while a value of 1 indicates that the lock is held. In order to acquire the lock, a thread must wait until an instant in which the lock is not held and signal that the lock is now held, as embodied by the compare-and-swap. Since the compare-and-swap is atomic, no other thread can also believe that it holds the lock. In order to release a lock, it is sufficient to (atomically) set the lock value to 0.

Interpretation of Abstract Predicates

I relate the lock implementation to the lock specification by giving a concrete interpretation of the abstract predicates. The predicates are interpreted not just as assertions about the internal state of the module, but also as assertions about the internal *interference* of the module: that is, how concurrent threads can modify shared parts of the module state. To describe this internal interference, separation logic is extended with shared region assertions $\boxed{P}_I^r$ and permission assertions $[\gamma]_\pi^r$.

The shared region assertion $\boxed{P}_I^r$ specifies that there is a shared region of memory, identified by label r, and that the entire shared region satisfies P. A shared region is not divided when it is shared, ensuring that all threads have a consistent view of it. This is captured by the logical equivalence: $\boxed{P}_I^r * \boxed{Q}_I^r \equiv \boxed{P \wedge Q}_I^r$. The possible changes that the shared region can undergo — the *actions* on the shared region — are specified by the *interference assertion* I.

The permission assertion $[\gamma]_\pi^r$ specifies that the thread has permission π to perform the action named γ on the shared region r. The action γ must be declared in region r's interference assertion. Following Boyland [Boy03], the permission can be

- a fractional permission, $\pi \in (0, 1)$, denoting that both the thread and the environment can perform the action, or
- full permission, $\pi = 1$, denoting that the thread can perform the action but the environment cannot.[3]

Permissions can be split (and recombined): $[\gamma]_{\pi_1+\pi_2}^r \equiv [\gamma]_{\pi_1}^r * [\gamma]_{\pi_2}^r$. This allows permissions to be split among threads so that, for example, each thread is able to acquire a lock. The total amount of permission for any given action is always 1, so having permission 1 is indeed exclusive.

[3]The state model also contains a zero permission, 0, denoting that the thread may not perform the action but the environment may. Zero permissions are not directly asserted, but are implied when no other permission value is asserted.

A permission can only be acted upon if it is held in the thread's local state. Thus if full permission to an action is in a shared region then no thread can perform that action. Why have an action at all if its full permission is in the shared state? Since actions can manipulate permissions, a thread may, by performing one action, acquire permission to perform another action. This pattern is a common one.

With the intuition behind these two new assertions established, I can give the interpretations of the lock predicates for the compare-and-swap lock. They are as follows:

$$\mathsf{isLock}(l) \equiv \exists r, \pi.\, [\textsc{Lock}]^r_\pi * \boxed{((l-1) \mapsto 0 * [\textsc{Unlock}]^r_1) \vee (l-1) \mapsto 1}^r_{I(l,r)}$$

$$\mathsf{Locked}(l) \equiv \exists r.\, [\textsc{Unlock}]^r_1 * \boxed{(l-1) \mapsto 1}^r_{I(l,r)}$$

The interpretation of the $\mathsf{isLock}(l)$ predicate specifies that the thread has, in its local state, permission π to perform the Lock. It also specifies that the shared region satisfies the module's invariant: either the lock is unlocked $(l-1) \mapsto 0$ and the region holds the full permission $[\textsc{Unlock}]^r_1$ to unlock the lock; or the lock is locked $(l-1) \mapsto 1$ and the unlocking permission is gone (taken by the thread that acquired the lock).

Meanwhile, the interpretation of the $\mathsf{Locked}(l)$ predicate specifies that the thread has exclusive permission $[\textsc{Unlock}]^r_1$ to unlock the lock, and that the lock is indeed locked $(l-1) \mapsto 1$.

Remark. Since the region name r is existentially quantified in both predicates, the question as to whether the predicates refer to the same region naturally arises, *e.g.* in the assertion $\mathsf{isLock}(l) * \mathsf{isLock}(l')$. When $l = l'$, the uniqueness of the heap cell at address $(l-1)$ ensures that the regions are the same. On the other hand, when $l \neq l'$ the regions have different contents (and interference), and so must be distinct.

The actions permitted on the lock's shared region are declared in the interference assertion $I(l, r)$. Each action declaration has the form $\gamma : P \rightsquigarrow Q$, where γ is the name of the action, P describes part of the shared region immediately before the action is performed, and Q describes what becomes of that part of the shared region immediately after the action is performed. The actions for the compare-and-swap lock are as follows:

$$I(l,r) \stackrel{\text{def}}{=} \left(\begin{array}{ll} \textsc{Lock} & : (l-1) \mapsto 0 * [\textsc{Unlock}]^r_1 \rightsquigarrow (l-1) \mapsto 1, \\ \textsc{Unlock} & : (l-1) \mapsto 1 \rightsquigarrow (l-1) \mapsto 0 * [\textsc{Unlock}]^r_1 \end{array} \right)$$

The LOCK action requires that the shared region contains the unlocked lock $(l-1) \mapsto 0$ together with full permission $[\text{UNLOCK}]_1^r$ to unlock the lock. The result of the action is that the lock is locked $(l-1) \mapsto 1$ and the unlock permission has moved to the local state of the thread performing the action (since $[\text{UNLOCK}]_1^r$ has gone from the shared region). The movement of $[\text{UNLOCK}]_1^r$ into the locking thread's local state allows the thread to release the lock later. Note the local state is not explicitly represented in the action; since interference only happens on the shared region, actions do not need to be prescriptive about local state.

The UNLOCK action requires that the shared region contains the locked lock $(l-1) \mapsto 1$. The result of the action is that the lock is unlocked $(l-1) \mapsto 0$ and the unlock permission $[\text{UNLOCK}]_1^r$ is returned to the shared region. The thread must have originally had $[\text{UNLOCK}]_1^r$ in its local state in order to move it to the shared state in accordance with the action.

Notice that UNLOCK is self-referential: the action moves exclusive permission on itself out of the local state. Consequently, a thread can only perform UNLOCK once, corresponding to the intuitive protocol that a thread can only release a lock once without acquiring it again. This self-reference is not problematic, since permissions *indirectly* reference actions by name rather than directly referencing them by their semantics.

It is important for the client that the abstract predicates be *self-stable* — that is, if the predicate holds then no (permitted) interference by the environment should change that. This makes it possible to abstract the module's internal interference.

Consider the $\mathsf{isLock}(x)$ predicate. There are two actions that the environment could potentially perform on the shared region: LOCK and UNLOCK. Performing the LOCK action simply takes the shared region from the first disjunct in the predicate definition to the second; conversely, performing the UNLOCK action takes the shared region from the second disjunct to the first. In both cases, the predicate remains stable.

Consider the $\mathsf{Locked}(x)$ predicate. Since the lock is already locked (*i.e.* $l - 1 \mapsto 1$), it is not possible for the environment to perform the LOCK action. Furthermore, since the full permission $[\text{UNLOCK}]_1^r$ to the unlock action is in the thread's local state, the environment cannot perform the UNLOCK action either. Hence the predicate is also stable.

In [DYDG+10], the proof rules required abstract predicates to be self-stable; here, in order to disentangle the abstraction mechanism from the semantic model, self-stability is exposed to the client explicitly with additional axioms. In

the logic, the stability of an arbitrary assertion P may be expressed as $P \to \boxed{\mathbf{R}}P$. The assertion $\boxed{\mathbf{R}}P$ describes those states which will still satisfy P no matter what permitted interference the environment may perform on them. This is the *weakest stable stronger assertion* with respect to P of Wickerson *et al.* [WDP10]. The implication $P \to \boxed{\mathbf{R}}P$ therefore means that, if P holds, then it will continue to hold, despite any permitted interference from the environment — that is, P is stable. The following two axioms should therefore be added to the lock module specification:

$$\mathsf{isLock}(x) \to \boxed{\mathbf{R}}\mathsf{isLock}(x)$$
$$\mathsf{Locked}(x) \to \boxed{\mathbf{R}}\mathsf{Locked}(x).$$

I have shown that these two axioms hold, but, in order for the implementation to be correct, the two axioms from §7.1.1 must also hold. The first of these, $\mathsf{isLock}(l) \leftrightarrow \mathsf{isLock}(l) * \mathsf{isLock}(l)$, follows from the definition of isLock by the fact that the fractional permission $[Lock]^r_\pi$ can be subdivided as $[Lock]^r_{\pi/2} * [Lock]^r_{\pi/2}$ and the fact that $*$ behaves additively (*i.e.*like $\wedge$) on shared region assertions. The second axiom, $\mathsf{Locked}(l) * \mathsf{Locked}(l) \leftrightarrow \text{false}$, follows from the fact that more than full permission an action cannot exist — $[\textsc{Unlock}]^r_1 * [\textsc{Unlock}]^r_1 \equiv \text{false}$.

Verifying the Implementation

Given the interpretations for the abstract predicates above, the lock implementation can be verified against its specification. Figures 7.1 and 7.2 present outline proofs for the three operations.

Consider first the `unlock` operation, which has the simplest proof. The proof first unfolds the abstract predicate in the precondition to its concrete interpretation. The precondition permits the atomic update $\langle [\mathtt{l} - 1] := 0 \rangle$ because it can be viewed as performing the unlock action — setting the heap cell at $\mathtt{l} - 1$ from 1 to 0 and transferring full permission $[\textsc{Unlock}]^r_1$ to the unlock action into the shared region — and the thread holds permission to perform that action. The result of the thread performing the action is shown in the third assertion in the proof; but this is not the postcondition of the atomic update because it is not stable. To stabilise the assertion, it is necessary to consider what updates the environment may make to the shared region: in this case, another thread could acquire the lock. This leads to the stable postcondition, which is weakened further to emp, which makes no assertion at all about the shared state.

For the `lock` operation, the key proof step is at the atomic compare-and-swap operation. If the operation fails then no change is made to the state; if the operation succeeds then it is interpreted as performing the Lock action (since

$\{\mathsf{isLock}(\mathtt{l})\}$
$\mathtt{lock}(\mathtt{l})\ \{$
$\quad \left\{\exists r, \pi.\ [\textsc{Lock}]^r_\pi * \boxed{((\mathtt{l}-1) \mapsto 0 * [\textsc{Unlock}]^r_1) \vee (\mathtt{l}-1) \mapsto 1}^r_{I(\mathtt{l},r)}\right\}$
$\quad \mathtt{local}\ \mathtt{b}\ \mathtt{in}$
$\quad\quad \mathtt{b} := \mathit{false};$
$\quad\quad \left\{\begin{array}{r}\left(\neg\mathtt{b} \wedge [\textsc{Lock}]^r_\pi * \boxed{((\mathtt{l}-1) \mapsto 0 * [\textsc{Unlock}]^r_1) \vee (\mathtt{l}-1) \mapsto 1}^r_{I(\mathtt{l},r)}\right) \\ \vee \left(\mathtt{b} \wedge [\textsc{Lock}]^r_\pi * [\textsc{Unlock}]^r_1 * \boxed{(\mathtt{l}-1) \mapsto 1}^r_{I(\mathtt{l},r)}\right)\end{array}\right\}$
$\quad\quad \mathtt{while}\ \neg b\ \mathtt{do}$
$\quad\quad\quad \left\{\neg\mathtt{b} \wedge [\textsc{Lock}]^r_\pi * \boxed{((\mathtt{l}-1) \mapsto 0 * [\textsc{Unlock}]^r_1) \vee (\mathtt{l}-1) \mapsto 1}^r_{I(\mathtt{l},r)}\right\}$
$\quad\quad\quad \langle \mathtt{b} := \mathtt{CAS}(\mathtt{l}-1, 0, 1)\rangle$
$\quad\quad\quad \left\{\begin{array}{r}\left(\neg\mathtt{b} \wedge [\textsc{Lock}]^r_\pi * \boxed{((\mathtt{l}-1) \mapsto 0 * [\textsc{Unlock}]^r_1) \vee (\mathtt{l}-1) \mapsto 1}^r_{I(\mathtt{l},r)}\right) \\ \vee \left(\mathtt{b} \wedge [\textsc{Lock}]^r_\pi * [\textsc{Unlock}]^r_1 * \boxed{(\mathtt{l}-1) \mapsto 1}^r_{I(\mathtt{l},r)}\right)\end{array}\right\}$
$\quad\quad \left\{\mathtt{b} \wedge [\textsc{Lock}]^r_\pi * [\textsc{Unlock}]^r_1 * \boxed{(\mathtt{l}-1) \mapsto 1}^r_{I(\mathtt{l},r)}\right\}$
$\quad \left\{\begin{array}{r}\exists r, \pi.\ [\textsc{Lock}]^r_\pi * \boxed{((\mathtt{l}-1) \mapsto 0 * [\textsc{Unlock}]^r_1) \vee (\mathtt{l}-1) \mapsto 1}^r_{I(\mathtt{l},r)} \\ * \exists r.\ [\textsc{Unlock}]^r_1 * \boxed{(\mathtt{l}-1) \mapsto 1}^r_{I(\mathtt{l},r)}\end{array}\right\}$
$\}$
$\{\mathsf{isLock}(\mathtt{l}) * \mathsf{Locked}(\mathtt{l})\}$

$\{\mathsf{Locked}(\mathtt{l})\}$
$\mathtt{unlock}(\mathtt{l})\ \{$
$\quad \left\{\exists r.\ [\textsc{Unlock}]^r_1 * \boxed{(\mathtt{l}-1) \mapsto 1}^r_{I(\mathtt{l},r)}\right\}$
$\quad \langle [\mathtt{l}-1] := 0\rangle$
$\quad\quad \left\{\exists r. \boxed{(\mathtt{l}-1) \mapsto 0 * [\textsc{Unlock}]^r_1}^r_{I(\mathtt{l},r)}\right\}$
$\quad$ *// Stabilise the assertion*
$\quad \left\{\exists r. \boxed{((\mathtt{l}-1) \mapsto 0 * [\textsc{Unlock}]^r_1) \vee (\mathtt{l}-1) \mapsto 1}^r_{I(\mathtt{l},r)}\right\}$
$\}$
$\{\mathrm{emp}\}$

Figure 7.1: Proof outline for compare-and-swap lock (`lock` and `unlock`)

$$
\begin{array}{l}
\{\mathrm{emp}\} \\
\mathtt{l} := \mathtt{makelock}(\mathtt{n})\ \{ \\
\quad \mathtt{local\ x\ in} \\
\quad\quad \mathtt{x} := \mathtt{alloc}(\mathtt{n}+1); \\
\quad\quad \{\exists x.\, \mathtt{x} = x \wedge x \mapsto - * (x+1) \mapsto - * \cdots * (x+n) \mapsto -\} \\
\quad\quad [\mathtt{x}] := 1; \\
\quad\quad \{\exists x.\, \mathtt{x} = x \wedge x \mapsto 1 * (x+1) \mapsto - * \cdots * (x+n) \mapsto -\} \\
\quad\quad \textit{// Create shared lock region} \\
\quad\quad \left\{ \exists l.\, \mathtt{x} = l - 1 \wedge \exists r.\, [\textsc{Lock}]_1^r * [\textsc{Unlock}]_1^r * \boxed{(l-1) \mapsto 1}_{I(\mathtt{l},r)}^{r} \right. \\
\quad\quad\quad\quad \left. * \, l \mapsto - * \cdots * (l + \mathtt{n} - 1) \mapsto - \right\} \\
\quad\quad \mathtt{l} := \mathtt{x} + 1 \\
\} \\
\{\exists l.\, \mathtt{l} = l \wedge \mathsf{isLock}(l) * \mathsf{Locked}(l) * l \mapsto - * \cdots * (l + \mathtt{n} - 1) \mapsto -\}
\end{array}
$$

Figure 7.2: Proof outline for compare-and-swap lock (`makelock`)

it has permission $[\textsc{Lock}]_\pi^r$) — setting the heap cell at $\mathtt{l} - 1$ from 0 to 1 and transferring the permission $[\textsc{Unlock}]_1^r$ to the thread's local state. The variable `b` records whether the `CAS` succeeded, so the lock must have been acquired once the loop has been exited. To repackage the postcondition in the form of the abstract predicates, the shared region assertion is duplicated and (for the isLock predicate) weakened.

Finally, for the `makelock` operation, the key proof step is the creation of a fresh shared region and its associated permissions. The proof system includes rules which allow repartitioning of state between local and shared regions (where the actual contents of the heap is unaffected) in a way that is consistent with the expectations of other threads. In particular, a fresh region shared region and full permissions to all of its actions can be created.

Remark. When a shared region is created, its region identifier is existentially quantified. This is because if a specific identifier were used then it would not be consistent with the possibility that a region with that identifier had already been created (possibly by another thread). This approach to region creation was inspired by the axiomatisation of heap allocation, wherein the address returned is existentially quantified. Essentially, region creation is treated as (demonically) nondeterministic.

7.1.3 The Proof System

I present an informal description of the proof system; the formal details are given in §7.3. Judgements in the proof system have the form $\Delta; \Gamma \vdash \{P\}\ \mathbb{C}\ \{Q\}$, where Δ contains predicate definitions and axioms, and Γ contains specifications for the functions called in $\mathbb{C}$. The pre- and postconditions P and Q are assertions which may use abstract predicates; their semantics (as sets of states) is therefore dependent on the interpretation of the abstract predicates themselves. The judgement should be read with a fault-avoiding partial correctness interpretation: given an interpretation of abstract predicates that satisfies Δ, and functions that meet the specifications in Γ, whenever the program $\mathbb{C}$ is run from a state satisfying P then it will not fault, but will either terminate in a state satisfying Q or not terminate at all.

The proof rule for atomic commands is the following:

$$\frac{\vdash_{\mathsf{SL}} \{p\}\ \mathbb{C}\ \{q\}}{\Delta; \Gamma \vdash \left\{\boxed{\mathbf{R}} \left[\begin{smallmatrix} p \\ q \end{smallmatrix}\right\rangle Q\right\}\ \langle \mathbb{C} \rangle\ \left\{\blacklozenge_{\mathbf{R}} Q\right\}}\ \textsc{Atomic}$$

The premiss of this rule is an ordinary separation logic triple, which is inferred using the separation logic proof rules. Its assertions therefore do not make any mention of permissions, shared regions or abstract predicates, but only of heap.

In the conclusion, the assertion $\left[\begin{smallmatrix} p \\ q \end{smallmatrix}\right\rangle Q$ specifies that the state is such that replacing a concrete subheap matching p with one satisfying q can be interpreted in a manner consistent with the permitted interference to give a state satisfying Q. One way of interpreting this is that, for the atomic command, the region boundaries are broken down and the update is performed, but when the bounderies are reestablished there must be some way of explaining the update that is consistent with the actions that the thread has permission to perform.

Since $\left[\begin{smallmatrix} p \\ q \end{smallmatrix}\right\rangle Q$ may not be stable, the precondition is taken to be the weakest stable assertion that is is stronger: $\boxed{\mathbf{R}} \left[\begin{smallmatrix} p \\ q \end{smallmatrix}\right\rangle Q$. Similarly Q may not be stable, and so the postcondition is taken to be the strongest stable assertion that is weaker: $\blacklozenge_{\mathbf{R}} Q$.

As we saw in the proof of `makelock`, a thread manipulate shared regions without making any update to the concrete state at all. This is embodied by the following proof rules:

$$\frac{\Delta; \Gamma \vdash \{P\}\ \mathbb{C}\ \{Q\}}{\Delta; \Gamma \vdash \left\{\diamondsuit_{\mathbf{G}} P\right\}\ \mathbb{C}\ \left\{Q\right\}}\ \textsc{Guar-L} \qquad \frac{\Delta; \Gamma \vdash \left\{P\right\}\ \mathbb{C}\ \left\{\diamondsuit_{\mathbf{G}} Q\right\}}{\Delta; \Gamma \vdash \{P\}\ \mathbb{C}\ \{Q\}}\ \textsc{Guar-R}$$

In these rules $\diamondsuit_{\mathbf{G}} P$ specifies that the state is such that, without updating the

concrete heap, it can be repartitioned in a manner consistent with the permitted interference to give a state satisfying P. This is equivalent to $\left[\begin{smallmatrix}\text{emp}\\ \text{emp}\end{smallmatrix}\right\rangle P$.

There are essentially three types of repartitioning that can take place: performing an action for which the thread holds permission; creating a fresh region, together with all of its associated permissions; and disposing of a region when all of its associated permissions are held. These may occur with a concrete update (using the ATOMIC rule) or without one (using the GUAR-L or GUAR-R rule).

Remark. In constructing a proof, we have a degree of choice about which repartitionings to perform and when. Since we could make different and inconsistent choices about these repartitionings, the conjunction rule must be omitted from the proof system.

The proof system also includes proof rules for dealing with abstract predicates. The following rule allows a client that is verified with respect to an abstract module to be verified with respect to a concrete module that correctly implements the abstract module specification:

$$\frac{\Delta \vdash \{P_1\}\ \mathbb{C}_1\ \{Q_1\} \quad \ldots \quad \Delta \vdash \{P_n\}\ \mathbb{C}_n\ \{Q_n\} \quad \Delta \vdash \Delta' \qquad \Delta'; \{P_1\}\ f_1\ \{Q_1\}, \ldots, \{P_n\}\ f_n\ \{Q_n\} \vdash \{P\}\ \mathbb{C}\ \{Q\}}{\vdash \{P\}\ \texttt{let}\ f_1 = \mathbb{C}_1, \ldots, f_n = \mathbb{C}_n\ \texttt{in}\ \mathbb{C}\ \{Q\}}$$

The premisses of the rule require that

- the implementation $\mathbb{C}_i$ of f_i satisfies the specification $\{P_i\}\ \mathbb{C}_i\ \{Q_i\}$, given the predicate definitions Δ;
- the predicate definitions Δ satisfy the abstract predicate axioms Δ' that are exposed to the client; and
- the client satsfies the specification $\{P\}\ \mathbb{C}\ \{Q\}$ given the predicate axioms Δ' and the function specifications $\{P_1\}\ f_1\ \{Q_1\}, \ldots, \{P_n\}\ f_n\ \{Q_n\}$.

Since the client is only verified with respect to the abstract specification of the module, any module implementation meeting the specification can be used.

The proof rule defined above is not one of the basic rules of the proof system, but it is derivable from them. A couple of side-conditions are omitted from the above rule. Firstly, P and Q should not use the abstract predicates. This is because abstract predicates have an undefined meaning in the conclusion of the rule, since there is no abstract predicate environment. Secondly, the predicate definitions must be realisable by some concrete interpretation of the predicates. A predicate definition such as $\mathsf{A} \leftrightarrow \neg\mathsf{A}$ is inconsistent. Syntactic conditions can be used to enforce the consistency of predicate definitions.

7.1.4 A Ticketed Lock Implementation

Let us now consider another, more complex, implementation of the lock module: the ticketed lock. Whereas the compare-and-swap lock allows any of the threads contending for the lock to succeed once the lock becomes available, the ticketed lock grants the lock to the first thread that registered its intent to acquire the lock. This provides a fairness guarantee: assuming every thread that acquires the lock eventually releases it, no thread that is waiting to acquire the lock will be denied it forever — a situation that is a possibility for the compare-and-swap lock. This algorithm is used in current versions of Linux. Despite the fact that the ticketed lock is quite different from the compare-and-swap lock, I show that it also satisfies the abstract specification given in §7.1.1.

The ticketed lock implementation is as follows:

```
lock(l) {
  local t, o in
    ⟨t := INCR(l.next)⟩;
    ⟨o := [l.owner]⟩;
    while t ≠ o do
      ⟨o := [l.owner]⟩
}
unlock(l) {
  ⟨INCR(l.owner)⟩
}
```

```
l := makelock(n) {
  local x in
    x := alloc(n + 2);
    l := x + 2;
    [l.owner] := 0
    [l.next] := 1
}
```

where

$$E.\texttt{next} \stackrel{\text{def}}{=} E - 1 \qquad\qquad E.\texttt{owner} \stackrel{\text{def}}{=} E - 2.$$

The operation $\texttt{INCR}(a)$ atomically increments the value stored at address a and returns the original value. As with `CAS`, instruction sets typically support `INCR` as a primitive atomic operation.

The intuition behind the algorithm is that $l.\texttt{next}$ stores the next available ticket for the lock l, and $l.\texttt{owner}$ stores the number of the ticket that currently has the right to the lock. A thread wishing to acquire the lock first takes a ticket by atomically reading and incrementing $l.\texttt{next}$. It then waits until its $l.\texttt{owner}$ matches the ticket it acquired, indicating it now holds the lock. To unlock the lock, the thread atomically increments $l.\texttt{owner}$, signalling the thread with the next ticket that it now holds the lock.

The algorithm is analogous to queuing systems in shops, in which each customer takes a numbered ticket on arrival and is served once his or her number is announced.

Interpretation of Abstract Predicates

The interpretations of the lock predicates for the ticketed lock are as follows:

$$\mathsf{isLock}(l) \equiv \exists r, \pi.\, [\textsc{Take}]^r_\pi * \boxed{\begin{array}{r}\exists k, k'.\, k \le k' * l.\texttt{owner} \mapsto k * l.\texttt{next} \mapsto k' \\ * \circledast k'' \ge k'.\, [\textsc{Next}(k'')]^r_1 * \mathrm{true}\end{array}}^r_{T(l,r)}$$

$$\mathsf{Locked}(l) \equiv \exists r, k.\, [\textsc{Next}(k)]^r_1 * \boxed{l.\texttt{owner} \mapsto k * \mathrm{true}}^r_{T(l,r)}$$

(Here $\circledast$ is the lifting of $*$ to sets; it is the multiplicative analogue of $\forall$.)

The actions for the ticketed lock are as follows:

$$T(l,r) \stackrel{\text{def}}{=} \left(\begin{array}{ll}\textsc{Take} & : \exists k.\, (l.\texttt{next} \mapsto k * [\textsc{Next}(k)]^r_1 \rightsquigarrow l.\texttt{next} \mapsto (k+1))\,, \\ \textsc{Next}(k) & : l.\texttt{owner} \mapsto k \rightsquigarrow l.\texttt{owner} \mapsto (k+1) * [\textsc{Next}(k)]^r_1\end{array}\right)$$

The Take action is performed by a thread in acquiring ticket, and Next is performed in making the lock available to the holder of the next ticket.

The predicate $\mathsf{isLock}(l)$ asserts that the shared region contains the `owner` and `next` fields of the lock l, together with full permission for the Next action for every ticket value that has not yet been claimed. It also asserts that current holder of the lock is no greater than the next available ticket value, and that the thread has permission to perform the Take action. The predicate $\mathsf{Locked}(l)$ asserts that the thread has the exclusive permission to increment the lock's `owner` field from its current value.

Stability of $\mathsf{isLock}(l)$ is ensured by the fact that it is stable under the Take action and the $\textsc{Next}(k)$ actions for each k less than the current value of $l.\texttt{next}$; no other actions can occur. Stability of $\mathsf{Lock}(l)$ is ensured by the fact that the predicate holds full permission on the action $\textsc{Next}(k)$, and no other action can affect the `owner` field while its value is k. The other two axioms follow simply from the predicate definitions, as for the compare-and-swap lock.

Verifying the Implementation

Given the interpretations of the inductive predicates above, the ticketed lock implementation can be verified against the lock specification. Figure 7.3 gives an outline proof of the `lock` and `unlock` operations; the proof of `makelock` is essentially the same as for the compare-and-swap lock, and so is omitted.

The proofs largely follow the intuition behind the algorithm. The `lock` operation increments the lock's `next` field, acquiring full permission for the Next action for its ticket value, `t`. It then loops, reading the lock's `owner` field, which is stably less than or equal to `t`; once it is equal to `t`, the field's value is stable, because only the $\textsc{Next}(t)$ action can change it. The `unlock`

$\{\mathsf{isLock}(\mathtt{l})\}$
$\mathtt{lock}(\mathtt{l})\ \{$
$\left\{\exists r, \pi.\, [\textsc{Take}]^r_\pi * \boxed{\begin{array}{r}\exists k, k'.\, k \le k' * l.\mathtt{owner} \mapsto k * l.\mathtt{next} \mapsto k' \\ * \circledast k'' \ge k'.\, [\textsc{Next}(k'')]^r_1 * \mathrm{true}\end{array}}^r_{T(\mathtt{l},r)}\right\}$
$\mathtt{local}\ \ \mathtt{t}, \mathtt{o}\ \ \mathtt{in}$
$\langle \mathtt{t} := \mathtt{INCR}(\mathtt{l}.\mathtt{next})\rangle\,;$
$\left\{\exists r, \pi.\, [\textsc{Take}]^r_\pi * [\textsc{Next}(\mathtt{t})]^r_1 * \boxed{\begin{array}{r}\exists k, k'.\, k \le \mathtt{t} < k * l.\mathtt{owner} \mapsto k * l.\mathtt{next} \mapsto k' \\ * \circledast k'' \ge k'.\, [\textsc{Next}(k'')]^r_1 * \mathrm{true}\end{array}}^r_{T(\mathtt{l},r)}\right\}$
$\langle \mathtt{o} := [\mathtt{l}.\mathtt{owner}]\rangle\,;$
$\left\{\exists r, \pi, k, k'.\, (\mathtt{o} = \mathtt{t} \to \mathtt{o} = k) \wedge [\textsc{Take}]^r_\pi * [\textsc{Next}(\mathtt{t})]^r_1 * \boxed{\begin{array}{r}k \le \mathtt{t} < k * l.\mathtt{owner} \mapsto k * l.\mathtt{next} \mapsto k' \\ * \circledast k'' \ge k'.\, [\textsc{Next}(k'')]^r_1 * \mathrm{true}\end{array}}^r_{T(\mathtt{l},r)}\right\}$
$\mathtt{while}\ \ \mathtt{t} \neq \mathtt{o}\ \ \mathtt{do}$
$\langle \mathtt{o} := [\mathtt{l}.\mathtt{owner}]\rangle$
$\left\{\exists r, \pi.\, [\textsc{Take}]^r_\pi * [\textsc{Next}(\mathtt{t})]^r_1 * \boxed{\begin{array}{r}\exists k'.\, \mathtt{t} < k * l.\mathtt{owner} \mapsto \mathtt{t} * l.\mathtt{next} \mapsto k' \\ * \circledast k'' \ge k'.\, [\textsc{Next}(k'')]^r_1 * \mathrm{true}\end{array}}^r_{T(\mathtt{l},r)}\right\}$
$\left\{\exists r, \pi.\, [\textsc{Take}]^r_\pi * \boxed{\begin{array}{r}\exists k, k'.\, k \le k' * l.\mathtt{owner} \mapsto k * l.\mathtt{next} \mapsto k' \\ * \circledast k'' \ge k'.\, [\textsc{Next}(k'')]^r_1 * \mathrm{true}\end{array}}^r_{T(\mathtt{l},r)} * \exists r, k.\, [\textsc{Next}(k)]^r_1 * \boxed{\mathtt{l}.\mathtt{owner} \mapsto k * \mathrm{true}}^r_{T(\mathtt{l},r)}\right\}$
$\}$
$\{\mathsf{isLock}(\mathtt{l}) * \mathsf{Locked}(\mathtt{l})\}$

$\{\mathsf{Locked}(\mathtt{l})\}$
$\mathtt{unlock}(\mathtt{l})\ \{$
$\left\{\exists r, k.\, [\textsc{Next}(k)]^r_1 * \boxed{l.\mathtt{owner} \mapsto k * \mathrm{true}}^r_{T(l,r)}\right\}$
$\langle \mathtt{INCR}(\mathtt{l}.\mathtt{owner})\rangle$
$\left\{\exists r, k.\, \boxed{\mathtt{l}.\mathtt{owner} \mapsto k + 1 * [\textsc{Next}(k)]^r_1 * \mathrm{true}}^r_{T(l,r)}\right\}$
// Stabilise the assertion
$\{\mathrm{emp}\}$
$\}$
$\{\mathrm{emp}\}$

Figure 7.3: Proof outline for the ticketed lock (`lock` and `unlock`)

operation increments the lock's `owner` field, which it may do since it holds exclusive permission to the NEXT action for the current value of the field.

7.1.5 Disposable Locks

For simplicity of exposition, I have so far omitted the disposal of locks from the lock module specification and implementations. In languages that provide garbage collection, explicit disposal is generally unnecessary; however, the language used here does not provide garbage collection, and so explicit disposal of dynamically allocated resources is good practice.

As remarked previously, in order to dispose a lock, a thread should have exclusive permission to that lock: no other thread should have an isLock for the same lock. Were this not the case, some other thread could attempt to acquire the lock once it has been disposed, with disastrous consequences. The solution is to associate fractional permissions with isLock predicates, ensuring that the total permission is 1. The previous axiom for splitting isLock is now replaced with the following:

$$\mathsf{isLock}(l, \pi_1 + \pi_2) \leftrightarrow \mathsf{isLock}(l, \pi_1) * \mathsf{isLock}(l, \pi_2)$$
$$\mathsf{isLock}(l, \pi) \rightarrow 0 < \pi \leq 1.$$

It is now possible to specify the operation $\texttt{disposelock}(l, n)$ which disposes of the lock and n-cell memory block at l. The revised specifications for the lock module's functions are then as follows:

$$\{\mathsf{isLock}(\mathit{l}, \pi)\}\ \texttt{lock}(\mathit{l})\ \{\mathsf{isLock}(\mathit{l}, \pi) * \mathsf{Locked}(\mathit{l})\}$$
$$\{\mathsf{Locked}(\mathit{l})\}\ \texttt{unlock}(\mathit{l})\ \{\mathrm{emp}\}$$
$$\left\{\mathrm{emp}\right\}\ \texttt{makelock}(\boldsymbol{n})\ \left\{\begin{array}{l}\exists l.\, \mathrm{ret} = l \wedge \mathsf{isLock}(l, 1) * \mathsf{Locked}(l) \\ \quad * \, l \mapsto - * \cdots * (l + \boldsymbol{n} - 1) \mapsto -\end{array}\right\}$$
$$\left\{\begin{array}{l}\mathsf{isLock}(\mathit{l}, 1) * \mathsf{Locked}(\mathit{l}) * \\ \mathit{l} \mapsto - * \cdots * (\mathit{l} + \boldsymbol{n} - 1) \mapsto -\end{array}\right\}\ \texttt{disposelock}(\mathit{l}, \boldsymbol{n})\ \left\{\mathrm{emp}\right\}.$$

$\{\mathsf{isLock}(l, 1) * \mathsf{Locked}(l) * l \mapsto - * \cdots * (l + n - 1) \mapsto -\}$
`disposelock`(l, n) {
$\left\{ \exists r.\, [\textsc{Lock}]_1^r * [\textsc{Unlock}]_1^r * \boxed{(l-1) \mapsto 1}^{r}_{I(l,r)} * \right.$
$\left. l \mapsto - * \cdots * (l + n - 1) \mapsto - \right\}$
// Dispose shared lock region
$\{(l-1) \mapsto 1 * l \mapsto - * \cdots * (l + n - 1) \mapsto -\}$
`dispose`$(l - 1, n + 1)$
{emp}
}
{emp}

Figure 7.4: Proof outline for compare-and-swap lock (`disposelock`)

Remark. The lock must be held when disposing the lock to ensure that the Locked predicate cannot escape and permit the lock to be released after it has been disposed. A different specification could allow the lock to be disposed even when it is not held by the disposing thread; such a specification would have to guarantee that no other thread holds the Locked predicate. This could be achieved by having `lock` and `unlock` transform an $\mathsf{isLock}(l, \pi)$ into a $\mathsf{Locked}(l, \pi)$ and vice-versa. An axiom would ensure that the total permission on isLock and Locked predicates never exceeds 1, so the lock may be safely disposed with $\mathsf{isLock}(l, 1)$.

Implementation and Verification

For both of the lock modules considered in this section, the implementation and verification for `disposelock` are very similar. I only consider the compare-and-swap lock here, for which the implementation is as follows:

```
disposelock(l, n) {
  dispose(l − 1, n + 1)
}
```

An outline proof for the correctness of the implementation is given in Figure 7.4. The key proof step is the disposal of the shared region for the lock by repartitioning. Region disposal is the inverse of region creation: the thread starts with full permissions to all actions on the region and ends up with the contents of the shared region in its local state.

7.2 Composing Abstract Specifications

In the previous section, I demonstrated that concurrent abstract predicates can be used to present and verify abstract specifications for concurrent modules. In this section, I show how clients of such modules, which themselves may be modules with abstract specifications, can be verified using these abstract specifications. In particular, I specify an abstract set module and prove the correctness of two different implementations. Both of the implementations assume a lock module satisfying the specification given in §7.1.

7.2.1 Set Specification

Consider a simple concurrent module that implements abstract sets. The module has three operations: `contains`, which determines whether a particular value belongs to the set; `add`, which adds a specified value to the set; and `remove`, which removes a specified value from the set. These operations can be given the following abstract specifications:

$$\begin{array}{rcl}
\{\mathsf{in}(s,v)\} & \texttt{contains}(s,v) & \{\mathrm{ret} \wedge \mathsf{in}(s,v)\} \\
\{\mathsf{out}(s,v)\} & \texttt{contains}(s,v) & \{\neg\mathrm{ret} \wedge \mathsf{out}(s,v)\} \\
\{\mathsf{own}(s,v)\} & \texttt{add}(s,v) & \{\mathsf{in}(s,v)\} \\
\{\mathsf{own}(s,v)\} & \texttt{remove}(s,v) & \{\mathsf{out}(s,v)\}
\end{array}$$

Here $\mathsf{in}(s,v)$ is an abstract predicate asserting that the set identified by s contains the value v. Correspondingly, $\mathsf{out}(s,v)$ asserts that the set s does not contain v. As a shorthand, $\mathsf{own}(s,v) \stackrel{\text{def}}{=} \mathsf{in}(s,v) \vee \mathsf{out}(s,v)$.

These abstract predicates not only express knowledge about whether the value v is in the set s, but also assert the *exclusive right* to change whether v belongs to s. Consequently, $\mathsf{out}(s,v)$ is not simply the negation of $\mathsf{in}(s,v)$. The exclusivity of permissions is captured by the following module axiom:

$$\mathsf{own}(s,v) * \mathsf{own}(s,v) \rightarrow \mathrm{false}.$$

The module also exposes the stability of its abstract predicates with the following axioms:

$$\mathsf{in}(s,v) \rightarrow \boxed{\mathbf{R}}\mathsf{in}(s,v)$$
$$\mathsf{out}(s,v) \rightarrow \boxed{\mathbf{R}}\mathsf{out}(s,v).$$

The abstract predicates permit disjoint reasoning about values in the set, even though they may be implemented by a single shared structure; any underlying sharing is hidden from the client. For example, consider the following

program, which concurrently removes two values from a set:

$$\texttt{remove}(s, v_1) \parallel \texttt{remove}(s, v_2)$$

If values v_1 and v_2 are distinct then the program should successfully remove both from the set, as captured by the following proof outline:

$$\begin{array}{c} \{\mathsf{own}(s, v_1) * \mathsf{own}(s, v_2)\} \\ \begin{array}{c|c} \{\mathsf{own}(s, v_1)\} & \{\mathsf{own}(s, v_2)\} \\ \texttt{remove}(s, v_1) & \texttt{remove}(s, v_2) \\ \{\mathsf{out}(s, v_1)\} & \{\mathsf{out}(s, v_2)\} \end{array} \\ \{\mathsf{out}(s, v_1) * \mathsf{out}(s, v_2)\} \end{array}$$

7.2.2 A Coarse-grained Set Implementation

A simple way of implementing a concurrent set is to take a sequential set implementation and simply ensure that the set operations are invoked in mutual exclusion by using a single lock for each set. Such an implementation is *course-grained*, since it does not break down the operations into smaller components (which may be safely performed concurrently).

A coarse-grained concurrent set can be implemented as follows:

```
r := contains(s, v) {      add(s, v) {      remove(s, v) {
  lock(s);                   lock(s);         lock(s);
  r := scontains(s, v);      sadd(s, v);      sremove(s, v);
  unlock(s)                  unlock(s)        unlock(s)
}                          }                }
```

This implementation assumes that a sequential set module, providing the operations `scontains`, `sadd` and `sremove`. It also assumes that s, which identifies the set to the sequential set module, is a lockable block.

The sequential set module is assumed to provide an abstract predicate $\mathsf{Set}(s, vs)$ that the (sequential) set identified by s holds the set of values vs. The Set predicate cannot be split, and so can only be held by one thread at once, thereby enforcing sequential access. Furthermore, the sequential set operations are assumed to satisfy the following specifications:

$$\begin{array}{lll} \{\mathsf{Set}(s, vs)\} & \texttt{scontains}(s, v) & \{(\mathrm{ret} \leftrightarrow v \in vs) \wedge \mathsf{Set}(s, vs)\} \\ \{\mathsf{Set}(s, vs)\} & \texttt{sadd}(s, v) & \{\mathsf{Set}(s, vs \cup \{v\})\} \\ \{\mathsf{Set}(s, vs)\} & \texttt{sremove}(s, v) & \{\mathsf{Set}(s, vs \setminus \{v\})\} \end{array}$$

Interpretation of Abstract Predicates

Informally, the interpretation of the $\mathsf{in}(s, v)$ predicate comprises three things: knowledge that s is lockable — $\mathsf{isLock}(s)$; knowledge that the shared, sequential set at s contains the value v — $\boxed{\mathsf{P}_{\in}(s, v, r)}^{r}_{C(s,r)}$; and permission to acquire the sequential set in order to change whether or not it contains v — $[\textsc{Change}(v)]^r_1$. The $\mathsf{out}(s, v)$ has a similar interpretation. Formally, in and out are defined as follows:

$$\mathsf{in}(s, v) \equiv \exists r.\, \mathsf{isLock}(s) * [\textsc{Change}(v)]^r_1 * \boxed{\mathsf{P}_{\in}(s, v, r)}^{r}_{C(s,r)}$$
$$\mathsf{out}(s, v) \equiv \exists r.\, \mathsf{isLock}(s) * [\textsc{Change}(v)]^r_1 * \boxed{\mathsf{P}_{\notin}(s, v, r)}^{r}_{C(s,r)}$$

Since a thread must remove the sequential set from the shared state in order to perform any operation on it, the assertion $\boxed{\mathsf{P}_{\in}(s, v, r)}^{r}_{C(s,r)}$ cannot always ensure that the set is in the shared state. The assertion must, however, provide assurance that, if the set is missing then the lock must have been acquired, and when the set is finally returned it will contain the value v. The $\textsc{Ret}(vs)$ action returns the set to the shared state with the contents vs. If some other thread has removed the set, then it should only have permission $[\textsc{Ret}(vs)]^r_1$ provided that $v \in r$. This is reflected in the definition of $\mathsf{P}_{\in}(s, v, r)$, which is as follows:

$$\mathsf{P}_{\lhd}(s, v, r) \equiv \exists vs.\, v \lhd vs \wedge \left(\begin{array}{r} \big(\mathsf{allrets}(r) * \mathsf{Set}(s, vs)\big) \vee \\ \big(\mathsf{Locked}(s) * ([\textsc{Ret}(vs)]^r_1 \mathrel{-\!\!*} \mathsf{allrets}(s))\big) \end{array} \right)$$
$$\text{where } \lhd = \in \text{ or } \lhd = \notin.$$

$$\mathsf{allrets}(r) \equiv \circledast ws.\, [\textsc{Ret}(ws)]^r_1$$

The actions for the coarse-grained set are as follows:

$$C(s, r) \stackrel{\text{def}}{=} \left(\begin{array}{lll} \textsc{Change}(v) : & \left(\begin{array}{r} \exists vs, ws.\, \mathsf{Set}(s, vs) * \\ [\textsc{Ret}(ws)]^r_1 \wedge \\ vs \setminus \{v\} = ws \setminus \{v\} \end{array} \right) & \rightsquigarrow \mathsf{Locked}(s), \\ \textsc{Ret}(ws) & : \mathsf{Locked}(s) \rightsquigarrow \mathsf{Set}(s, ws) * [\textsc{Ret}(ws)]^r_1 & \end{array} \right)$$

The $\textsc{Change}(v)$ action is performed by a thread in order to acquire the shared, sequential set in order to (at most) change it with respect to whether it contains the value v. In order to legitimately acquire the sequential set, the thread must already hold the lock; this is enforced by the fact that the thread must give up the $\mathsf{Locked}(s)$ predicate in performing the action. In exchange, the thread acquires the set $\mathsf{Set}(s, vs)$, whose contents is some vs, together with permission $[\textsc{Ret}(ws)]^r_1$ to return the set with contents ws, which differs from vs only as to whether v is in the set.

The $\textsc{Ret}(ws)$ action is performed by a thread in order to return the set $\mathsf{Set}(s, ws)$ to the shared state. It also returns the permission $[\textsc{Ret}(ws)]^r_1$ in doing so, but reacquires the $\mathsf{Locked}(s)$ predicate, enabling it to release the lock.

The predicate axiom $\mathsf{own}(s, v) * \mathsf{own}(s, v) \rightarrow \text{false}$ holds by the fact that $\mathsf{own}(s, v)$ entails exclusive permission $[\textsc{Change}(v)]^r_1$ to alter the set with respect to the value v, and two such permissions cannot be combined.

Remark. The above argument is subtly flawed. The assertion $\exists r.\, \mathsf{P}_{\in}(s, v, r)$ is not precise, so it is possible to have two $\mathsf{in}(s, v)$ predicates that refer to different shared regions! One way of fixing this would be to use the disposable lock specification and include the predicate $\exists \pi.\, \pi > 0.5 \wedge \mathsf{isLock}(s, \pi)$ in the shared region at all times. This predicate is precise, since the total permission cannot exceed 1, so two own predicates for s must always refer to the same shared region.

The stability axiom for $\mathsf{in}(s, v)$ holds because the only actions available to other threads are $\textsc{Change}(w)$ for some $w \neq v$, and $\textsc{Ret}(vs)$ for some vs with $v \in vs$. The assertion $\mathsf{in}(s, v)$ is stable under both of these actions: $\textsc{Change}(w)$ requires the disjunct $\mathsf{allrets}(r) * \mathsf{Set}(s, vs)$ to hold and leaves the disjunct $\mathsf{Locked}(s) * ([\textsc{Ret}(vs)]^r_1 \mathrel{-\!\!*} \mathsf{allrets}(s))$ holding; $\textsc{Ret}(vs)$ does the reverse. A similar argument holds for the stability of $\mathsf{out}(s, v)$.

Verifying the Implementation

Given the interpretations of the inductive predicates above, the coarse-grained set implementation can be verified against the set specification. Figure 7.5 gives an outline proof of the `add` operation when the value to be added is initially absent from the set; the other cases have similar proofs.

The `add` operation first acquires $\mathsf{Locked}(s)$ by calling `lock`. It is now able to invoke the $\textsc{Change}(v)$ action to swap the $\mathsf{Locked}(s)$ predicate for the set $\mathsf{Set}(s, vs)$ and permission $[\textsc{Ret}(vs \cup \{v\})]^r_1$ to return it having only added v to the set. Once v is added to the set, the $\textsc{Ret}$ action is used to return it (and the permission to return it) to the shared region, recovering the $\mathsf{Locked}(s)$ predicate. Finally, the lock is released and the shared set continues to contain the value v.

7.2.3 A Lock-coupling List Implementation

The coarse-grained set implementation guards access to a shared set with a single lock, forcing threads to use the set in mutual exclusion. I now consider

$\{\mathsf{out}(s, v)\}$
$\mathtt{add}(s, v)$ {
$\left\{\exists r.\, \mathsf{isLock}(s) * [\textsc{Change}(v)]_1^r * \boxed{\mathsf{P}_{\notin}(s, v, r)}^r_{C(s,r)}\right\}$
$\mathtt{lock}(s)$;
$\left\{\exists r.\, \mathsf{isLock}(s) * \mathsf{Locked}(s) * [\textsc{Change}(v)]_1^r * \boxed{\mathsf{P}_{\notin}(s, v, r)}^r_{C(s,r)}\right\}$
// use Change *to extract* Set *predicate and* Ret *permission*
$\left\{\begin{array}{l}\exists r, vs.\, \mathsf{isLock}(s) * \mathsf{Set}(s, vs) * [\textsc{Ret}(vs \cup \{v\})]_1^r * [\textsc{Change}(v)]_1^r * \\ \qquad\qquad \boxed{\mathsf{Locked}(s) * ([\textsc{Ret}(vs \cup \{v\})]_1^r \mathrel{-\!\!*} \mathsf{allrets}(s))}^r_{C(s,r)}\end{array}\right\}$
$\mathtt{sadd}(s, v)$;
$\left\{\begin{array}{l}\exists r, vs.\, \mathsf{isLock}(s) * \mathsf{Set}(s, vs \cup v) * [\textsc{Ret}(vs \cup \{v\})]_1^r * [\textsc{Change}(v)]_1^r * \\ \qquad\qquad \boxed{\mathsf{Locked}(s) * ([\textsc{Ret}(vs \cup \{v\})]_1^r \mathrel{-\!\!*} \mathsf{allrets}(s))}^r_{C(s,r)}\end{array}\right\}$
// use Ret *to return* Set *predicate and* Ret *permission*
$\left\{\exists r.\, \mathsf{isLock}(s) * \mathsf{Locked}(s) * [\textsc{Change}(v)]_1^r * \boxed{\mathsf{P}_{\in}(s, v, r)}^r_{C(s,r)}\right\}$
$\mathtt{unlock}(s)$
$\left\{\exists r.\, \mathsf{isLock}(s) * [\textsc{Change}(v)]_1^r * \boxed{\mathsf{P}_{\in}(s, v, r)}^r_{C(s,r)}\right\}$
}
$\{\mathsf{in}(s, v)\}$

Figure 7.5: Proof outline for the coarse-grained set (`add` — `out` case)

$E.\texttt{val} \stackrel{\text{def}}{=} E$

$E.\texttt{next} \stackrel{\text{def}}{=} E + 1$

```
p, c := locate(s, v) {
  p := s;
  lock(p);
  c := [p.next];
  while [c.val] < v do
    lock(c);
    unlock(p);
    p := c;
    c := [p.next]
}

r := contains(s, v) {
  local p, c in
    p, c := locate(s, v);
    r := ([c.val] = v);
    unlock(p)
}
```

```
add(s, v) {
  local p, c, z in
    p, c := locate(s, v);
    if [c.val] ≠ v then
      z := makelock(2);
      unlock(z);
      [z.value] := v;
      [z.next] := c;
      [p.next] := z   ;
    unlock(p)
}

remove(s, v) {
  local p, c, z in
    p, c := locate(s, v);
    if [c.val] = v then
      lock(c);
      z := [c.next];
      [p.next] := z;
      disposelock(c, 2)   ;
    unlock(p)
}
```

Figure 7.6: Lock-coupling list implementation of the set module

an implementation that uses a finer granularity of locking: a lock-coupling list.

The lock-coupling list implements a set with a sorted linked-list whose nodes each record a value belonging to the set. Each of these nodes can be locked individually, so multiple threads can access the set concurrently, holding locks to different parts of the list. The code for the algorithm (adapted from [HS08, §9.5]) is given in Figure 7.6.

The three module functions use the function `locate`(s, v), which traverses the list from the head node, until it reaches the position for a node holding value v — the return value c is either the node with value v, if present, or the node with the smallest value greater than v that is in the list; the return value p is c's predecessor. To ensure that every value has a successor and predecessor node, the first and last nodes of the list are special dummy nodes that do not represent actual values held in the set; they have values $-\infty$ and ∞ respectively. The traversal begins by locking the head node, which resides at address s and proceeds to along the list performing hand-over-hand locking. That is, the node following the currently-locked node is locked, then the previously-held lock is released. When `locate` returns, the lock on p is held by the thread, but the lock on c is not.

No thread accesses a node that is locked by another thread, so it cannot traverse past a locked node. Consequently, a thread cannot overtake any other threads that are accessing the list. Nodes can be added to and removed from locked segments of the list: if a thread holds the lock to a node then it can insert a new node immediately after it, providing this new node preserves the sorted nature of the list; if a thread holds the locks on two sequential nodes then it can remove the second of them from the list (and subsequently dispose of it).

The algorithm relies on and ensures a number of invariant properties of the nodes in the list.

- The head node is always in the list.
- If a node is in the list and is locked by a thread, no other thread can remove the node from the list or change which node is its successor. (Consequently, the successor cannot be removed by another thread.)
- While a node is in the list, its value remains invariant.

Interpretation of Abstract Predicates

The interpretation of the $\mathsf{in}(s, v)$ and $\mathsf{out}(s, v)$ predicates at a high level resembles the interpretation for the coarse-grained implementation: it includes

permission to acquire a lock for the set, permission to change the set with respect to whether v is present, and knowledge that the set represented in the shared state includes or does not include the value v.

$$\mathsf{in}(s,v) \equiv \exists r,\pi.\, \mathsf{isLock}(s,\pi) * [\textsc{Change}(v)]_1^r * \boxed{\mathsf{L}_{\in}(s,v,r)}_{F(s,r)}^r$$
$$\mathsf{out}(s,v) \equiv \exists r,\pi.\, \mathsf{isLock}(s,\pi) * [\textsc{Change}(v)]_1^r * \boxed{\mathsf{L}_{\notin}(s,v,r)}_{F(s,r)}^r$$

The difference is that the lock only controls access to the first node, the change permission allows the thread to remove nodes individually (as opposed to the whole set at once) in order to manipulate them, and the shared state comprises a list with gaps, where other threads have taken nodes into their shared states. In order to elaborate this definition, I first define predicates for individual list nodes.

$$\mathsf{val}(a,v) \equiv a.\mathtt{val} \mapsto v$$
$$\mathsf{link}(a,b) \equiv a.\mathtt{next} \mapsto b * \mathsf{isLock}(b,1)$$
$$\mathsf{node}(a,b,v) \equiv \mathsf{val}(a,v) * \mathsf{link}(a,b)$$

The $\mathsf{node}(a,b,v)$ predicate comprises two components: the value field of node a, which contains v, represented by the $\mathsf{val}(a,v)$ predicate; and the link from node a to its successor, b, represented by $\mathsf{link}(a,b)$. When a thread owns a node it has exclusive permission to lock that node's successor — this is the nature of hand-over-hand locking — therefore the link component of a node comprises the isLock permission for its successor. The val component of a node remains in the shared region while that value is still in the set, whereas the link component is (temporarily) removed from the shared region by the thread that holds the lock on the node.

The actions for the fine-grained set module are defined by the following interference assertion:

$$F(r) \stackrel{\text{def}}{=}$$
$$\left(\begin{array}{l}
\textsc{Change}(v):\ \exists n,t.\,([\textsc{Ret}(n,t,v)]_1^r * \mathsf{link}(n,t) \rightsquigarrow \mathsf{Locked}(n)) \\
\textsc{Ret}(n,t,v): \\
\quad \left\{\begin{array}{rcl}
\mathsf{Locked}(n) & \rightsquigarrow & \mathsf{link}(n,t) * [\textsc{Ret}(n,t,v)]_1^r \\
\exists v_1,v_2.\begin{pmatrix}\mathsf{Locked}(n)\\ *\,\mathsf{val}(n,v_1)\\ *\,\mathsf{val}(t,v_2)\end{pmatrix} & \rightsquigarrow & \begin{pmatrix}\exists y.\,[\textsc{Ret}(n,t,v)]_1^r * \mathsf{node}(y,t,v)\\ *\,\mathsf{link}(n,y) * \mathsf{val}(n,v_1)\\ *\,\mathsf{val}(t,v_2) \wedge v_1 < v < v_2\end{pmatrix} \\
\begin{pmatrix}\mathsf{Locked}(n) *\\ \mathsf{Locked}(t) * \mathsf{val}(t,v)\end{pmatrix} & \rightsquigarrow & \begin{pmatrix}\exists y.\,[\textsc{Ret}(n,t,v)]_1^r *\\ [\textsc{Ret}(t,y,v)]_1^r * \mathsf{link}(n,y)\end{pmatrix}
\end{array}\right.
\end{array}\right)$$

The CHANGE and RET actions are analogous to their counterparts for the coarse-grained set in that they allow a thread to remove part of the shared state by holding a lock and to return that part with some limited modification. Rather than removing the entire set, only a single link of the list is removed with each lock. Both CHANGE and RET are parameterised by v, the value that the thread may add or remove from the list.

The CHANGE(v) action allows a thread, having locked some node n, to take the link of that node (that is, the `next` pointer and the permission to lock node's successor) from the shared state, together with a permission that will allow the thread to return that link, possibly having added or removed a node with value v; in exchange, the thread gives up the $\mathsf{Locked}(n)$ predicate.

The three RET(n, t, v) actions deal with returning the link of node n to the shared state. The first simply allows the same linked to be returned as was originally removed. In returning the link, the thread also gives up the RET permission, but regains the $\mathsf{Locked}(n)$ predicate. This action is used to traverse parts of the list that the thread does not change.

The second allows a link from n to t to be replaced with a link from n to a new node at some address y, having value v, that in turn links to the node at t. The action requires that v falls between the values of the nodes at n and t, in order to preserve the sorted order of the list. As before, the thread gives up the RET permission and regains the $\mathsf{Locked}(n)$ predicate. This action is used to add a new the value v to the set, in the form of the node y.

The third action allows the link from n to t to be replaced with one from n to y, where there was formally a link from t to y. To do this, the thread must also have locked the node at t and obtained its corresponding RET permission and $\mathsf{link}(t, y)$ from the shared state. Node t must also have the value v; in performing the action, the value component of t is removed from the shared state, and its link component is not returned. The RET permissions for both of the nodes are returned to the shared state, and both the lock predicates $\mathsf{Locked}(n)$ and $\mathsf{Locked}(t)$ are regained by the thread. This action is used to remove the value v from the set, by removing the node t, which has that value.

The predicates $\mathsf{L}_{\in}(s, v, r)$ and $\mathsf{L}_{\notin}(s, v, r)$ describe lists containing and not containing value v. These lists can have gaps in where threads have removed links into their local states. The list captured by the slsg (for 'sorted list segment with gaps') predicate. The predicates also assert the presence of all RET permissions, except for ones corresponding to gaps in the list, which is captured by the rets predicate. For both the slsg and rets predicates, the carrier set S tracks the missing links from the list. None of the missing RET permissions can

$$\mathsf{lsg}(x, y, S, \varnothing) \equiv x = y \wedge S = \emptyset \wedge \mathrm{emp}$$

$$\mathsf{lsg}(x, y, S \uplus \{(x, z)\}, v \cdot vs) \equiv \mathsf{Locked}(x) * \mathsf{val}(x, v) * \mathsf{lsg}(z, y, S, vs)$$

$$\mathsf{lsg}(x, y, S, v \cdot vs) \equiv x \neq y \wedge \mathsf{node}(x, v, z) * \mathsf{lsg}(z, y, S, vs)$$

$$\mathsf{slsg}(x, y, S, vs) \equiv \mathsf{lsg}(x, y, S, vs) \wedge \mathsf{sorted}(vs) \wedge \exists vs'.\, vs = -\infty \cdot vs' \cdot \infty$$

$$\begin{aligned}\mathsf{rets}(S, r) \equiv {} & \circledast(x, y) \notin S.\, \circledast v.\, [\textsc{Ret}(x, y, v)]_1^r * \\ & \circledast(x, y) \in S.\, \exists w.\, \circledast v \neq w.\, [\textsc{Ret}(x, y, v)]_1^r \wedge \\ & \forall x, y, w, z.\, (x, y) \in S \wedge (w, z) \in S \rightarrow (x = w \leftrightarrow y = z)\end{aligned}$$

$$\mathsf{myrets}(v, r) \equiv \circledast x, y.\, [\textsc{Ret}(x, y, v)]_1^r * \mathrm{true}$$

$$\mathsf{L}_\triangleleft(s, v, r) \equiv \exists vs, S.\, v \triangleleft vs \wedge \mathsf{slsg}(s, \mathit{nil}, S, vs) * (\mathsf{rets}(S, r) \wedge \mathsf{myrets}(v, r))$$

where $\triangleleft = \in$ or $\triangleleft = \notin$

Figure 7.7: Auxiliary predicates for lock-coupling list

correspond to the value v, to ensure that no other thread will change whether v is in the list; this is captured by the myrets predicate. These predicates are formally defined in Figure 7.7.

The abstract predicate axiom $\mathsf{own}(s, v) * \mathsf{own}(s, v) \rightarrow \mathrm{false}$ follows from the fact that exclusive permissions to $\textsc{Change}(v)$ cannot be combined.

To check that stability axioms for in and out, observe that when the environment performs the $\textsc{Change}(v')$ action, it simply corresponds to adding (n, t) to the carrier set S. The first $\textsc{Ret}(n, t, v')$ action corresponds to removing (n, t) from the set S. The second $\textsc{Ret}(n, t, v')$ action corresponds to removing (n, t) from S and also inserting v' in the list vs. The third $\textsc{Ret}(n, t, v')$ action corresponds to removing (n, t) and (t, y) from S and v' from the list vs. Since it must be that $v' \neq v$ in each of these cases, the predicates are stable.

Verifying the Implementation

Given the interpretations of the abstract predicates above, the fine-grained set implementation can be verified against the set specification. Figures 7.8 and 7.9 give an outline proof of the `locate` function, while Figure 7.10 gives an outline

$\{\mathsf{in}(s, v)\}$

$p, c := \texttt{locate}(s, v)\ \{$

$\left\{ \exists r.\ \boxed{\exists vs, S.\ v \in vs \wedge \mathsf{slsg}(s, \mathit{nil}, S, vs) * (\mathsf{rets}(S, r) \wedge \mathsf{myrets}(v, r))}^{r}_{F(r)} * \exists \pi.\ \mathsf{isLock}(s, \pi) * [\textsc{Change}(v)]^r_1 \right\}$

$p := s;$

$\texttt{lock}(p);$

$\left\{ \exists r.\ \boxed{\exists vs, S.\ v \in vs \wedge \mathsf{slsg}(s, \mathit{nil}, S, vs) * (\mathsf{rets}(S, r) \wedge \mathsf{myrets}(v, r))}^{r}_{F(r)} * \exists \pi.\ \mathsf{isLock}(s, \pi) * \mathsf{Locked}(s) * [\textsc{Change}(v)]^r_1 \wedge p = s \right\}$

// By $\textsc{Change}(v)$ *using* $\mathsf{Locked}(s)$

$\left\{ \exists r, z.\ \boxed{\begin{array}{l}\exists vs, S.\ v \in vs \wedge \{(s, z)\} \subseteq S \wedge \mathsf{slsg}(s, \mathit{nil}, S, vs) \\ * \left(\mathsf{rets}(S, r) \wedge ([\textsc{Ret}(s, z, v)]^r_1 \mathrel{-\!\!*} \mathsf{myrets}(v, r))\right)\end{array}}^{r}_{F(r)} * \mathsf{link}(s, z) * [\textsc{Ret}(s, z, v)]^r_1 * \exists \pi.\ \mathsf{isLock}(s, \pi) * [\textsc{Change}(v)]^r_1 \wedge p = s \right\}$

$c := [p.\texttt{next}];$

$\left\{ \exists r.\ \boxed{\begin{array}{l}(\exists v'.\ \mathsf{val}(c, v') * \mathrm{true}) \wedge \\ \quad \exists vs, S.\ v \in vs \wedge \{(p, c)\} \subseteq S \wedge \mathsf{slsg}(s, \mathit{nil}, S, vs) \\ \qquad * (\mathsf{rets}(S, r) \wedge ([\textsc{Ret}(p, c, v)]^r_1 \mathrel{-\!\!*} \mathsf{myrets}(v, r)))\end{array}}^{r}_{F(r)} * \mathsf{link}(p, c) * [\textsc{Ret}(p, c, v)]^r_1 * \exists \pi.\ \mathsf{isLock}(s, \pi) * [\textsc{Change}(v)]^r_1 \right\}$

$\texttt{while}\ [c.\texttt{val}] < v\ \texttt{do}$

$\ldots$ *// see Figure 7.9*

$\}$

$\left\{ \exists r.\ \boxed{\begin{array}{l}(\exists v'.\ \mathsf{val}(p, v') * v' < v) \wedge (\exists v''.\ \mathsf{val}(c, v'') * v'' \geq v) \\ \quad \exists vs, S.\ v \in vs \wedge \{(p, c)\} \subseteq S \wedge \mathsf{slsg}(s, \mathit{nil}, S, vs) \\ \qquad * \left(\mathsf{rets}(S, r) \wedge ([\textsc{Ret}(p, c, v)]^r_1 \mathrel{-\!\!*} \mathsf{myrets}(v, r))\right)\end{array}}^{r}_{F(r)} * \mathsf{link}(p, c) * [\textsc{Ret}(p, c, v)]^r_1 * \exists \pi.\ \mathsf{isLock}(s, \pi) * [\textsc{Change}(v)]^r_1 \right\}$

Figure 7.8: Proof outline for `locate`

$$\left\{\begin{array}{l}\exists r.\ \boxed{\begin{array}{l}(\exists v'.\,\mathsf{val}(c,v')*v'<v)\wedge\\ \quad \exists vs, S.\, v\in vs\wedge\{(p,c)\}\subseteq S\wedge \mathsf{slsg}(s,\mathit{nil},S,vs)\\ \qquad *\,(\mathsf{rets}(S,r)\wedge([\textsc{Ret}(p,c,v)]_1^r\mathrel{-\!\!*}\mathsf{myrets}(v,r)))\end{array}}^{r}_{F(r)}\\ \quad *\,\mathsf{link}(p,c)*[\textsc{Ret}(p,c,v)]_1^r*\exists\pi.\,\mathsf{isLock}(s,\pi)*[\textsc{Change}(v)]_1^r\end{array}\right\}$$

`lock(`c`);`

$$\left\{\begin{array}{l}\exists r.\ \boxed{\begin{array}{l}(\exists v'.\,\mathsf{val}(c,v')*v'<v)\wedge\\ \quad \exists vs, S.\, v\in vs\wedge\{(p,c)\}\subseteq S\wedge \mathsf{slsg}(s,\mathit{nil},S,vs)\\ \qquad *\,(\mathsf{rets}(S,r)\wedge([\textsc{Ret}(p,c,v)]_1^r\mathrel{-\!\!*}\mathsf{myrets}(v,r)))\end{array}}^{r}_{F(r)}\\ \quad *\,\mathsf{link}(p,c)*[\textsc{Ret}(p,c,v)]_1^r*\exists\pi.\,\mathsf{isLock}(s,\pi)*\\ \qquad [\textsc{Change}(v)]_1^r*\mathsf{Locked}(s)\end{array}\right\}$$

// By $\textsc{Change}(v)$ *using* $\mathsf{Locked}(c)$

$$\left\{\begin{array}{l}\exists r,z.\ \boxed{\begin{array}{l}(\exists v'.\,\mathsf{val}(c,v')*v'<v)\wedge\exists vs, S.\, v\in vs\wedge\\ \{(p,c),(c,z)\}\subseteq S\wedge\mathsf{slsg}(s,\mathit{nil},S,vs)*\big(\mathsf{rets}(S,r)\wedge\\ \quad([\textsc{Ret}(p,c,v)]_1^r*[\textsc{Ret}(c,z,v)]_1^r\mathrel{-\!\!*}\mathsf{myrets}(v,r))\big)\end{array}}^{r}_{F(r)}\\ \quad *\,\mathsf{link}(p,c)*[\textsc{Ret}(p,c,v)]_1^r*\exists\pi.\,\mathsf{isLock}(s,\pi)*\\ \qquad [\textsc{Change}(v)]_1^r*[\textsc{Ret}(c,z,v)]_1^r*\mathsf{link}(c,z)\end{array}\right\}$$

// By $\textsc{Ret}(p,c,v)$ *(first case)*

$$\left\{\begin{array}{l}\exists r,z.\ \boxed{\begin{array}{l}(\exists v'.\,\mathsf{val}(c,v')*v'<v)\wedge\exists vs, S.\, v\in vs\wedge\\ \{(c,z)\}\subseteq S\wedge\mathsf{slsg}(s,\mathit{nil},S,vs)*\big(\mathsf{rets}(S,r)\wedge\\ \quad([\textsc{Ret}(c,z,v)]_1^r\mathrel{-\!\!*}\mathsf{myrets}(v,r))\big)\end{array}}^{r}_{F(r)}\\ \quad *\,\mathsf{Locked}(p)*\exists\pi.\,\mathsf{isLock}(s,\pi)*\\ \qquad [\textsc{Change}(v)]_1^r*[\textsc{Ret}(c,z,v)]_1^r*\mathsf{link}(c,z)\end{array}\right\}$$

`unlock(`p`);`

$p := c$;

$c :=$ `[`p`.next]`

$$\left\{\begin{array}{l}\exists r.\ \boxed{\begin{array}{l}(\exists v'.\,\mathsf{val}(p,v')*v'<v)\wedge\\ \exists vs, S.\, v\in vs\wedge\{(p,c)\}\subseteq S\wedge\mathsf{slsg}(s,\mathit{nil},S,vs)\\ \quad *\,\big(\mathsf{rets}(S,r)\wedge([\textsc{Ret}(p,c,v)]_1^r\mathrel{-\!\!*}\mathsf{myrets}(v,r))\big)\end{array}}^{r}_{F(r)}\\ *\,\mathsf{link}(p,c)*[\textsc{Ret}(p,c,v)]_1^r*\exists\pi.\,\mathsf{isLock}(s,\pi)*[\textsc{Change}(v)]_1^r\end{array}\right\}$$

Figure 7.9: Proof outline for `locate` loop body

$\{\mathsf{in}(s, v)\}$
$\texttt{remove}(s, v)$ {
  $\texttt{local } p, c, z \texttt{ in}$
    $p, c := \texttt{locate}(s, v);$
    $\left\{ \exists r. \boxed{\begin{array}{c}(\exists v'.\, \mathsf{val}(p, v') * v' < v) \wedge (\exists v''.\, \mathsf{val}(c, v'') * v'' \geq v) \wedge \\ \exists vs, S.\, v \in vs \wedge \{(p, c)\} \subseteq S \wedge \mathsf{slsg}(s, \mathit{nil}, \mathcal{S}, vs) \\ * \left(\mathsf{rets}(S, r) \wedge ([\textsc{Ret}(p, c, v)]_1^r \mathrel{-\!\!*} \mathsf{myrets}(v, r))\right)\end{array}}^{r}_{F(r)} * \mathsf{link}(p, c) * [\textsc{Ret}(p, c, v)]_1^r * \exists \pi.\, \mathsf{isLock}(s, \pi) * [\textsc{Change}(v)]_1^r \right\}$
    $\texttt{if } [c.\texttt{val}] = v \texttt{ then}$
      $\texttt{lock}(c);$
      $\left\{ \exists r. \boxed{\begin{array}{c}(\mathsf{val}(c, v) * \mathrm{true}) \wedge \\ \exists vs, S.\, v \in vs \wedge \{(p, c)\} \subseteq S \wedge \mathsf{slsg}(s, \mathit{nil}, \mathcal{S}, vs) \\ * \left(\mathsf{rets}(S, r) \wedge ([\textsc{Ret}(p, c, v)]_1^r \mathrel{-\!\!*} \mathsf{myrets}(v, r))\right)\end{array}}^{r}_{F(r)} * \mathsf{link}(p, c) * [\textsc{Ret}(p, c, v)]_1^r * \exists \pi.\, \mathsf{isLock}(s, \pi) * [\textsc{Change}(v)]_1^r * \mathsf{Locked}(c) \right\}$
      // *By* $\textsc{Change}(v)$ *using* $\mathsf{Locked}(c)$
      $\left\{ \exists r, z. \boxed{\begin{array}{c}(\mathsf{val}(c, v) * \mathrm{true}) \wedge \exists vs, S.\, v \in vs \wedge \{(p, c), (c, z)\} \subseteq S \\ \wedge \mathsf{slsg}(s, \mathit{nil}, \mathcal{S}, vs) * \left(\mathsf{rets}(S, r) \wedge \right. \\ \left. ([\textsc{Ret}(p, c, v)]_1^r * [\textsc{Ret}(c, z, v)]_1^r \mathrel{-\!\!*} \mathsf{myrets}(v, r))\right)\end{array}}^{r}_{F(r)} * \mathsf{link}(p, c) * [\textsc{Ret}(p, c, v)]_1^r * \exists \pi.\, \mathsf{isLock}(s, \pi) * [\textsc{Change}(v)]_1^r * [\textsc{Ret}(c, z, v)]_1^r * \mathsf{link}(c, z) \right\}$
      $z := [c.\texttt{next}]\,; [p.\texttt{next}] := z;$
      $\left\{ \exists r. \boxed{\begin{array}{c}(\mathsf{val}(c, v) * \mathrm{true}) \wedge \exists vs, S.\, v \in vs \wedge \{(p, c), (c, z)\} \subseteq S \\ \wedge \mathsf{slsg}(s, \mathit{nil}, \mathcal{S}, vs) * \left(\mathsf{rets}(S, r) \wedge \right. \\ \left. ([\textsc{Ret}(p, c, v)]_1^r * [\textsc{Ret}(c, z, v)]_1^r \mathrel{-\!\!*} \mathsf{myrets}(v, r))\right)\end{array}}^{r}_{F(r)} * \mathsf{link}(p, z) * [\textsc{Ret}(p, c, v)]_1^r * \exists \pi.\, \mathsf{isLock}(s, \pi) * [\textsc{Change}(v)]_1^r * [\textsc{Ret}(c, z, v)]_1^r * \mathsf{isLock}(c, 1) * c.\texttt{next} \mapsto z \right\}$
      // *By* $\textsc{Ret}(p, c, v)$ *(third case)*
      $\left\{ \exists r. \boxed{\exists vs, S.\, v \notin vs \wedge \mathsf{slsg}(s, \mathit{nil}, \mathcal{S}, vs) * \left(\mathsf{rets}(S, r) \wedge \mathsf{myrets}(v, r)\right)}^{r}_{F(r)} * \mathsf{Locked}(p) * \exists \pi.\, \mathsf{isLock}(s, \pi) * [\textsc{Change}(v)]_1^r * \mathsf{isLock}(c, 1) * c.\texttt{next} \mapsto z * \mathsf{Locked}(c) * c.\texttt{value} \mapsto v \right\}$
      $\texttt{disposelock}(c, 2)$;
    $\left\{ \exists r. \boxed{\exists vs, S.\, v \notin vs \wedge \mathsf{slsg}(s, \mathit{nil}, \mathcal{S}, vs) * \left(\mathsf{rets}(S, r) \wedge \mathsf{myrets}(v, r)\right)}^{r}_{F(r)} * \mathsf{Locked}(p) * \exists \pi.\, \mathsf{isLock}(s, \pi) * [\textsc{Change}(v)]_1^r \right\}$
    $\texttt{unlock}(p)$
}
$\{\mathsf{out}(s, v)\}$

Figure 7.10: Proof outline for the fine-grained set (`remove` — in case)

proof of the `remove` operation in the case where v is in the set.

The main loop of `locate` searches through the list checking if the current element, c, has value less than the value v being searched for. At the start of the loop body, the thread has locked p and used the $\textsc{Change}(v)$ action to acquire $\mathsf{link}(p, c)$ and full permission $[\textsc{Ret}(p, c, v)]^r_1$. The first step of the loop locks the current element c, which is permitted because the link predicate incorporates $\mathsf{isLock}(c, 1)$. The $\textsc{Change}(v)$ action is then invoked to swap this lock for the link component of node c and the corresponding $\textsc{Ret}$ permission. At this point, the thread holds locks to two nodes, both of which precede the location being sought, so the first can be unlocked. To allow this to happen, the first $\textsc{Ret}(p, c, v)$ is invoked to return the link of the p node and the $\textsc{Ret}$ permission in exchange for the necessary $\mathsf{Locked}(p)$ permission. The p and c pointers are then advanced along the list.

As with the coarse-grained set, locking and unlocking are two-step processes. First the lock is acquired in local state, then a permission is used to retrieve the resource that is guarded by the lock from the shared state, giving up permission to release the lock in exchange. To release the lock, the guarded resource is returned to the shared state in exchange for the permission to release the lock, which is then unlocked locally.

Once the `locate` function is complete, node p is locked and has a value less then v, while its successor, c, has a value greater than or equal to v. From here, the `remove` operation checks that c holds the value v, which it must if v is in the set (since the set is sorted). In order to remove c, the node is first locked and the $\textsc{Change}$ action invoked to acquire its link from the shared state. Node p's link is then updated to point past c, to z. The third $\textsc{Ret}(p, c, v)$ action is used to return the link to the state, together with the $\textsc{Ret}$ permissions, and recover the $\mathsf{Locked}(p)$, $\mathsf{Locked}(c)$ and $\mathsf{val}(c, v)$ predicates. The thread now has sufficient resources to safely dispose the c node, which it does before finally unlocking p. At this point, it is assured that the value v has been removed from the set.

I omit the other proof cases here, which follow along the intuition behind the algorithm in a similar fashion to the remove case.

7.3 Semantics and Soundness

In this section I formalise the semantic basis and proof system that I have informally introduced. This culminates in a proof of the soundness of the concurrent abstract predicate proof system.

7.3.1 State Model

The state model for the logic — the set of *worlds*, World — is an abstraction of machine states that represents the view that a particular thread — the *subject thread* or simply *subject* — has of the state. As well as (partially) describing the actual state of the machine's memory, worlds also constrain the future evolution of the state, terms of both how the subject thread and other threads — the *environment* — can modify the state.

Worlds consist of three components: the *local state*, which can be accessed and updated exclusively by the subject thread, the *shared state*, which can be accessed and manipulated by both the subject and environment in accordance with permissions held, and the *action model*, which determines exactly what updates are permissible on the shared state. The local state component is from the set of logical states, LState (Definition 7.7), the shared state component is from the set of shared states, SState (Definition 7.2), and the action model component is from the set of action models, AMod (Definition 7.3). Worlds obey a well-formedness condition, wf, that ensures that the different portions of state are consistent with each other. I defer the formal definition of well-formedness (Definition 7.10).

Definition 7.1 (World)**.** The set of *worlds*, World, ranged over by $w, w', w_1, \ldots$, is defined as follows:

$$\mathsf{World} \stackrel{\text{def}}{=} \{(l, s, \zeta) \in \mathsf{LState} \times \mathsf{SState} \times \mathsf{AMod} \mid wf(l, s, \zeta)\}\,.$$

For world $w = (l, s, \zeta) \in \mathsf{World}$, $w_{\mathrm{L}} = l \in \mathsf{LState}$ stands for the local state, $w_{\mathrm{S}} = s \in \mathsf{SState}$ stands for the shared state, and $w_{\mathrm{A}} = \zeta \in \mathsf{AMod}$ stands for the action model.

A triple $(l, s, \zeta) \in \mathsf{LState} \times \mathsf{SState} \times \mathsf{AMod}$ that is not necessarily a well-formed world is called a *pre-world*.

Shared states are further divided into *regions*. This division into regions supports encapsulation: each shared object typically inhabits its own region, and the interference on that region is specific to that object. It also supports dynamic construction and destruction of shared objects: shared regions are typically created and destroyed at the same time as a shared object.

Each region is a logical state from the set LState, and is identified by a *region identifier* from the infinite set RID, ranged over by $r, r', r_1, \ldots$.

Definition 7.2 (Shared State)**.** The set of *shared states*, SState, ranged over by $s, s', s_1, \ldots$, is defined as follows:

$$\mathsf{SState} \stackrel{\text{def}}{=} \mathsf{RID} \rightharpoonup_{\text{fin}} \mathsf{LState}.$$

Action models determine the possible updates, or *actions*, that can be performed on the shared state. Actions, which comprise the set Action, affect a single shared region, but can be dependent on other regions. Each action in the model is identified by a *token*, which includes the identifier of the region that the action applies to.

Definition 7.3 (Action Model)**.** The set of *action models*, AMod, ranged over by $\zeta, \zeta', \zeta_1, \ldots$, is defined as follows:

$$\mathsf{AMod} \stackrel{\text{def}}{=} \mathsf{Token} \rightharpoonup \mathsf{Action}.$$

A token comprises a region identifier from the set RID, an *action name* from the set AName (literal action names are written in small-caps: ACTION), ranged over by $\gamma, \gamma', \gamma_1, \ldots$, and a sequence of *action parameters* from the set Val^*.

Definition 7.4 (Token)**.** The set of *tokens*, Token, ranged over by $t, t', t_1, \ldots$, is defined as follows:

$$\mathsf{Token} \stackrel{\text{def}}{=} \mathsf{RID} \times \mathsf{AName} \times \mathsf{Val}^*.$$

An action is represented as a relation between the initial shared state and the new logical state of the shared region being updated.

Definition 7.5 (Action)**.** The set of *actions*, Action, ranged over by a, a', a_1, is defined as follows:

$$\mathsf{Action} \stackrel{\text{def}}{=} \mathsf{SState} \times \mathsf{LState}$$

Note that an action may be ambiguous without knowing the region identifier that it applies to.

Definition 7.6 (Action Model Combination)**.** The operation $\sqcup : \mathsf{AMod} \times \mathsf{AMod} \to \mathsf{AMod}$ combines action models, taking the union of actions defined in both. It is defined as follows:

$$(\zeta \sqcup \zeta')(t) \stackrel{\text{def}}{=} \begin{cases} \zeta(t) \cup \zeta'(t) & \text{if } t \in (\operatorname{dom}\zeta) \cap (\operatorname{dom}\zeta') \\ \zeta(t) & \text{if } t \in (\operatorname{dom}\zeta) \setminus (\operatorname{dom}\zeta') \\ \zeta'(t) & \text{otherwise.} \end{cases}$$

Logical states combine a separation algebra [COY07] representing machine states — here, I assume it to be the heap separation algebra, $(\mathsf{Heap}, *, \mathrm{emp})$ — with a separation algebra representing *permission assignments* for actions, $(\mathsf{Perm}, \oplus, \mathbf{0}_{\mathsf{Perm}})$.

Definition 7.7 (Logical State)**.** The set of *logical states*, LState, ranged over by $l, l', l_1, \ldots$, is defined as follows:

$$\mathsf{LState} \stackrel{\text{def}}{=} \mathsf{Heap} \times \mathsf{Perm}.$$

For logical state $l = (h, \phi) \in \mathsf{LState}$, $l_{\mathrm{H}} = h \in \mathsf{Heap}$ stands for the heap component, and $l_{\mathrm{P}} = \phi \in \mathsf{Perm}$ stands for the permission assignment component.

Permission assignments associate tokens, which indirectly reference actions, with *permission values*. Here, the set of permission values is taken to be the closed interval $[0, 1]$. A thread holding permission 0 on a token in its local state may not perform the associated action; a thread holding permission greater than 0 on a token may perform the action; a thread holding permission 1 is the only thread which may perform the action. Permission values form a separation algebra $([0, 1], +_{[0,1]}, 0)$, where $+_{[0,1]}$ is $+$ from the real numbers restricted to the codomain $[0, 1]$. Permission assignments form a separation algebra in a pointwise fashion.

Definition 7.8 (Permission assignments)**.** The *permission assignment separation algebra* $(\mathsf{Perm}, \oplus, \mathbf{0}_{\mathsf{Perm}})$, where Perm is ranged over by $\phi, \phi', \phi_1, \ldots$, is defined as follows:

$$\begin{aligned} \mathsf{Perm} &\stackrel{\text{def}}{=} \mathsf{Token} \to [0, 1] \\ (\phi_1 \oplus \phi_2)(t) &\stackrel{\text{def}}{=} \phi_1(t) +_{[0,1]} \phi_2(t) \\ \mathbf{0}_{\mathsf{Perm}}(t) &\stackrel{\text{def}}{=} 0. \end{aligned}$$

This is the fractional permission model of Boyland [Boy03]. Other permissions models are also possible. In [DFPV09], Dodds *et al.* use a permissions model that has both *guarantee* permissions — such as those used here, which permit an action — and *deny* permissions — which do not permit an action, but carry the knowledge that no other thread can hold a guarantee permission, and hence perform the action. Bornat *et al.* [BCOP05] and Parkinson [Par05] have considered other permissions models (in the context of concurrent separation logic) which could also be applied here.

Remark. The permissions model itself does not define the actual semantics of the permissions, which are embodied in the definitions of the rely and guarantee, and derive from the action model.

Since logical states are defined as a product of two separation algebras, they also form a separation algebra in the natural way: $(\mathsf{LState}, \odot, (\mathrm{emp}, \mathbf{0}_{\mathsf{Perm}}))$,

where $(h, \phi) \odot (h', \phi') \stackrel{\text{def}}{=} (h * h', \phi \oplus \phi')$. Worlds also form a separation algebra — nearly. Before defining composition for worlds, I first define well-formedness, which itself depends on the notion of collapsing a world to a logical state.

Definition 7.9 (LState Collapse)**.** The LState *collapse* $\lfloor(\cdot)\rfloor : \mathsf{LState} \times \mathsf{SState} \times \mathsf{AMod} \rightharpoonup \mathsf{LState}$, a partial function that collapses a pre-world to a logical state, is defined as follows:

$$\lfloor(l, s, \zeta)\rfloor \stackrel{\text{def}}{=} l \odot \left(\prod_{r \in \operatorname{dom} s}^{\odot} s(r) \right).$$

The collapse simply composes the logical states of the local portion and each of the shared regions. If more than one region contains the same heap cell, or if the total permission on a token exceeds 1, the collapse is not defined. Part of the well-formedness condition for worlds ensures full disjointness by requiring the collapse of a world to be defined. The well-formedness condition also ensures that no permissions are held on any tokens that do not correspond to actions in the action model of the appropriate shared region, and that no actions are defined for regions that do not exist.

Definition 7.10 (Well-formedness)**.** The well-formedness predicate *wf* is defined as follows:

$$\begin{aligned} wf(l, s, \zeta) \iff & \lfloor(l, s, \zeta)\rfloor \text{ is defined and} \\ & \text{for all } (r, \gamma, \overrightarrow{v}) \in \mathsf{Token}, \\ & \quad \lfloor(l, s, \zeta)\rfloor_{\mathrm{P}}(r, \gamma, \overrightarrow{v}) > 0 \implies (r, \gamma, \overrightarrow{v}) \in \operatorname{dom} \zeta \\ & \quad \text{and } (r, \gamma, \overrightarrow{v}) \in \operatorname{dom} \zeta \implies r \in \operatorname{dom} s. \end{aligned}$$

Figure 7.11 illustrates the structure of a world. Figure 7.12 illustrates an example of a well-formed world, such as might be encountered by the client of a lock module. Figure 7.13 illustrates an example of a triple of logical state, shared state and action model that is not a well-formed world: the heap cell at address 2 is present in both the local and shared state; the total permission for the r, UNLOCK token exceeds 1; a non-zero permission exists for the r, SPLIT token, yet the action model does not define an action corresponding to that token; and the action model defines an action for the region s, which does not exists.

It is necessary to relax the definition of a separation algebra slightly in order to accommodate worlds. Specifically, instead of a single identity element a separation algebra may have multiple identity elements. This reflects the approach taken to the definition of context algebras in §2.2.

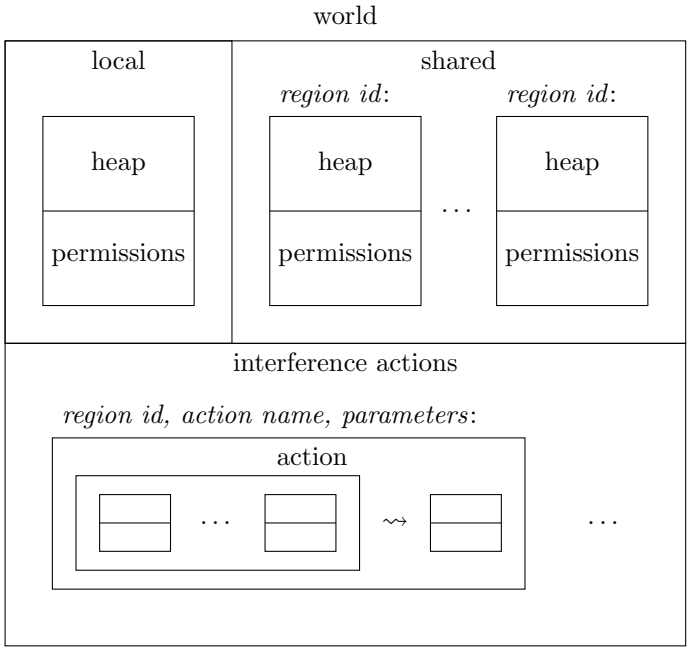


Figure 7.11: Representation of the structure of a world

Definition 7.11 (World Separation Algebra)**.** *World composition* $\bullet : \mathsf{World} \times \mathsf{World} \rightharpoonup \mathsf{World}$ is defined as follows:

$$w \bullet w' \stackrel{\text{def}}{=} \begin{cases} (w_{\mathrm{L}} \odot w'_{\mathrm{L}}, w_{\mathrm{S}}, w_{\mathrm{A}}) & \text{if } w_{\mathrm{S}} = w'_{\mathrm{S}} \text{ and } w_{\mathrm{A}} = w'_{\mathrm{A}} \\ & \text{and } \lfloor w \rfloor_{\mathrm{P}} \oplus ((w')_{\mathrm{L}})_{\mathrm{P}} \text{ is defined} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

The *empty world set* $\mathbf{0}_{\mathsf{World}} \subseteq \mathsf{World}$ is defined as follows:

$$\mathbf{0}_{\mathsf{World}} \stackrel{\text{def}}{=} \{(l, s, \zeta) \in \mathsf{World} \mid l = (\mathrm{emp}, \mathbf{0}_{\mathsf{Perm}})\}\,.$$

It is easy to see that world composition preserves well-formedness, is cancellative, commutative and that $\mathbf{0}_{\mathsf{World}}$ is the identity. Hence, the *world separation algebra* is defined to be $(\mathsf{World}, \bullet, \mathbf{0}_{\mathsf{World}})$.

It is perhaps easiest to view the composition operator on worlds in terms of decomposing a world into two: each gets a disjoint piece of the local state and exactly the same shared state. Conversely, to combine two worlds, their shared states must be identical and their local states disjoint. This may seem at odds with the fact that different threads may make different assertions about the shared state; when the threads join together, however, each of their assertions about the shared state must be true.

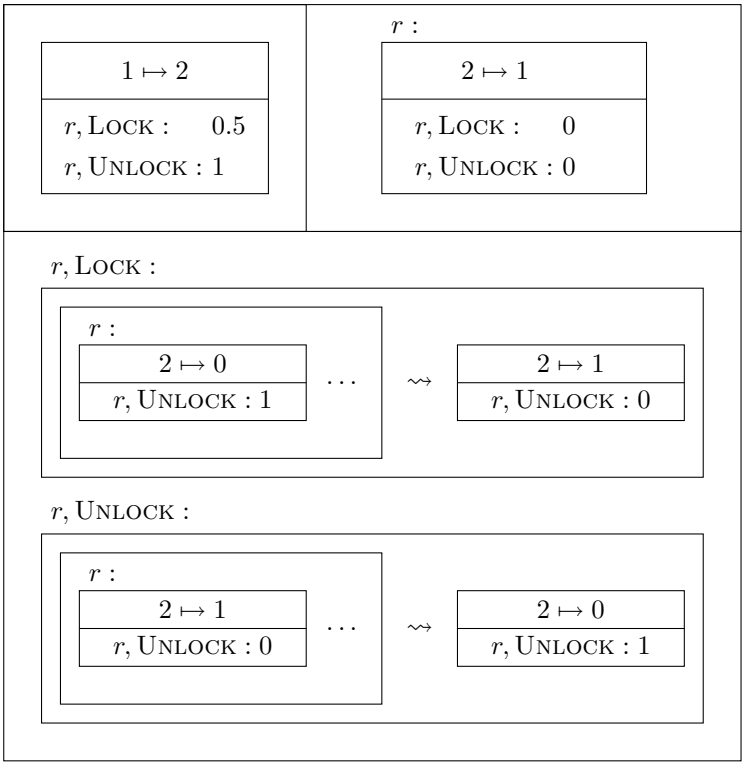


Figure 7.12: Example of a well-formed world

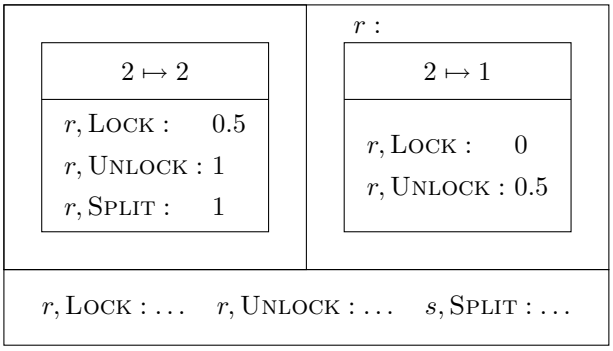


Figure 7.13: Example that is not a well-formed world

In this chapter, I explicitly distinguish the syntax of assertions from their semantics. Sets of worlds are therefore not assertions in their own right, but what I term *semantic predicates*, which are ranged over by $\mathcal{Q}, \mathcal{Q}', \mathcal{Q}_1, \ldots$.

7.3.2 Interference Model

The interference model for the logic defines how the subject thread's world may be updated, both by the subject thread itself and by the environment. These are captured by the *guarantee* and *rely* relations, respectively.

The notions of guarantee and rely come from Jones's rely/guarantee verification method [Jon81]. A key difference with Jones's approach is that the rely and guarantee are not specific to a particular program, but are completely general. The reason for this is that the permissions and action environments of a world encode the information about how the subject or environment can update the world — the rely and guarantee relations essentially provide a semantics for permissions.

Guarantee

The guarantee describes how the subject thread may update the world, w. The subject is free to modify the local heap in any way it pleases, but any change to a shared region, r, must correspond to some action, $\gamma(\overrightarrow{v})$, in r's action environment, for which it has sufficient permission, *i.e.* $(w_{\mathrm{L}})_{\mathrm{P}}(r, \gamma, v) > 0$.

It is important for permitted updates to preserve the total amount of permission in the world, $\lfloor w \rfloor_{\mathrm{P}}$. If threads could acquire permissions out of thin air, then they would be able to introduce interference that was not anticipated by the environment. On the other hand, it is perfectly permissible for heap cells to be allocated or disposed by an update.

The exceptions to preserving permissions are region creation and destruction. When a new region is created, the subject thread acquires all of the region's associated permissions and provides the region's initial contents from its local state. Conversely, in order to destroy of a region, the subject must hold all of the region's associated permissions. On destroying a region, the contents of the region are moved to the subject's local state. In the definition below, the G_c relation corresponds to region creation and its inverse, G_c^{-1}, corresponds to region destruction.

Definition 7.12 (Guarantee Relation)**.** The *guarantee relation* $\mathrm{G} \subseteq \mathsf{World} \times$

World is defined as follows:

$$\begin{aligned} w \mathrel{\mathrm{G}} w' \iff \Big(& \lfloor w \rfloor_{\mathrm{P}} = \lfloor w' \rfloor_{\mathrm{P}} \text{ and } w_{\mathrm{A}} = {w'}_{\mathrm{A}} \text{ and} \\ & \Big(w_{\mathrm{S}} = {w'}_{\mathrm{S}} \text{ or} \\ & \quad \text{there exist } r, \gamma, \overrightarrow{v} \text{ s.t. } (w_{\mathrm{L}})_{\mathrm{P}}(r, \gamma, \overrightarrow{v}) > 0 \text{ and} \\ & \quad (w_{\mathrm{S}}, {w'}_{\mathrm{S}}(r)) \in w_{\mathrm{A}}(r, \gamma, \overrightarrow{v}) \text{ and} \\ & \quad \text{for all } r' \neq r,\ w_{\mathrm{S}}(r') = {w'}_{\mathrm{S}}(r') \Big) \Big) \text{ or} \\ & w \mathrel{\mathrm{G}_c} w' \text{ or } w \mathrel{\mathrm{G}_c^{-1}} w' \end{aligned}$$

where

$$\begin{aligned} w \mathrel{\mathrm{G}_c} w' \iff & \text{there exist } r, \zeta, l_1, l_2 \text{ s.t. } r \notin \operatorname{dom} w_{\mathrm{S}} \text{ and } {w'}_{\mathrm{S}} = w_{\mathrm{S}}[r \mapsto l_1] \\ & \text{and } w_{\mathrm{L}} = l_1 \odot l_2 \text{ and } {w'}_{\mathrm{L}} = l_2 \odot (\mathrm{emp}, \mathit{perms}(\zeta)) \\ & \text{and } {w'}_{\mathrm{A}} = w_{\mathrm{A}} \sqcup \zeta \\ & \text{and for all } r', \gamma, \overrightarrow{v},\ (r', \gamma, \overrightarrow{v}) \in \zeta \implies r' = r \end{aligned}$$

$$\mathit{perms}(\zeta)(t) \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } t \in \operatorname{dom} \zeta \\ 0 & \text{otherwise.} \end{cases}$$

The *repartitioning guarantee relation* $\overline{\mathrm{G}} \subseteq \mathsf{World} \times \mathsf{World}$ permits only updates that do not involve a change to the overall heap — that is, updates that only repartition state between regions. It is defined as follows:

$$\overline{\mathrm{G}} \stackrel{\text{def}}{=} \mathrm{G} \cap \{(w, w') \mid \lfloor w \rfloor_{\mathrm{H}} = \lfloor w' \rfloor_{\mathrm{H}}\}.$$

The *single-step closed guarantee relation* $\widehat{\mathrm{G}} \subseteq \mathsf{World} \times \mathsf{World}$ permits multiple guarantee steps, only one of which is not a repartitioning. It is defined as follows:

$$\widehat{\mathrm{G}} \stackrel{\text{def}}{=} \overline{\mathrm{G}}^* \fatsemi \mathrm{G} \fatsemi \overline{\mathrm{G}}^*.$$

The purpose of the repartitioning guarantee relation is that it represents *logical* updates that the subject thread can perform at any time, without performing any concrete update. The single-step closed guarantee relation represents the permissible updates for the subject thread performing a single atomic update: any number of purely logical updates, and one, possibly concrete update.

Remark. The reason that $\widehat{\mathrm{G}}$ only permits one concrete update is rather subtle. Suppose that the subject thread performs an atomic update from w to w', and that, for some w'', w G w'' G w'. It is possible that w'' has some additional concrete state — a particular allocated heap cell — that is neither present in w nor w'. If another thread held that resource in its local state, then it might assume that the original thread could not make the update from w to w', since it could not enter the intermediate w'' state.

Rely

The rely describes how the environment may update the world, w. The environment may not touch the subject thread's local state, but it may change a shared region, r, in accordance with some action, $\gamma(\overrightarrow{v})$, for which some permission is not fully accounted for, *i.e.* $\lfloor w \rfloor_{\mathrm{P}}(r, \gamma, \overrightarrow{v}) < 1$. This is because when a permission is not fully accounted for in the world, it implies that it could belong to the local state of some other thread, which would then be entitled to perform the action.

The rely must also account for regions being created and destroyed by the environment. The initial contents of a created region are essentially arbitrary, since they come from the local state of the thread which created it, however, no permissions for new region are initially present in the world. Conversely, the environment may destroy a region if no permissions for the region are present in the world, and its contents simply disappear from the subject's perspective. In the definition below, R_c corresponds to region creation and its inverse, R_c^{-1}, corresponds to region destruction.

Definition 7.13 (Rely Relation)**.** The *rely relation* $\mathrm{R} \subseteq \mathsf{World} \times \mathsf{World}$ is defined as follows:

$$
\begin{aligned}
w \ \mathrm{R} \ w' \iff \Big(& w_{\mathrm{L}} = {w'}_{\mathrm{L}} \text{ and } w_{\mathrm{A}} = {w'}_{\mathrm{A}} \text{ and} \\
& \quad \text{there exist } r, \gamma, \overrightarrow{v} \text{ s.t. } \lfloor w \rfloor_{\mathrm{P}}(r, \gamma, \overrightarrow{v}) < 1 \text{ and} \\
& \quad\quad (w_{\mathrm{S}}, {w'}_{\mathrm{S}}(r)) \in w_{\mathrm{A}}(r, \gamma, \overrightarrow{v}) \text{ and} \\
& \quad\quad \text{for all } r' \neq r,\ w_{\mathrm{S}}(r') = {w'}_{\mathrm{S}}(r') \Big) \text{ or} \\
& w \ \mathrm{R}_c \ w' \text{ or } w \ \mathrm{R}_c^{-1} \ w'
\end{aligned}
$$

where

$$\begin{aligned} w\ \mathrm{R}_c\ w' \iff{}& \text{there exist } r, \zeta, l \text{ s.t. } r \notin \operatorname{dom} w_{\mathrm{S}} \text{ and } w_{\mathrm{L}} = {w'}_{\mathrm{L}} \text{ and} \\ & {w'}_{\mathrm{S}} = w_{\mathrm{S}}[r \mapsto l] \text{ and for all } \gamma, \overrightarrow{v},\ \lfloor w' \rfloor_{\mathrm{P}}(r, \gamma, \overrightarrow{v}) = 0 \text{ and} \\ & {w'}_{\mathrm{A}} = w_{\mathrm{A}} \sqcup \zeta \text{ and} \\ & \text{for all } r', \gamma, \overrightarrow{v},\ (r', \gamma, \overrightarrow{v}) \in \zeta \implies r' = r. \end{aligned}$$

Definition 7.14 (Stability)**.** For $\mathcal{Q} \in \mathcal{P}(\mathsf{World})$, *stable* $(\mathcal{Q})$ holds if and only if, for all worlds $w, w' \in \mathsf{World}$, if $w \in \mathcal{Q}$ and $w\ \mathrm{R}\ w'$ then $w' \in \mathcal{Q}$.

7.3.3 Assertions

I now present an assertion language for defining predicates over worlds. As well as standard separation logic connectives, the logic includes a number of connectives to deal with permissions, shared regions, abstract predicates and updates.

Definition 7.15 (Assertions)**.** The set of *assertions* Assn, ranged over by $P, Q, P_1, \ldots$, the set of *basic assertions* BAssn, ranged over by $p, q, p_1, \ldots$, and the set of *interference assertions* IAssn, ranged over by $I, I_1, \ldots$, are defined inductively as follows:

$$\begin{aligned} P ::= {}& \mathrm{emp} \mid E_1 \mapsto E_2 \mid P * Q \mid P \mathrel{-\!\!*} Q \mid \mathrm{false} \mid P \to Q \mid \exists x.\, P \\ & \mid \circledast x.\, P \mid \mathrm{all}(I, R) \mid [\gamma(E_1, \ldots, E_n)]^R_\pi \mid \boxed{P}^R_I \mid \alpha(E_1, \ldots, E_n) \\ & \mid \diamondsuit\!\!\!\!\boxed{\mathbf{G}} P \mid \blacklozenge\!\!\!\!\boxed{\mathbf{R}} P \mid \boxed{\mathbf{R}} P \mid \left[\begin{smallmatrix} p \\ q \end{smallmatrix}\right\rangle P \end{aligned}$$

$$p ::= \mathrm{emp} \mid E_1 \mapsto E_2 \mid p * q \mid p \mathrel{-\!\!*} q \mid \mathrm{false} \mid p \to q \mid \exists x.\, p \mid \circledast x.\, p$$

$$I ::= \gamma(\overrightarrow{x}) : \ \exists \overrightarrow{y}.\, (P \rightsquigarrow Q) \mid I_1, I_2$$

where

- $E_1, E_2, R, \pi \in \mathsf{Expr}$ range over expressions, for which an appropriate syntax is assumed that includes basic arithmetic operators,
- $x, y \in \mathsf{LVar}$ range over logical variables ($\overrightarrow{x}$ being a vector of logical variables),
- $\gamma \in \mathsf{AName}$ ranges over action names, and
- $\alpha \in \mathsf{APName}$ ranges over abstract predicate names (literal abstract predicate names are written in sans-serif: $\mathsf{absPred}$).

Basic assertions describe only heap, and so their syntax is the standard separation logic syntax: connectives for describing empty and single-celled heaps, the composition of disjoint heaps and its adjoint, and basic predicate logic operators. The $\circledast$ operator, which is not commonly used in separation logic, is the multiplicative analogue of $\forall$: it describes a heap that is composed of a pieces satisfying p for every value of x. (In the literature, the syntax $\forall^*$ is sometimes used instead.)

Assertions include the same connectives as basic assertions, except that these are now interpreted over worlds. To these are added connectives for describing permissions: $[\gamma(E_1, \ldots, E_n)]^R_\pi$ describes permission value π for the action γ on region R with parameters $E_1, \ldots, E_n$; and $\mathrm{all}(I, R)$ describes full permission for all of the actions on R defined by the interference assertion I. Unboxed assertions describe only the local state of the world; they make no assertion whatever as to the contents of shared regions.

Boxed assertions $\boxed{P}^R_I$ assert that contents of the shared region identified by R satisfies P, that the action model's interference for that region is specified by I, and that the local state is empty. A consequence of the definition of world composition is that separating conjunction between shared state assertions behaves as (additive) conjunction. In particular:

$$\boxed{P}^r_I * \boxed{Q}^{r'}_{I'} \equiv \boxed{P}^r_I \wedge \boxed{Q}^{r'}_{I'}$$
$$\boxed{P}^r_I * \boxed{Q}^r_I \equiv \boxed{P \wedge Q}^r_I$$

Since the contents of a boxed assertion is itself an assertion, there are some subtleties in the interpretation of these assertions. In particular, boxes may be nested inside each other. Of course, shared regions are not nested in the model; in the semantics this is resolved by the fact that boxes have the same semantics whether or not they occur inside another box or not. Hence:

$$\boxed{\boxed{P}^r_I * Q}^{r'}_{I'} \equiv \boxed{P}^r_I * \boxed{Q}^{r'}_{I'}$$

(Note: this holds even when $r = r'$.)

In the examples discussed so far, I have not used explicit nesting, but in §7.2 there is implicit nesting through abstract predicates: the concrete interpretation of such predicates as $\mathsf{Locked}(l)$ make assertions about a shared region, yet they themselves appear in shared regions. For example, in the proof of the coarse-grained set, the following shared region assertion appears:

$$\boxed{\mathsf{Locked}(s) * ([\textsc{Ret}(vs \cup \{v\})]^r_1 \mathrel{-\!\!*} \mathsf{allrets}(s))}^r_{C(s,r)}$$

Using the compare-and-swap lock implementation, the above assertion is semantically equivalent to the following:

$$\boxed{\exists r'.\, [\textsc{Unlock}]_1^{r'} * \boxed{(l-1) \mapsto 1}^{r'}_{I(l,r')} * ([\textsc{Ret}(vs \cup \{v\})]_1^r \mathrel{-\!\!*} \mathsf{allrets}(s))}^{r}_{C(s,r)}$$

Which in turn is equivalent to:

$$\exists r'.\, \boxed{[\textsc{Unlock}]_1^{r'} * ([\textsc{Ret}(vs \cup \{v\})]_1^r \mathrel{-\!\!*} \mathsf{allrets}(s))}^{r}_{C(s,r)} * \boxed{(l-1) \mapsto 1}^{r'}_{I(l,r')}$$

Remark. It may be helpful to consider shared state assertions as akin to pure assertions — assertions about logical variables that make no assertion at all about the world and which hold solely on the basis of the variable interpretation. Shared regions here do make an assertion about the local state — that it is empty — but they could just as easily be defined to make no assertion about the local state at all, in which case they hold solely on the basis of the shared state.

Abstract predicates $\gamma(E_1, \ldots, E_n)$ are interpreted according to a predicate environment. Essentially, they are predicate-valued variables. (Note that the logic does not include quantification over such variables.)

The repartitioning update modality $\diamondsuit\!\!\!\!\boxed{\mathrm{G}} P$ asserts that it is possible to repartition the world in accordance with $\overline{\mathrm{G}}^*$ such that the resulting world satisfies P. This can be thought of as that most general set of worlds such that the subject thread can ensure that P holds without making any concrete update.

The explicit stabilisation assertions $\diamondsuit\!\!\!\!\blacksquare\!\!\!\!\blacklozenge P$ and $\boxed{\mathbf{R}} P$ correspond to the strongest weaker and weakest stronger stable assertions than P. $\diamondsuit\!\!\!\!\blacksquare\!\!\!\!\blacklozenge P$ is satisfied by every world that can be obtained by performing an R^* transition from on satisfying P. $\boxed{\mathbf{R}} P$ is satisfied by a world if every R^* transition that is performed on it results in a world satisfying P. Both of these concepts are due to Wickerson *et al.* [WDP10], who use the notation $\lceil P \rceil$ for $\diamondsuit\!\!\!\!\blacksquare\!\!\!\!\blacklozenge P$ and $\lfloor P \rfloor$ for $\boxed{\mathbf{R}} P$.

Finally, the update operator $\left[\begin{smallmatrix} p \\ q \end{smallmatrix}\right\rangle$ can be read as "the weakest liberal guaranteed precondition of the best local action specified by $\{p\} - \{q\}$". The best local action [COY07] can be viewed as the least-refined heap operation that satisfies the given specification and all framed versions of the specification. Semantically, this can be thought of as the function $bla[p,q] : \mathsf{Heap} \to \mathcal{P}(\mathsf{Heap}) \cup \{\lightning\}$ defined as follows:

$$bla[p,q](h) = \begin{cases} \lightning & \text{if there are no } h_0, h_1 \text{ with} \\ & \quad h = h_0 * h_1 \text{ and } h_1 \in [\![p]\!] \\ \bigcap\limits_{\substack{h_1 \in [\![p]\!] \\ h_0 * h_1 = h}} \{h_0 * h_1' \mid h_1' \in [\![q]\!]\} & \text{otherwise.} \end{cases}$$

The weakest liberal precondition of an operation with respect to some postcondition describes all states such that performing the operation results in only states satisfying the postcondition. In terms of worlds, the weakest liberal precondition describes those worlds that can be collapsed to heaps such that every outcome of the operation on that heap is a heap which is the collapse of a world satisfying the postcondition. The weakest liberal *guaranteed* precondition further restricts this so that the worlds must be related by the guarantee $\widehat{\mathrm{G}}$. Thus, in essence, $\left[\begin{smallmatrix} p \\ q \end{smallmatrix}\right\rangle P$ describes all those worlds that, subject to replacing one concrete subheap satisfying p with one satisfying q, can be repartitioned into a world satisfying P in a manner permitted by the guarantee.

Interference assertions, which must be interpreted in the context of a region identifier r, are essentially collections of assertions of the form $\gamma(\overrightarrow{x}) : \exists \overrightarrow{y}.\, (P \rightsquigarrow Q)$. Each of these specifies behaviour associated with the action name γ: the token $r, \gamma, \overrightarrow{v}$ corresponds to updating some portion of region r that satisfies P to satisfy Q; P and Q here are interpreted with $\overrightarrow{x}$ bound to $\overrightarrow{v}$ and $\overrightarrow{y}$ bound to any $\overrightarrow{w}$ (but the same for both P and Q). Multiple actions are combined with $\sqcup$, which even allows for more than one definition for the same action name; an example of this is in the interference assertion for the fine-grained set implementation, in which three actions are defined for $\textsc{Ret}(n, t, v)$.

Assertion Semantics

Logical variables in assertions range over the set of values Val. I assume that $\mathsf{RID} \cup (0, 1] \subseteq \mathsf{Val}$, so that variables may range over region identifiers and (nonzero) permission values. In the semantics of assertions, free variables are evaluated using a *variable interpretation* (or simply *interpretation*), which maps variables to values. The set of variable interpretations is $\mathsf{Interp} \stackrel{\text{def}}{=} \mathsf{LVar} \to \mathsf{Val}$, and is ranged over by $i, i', i_1, \ldots$.

I assume an appropriate semantics for expressions $[\![(\cdot)]\!]_{(\cdot)} : \mathsf{Expr} \times \mathsf{Interp} \to \mathsf{Val}$. The semantics for basic assertions, $[\![(\cdot)]\!]_{(\cdot)} : \mathsf{BAssn} \times \mathsf{Interp} \to \mathcal{P}(\mathsf{Heap})$, is just the standard separation logic semantics.

Abstract predicates α are essentially predicate-valued variables. They are interpreted using a *predicate environment*, which maps abstract predicates to their semantic definitions. The set of predicate environments is $\mathsf{PEnv} \stackrel{\text{def}}{=} \mathsf{APName} \times \mathsf{Val}^* \to \mathcal{P}(\mathsf{World})$, and is ranged over by $\delta, \delta', \delta_1, \ldots$.

Definition 7.16 (Assertion Semantics)**.** The semantics of assertions $[\![(\cdot)]\!]_{(\cdot),(\cdot)} : \mathsf{Assn} \times \mathsf{PEnv} \times \mathsf{Interp} \to \mathcal{P}(\mathsf{World})$ is defined as follows:

$$[\![P]\!]_{\delta,i} \stackrel{\text{def}}{=} \left\{ (l, s, \zeta) \in (\!|P|\!)_{\delta,i} \;\middle|\; wf(l, s, \zeta) \right\}$$

where $(\!|(\cdot)|\!)_{(\cdot),(\cdot)} : \mathsf{Assn} \times \mathsf{PEnv} \times \mathsf{Interp} \to \mathcal{P}(\mathsf{LState} \times \mathsf{SState} \times \mathsf{AMod})$ and $[\![(\cdot)]\!]^{(\cdot)}_{(\cdot),(\cdot)} : \mathsf{IAssn} \times \mathsf{RID} \times \mathsf{PEnv} \times \mathsf{Interp} \to \mathsf{AMod}$ are defined as follows:

$$(\!|\mathrm{emp}|\!)_{\delta,i} \stackrel{\text{def}}{=} \{((\mathrm{emp}, \mathbf{0}_{\mathsf{Perm}}), s, \zeta) \mid s \in \mathsf{SState} \text{ and } \zeta \in \mathsf{AMod}\}$$

$$(\!|E_1 \mapsto E_2|\!)_{\delta,i} \stackrel{\text{def}}{=} \left\{ (l, s, \zeta) \;\middle|\; \begin{array}{c} l_{\mathrm{H}} = [[\![E_1]\!]_i \mapsto [\![E_2]\!]_i] \text{ and} \\ l_{\mathrm{P}} = \mathbf{0}_{\mathsf{Perm}} \text{ and} \\ s \in \mathsf{SState} \text{ and } \zeta \in \mathsf{AMod} \end{array} \right\}$$

$$(\!|P * Q|\!)_{\delta,i} \stackrel{\text{def}}{=} \left\{ w_1 \bullet w_2 \;\middle|\; w_1 \in (\!|P|\!)_{\delta,i} \text{ and } w_2 \in (\!|Q|\!)_{\delta,i} \right\}$$

$$(\!|P -\!\!* Q|\!)_{\delta,i} \stackrel{\text{def}}{=} \left\{ w \;\middle|\; \begin{array}{r} \text{for all } w_1, w_2,\ w_2 = w \bullet w_1 \text{ and } w_1 \in (\!|P|\!)_{\delta,i} \\ \implies w_2 \in (\!|Q|\!)_{\delta,i} \end{array} \right\}$$

$$(\!|\mathrm{false}|\!)_{\delta,i} \stackrel{\text{def}}{=} \emptyset$$

$$(\!|P \to Q|\!)_{\delta,i} \stackrel{\text{def}}{=} \left\{ w \;\middle|\; w \in (\!|P|\!)_{\delta,i} \implies w \in (\!|Q|\!)_{\delta,i} \right\}$$

$$(\!|\exists x.\, P|\!)_{\delta,i} \stackrel{\text{def}}{=} \bigcup_{v \in \mathsf{Val}} (\!|P|\!)_{\delta, i[x \mapsto v]}$$

$$(\!|\circledast x.\, P|\!)_{\delta,i} \stackrel{\text{def}}{=} \left\{ \prod_{v \in \mathsf{Val}}^{\bullet} w_v \;\middle|\; \text{for all } v \in \mathsf{Val},\ w_v \in (\!|P|\!)_{\delta, i[x \mapsto v]} \right\}$$

$$(\!|\mathrm{all}(I, R)|\!)_{\delta,i} \stackrel{\text{def}}{=} \left\{ ((\mathrm{emp}, \mathit{perms}([\![I]\!]^r_{\delta,i})), s, \zeta) \;\middle|\; \begin{array}{r} r = [\![R]\!]_{\delta,i} \text{ and} \\ s \in \mathsf{SState} \text{ and} \\ \zeta \in \mathsf{AMod} \end{array} \right\}$$

$$\left(\!\left|\left[\gamma(\overrightarrow{E})\right]^R_\pi\right|\!\right)_{\delta,i} \stackrel{\text{def}}{=} \left\{ \left(\left(\mathrm{emp}, [([\![R]\!]_i, \gamma, [\![\overrightarrow{E}]\!]_i) \mapsto [\![\pi]\!]_i]\right), s, \zeta\right) \;\middle|\; \begin{array}{c} [\![\pi]\!]_i \in (0, 1] \\ \text{and} \\ s \in \mathsf{SState} \\ \text{and} \\ \zeta \in \mathsf{AMod} \end{array} \right\}$$

$$\left(\!\left|\boxed{P}^R_I\right|\!\right)_{\delta,i} \stackrel{\text{def}}{=} \left\{ ((\mathrm{emp}, \mathbf{0}_{\mathsf{Perm}}), s, \zeta) \;\middle|\; \begin{array}{l} \text{there exist } l, r \text{ s.t.} \\ \quad (l, s, \zeta) \in (\!|P|\!) \text{ and} \\ \quad r = [\![R]\!]_i \text{ and } s(r) = l \text{ and} \\ \quad \text{for all } \gamma, \overrightarrow{v}, \\ \quad\quad \zeta(r, \gamma, \overrightarrow{v}) = [\![I]\!]^r_{\delta,i}(r, \gamma, \overrightarrow{v}) \end{array} \right\}$$

$$\left(\!\left|\alpha(\overrightarrow{E})\right|\!\right)_{\delta,i} \stackrel{\text{def}}{=} \delta(\alpha, [\![\overrightarrow{E}]\!]_i)$$

$$\left(\!\left|\diamondsuit_{\mathrm{G}} P\right|\!\right)_{\delta,i} \stackrel{\text{def}}{=} \left\{ w \;\middle|\; \text{there exists } w' \text{ s.t. } w\ \overline{\mathrm{G}}^*\ w' \text{ and } w' \in (\!|P|\!)_{\delta,i} \right\}$$

$$\left(\!\left|\diamondsuit_{\mathrm{R}} P\right|\!\right)_{\delta,i} \stackrel{\text{def}}{=} \left\{ w \;\middle|\; \text{there exists } w' \text{ s.t. } w'\ \mathrm{R}^*\ w \text{ and } w' \in (\!|P|\!)_{\delta,i} \right\}$$

$$(\!|\boxed{\mathrm{R}} P|\!)_{\delta,i} \stackrel{\text{def}}{=} \left\{ w \;\middle|\; \text{for all } w',\ w\ \mathrm{R}^*\ w' \implies w' \in (\!|P|\!)_{\delta,i} \right\}$$

$$\left(\!\left|\left[\frac{p}{q}\right\rangle P\right|\!\right)_{\delta,i} \stackrel{\text{def}}{=} \left\{ w \;\middle|\; \begin{array}{r} \text{there exist } h_1 \in [\![p]\!]_i, h' \text{ s.t. } \lfloor w \rfloor_{\mathrm{H}} = h_1 * h' \text{ and} \\ \text{for all } h_2 \in [\![q]\!]_i, \text{ there exists } w' \in (\!|P|\!)_{\delta,i} \text{ s.t.} \\ \lfloor w' \rfloor_{\mathrm{H}} = h_2 * h' \text{ and } w \;\widehat{\mathrm{G}}\; w' \end{array} \right\}$$

$$[\![\gamma(\overrightarrow{x}) : \exists \overrightarrow{y}.\, (P \rightsquigarrow Q)]\!]^r_{\delta,i}\,(r', \gamma', \overrightarrow{v}) \stackrel{\text{def}}{=}$$

$$\begin{cases} \left\{ (s,l) \;\middle|\; \begin{array}{c} \text{there exist } l_0, l_1, l_2, \overrightarrow{u} \text{ s.t.} \\ s(r) = l_0 \odot l_1 \text{ and} \\ l = l_0 \odot l_2 \text{ and} \\ (l_1, s) \in (\!|P|\!)_{\delta,i[\overrightarrow{x}\mapsto\overrightarrow{v}][\overrightarrow{y}\mapsto\overrightarrow{u}]} \text{ and} \\ (l_2, s[r \mapsto l]) \in (\!|Q|\!)_{\delta,i[\overrightarrow{x}\mapsto\overrightarrow{v}][\overrightarrow{y}\mapsto\overrightarrow{u}]} \end{array} \right\} & \begin{array}{l} \text{if } r = r' \text{ and} \\ \quad \gamma = \gamma' \text{ and} \\ \quad \text{len } \overrightarrow{x} = \text{len } \overrightarrow{v} \end{array} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$[\![I_1, I_2]\!]^r_{\delta,i} \stackrel{\text{def}}{=} [\![I_1]\!]^r_{\delta,i} \sqcup [\![I_1]\!]^r_{\delta,i}\,.$$

7.3.4 Programming Language and Proof System

The proof system presented here is for a simple Dijkstra-style language. Many common constructs can be encoded elegantly in the language.

Definition 7.17 (Programming Language)**.** Assuming a set of basic commands Cmd, ranged over by c, the language $\mathcal{L}$, ranged over by $\mathbb{C}, \mathbb{C}', \mathbb{C}_1, \ldots$, is defined as follows:

$$\begin{array}{c} \mathbb{C} ::= \texttt{skip} \;\mid\; c \;\mid\; f \;\mid\; \langle \mathbb{C} \rangle \;\mid\; \mathbb{C}_1; \mathbb{C}_2 \;\mid\; \mathbb{C}_1 + \mathbb{C}_2 \;\mid\; \mathbb{C}^* \\ \mid\; \mathbb{C}_1 \parallel \mathbb{C}_2 \;\mid\; \texttt{let } f_1 = \mathbb{C}_1, \ldots, f_n = \mathbb{C}_n \texttt{ in } \mathbb{C} \end{array}$$

Function names $f_1, \ldots, f_n$ defined by the same `let` block are assumed to be pairwise distinct.

Judgements of the proof system have the form $\Delta; \Gamma \vdash \{P\}\ \mathbb{C}\ \{Q\}$. Here, $\Delta \in \mathsf{Assn}$ is an assertion about the abstract predicates, Γ is a set of assertions of the form $\{P'\}\ f\ \{Q'\}$ specifying the functions called in $\mathbb{C}$, and $P, Q \in \mathsf{Assn}$ are pre- on postconditions of the program $\mathbb{C}$. The judgement should be read with a fault-avoiding partial correctness interpretation: given an interpretation of abstract predicates that satisfies Δ, and functions that meet the specifications in Γ, whenever the program $\mathbb{C}$ is run from a state satisfying P then it will not fault, but will either terminate in a state satisfying Q or not terminate at all.

The premisses of some of the proof rules include certain other judgements. The judgement $\vdash_{\mathsf{SL}} \{p\}\ \mathbb{C}\ \{q\}$ is the standard separation logic proof judgement — it holds if the triple $\{p\}\ \mathbb{C}\ \{q\}$ is provable in separation logic. The predicate entailment judgement $\Delta \vdash P$ asserts that for any variable interpretation and

predicate environments for which Δ is valid (*i.e.*, satisfied by all worlds), P is also valid.

Definition 7.18 (Predicate Entailment Judgement)**.** For $\Delta, P \in \mathsf{Assn}$, the judgement $\Delta \vdash P$ is defined to hold if and only if, for all δ, i with $[\![\Delta]\!]_{\delta,i} = \mathsf{World}$, $[\![P]\!]_{\delta,i} = \mathsf{World}$.

The predicate definition safety judgement asserts that a new abstract predicate definition is realisable in a manner that does not impact the possible interpretations of other abstract predicates.

Definition 7.19 (Predicate Definition Safety Judgement)**.** For $\Delta \in \mathsf{Assn}$, $\alpha \in \mathsf{APName}$, $\overrightarrow{x} \in \mathsf{LVar}^*$ and $R \in \mathsf{Assn}$, the judgement $\Delta \Uparrow \alpha, \overrightarrow{x}, R$ is defined to hold if and only if, for all δ, i with $[\![\Delta]\!]_{\delta,i} = \mathsf{World}$, there exists some δ' such that, for all α', $\overrightarrow{v}$

$$\delta'(\alpha', \overrightarrow{v}) = \begin{cases} [\![R]\!]_{\delta', i[\overrightarrow{x} \mapsto \overrightarrow{v}]} & \text{if } \alpha' = \alpha \\ \delta(\alpha', \overrightarrow{v}) & \text{otherwise.} \end{cases}$$

This judgement may appear to be quite awkward to check in general, but it is easy to syntactically constrain predicate definitions so that it holds. In particular, the existence of an appropriate δ' corresponds to the existence of a certain fixed-point. If the abstract predicate α only occurs positively (*i.e.*, under an even number of negations) in its definition R then the predicate definition is monotone and so a fixed-point exists by the Knaster-Tarski theorem [Tar55]. This restriction is typical when defining inductive predicates.

Definition 7.20 (Proof System)**.** The derivation rules for judgements of the form $\Delta; \Gamma \vdash \{P\}\ \mathbb{C}\ \{Q\}$ are given in Figure 7.14.

7.3.5 Language Semantics

I give a small-step operational semantics for the programming language. This semantics assumes that basic commands c have a relational semantics $[\![c]\!] \subseteq \mathsf{Heap} \times (\mathsf{Heap} \cup \{\lightning\})$; I implicitly identify c with its semantics $[\![c]\!]$.The one-step judgement $(\mathbb{C}, h) \xrightarrow{\eta} (\mathbb{C}', h')$ expresses that, in the presence of functions η, the program $\mathbb{C}$ in the (heap) state h reduces in one step to the program $\mathbb{C}'$ and state h'. The function definition environment η is simply a partial function mapping function names to the programs that implement them.

Definition 7.21 (Operational Semantics)**.** The derivation rules for judgements of the form $(\mathbb{C}, h) \xrightarrow{\eta} (\mathbb{C}', h')$ and $(\mathbb{C}, h) \xrightarrow{\eta} \lightning$ are defined in Figure 7.15.

$$\frac{\vdash_{\mathsf{SL}} \{p\}\ \mathbb{C}\ \{q\}}{\Delta;\Gamma \vdash \left\{\boxed{\mathbf{R}}\left[\begin{smallmatrix}p\\q\end{smallmatrix}\right\rangle Q\right\}\ \langle\mathbb{C}\rangle\ \left\{\diamondsuit Q\right\}}\ \textsc{Atomic} \qquad \frac{\vdash_{\mathsf{SL}} \{p\}\ \mathbb{C}\ \{q\}}{\Delta;\Gamma \vdash \{p\}\ \mathbb{C}\ \{q\}}\ \textsc{Prim}$$

$$\frac{\begin{array}{l}\Delta \vdash Q_1 \to \boxed{\mathbf{R}}Q_1 \quad \Delta;\Gamma \vdash \{P_1\}\ \mathbb{C}_1\ \{Q_1\}\\ \Delta \vdash Q_2 \to \boxed{\mathbf{R}}Q_2 \quad \Delta;\Gamma \vdash \{P_2\}\ \mathbb{C}_2\ \{Q_2\}\end{array}}{\Delta;\Gamma \vdash \{P_1 * P_2\}\ \mathbb{C}_1 \parallel \mathbb{C}_2\ \{Q_1 * Q_2\}}\ \textsc{Par}$$

$$\frac{\Delta \vdash R \to \boxed{\mathbf{R}}R \quad \Delta;\Gamma \vdash \{P\}\ \mathbb{C}\ \{Q\}}{\Delta;\Gamma \vdash \{R * P\}\ \mathbb{C}\ \{R * Q\}}\ \textsc{Frame}$$

$$\frac{\Delta \vdash P \to P' \quad \Delta;\Gamma \vdash \{P'\}\ \mathbb{C}\ \{Q'\} \quad \Delta \vdash Q' \to Q}{\Delta;\Gamma \vdash \{P\}\ \mathbb{C}\ \{Q\}}\ \textsc{Cons}$$

$$\frac{\Delta;\Gamma \vdash \{P\}\ \mathbb{C}\ \{Q\}}{\Delta;\Gamma \vdash \left\{\diamondsuit P\right\}\ \mathbb{C}\ \left\{Q\right\}}\ \textsc{Guar-L} \qquad \frac{\Delta;\Gamma \vdash \left\{P\right\}\ \mathbb{C}\ \left\{\diamondsuit Q\right\}}{\Delta;\Gamma \vdash \{P\}\ \mathbb{C}\ \{Q\}}\ \textsc{Guar-R}$$

$$\frac{\Delta \vdash \Delta' \quad \Delta';\Gamma \vdash \{P\}\ \mathbb{C}\ \{Q\}}{\Delta;\Gamma \vdash \{P\}\ \mathbb{C}\ \{Q\}}\ \textsc{Pred-I}$$

$$\frac{\alpha \notin \Delta, \Gamma, P, Q \quad \Delta \Uparrow \alpha, \overrightarrow{x}, R \quad \Delta \wedge (\forall \overrightarrow{x}.\, \alpha(\overrightarrow{x}) \leftrightarrow R); \Gamma \vdash \{P\}\ \mathbb{C}\ \{Q\}}{\Delta;\Gamma \vdash \{P\}\ \mathbb{C}\ \{Q\}}\ \textsc{Pred-E}$$

$$\frac{\begin{array}{l}\Delta;\Gamma \vdash \{P_1\}\ \mathbb{C}_1\ \{Q_1\} \quad \ldots \quad \Delta;\Gamma \vdash \{P_n\}\ \mathbb{C}_n\ \{Q_n\}\\ \Delta;\Gamma, \{P_1\}\ f_1\ \{Q_1\}, \ldots, \{P_n\}\ f_n\ \{Q_n\} \vdash \{P\}\ \mathbb{C}\ \{Q\}\end{array}}{\Delta;\Gamma \vdash \{P\}\ \texttt{let}\ f_1 = \mathbb{C}_1, \ldots, f_n = \mathbb{C}_n\ \texttt{in}\ \mathbb{C}\ \{Q\}}\ \textsc{Let}$$

$$\frac{\{P\}\ f\ \{Q\} \in \Gamma}{\Delta;\Gamma \vdash \{P\}\ f\ \{Q\}}\ \textsc{Call} \qquad \frac{\Delta;\Gamma \vdash \{P\}\ \mathbb{C}\ \{P\}}{\Delta;\Gamma \vdash \{\boxed{\mathbf{R}}P\}\ \mathbb{C}^*\ \{P\}}\ \textsc{Loop}$$

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

$$\frac{\Delta;\Gamma \vdash \{P\}\ \mathbb{C}_1\ \{Q\} \quad \Delta;\Gamma \vdash \{P\}\ \mathbb{C}_2\ \{Q\}}{\Delta;\Gamma \vdash \{P\}\ \mathbb{C}_1 + \mathbb{C}_2\ \{Q\}}\ \textsc{Choice}$$

$$\frac{\begin{array}{l}\Delta;\Gamma \vdash \{P_1\}\ \mathbb{C}\ \{Q_1\}\\ \Delta;\Gamma \vdash \{P_2\}\ \mathbb{C}\ \{Q_2\}\end{array}}{\Delta;\Gamma \vdash \{P_1 \vee P_2\}\ \mathbb{C}\ \{Q_1 \vee Q_2\}}\ \textsc{Disj} \qquad \frac{\Delta;\Gamma \vdash \{P\}\ \mathbb{C}\ \{Q\}}{\Delta;\Gamma \vdash \{\exists x.\, P\}\ \mathbb{C}\ \{\exists x.\, Q\}}\ \textsc{Exist}$$

Figure 7.14: The rules of the concurrent abstract predicates proof system

$$\frac{(\mathbb{C}, h) \xrightarrow{\eta[f_1 \mapsto \mathbb{C}_1 \ldots f_n \mapsto \mathbb{C}_n]} (\mathbb{C}', h')}{(\texttt{let}\ \ f_1 = \mathbb{C}_1, \ldots, f_n = \mathbb{C}_n\ \ \texttt{in}\ \ \mathbb{C}, h) \xrightarrow{\eta} (\texttt{let}\ \ f_1 = \mathbb{C}_1, \ldots, f_n = \mathbb{C}_n\ \ \texttt{in}\ \ \mathbb{C}', h')}$$

$$\frac{}{(\text{let}\ \ldots\ \text{in}\ \texttt{skip}, h) \xrightarrow{\eta} (\texttt{skip}, h)} \qquad \frac{f \in \operatorname{dom} \eta}{(f, h) \xrightarrow{\eta} (\eta(f), h)} \qquad \frac{(h, h') \in c \quad h' \neq \lightning}{(c, h) \xrightarrow{\eta} (\texttt{skip}, h')}$$

$$\frac{(\mathbb{C}, h) \xrightarrow{\eta} (\mathbb{C}_1, h')}{(\mathbb{C}; \mathbb{C}', h) \xrightarrow{\eta} (\mathbb{C}_1; \mathbb{C}', h')} \qquad \frac{}{(\texttt{skip}; \mathbb{C}, h) \xrightarrow{\eta} (\mathbb{C}, h)} \qquad \frac{}{(\mathbb{C}^*, h) \xrightarrow{\eta} (\texttt{skip} + (\mathbb{C}; \mathbb{C}^*), h)}$$

$$\frac{}{(\mathbb{C}_1 + \mathbb{C}_2, h) \xrightarrow{\eta} (\mathbb{C}_1, h)} \qquad \frac{}{(\mathbb{C}_1 + \mathbb{C}_2, h) \xrightarrow{\eta} (\mathbb{C}_2, h)} \qquad \frac{(\mathbb{C}, h) \xrightarrow{\eta}{}^* (\texttt{skip}, h')}{(\langle \mathbb{C} \rangle, h) \xrightarrow{\eta} (\texttt{skip}, h')}$$

$$\frac{(\mathbb{C}_1, h) \xrightarrow{\eta} (\mathbb{C}_1', h')}{(\mathbb{C}_1 \| \mathbb{C}_2, h) \xrightarrow{\eta} (\mathbb{C}_1' \| \mathbb{C}_2, h')} \qquad \frac{(\mathbb{C}_2, h) \xrightarrow{\eta} (\mathbb{C}_2', h')}{(\mathbb{C}_1 \| \mathbb{C}_2, h) \xrightarrow{\eta} (\mathbb{C}_1 \| \mathbb{C}_2', h')}$$

$$\frac{}{(\texttt{skip} \| \texttt{skip}, h) \xrightarrow{\eta} (\texttt{skip}, h)} \qquad \frac{(\mathbb{C}_1, h) \xrightarrow{\eta} \lightning}{(\mathbb{C}_1; \mathbb{C}_2, h) \xrightarrow{\eta} \lightning} \qquad \frac{(\mathbb{C}_1, h) \xrightarrow{\eta} \lightning}{(\mathbb{C}_1 \| \mathbb{C}_2, h) \xrightarrow{\eta} \lightning}$$

$$\frac{(\mathbb{C}_2, h) \xrightarrow{\eta} \lightning}{(\mathbb{C}_1 \| \mathbb{C}_2, h) \xrightarrow{\eta} \lightning} \qquad \frac{f \notin \operatorname{dom} \eta}{(f, h) \xrightarrow{\eta} \lightning} \qquad \frac{(h, \lightning) \in c}{(c, h) \xrightarrow{\eta} \lightning} \qquad \frac{(\mathbb{C}, h) \xrightarrow{\eta}{}^* \lightning}{(\langle \mathbb{C} \rangle, h) \xrightarrow{\eta} \lightning}$$

Figure 7.15: Small-step operational semantics

The judgement $(\mathbb{C}, h) \xrightarrow{\eta}{}^* (\mathbb{C}', h')$ holds if, for some finite sequence of pairs $(\mathbb{C}_1, h_1), \ldots, (\mathbb{C}_n, h_n)$ with $(\mathbb{C}_1, h_1) = (\mathbb{C}, h)$ and $(\mathbb{C}_n, h_n) = (\mathbb{C}', h')$,

$$(\mathbb{C}_1, h_1) \xrightarrow{\eta} (\mathbb{C}_2, h_2) \xrightarrow{\eta} \ldots \xrightarrow{\eta} (\mathbb{C}_n, h_n).$$

The judgement $(\mathbb{C}, h) \xrightarrow{\eta}{}^* \lightning$ holds if, for some $(\mathbb{C}', h')$,

$$(\mathbb{C}, h) \xrightarrow{\eta}{}^* (\mathbb{C}', h') \xrightarrow{\eta} \lightning.$$

That is, $\xrightarrow{\eta}{}^*$ is the reflexive, transitive closure of $\xrightarrow{\eta}$.

While the operational semantics defines how programs transform heaps, the proof system makes assertions about how programs transform worlds. In fact, judgements of the proof system not only assert how programs transform worlds, but that they do so causing only interference consistent with the guarantee and tolerating interference consistent with the rely.

Configuration safety lifts the operational semantics to the level of worlds, taking into account interference. Configuration safety was developed by Vafeiadis [DYDG+10] as an approach to proving the soundness of RGSep [VP07, Vaf07] that avoids introducing an extra level of semantics. I follow his approach in §7.3.6 to establish the soundness of the concurrent abstract predicates proof system.

Definition 7.22 (Configuration Safety)**.** Let $\mathbb{C} \in \mathcal{L}$, $w \in \mathsf{World}$, $\eta \in \mathsf{FEnv}$ and $\mathcal{Q} \in \mathcal{P}(\mathsf{World})$. $\mathit{safe}_0(\mathbb{C}, w, \eta, \mathcal{Q})$ always holds. $\mathit{safe}_{n+1}(\mathbb{C}, w, \eta, \mathcal{Q})$ holds if and only if the following four conditions hold:

1. for all w' with $w\ \mathrm{R}^*\ w'$, $\mathit{safe}_n(\mathbb{C}, w', \eta, \mathcal{Q})$ holds;
2. $(\mathbb{C}, \lfloor w \rfloor_{\mathrm{H}}) \overset{\eta}{\not\to} \lightning$;
3. for all $\mathbb{C}', h'$ with $(\mathbb{C}, \lfloor w \rfloor_{\mathrm{H}}) \xrightarrow{\eta} (\mathbb{C}', h')$, there exists w' such that $w\ \widehat{\mathrm{G}}\ w'$, $h' = \lfloor w' \rfloor_{\mathrm{H}}$ and $\mathit{safe}_n(\mathbb{C}', w', \eta, \mathcal{Q})$ holds; and
4. if $\mathbb{C} = \mathtt{skip}$, then there exists w' such that $w\ \overline{\mathrm{G}}^*\ w'$ and $w' \in \mathcal{Q}$.

This definition asserts that a configuration is safe provided that: changing the world in a way that respects the rely is still safe; the program does not fault; if the program can make a step on the concrete heap corresponding to the world, there must be a corresponding step in the logical world that is permitted by the guarantee; and if the configuration has terminated then the postcondition holds, up to some repartitioning permitted by the guarantee.

Using configuration safety, I can now formally define the semantics of proof judgements operationally.

Definition 7.23 (Judgement Semantics)**.** $\Delta; \Gamma \models \{P\}\ \mathbb{C}\ \{Q\}$ holds if and only if, for all $n \in \mathbb{N}$, $i \in \mathsf{Interp}$, $\delta \in \mathsf{PEnv}$ with $[\![\Delta]\!]_{\delta,i} = \mathsf{World}$, $\eta \in [\![\Gamma]\!]_{n,\delta,i}$ and $w \in [\![P]\!]_{\delta,i}$, $\mathit{safe}_{n+1}\left(\mathbb{C}, w, \eta, [\![Q]\!]_{\delta,i}\right)$ holds, where

$$[\![\Gamma]\!]_{n,\delta,i} \overset{\text{def}}{=} \left\{ \eta \in \mathsf{FEnv} \;\middle|\; \begin{array}{r} \text{for all } \{P\}\ f\ \{Q\} \in \Delta \text{ and } w \in [\![P]\!]_{\delta,i}\,, \\ \mathit{safe}_n\left(\eta(f), w, \eta, [\![Q]\!]_{\delta,i}\right) \end{array} \right\}.$$

7.3.6 Soundness

In order to establish soundness of the proof system, I assume that each primitive command c is local on the heap. That is, each obeys the fault locality and safety monotonicity properties.

Assumption 7.1 (Primitive Locality)**.** Assume that each primitive update, c, satisfies the following conditions:

1. if $(h * h_1, \lightning) \in c$ then $(h_1, \lightning) \in c$; and
2. if $(h*h_1, h') \in c$ and $(h_1, \lightning) \notin c$ then there exists an h_2 such that $(h_1, h_2) \in c$ and $h' = h * h_2$.

Theorem 102 (Soundness)**.** *If* $\Delta; \Gamma \vdash \{P\}\ \mathbb{C}\ \{Q\}$ *then* $\Delta; \Gamma \models \{P\}\ \mathbb{C}\ \{Q\}$.

The proof is by induction on the structure of the derivation for $\Delta; \Gamma \vdash \{P\}\ \mathbb{C}\ \{Q\}$. The cases for Prim, Par, Frame, Atomic and Pred-E are covered by Lemmata 105, 118, 120, 121 and 122 respectively, detailed below. The remaining cases are (relatively) simple, and so are omitted.

Lemma 103 (Concrete Frame Property)**.** *The frame property holds for the operational semantics over single steps and multiple steps.*

- *If* $(\mathbb{C}, h * h_1) \xrightarrow{\eta} (\mathbb{C}', h')$ *and* $(\mathbb{C}, h_1) \overset{\eta}{\not\rightarrow} \lightning$*, then there exists an* h_2 *such that* $(\mathbb{C}, h_1) \xrightarrow{\eta} (\mathbb{C}', h_2)$ *and* $h' = h * h_2$.
- *If* $(\mathbb{C}, h * h_1) \xrightarrow{\eta^*} (\mathbb{C}', h')$ *and* $(\mathbb{C}, h_1) \overset{\eta}{\not\rightarrow} \lightning$*, then there exists an* h_2 *such that* $(\mathbb{C}, h_1) \xrightarrow{\eta^*} (\mathbb{C}', h_2)$ *and* $h' = h * h_2$.

Proof. The proof is by induction on the structure of the derivation or derivation sequence.

Consider the single-step case, where there is a derivation of $(\mathbb{C}, h * h_1) \xrightarrow{\eta} (\mathbb{C}', h')$. Consider the last rule applied in this derivation. For most of the rules, the derivation for $(\mathbb{C}, h_1) \xrightarrow{\eta} (\mathbb{C}', h_2)$ is either trivial or follows from the single-step case of the inductive hypothesis. For the primitive update case, the derivation for $(c, h_1) \xrightarrow{\eta} (\texttt{skip}, h_2)$ follows from Primitive Locality (Assumption 7.1). For the atomic case, the derivation for $(\langle \mathbb{C} \rangle, h_1) \xrightarrow{\eta} (\texttt{skip}, h_2)$ follows from the multiple-step case of the inductive hypothesis.

Consider the multiple-step case, where there is a derivations sequence for $(\mathbb{C}, h * h_1) \xrightarrow{\eta\ k} (\mathbb{C}', h')$. A derivation for $(\mathbb{C}, h_1) \xrightarrow{\eta^*} (\mathbb{C}', h_2)$ can be obtained by applying the single-step case of the inductive hypothesis to get the first step and the multiple-step case of the inductive hypothesis to get the remaining steps. □

Lemma 104 (Concrete Safety Monotonicity)**.** *Safety monotonicity holds for the operational semantics over single steps and multiple steps.*

- *If* $(\mathbb{C}, h * h_1) \xrightarrow{\eta} \lightning$ *then* $(\mathbb{C}, h_1) \xrightarrow{\eta} \lightning$.
- *If* $(\mathbb{C}, h * h_1) \xrightarrow{\eta^*} \lightning$ *then* $(\mathbb{C}, h_1) \xrightarrow{\eta^*} \lightning$.

Proof. The proof is by induction on the structure of the derivation or derivation sequence.

Consider the single-step case, where there is a derivation of $(\mathbb{C}, h * h_1) \xrightarrow{\eta} \lightning$. Consider the last rule applied in this derivation. For most of the rules, the derivation for $(\mathbb{C}, h_1) \xrightarrow{\eta} \lightning$ is either trivial or follows from the single-step case of the inductive hypothesis. For the primitive update case, the derivation for $(c, h_1) \xrightarrow{\eta} \lightning$ follows from Primitive Locality (Assumption 7.1). For the atomic case, the derivation for $(\langle \mathbb{C} \rangle, h_1) \xrightarrow{\eta} \lightning$ follows from the multiple-step case of the inductive hypothesis.

Consider the multiple-step case, where there is a derivation sequence for $(\mathbb{C}, h * h_1) \xrightarrow{\eta}{}^{*} \lightning$. Suppose this derivation has $k+1$ steps, and so, for some $\mathbb{C}', h'$, $(\mathbb{C}, h * h_1) \xrightarrow{\eta}{}^{k} (\mathbb{C}', h')$ and $(\mathbb{C}', h') \xrightarrow{\eta} \lightning$. A derivation sequence for $(\mathbb{C}, h_1) \xrightarrow{\eta}{}^{*} \lightning$ can be obtained by applying Lemma 103 and the single-step case of the inductive hypothesis. □

Lemma 105 (Primitive Safety)**.** *Suppose that* $(\mathbb{C}, h) \overset{\eta}{\not\rightarrow}{}^{*} \lightning$ *and for all* h' *with* $(\mathbb{C}, h) \xrightarrow{\eta}{}^{*} (\mathtt{skip}, h')$*,* $h' \in [\![q]\!]_i$*. Then for all* w *with* $(w_{\mathrm{L}})_{\mathrm{H}} = h$*,*

$$\mathit{safe}_n \left(\mathbb{C}, w, \eta, [\![q]\!]_{\delta,i} \right).$$

Proof. The proof is by induction on n. When $n = 0$, the result is trivial. Consider the inductive case. It must be that $w = ((h, \phi), s, \zeta)$, for some ϕ, s and ζ. Consider each clause of the definition of configuration safety.

Clause 1:

If $w \ \mathrm{R}^* \ w'$ then $(w'_{\mathrm{L}})_{\mathrm{H}}$ also, and so, by the inductive hypothesis,

$$\mathit{safe}_{n-1} \left(\mathbb{C}, w', \eta, [\![q]\!]_{\delta,i} \right).$$

Clause 2:

It must be that $\lfloor w \rfloor_{\mathrm{H}} = h * h_0$ for some h_0. Since $(\mathbb{C}, h) \overset{\eta}{\not\rightarrow} \lightning$, by Lemma 104 $(\mathbb{C}, \lfloor w \rfloor_{\mathrm{H}}) \overset{\eta}{\not\rightarrow} \lightning$.

Clause 3:

Suppose that $(\mathbb{C}, \lfloor w \rfloor_{\mathrm{H}}) \xrightarrow{\eta} (\mathbb{C}', h_1)$. By Lemma 103, there is some h' such that $(\mathbb{C}, h) \xrightarrow{\eta} (\mathbb{C}', h')$ and $h_1 = h' * h_0$. Let $w' = ((h', \phi), s, \zeta)$, which is well-formed by construction, having $\lfloor w' \rfloor_{\mathrm{H}} = h'$. Since only the local heap is changed, $w \ \widehat{\mathrm{G}} \ w'$. It must be that $(\mathbb{C}', h') \overset{\eta}{\not\rightarrow}{}^{*} \lightning$ and for all h'' with $(\mathbb{C}', h') \xrightarrow{\eta}{}^{*} (\mathtt{skip}, h'')$, $h'' \in [\![q]\!]_i$, and so, by the inductive hypothesis, $\mathit{safe}_{n-1} \left(\mathbb{C}', w', \eta, [\![q]\!]_{\delta,i} \right)$.

Clause 4:

Suppose that $\mathbb{C} = \mathtt{skip}$. It must be that $h = h' \in [\![q]\!]_i$. Since $(w_{\mathrm{L}})_{\mathrm{H}} = h$, it must be that $w \in [\![q]\!]_{\delta,i}$. Furthermore, $w \ \overline{\mathrm{G}}^{*} \ w$, as required. □

Lemma 106 (Skip Safety)**.** *If stable* $(\mathcal{Q})$ *and* $w \in \mathcal{Q}$ *then* $\mathit{safe}_n (\mathtt{skip}, w, \eta, \mathcal{Q})$*.*

Proof. The proof is by induction on n. When $n = 0$, the result is trivial. Consider the inductive case. By the definition of stability, any w' such that $w\ \mathrm{R}^*\ w'$ also satisfies $\mathcal{Q}$. Hence, the first clause of the definition of safety holds by the inductive hypothesis. The second and third clauses hold trivially since no reduction is possible from `skip`. The fourth clause holds simply by picking $w' = w$. □

Lemma 107. *If* $w_1\ \mathrm{G}\ w_1'$, $w_1 \bullet w_2$ *and* $w_1' \bullet w_2'$ *are defined, and* $(w_2)_{\mathrm{L}} = (w_2')_{\mathrm{L}}$, *then* $(w_1 \bullet w_2)\ \mathrm{G}\ (w_1' \bullet w_2')$.

Proof. By the definition of state composition, $(w_1)_{\mathrm{S}} = (w_1 \bullet w_2)_{\mathrm{S}}$ and $(w_1')_{\mathrm{S}} = (w_1' \bullet w_2')_{\mathrm{S}}$. Also, by permission composition, if $((w_1)_{\mathrm{L}})_{\mathrm{P}}(r, \gamma, \overrightarrow{v}) \in (0, 1]$ then $((w_1 \bullet w_2)_{\mathrm{L}})_{\mathrm{P}}(r, \gamma, \overrightarrow{v}) \in (0, 1]$, so any action permitted on w_1 is also permitted on $w_1 \bullet w_2$. If a region is created in w_1', then that region must not be in w_1, and so also not in $w_1 \bullet w_2$; hence its creation is permitted by the guarantee relation on the latter. If a region is destroyed in w_1' then all permission to it must be in $(w_1)_{\mathrm{L}}$ and so also in $(w_1 \bullet w_2)_{\mathrm{L}}$; hence its destruction is permitted by the guarantee relation on the latter. The fact that $(w_1 \bullet w_2)\ \mathrm{G}\ (w_1' \bullet w_2')$ then follows by the definition of the guarantee relation. □

Lemma 108. *If* $w_1\ \overline{\mathrm{G}}^*\ w_1'$ *and* $w_1 \bullet w_2$ *is defined then*

$$(w_1 \bullet w_2)\ \overline{\mathrm{G}}^*\ (w_1' \bullet ((w_2)_{\mathrm{L}}, (w_1')_{\mathrm{S}}, (w_1')_{\mathrm{A}})).$$

Proof. The proof is by induction on the number of $\overline{\mathrm{G}}$ steps. The base case is trivial; consider the inductive case of $n + 1$ steps.

For some w_1'', $w_1\ \overline{\mathrm{G}}^n\ w_1''\ \overline{\mathrm{G}}\ w_1'$. By the inductive hypothesis,

$$(w_1 \bullet w_2)\ \overline{\mathrm{G}}^*\ (w_1'' \bullet ((w_2)_{\mathrm{L}}, (w_1'')_{\mathrm{S}}, (w_1'')_{\mathrm{A}})).$$

By the definition of $\overline{\mathrm{G}}$, $\lfloor w_1'' \rfloor_{\mathrm{H}} = \lfloor w_1' \rfloor_{\mathrm{H}}$. Furthermore, for all $r, \gamma, \overrightarrow{v}$ with

$$\lfloor ((w_2)_{\mathrm{L}}, (w_1'')_{\mathrm{S}}, (w_1'')_{\mathrm{A}}) \rfloor_{\mathrm{P}}(r, \gamma, \overrightarrow{v}) > 0$$

it must be that $((w_1)_{\mathrm{L}})_{\mathrm{P}}(r, \gamma, \overrightarrow{v}) < 1$, and so the region r cannot be destroyed by the final $\overline{\mathrm{G}}$ step and its permissions are preserved. Thus $wf((w_2)_{\mathrm{L}}, (w_1')_{\mathrm{S}}, (w_1')_{\mathrm{A}})$. Moreover, $w_1' \bullet ((w_2)_{\mathrm{L}}, (w_1')_{\mathrm{S}}, (w_1')_{\mathrm{A}})$ is defined. Hence, by Lemma 107,

$$(w_1'' \bullet ((w_2)_{\mathrm{L}}, (w_1'')_{\mathrm{S}}, (w_1'')_{\mathrm{A}}))\ \overline{\mathrm{G}}\ (w_1' \bullet ((w_2)_{\mathrm{L}}, (w_1')_{\mathrm{S}}, (w_1')_{\mathrm{A}})),$$

and so $(w_1 \bullet w_2)\ \overline{\mathrm{G}}^*\ (w_1' \bullet ((w_2)_{\mathrm{L}}, (w_1')_{\mathrm{S}}, (w_1')_{\mathrm{A}}))$, as required. □

Lemma 109 (Repartitioning Guarantee Locality)**.** *If* $w_1\ \overline{\mathrm{G}}^*\ w_1'$, $w_2\ \mathrm{R}^*\ w_2'$, *and* $w_1 \bullet w_2$ *and* $w_1' \bullet w_2'$ *are defined, then* $(w_1 \bullet w_2)\ \overline{\mathrm{G}}^*\ (w_1' \bullet w_2')$.

Proof. It must be that $w_2' = ((w_2)_{\mathrm{L}}, (w_1')_{\mathrm{S}}, (w_1')_{\mathrm{A}})$ by the definitions of R and $\bullet$. Therefore the result follows directly from Lemma 108. □

Lemma 110 (Step Guarantee Locality)**.** *If* $w_1 \mathrel{\widehat{\mathrm{G}}} w_1'$, $w_2 \mathrel{\mathrm{R}^*} w_2'$, *and* $w_1 \bullet w_2$ *and* $w_1' \bullet w_2'$ *are defined, then* $(w_1 \bullet w_2) \mathrel{\widehat{\mathrm{G}}} (w_1' \bullet w_2')$.

Proof. For some w_1'', w_1''', $w_1 \mathrel{\overline{\mathrm{G}}^*} w_1''$ G $w_1''' \mathrel{\overline{\mathrm{G}}^*} w_1'$. Furthermore, it must be that $w_2' = ((w_2)_{\mathrm{L}}, (w_1')_{\mathrm{S}}, (w_1')_{\mathrm{A}})$ by the definitions of R and $\bullet$.

By Lemma 108, $(w_1 \bullet w_2) \mathrel{\overline{\mathrm{G}}^*} (w_1'' \bullet ((w_2)_{\mathrm{L}}, (w_1'')_{\mathrm{S}}, (w_1'')_{\mathrm{A}}))$. By the definition of $\overline{\mathrm{G}}$, $\lfloor w_1''' \rfloor_{\mathrm{H}} = \lfloor w_1' \rfloor_{\mathrm{H}}$. Furthermore, for all $r, \gamma, \overrightarrow{v}$ with $((w_2)_{\mathrm{L}})_{\mathrm{P}}(r, \gamma, \overrightarrow{v}) > 0$ it must be that $\lfloor w_1' \rfloor_{\mathrm{P}}(r, \gamma, \overrightarrow{v}) < 1$, and so the region r cannot be created between w_1''' and w_1', and its permissions must be preserved. Thus $wf((w_2)_{\mathrm{L}}, (w_1''')_{\mathrm{S}}, (w_1''')_{\mathrm{A}})$. Moreover, $w_1''' \bullet ((w_2)_{\mathrm{L}}, (w_1''')_{\mathrm{S}}, (w_1''')_{\mathrm{A}})$ is defined. Hence, by Lemma 107,

$$(w_1'' \bullet ((w_2)_{\mathrm{L}}, (w_1'')_{\mathrm{S}}, (w_1'')_{\mathrm{A}})) \text{ G } (w_1''' \bullet ((w_2)_{\mathrm{L}}, (w_1''')_{\mathrm{S}}, (w_1''')_{\mathrm{A}})).$$

By Lemma 108,

$$(w_1''' \bullet ((w_2)_{\mathrm{L}}, (w_1''')_{\mathrm{S}}, (w_1''')_{\mathrm{A}})) \mathrel{\overline{\mathrm{G}}^*} (w_1' \bullet w_2')$$

and so $(w_1 \bullet w_2) \mathrel{\widehat{\mathrm{G}}} (w_1' \bullet w_2')$, as required. □

Lemma 111 (Rely Decomposition)**.** *If* $w = w_1 \bullet w_2$ *and* $w \mathrel{\mathrm{R}^*} w'$, *then there exist* w_1', w_2' *such that* $w_1 \mathrel{\mathrm{R}^*} w_1'$, $w_2 \mathrel{\mathrm{R}^*} w_2'$ *and* $w' = w_1' \bullet w_2'$.

Proof. Suppose that w R w'; for multiple rely steps, the result follows from this case by induction. Let $w_1' = ((w_1)_{\mathrm{L}}, (w')_{\mathrm{S}}, (w')_{\mathrm{A}})$ and $w_2' = ((w_2)_{\mathrm{L}}, (w')_{\mathrm{S}}, (w')_{\mathrm{A}})$. By the definition of state composition, $(w_1)_{\mathrm{S}} = (w_2)_{\mathrm{S}} = w_{\mathrm{S}}$. Since a rely step does not affect the local state, $w' = w_1' \bullet w_2'$. Furthermore, w_1 and w_2 have no more permissions that w and so any change to the shared state allowed by the rely on w is also allowed by the rely on w_1 and w_2. Hence, $w_1 \mathrel{\mathrm{R}^*} w_1'$ and $w_2 \mathrel{\mathrm{R}^*} w_2'$. □

Corollary 112 (Stability Composition)**.** *If stable* $(\mathcal{Q}_1)$ *and stable* $(\mathcal{Q}_2)$ *then stable* $(\mathcal{Q}_1 \bullet \mathcal{Q}_2)$.

Proof. Suppose that $w \in \mathcal{Q}_1 \bullet \mathcal{Q}_2$ and w R w'. It must be that $w = w_1 \bullet w_2$ with $w_1 \in \mathcal{Q}_1$ and $w_2 \in \mathcal{Q}_2$. By Lemma 111, there are w_1' and w_2' with $w_1 \mathrel{\mathrm{R}^*} w_1'$, $w_2 \mathrel{\mathrm{R}^*} w_2'$ and $w = w_1' \bullet w_2'$. By the stability assumptions, $w_1' \in \mathcal{Q}_1$ and $w_2' \in \mathcal{Q}_2$, and so $w' \in \mathcal{Q}_1 \bullet \mathcal{Q}_2$, as required. □

Lemma 113. *If* $w_1 \bullet w_2$ *is defined,* w_1 G w_1', *and* $\lfloor w_1' \rfloor_{\mathrm{H}} * ((w_2)_{\mathrm{L}})_{\mathrm{H}}$ *is defined, then* $w_2' = ((w_2)_{\mathrm{L}}, (w_1')_{\mathrm{S}}, (w_1')_{\mathrm{A}})$ *is well-formed,* w_2 R w_2' *and* $w_1' \bullet w_2'$ *is defined.*

Proof. Suppose that an action is performed to transform w_1 to w_1'. Total permission is unchanged in performing an action, no shared regions are created or destroyed, the action model stays the same, and the total heap remains compatible with $((w_2)_{\mathrm{L}})_{\mathrm{H}}$, so $w_2' = ((w_2)_{\mathrm{L}}, (w_1')_{\mathrm{S}}, (w_1')_{\mathrm{A}})$ is well-formed. The action performed must have been identified by some token t with $((w_1)_{\mathrm{L}})_{\mathrm{P}}(t) > 0$. Since $w_1 \bullet w_2$ is defined, $\lfloor w_2 \rfloor_{\mathrm{P}}(t) < 1$ and so $w_2 \ \mathrm{R} \ w_2'$. Furthermore, $\lfloor w_1' \rfloor_{\mathrm{P}} \oplus ((w_2')_{\mathrm{L}})_{\mathrm{H}} = \lfloor w_1 \rfloor_{\mathrm{P}} \oplus ((w_2)_{\mathrm{L}})_{\mathrm{H}}$ is defined, and so $w_1' \bullet w_2'$ is defined.

Suppose that a new shared region is created to transform w_1 to w_1'. Total heap is unchanged, and the new region cannot have had any permissions in w_2, so $w_2' = ((w_2)_{\mathrm{L}}, (w_1')_{\mathrm{S}}, (w_1')_{\mathrm{A}})$ is well-formed. Rely permits such a region to be created, and so $w_2 \ \mathrm{R} \ w_2'$. Furthermore, $w_1' \bullet w_2'$ is defined.

Suppose that a shared region is destroyed to transform w_1 to w_1'. Then no reference to this region can be present in $(w_2)_{\mathrm{L}}$, so $w_2' = ((w_2)_{\mathrm{L}}, (w_1')_{\mathrm{S}}, (w_1')_{\mathrm{A}})$ is well-formed. Rely permits such region destruction, and so $w_2 \ \mathrm{R} \ w_2'$. Furthermore, $w_1' \bullet w_2'$ is defined. □

Lemma 114 (Guarantee/Rely Compatability)**.** *If $w_1 \bullet w_2$ is defined and $w_1 \ \overline{\mathrm{G}}^* \ w_1'$ then there exists w_2' with $w_2 \ \mathrm{R}^* \ w_2'$ and $w_1' \bullet w_2'$ defined.*

Proof. The particular choice of w_2' is $((w_2)_{\mathrm{L}}, (w_1')_{\mathrm{S}}, (w_1')_{\mathrm{A}})$. The result follows from Lemma 113 by induction on the number of $\overline{\mathrm{G}}$ steps, observing that, whenever $w_1 \bullet w_2$ is defined and $w_1 \ \overline{\mathrm{G}}^* \ w_1'$, $\lfloor w_1' \rfloor_{\mathrm{H}} * ((w_2)_{\mathrm{L}})_{\mathrm{H}}$ is defined. □

Lemma 115. *If $w_1 \bullet w_2$ is defined, $w_1 \ \widehat{\mathrm{G}} \ w_2$, and $\lfloor w_1' \rfloor_{\mathrm{H}} * ((w_2)_{\mathrm{L}})_{\mathrm{H}}$ is defined, then $w_2' = ((w_2)_{\mathrm{L}}, (w_1')_{\mathrm{S}}, (w_1')_{\mathrm{A}})$ is well-formed, $w_2 \ \mathrm{R}^* \ w_2'$ and $w_1' \bullet w_2'$ is defined.*

Proof. This result also follows from Lemma 113 by induction on the number of $\overline{\mathrm{G}}$ steps. □

Lemma 116 (Abstract Frame Property)**.** *If*

$$(\mathbb{C}, \lfloor w_1 \bullet w_2 \rfloor_{\mathrm{H}}) \xrightarrow{\eta} (\mathbb{C}', h')$$
$$\mathit{safe}_{n+1}(\mathbb{C}, w_1, \eta, \mathcal{Q})$$

then there exist w_1', w_2' such that

$$(\mathbb{C}, \lfloor w_1 \rfloor_{\mathrm{H}}) \xrightarrow{\eta} (\mathbb{C}', \lfloor w_1' \rfloor_{\mathrm{H}})$$
$$h' = \lfloor w_1' \bullet w_2' \rfloor_{\mathrm{H}}$$
$$w_1 \ \widehat{\mathrm{G}} \ w_1' \qquad w_2 \ \mathrm{R}^* \ w_2'$$
$$\mathit{safe}_n(\mathbb{C}', w_1', \eta, \mathcal{Q}).$$

Proof. Let $h_2 = ((w_2)_{\mathrm{L}})_{\mathrm{H}}$. By the definition of world composition, $\lfloor w_1 \bullet w_2 \rfloor_{\mathrm{H}} = \lfloor w_1 \rfloor_{\mathrm{H}} * h_2$.

By Concrete Frame (Lemma 103), there exists an h_1' such that

$$(\mathbb{C}, \lfloor w_1 \rfloor_{\mathrm{H}}) \xrightarrow{\eta} (\mathbb{C}', h_1')$$

and $h_1' * h_2 = h'$. By the definition of safety (third clause), there must exist a w_1' such that $w_1 \mathrel{\widehat{\mathrm{G}}} w_1'$, $\lfloor w_1' \rfloor_{\mathrm{H}} = h_1'$ and $\mathit{safe}_n(\mathbb{C}', w_1', \eta, \mathcal{Q})$.

Now let $w_2' = ((w_2)_{\mathrm{L}}, (w_1')_{\mathrm{S}}, (w_1')_{\mathrm{A}})$. By Lemma 113, w_2' is well-formed, $w_2 \mathrel{\mathrm{R}^*} w_2'$ and $w_1' \bullet w_2'$ is defined. By construction $h' = h_1' * h_2 = \lfloor w_1' \rfloor_{\mathrm{H}} * ((w_2')_{\mathrm{L}})_{\mathrm{H}} = \lfloor w_1' \bullet w_2' \rfloor_{\mathrm{H}}$, as required. □

Lemma 117 (Parallel Safety)**.** *If*

$$w = w_1 \bullet w_2 \tag{7.1}$$

$$\mathit{safe}_n(\mathbb{C}_1, w_1, \eta, \mathcal{Q}_1) \tag{7.2}$$

$$\mathit{safe}_n(\mathbb{C}_2, w_2, \eta, \mathcal{Q}_2) \tag{7.3}$$

$$\mathit{stable}(\mathcal{Q}_1) \tag{7.4}$$

$$\mathit{stable}(\mathcal{Q}_2) \tag{7.5}$$

then

$$\mathit{safe}_n(\mathbb{C}_1 \parallel \mathbb{C}_2, w, \eta, \mathcal{Q}_1 \bullet \mathcal{Q}_2)\,.$$

Proof. The proof is by induction on n. When $n = 0$, the result is trivial. Consider the inductive case. Consider each clause of the definition of configuration safety.

Clause 1: if $w \mathrel{\mathrm{R}^*} w'$ then $\mathit{safe}_{n-1}(\mathbb{C}_1 \parallel \mathbb{C}_2, w', \eta, \mathcal{Q}_1 \bullet \mathcal{Q}_2)$.
By Rely Decomposition (Lemma 111), there exist w_1' and w_2' with $w_1 \mathrel{\mathrm{R}^*} w_1'$ and $w_2 \mathrel{\mathrm{R}^*} w_2'$. By (7.2) and (7.3), this implies that $\mathit{safe}_{n-1}(\mathbb{C}_1, w_1', \eta, \mathcal{Q}_1)$ and $\mathit{safe}_{n-1}(\mathbb{C}_2, w_2', \eta, \mathcal{Q}_2)$. Hence, by the inductive hypothesis,

$$\mathit{safe}_{n-1}(\mathbb{C}_1 \parallel \mathbb{C}_2, w', \eta, \mathcal{Q}_1 \bullet \mathcal{Q}_2)\,.$$

Clause 2: $(\mathbb{C}_1 \parallel \mathbb{C}_2, \lfloor w_1 \bullet w_2 \rfloor_{\mathrm{H}}) \overset{\eta}{\not\to} \lightning$.
If $(\mathbb{C}_1 \parallel \mathbb{C}_2, \lfloor w_1 \bullet w_2 \rfloor_{\mathrm{H}}) \xrightarrow{\eta} \lightning$ then, by the operational semantics, it must be that either $(\mathbb{C}_1, \lfloor w_1 \bullet w_2 \rfloor_{\mathrm{H}}) \xrightarrow{\eta} \lightning$ or $(\mathbb{C}_2, \lfloor w_1 \bullet w_2 \rfloor_{\mathrm{H}}) \xrightarrow{\eta} \lightning$. By Concrete Safety Monotonicity (Lemma 104), this would mean that either $(\mathbb{C}_1, \lfloor w_1 \rfloor_{\mathrm{H}}) \xrightarrow{\eta} \lightning$ or $(\mathbb{C}_2, \lfloor w_2 \rfloor_{\mathrm{H}}) \xrightarrow{\eta} \lightning$, but this cannot be the case by (7.2) and (7.3). Therefore, $(\mathbb{C}_1 \parallel \mathbb{C}_2, \lfloor w_1 \bullet w_2 \rfloor_{\mathrm{H}}) \overset{\eta}{\not\to} \lightning$.

Clause 3: if $(\mathbb{C}_1 \parallel \mathbb{C}_2, \lfloor w_1 \bullet w_2 \rfloor_{\mathrm{H}}) \xrightarrow{\eta} (\mathbb{C}', h')$ then $w \mathrel{\widehat{\mathrm{G}}} w'$, $h' = \lfloor w' \rfloor_{\mathrm{H}}$ and $\mathit{safe}_{n-1}(\mathbb{C}', w', \eta, \mathcal{Q}_1 \bullet \mathcal{Q}_2)$, for some w'.

There are two cases to consider here: either $\mathbb{C}_1 = \mathbb{C}_2 = \texttt{skip}$, or $\mathbb{C}_1$ or $\mathbb{C}_2$ is executed for one step.

In the first case, the semantics ensure that $\mathbb{C}' = \texttt{skip}$ and $h' = \lfloor w \rfloor_{\mathrm{H}}$. If $n-1 = 0$ then $w' = w$ suffices trivially; assume that $n-1 > 0$. By (7.2) (clause 4), there exist w_1' with $w_1 \; \overline{\mathrm{G}}^* \; w_1'$ and $w_1' \in \mathcal{Q}_1$. By Lemma 114, there exists w_2' with $w_2 \; \mathrm{R}^* \; w_2'$ and $w_1' \bullet w_2'$ defined. By (7.3) (clause 1), $\mathit{safe}_{n-1}\,(\texttt{skip}, w_2', \eta, \mathcal{Q}_2)$ holds. Since $n-1 > 0$, this means that there is some w_2'' such that $w_2' \; \overline{\mathrm{G}}^* \; w_2''$ and $w_2'' \in \mathcal{Q}_2$. By Lemma 114, there exists w_1'' with $w_1' \; \mathrm{R}^* \; w_1''$ and $w_1'' \bullet w_2''$ defined. By (7.4), $w_1'' \in \mathcal{Q}_1$, and so $w_1'' \bullet w_2'' \in \mathcal{Q}_1 \bullet \mathcal{Q}_2$. By Lemma 109, $(w_1 \bullet w_2) \; \overline{\mathrm{G}}^* \; (w_1' \bullet w_2')$ and $(w_1' \bullet w_2') \; \overline{\mathrm{G}}^* \; (w_1'' \bullet w_2'')$, and so taking $w' = w_1'' \bullet w_2''$ gives $w \; \overline{\mathrm{G}}^* \; w'$. By the definition of $\overline{\mathrm{G}}$, $\lfloor w' \rfloor_{\mathrm{H}} = \lfloor w \rfloor_{\mathrm{H}} = h'$. Applying Corollary 112 to (7.4) and (7.5) gives that $\mathit{stable}\,(\mathcal{Q}_1 \bullet \mathcal{Q}_2)$. Finally, by Skip Safety (Lemma 106), $\mathit{safe}_{n-1}\,(\mathbb{C}', w', \eta, \mathcal{Q}_1 \bullet \mathcal{Q}_2)$, as required.

For the second case, assume that $\mathbb{C}_1$ is executed for one step; the case for $\mathbb{C}_2$ is essentially identical. The semantics require that $(\mathbb{C}_1, \lfloor w_1 \bullet w_2 \rfloor_{\mathrm{H}}) \xrightarrow{\eta} (\mathbb{C}_1', h')$ for some $\mathbb{C}_1'$ with $\mathbb{C}' = \mathbb{C}_1' \parallel \mathbb{C}_2$. By the Abstract Frame Property (Lemma 116), given (7.2), there are w_1' and w_2' with

$$(\mathbb{C}_1, \lfloor w_1 \rfloor_{\mathrm{H}}) \xrightarrow{\eta} (\mathbb{C}_1', \lfloor w_1' \rfloor_{\mathrm{H}}) \tag{7.6}$$

$$h' = \lfloor w_1' \bullet w_2' \rfloor_{\mathrm{H}} \tag{7.7}$$

$$w_1 \; \widehat{\mathrm{G}} \; w_1' \qquad w_2 \; \mathrm{R}^* \; w_2' \tag{7.8}$$

$$\mathit{safe}_{n-1}\,(\mathbb{C}_1', w_1', \eta, \mathcal{Q}_1)\,. \tag{7.9}$$

Let $w' = w_1' \bullet w_2'$, so by definition $h' = \lfloor w' \rfloor_{\mathrm{H}}$. Furthermore, given (7.8), Lemma 110 implies that $w \; \widehat{\mathrm{G}} \; w'$. From (7.3) (clause 1), $\mathit{safe}_{n-1}\,(\mathbb{C}_2, w_2', \eta, \mathcal{Q}_2)$ holds. Finally, applying the inductive hypothesis with this and (7.9) gives $\mathit{safe}_{n-1}\,(\mathbb{C}', w', \eta, \mathcal{Q}_1 \bullet \mathcal{Q}_2)$, as required.

Clause 4 holds trivially since $\mathbb{C}_1 \parallel \mathbb{C}_2$ is not `skip`. □

Lemma 118 (Parallel Soundness)**.** *If*

$$\Delta \models Q_1 \rightarrow \boxed{\mathbf{R}} Q_1 \qquad \Delta; \Gamma \models \{P_1\}\ \mathbb{C}_1\ \{Q_1\}$$
$$\Delta \models Q_2 \rightarrow \boxed{\mathbf{R}} Q_2 \qquad \Delta; \Gamma \models \{P_2\}\ \mathbb{C}_2\ \{Q_2\}$$

then $\Delta; \Gamma \models \{P_1 * P_2\}\ \mathbb{C}_1 \parallel \mathbb{C}_2\ \{Q_1 * Q_2\}$.

Proof. Fix n, i, δ with $[\![\Delta]\!]_{\delta,i} = \mathsf{World}$, $\eta \in [\![\Gamma]\!]_{n,\delta,i}$ and $w \in [\![P_1 * P_1]\!]_{\delta,i}$. It must be that $w = w_1 \bullet w_2$ for some w_1, w_2 with $w_1 \in [\![P_1]\!]$ and $w_2 \in [\![P_2]\!]$. By the assumptions, $\mathit{safe}_{n+1}\left(\mathbb{C}_1, w_1, \eta, [\![Q_1]\!]_{\delta,i}\right)$ and $\mathit{safe}_{n+1}\left(\mathbb{C}_2, w_2, \eta, [\![Q_2]\!]_{\delta,i}\right)$. Furthermore, $\mathit{stable}\left([\![Q_1]\!]_{\delta,i}\right)$ and $\mathit{stable}\left([\![Q_2]\!]_{\delta,i}\right)$, and so, by Parallel Safety (Lemma 117), $\mathit{safe}_n\left(\mathbb{C}_1 \parallel \mathbb{C}_2, w, \eta, [\![Q_1 * Q_2]\!]_{\delta,i}\right)$, as required. □

Remark. The PAR rule requires that the postconditions of the two threads be stable, but why? The real problem is that each thread is allowed to make repartitioning steps before the two threads join, so which thread gets the last repartitioning? Whichever it is, all of the other threads must have postconditions that are stable under this repartitioning. Enforcing that all thread postconditions are stable ensures this.

Lemma 119 (Frame Safety)**.** *If*

$$w = w_1 \bullet w_2 \tag{7.10}$$

$$\mathit{safe}_n\,(\mathbb{C}, w_1, \eta, \mathcal{Q}_1) \tag{7.11}$$

$$w_2 \in \mathcal{Q}_2 \tag{7.12}$$

$$\mathit{stable}\,(\mathcal{Q}_2) \tag{7.13}$$

then

$$\mathit{safe}_n\,(\mathbb{C}, w, \eta, \mathcal{Q}_1 \bullet \mathcal{Q}_2)\,.$$

Proof. The proof is by induction on n. When $n = 0$, the result is trivial. Consider the inductive case. Consider each clause of the definition of configuration safety.

Clause 1: if $w\ \mathrm{R}^*\ w'$ then $\mathit{safe}_{n-1}\,(\mathbb{C}, w'', \eta, \mathcal{Q}_1 \bullet \mathcal{Q}_2)$.
By Rely Decomposition (Lemma 111), there exists w_1' and w_2' with $w_1\ \mathrm{R}^*\ w_1'$ and $w_2\ \mathrm{R}^*\ w_2'$. By (7.11) (clause 1), $\mathit{safe}_{n-1}\,(\mathbb{C}, w_1', \eta, \mathcal{Q}_1)$. By (7.13), $w_2' \in \mathcal{Q}_2$. Hence, by the inductive hypothesis, $\mathit{safe}_{n-1}\,(\mathbb{C}, w', \eta, \mathcal{Q}_1 \bullet \mathcal{Q}_2)$.

Clause 2: $(\mathbb{C}, \lfloor w_1 \bullet w_2 \rfloor_{\mathrm{H}}) \overset{\eta}{\not\to} \lightning$.
If $(\mathbb{C}, \lfloor w_1 \bullet w_2 \rfloor_{\mathrm{H}}) \overset{\eta}{\to} \lightning$ then Concrete Safety Monotonicity (Lemma 104) would mean that $(\mathbb{C}, \lfloor w_1 \rfloor_{\mathrm{H}}) \overset{\eta}{\to} \lightning$. This cannot be the case by (7.11) (clause 2), and so $(\mathbb{C}, \lfloor w_1 \bullet w_2 \rfloor_{\mathrm{H}}) \overset{\eta}{\not\to} \lightning$.

Clause 3: if $(\mathbb{C}, \lfloor w_1 \bullet w_2 \rfloor_{\mathrm{H}}) \overset{\eta}{\to} (\mathbb{C}', h')$ then $w\ \widehat{\mathrm{G}}\ w'$, $h' = \lfloor w' \rfloor_{\mathrm{H}}$ and $\mathit{safe}_{n-1}\,(\mathbb{C}', w', \eta, \mathcal{Q}_1 \bullet \mathcal{Q}_2)$, for some w'.
By the Abstract Frame Property (Lemma 116), given (7.11), there exist w_1 and w_2 such that

$$(\mathbb{C}, \lfloor w_1 \rfloor_{\mathrm{H}}) \overset{\eta}{\to} (\mathbb{C}', \lfloor w_1' \rfloor_{\mathrm{H}}) \tag{7.14}$$

$$h' = \lfloor w_1' \bullet w_2' \rfloor_{\mathrm{H}} \tag{7.15}$$

$$w_1\ \widehat{\mathrm{G}}\ w_1' \qquad w_2\ \mathrm{R}^*\ w_2' \tag{7.16}$$

$$\mathit{safe}_{n-1}\,(\mathbb{C}', w_1', \eta, \mathcal{Q}_1)\,. \tag{7.17}$$

Let $w' = w'_1 \bullet w'_2$, so by definition $h' = \lfloor w' \rfloor_{\mathrm{H}}$. Furthermore, given (7.16), Lemma 110 implies that $w \mathrel{\widehat{\mathrm{G}}} w'$. By (7.13), $w'_2 \in \mathcal{Q}_2$, and so, by the inductive hypothesis, $\mathit{safe}_{n-1}(\mathbb{C}', w', \eta, \mathcal{Q}_1 \bullet \mathcal{Q}_2)$, as required.

Clause 4: if $\mathbb{C} = \texttt{skip}$ then $w \mathrel{\overline{\mathrm{G}}^*} w'$ for some $w' \in \mathcal{Q}_1 \bullet \mathcal{Q}_2$.
By (7.11), there exists $w'_1 \in \mathcal{Q}_1$ with $w_1 \mathrel{\overline{\mathrm{G}}^*} w'_1$. By Lemma 114, there exists w'_2 with $w_2 \mathrel{\mathrm{R}^*} w'_2$ and $w'_1 \bullet w'_2$ defined. Let $w = w'_1 \bullet w'_2$. By (7.13), $w'_2 \in \mathcal{Q}_2$, and so $w \in \mathcal{Q}_1 \bullet \mathcal{Q}_2$, as required. □

Lemma 120 (Frame Soundness)**.** *If*

$$\Delta \models R \to \boxed{\mathbf{R}} R$$
$$\Delta; \Gamma \models \{P\}\ \mathbb{C}\ \{Q\}$$

then

$$\Delta; \Gamma \models \{R * P\}\ \mathbb{C}\ \{R * Q\}\,.$$

Proof. Fix n, i, δ with $[\![\Delta]\!]_{\delta,i} = \mathsf{World}$, $\eta \in [\![\Gamma]\!]_{n,\delta,i}$ and $w \in [\![R * P]\!]_{\delta,i}$. It must be that $w = w_1 \bullet w_2$ for some $w_1 \in P$ and $w_2 \in R$. Furthermore, $\mathit{safe}_n\left(\mathbb{C}, w_1, \eta, [\![Q]\!]_{\delta,i}\right)$ and $\mathit{stable}\left([\![R]\!]_{\delta,i}\right)$ hold. Thus, by Frame Safety (Lemma 119), $\mathit{safe}_n\left(\mathbb{C}, w, \eta, [\![R * Q]\!]_{\delta,i}\right)$. □

Lemma 121 (Atomic Soundness)**.** *If*

$$\vdash_{\mathsf{SL}} \{p\}\ \mathbb{C}\ \{q\}$$

then

$$\Delta; \Gamma \models \left\{\boxed{\mathbf{R}}\left\langle \frac{p}{q} \right\rangle Q\right\}\ \langle \mathbb{C} \rangle\ \left\{ \blacklozenge_{\mathbf{R}} Q \right\}.$$

Proof. It is necessary to show that, for all n, i, δ with $[\![\Delta]\!]_{\delta,i} = \mathsf{World}$, $\eta \in [\![\Gamma]\!]_{n,\delta,i}$ and $w \in \left[\!\left[\boxed{\mathbf{R}}\left\langle \frac{p}{q} \right\rangle Q\right]\!\right]_{\delta,i}$, $\mathit{safe}_n\left(\langle \mathbb{C} \rangle, w, \eta, \left[\!\left[\blacklozenge_{\mathbf{R}} Q \right]\!\right]_{\delta,i}\right)$. The proof of this is by induction on n. The case where $n = 0$ is trivial; consider the inductive case. Consider each clause of the definition of configuration safety.

Clause 1: if $w \mathrel{\mathrm{R}^*} w'$ then $\mathit{safe}_{n-1}\left(\langle \mathbb{C} \rangle, w', \eta, \left[\!\left[\blacklozenge_{\mathbf{R}} Q \right]\!\right]_{\delta,i}\right)$.
Fix w'' with $w' \mathrel{\mathrm{R}^*} w''$. By transitivity, $w \mathrel{\mathrm{R}^*} w''$, and so $w'' \in \left[\!\left[\left\langle \frac{p}{q} \right\rangle Q\right]\!\right]_{\delta,i}$. Since the choice of w'' was arbitrary, $w' \in \left[\!\left[\boxed{\mathbf{R}}\left\langle \frac{p}{q} \right\rangle Q\right]\!\right]_{\delta,i}$. Thus, by the inductive hypothesis, $\mathit{safe}_{n-1}\left(\langle \mathbb{C} \rangle, w', \eta, \left[\!\left[\blacklozenge_{\mathbf{R}} Q \right]\!\right]_{\delta,i}\right)$, as required.

Clause 2: $(\langle \mathbb{C} \rangle, \lfloor w \rfloor_{\mathrm{H}}) \overset{\eta}{\not\to} \lightning$.
By definition, there exist $h_1 \in [\![p]\!]_i$ and h' such that $\lfloor w \rfloor_{\mathrm{H}} = h_1 * h'$. By the

soundness of separation logic, $(\mathbb{C}, h_1) \overset{\eta}{\not\to}^* \lightning$. By Concrete Safety Monotonicity, therefore, $(\mathbb{C}, \lfloor w \rfloor_{\mathrm{H}}) \overset{\eta}{\not\to}^* \lightning$. Hence, $(\langle \mathbb{C} \rangle, \lfloor w \rfloor_{\mathrm{H}}) \overset{\eta}{\not\to} \lightning$, as required.

Clause 3: if $(\langle \mathbb{C} \rangle, \lfloor w \rfloor_{\mathrm{H}}) \xrightarrow{\eta} (\mathbb{C}', h')$ then $w \mathrel{\widehat{\mathrm{G}}} w'$, $h' = \lfloor w' \rfloor_{\mathrm{H}}$ and

$$\mathit{safe}_{n-1}\left(\mathbb{C}', w', \eta, \left[\!\left[\diamondsuit Q \right]\!\right]_{\delta,i}\right),$$

for some w'.

It must be the case that $\mathbb{C}' = \texttt{skip}$ and $(\mathbb{C}, \lfloor w \rfloor_{\mathrm{H}}) \xrightarrow{\eta}^* (\texttt{skip}, h')$. By definition, there exist $h_1 \in [\![p]\!]_i$ and h_0 such that $\lfloor w \rfloor_{\mathrm{H}} = h_1 * h_0$. By the soundness of separation logic, $(\mathbb{C}, h_1) \overset{\eta}{\not\to}^* \lightning$. Hence, by the Concrete Frame Property (Lemma 103) there must be some h_2 such that $(\mathbb{C}, h_1) \xrightarrow{\eta}^* (\texttt{skip}, h_2)$ and $h' = h_2 * h_0$. By the soundness of separation logic again, $h_2 \in [\![q]\!]_i$. Thus, there exists $w' \in [\![Q]\!]_{\delta,i}$ with $\lfloor w' \rfloor_{\mathrm{H}} = h_2 * h_0 = h'$ and $w \mathrel{\widehat{\mathrm{G}}} w'$. This fulfils the first two requirements; for the final requirement, note that $w' \in \left[\!\left[\diamondsuit Q \right]\!\right]_{\delta,i}$ and $\mathit{stable}\left(\left[\!\left[\diamondsuit Q \right]\!\right]_{\delta,i}\right)$, and so, by Skip Safety (Lemma 106), $\mathit{safe}_{n-1}\left(\mathbb{C}', w', \eta, \left[\!\left[\diamondsuit Q \right]\!\right]_{\delta,i}\right)$, as required.

Clause 4 holds trivially, since $\langle \mathbb{C} \rangle$ is not $\texttt{skip}$. □

Lemma 122 (Predicate Elimination Soundness)**.** *If*

$$\alpha \notin \Delta, \Gamma, P, Q$$
$$\Delta \Uparrow \alpha, \overrightarrow{x}, R$$
$$\Delta \wedge (\forall \overrightarrow{x}.\, \alpha(\overrightarrow{x}) \leftrightarrow R); \Gamma \models \{P\}\ \mathbb{C}\ \{Q\}$$

then

$$\Delta; \Gamma \models \{P\}\ \mathbb{C}\ \{Q\}$$

Proof. Fix n, i, δ with $[\![\Delta]\!]_{\delta,i} = \mathsf{World}$, $\eta \in [\![\Gamma]\!]_{n,\delta,i}$ and $w \in [\![P]\!]_{\delta,i}$. Since $\Delta \Uparrow \alpha, \overrightarrow{x}, R$, there exists δ' with

$$\delta'(\alpha', \overrightarrow{v}) = \begin{cases} [\![R]\!]_{\delta, \prime i[\overrightarrow{x} \mapsto v]} & \text{if } \alpha' = \alpha \\ \delta(\alpha', \overrightarrow{v}) & \text{otherwise.} \end{cases}$$

Since α is fresh, $[\![\Delta]\!]_{\delta',i} = \mathsf{World}$, and by construction, $[\![\forall \overrightarrow{x}.\, \alpha(\overrightarrow{x}) \leftrightarrow R]\!]_{\delta',i} = \mathsf{World}$, so $[\![\Delta \wedge (\forall \overrightarrow{x}.\, \alpha(\overrightarrow{x}) \leftrightarrow R)]\!]_{\delta',i} = \mathsf{World}$. Also, since α is fresh, $[\![\Gamma]\!]_{n,\delta',i} = [\![\Gamma]\!]_{n,\delta,i}$, $[\![P]\!]_{\delta',i} = [\![P]\!]_{\delta,i}$ and $[\![Q]\!]_{\delta',i} = [\![Q]\!]_{\delta,i}$. Therefore $\eta \in [\![\Gamma]\!]_{n,\delta',i}$ and $w \in [\![P]\!]_{\delta',i}$, and $\mathit{safe}_{n+1}\left(\mathbb{C}, w, \eta, [\![Q]\!]_{\delta',i}\right)$ as required. □

Chapter 8

Conclusions

Throughout this thesis, I have considered numerous problems concerning abstract data structures, expressivity and decidability problems of spatial logics for describing them, and proving sequential and concurrent programs that implement them. In doing so, I have answered numerous questions and found new techniques and solutions, but have also turned up new and interesting questions along the way.

As to expressivity, I have shown that the adjunct connectives can be eliminated from multi-holed context logic for trees, but not for single-holed context logic. The question for single-holed context logic with context composition, however, remains open and appears to be a difficult one. A related open problem is the question of whether or not multi-holed context logic is more expressive than single-holed context logic (when restricted to pure trees).

On the matter of decidability, I have demonstrated decision procedures for multi-holed context logic on trees, terms and sequences, including quantification over hole labels. Open questions include the lower bounds on the complexity of decidability, whether efficient decision procedures can be implemented in practice, and whether these decision procedures can be usefully deployed toward automating reasoning about programming. Some significant hurdles are likely involved in this last point, since quantification is common in reasoning about programs, but tends to make validity undecidable.

Concerning reasoning about programs that implement abstract data structures, I have presented and illustrated two techniques for establishing the correctness of data structure implementations that present a local view of the abstract structure. An interesting question is whether abstract predicates can be extended to express these techniques, perhaps by embedding abstract predicates in a form of dynamic logic. The concurrency implications of locality

refinement also remain to be explored. In ongoing work, Wheelhouse's is applying the locality-preserving technique to segment logic [GW09], which will hopefully shed some light in this area.

I also presented concurrent abstract predicates, a proof system that combines an abstraction mechanism with a fine-grained low-level permissions system. Already, this work has seen a number of interesting developments. Dodds, Jagannathan and Parkinson have applied concurrent abstract predicates to reasoning about deterministic parallelism [DJP11] — the injection of concurrency into sequential code in a manner that preserves the semantics of the original program. They extended the system with *higher-order abstract predicates*, that is, predicates that are themselves parametrised by assertions, which permit more elegant specifications of concurrent modules. For instance, a lock predicate can specify the exact resource it is protecting, rather than relying on the client to play token games to retrieve the resource from a shared state. This can drastically simplify client proofs; for instance, the set implementations discussed in §7.2 can be verified without explicit shared state assertions.

da Rocha Pinto *et al.* have applied concurrent abstract predicates to reasoning about concurrent indexes [da 10, dRPDYGW11], which are pervasive in computer systems such as databases and file systems. In particular, da Rocha Pinto showed in his masters' thesis that a sophisticated concurrent B-tree correctly implements an abstract index specification using concurrent abstract predicates.

A more theoretical question concerning the CAP permissions system is how it can be generalised. The soundness proof presented in §7.3.6 appears to depend on a few key properties of the resource model, the rely and guarantee relations and their relation to the underlying operational semantics, suggesting that such a generalisation should be possible.

Bibliography

[BCOP05] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *POPL*, pages 259–270, 2005.

[BK10] James Brotherston and Max Kanovich. Undecidability of propositional separation logic and its neighbours. In *Proceedings of LICS-25*, pages 137–146, 2010.

[BKMW01] A. Brüggemann-Klein, M. Murata, and D. Wood. Regular tree and regular hedge languages over unranked alphabets: Version 1, 2001.

[Boj07] Mikołaj Bojańczyk. Forest expressions. In *CSL '07: Proceedings of the 16th EACSL Annual Conference on Computer Science and Logic*, 2007. To appear.

[Boy03] John Boyland. Checking interference with fractional permissions. In *SAS*, 2003.

[BS09] Michael Benedikt and Luc Segoufin. Regular tree languages definable in fo and in fomod. *ACM Transactions on Computational Logic*, 11(1):1–32, 2009.

[Büc60] J. R. Büchi. Weak second-order arithmetic and finite automata. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 6:66–92, 1960.

[BW06] Mikolaj Bojańczyk and Igor Walukiewicz. Forest algebras, 2006. http://hal.archives-ouvertes.fr/hal-00105796/en/.

[CCG03] Cristiano Calcagno, Luca Cardelli, and Andrew D. Gordon. Deciding validity in a spatial logic for trees. *SIGPLAN Not.*, 38(3):62–73, 2003.

[CDG+07] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 2007. release October, 12th 2007.

[CDYG07] Cristiano Calcagno, Thomas Dinsdale-Young, and Philippa Gardner. Adjunct elimination in Context Logic for trees. In *APLAS 2007*, LNCS, Heidelberg, 2007. Springer. To appear.

[CDYG10] Cristiano Calcagno, Thomas Dinsdale-Young, and Philippa Gardner. Adjunct elimination in context logic for trees. *Inf. Comput.*, 208:474–499, May 2010.

[CG00] Luca Cardelli and Andrew D. Gordon. Anytime, anywhere: modal logics for mobile ambients. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 365–377, New York, NY, USA, 2000. ACM Press.

[CG04] G. Conforti and G. Ghelli. Decidability of freshness, undecidability of revelation. In *Proc. of Foundations of Software Science and Computation Structures (FOSSACS04)*, LNCS 2987, pages 105–120, Barcelona, Spain, March 2004. Springer.

[CGZ04] C. Calcagno, P. Gardner, and U. Zarfaty. A context logic for tree update. In *LRPP '04: Proceedings of Workshop on Logics for Resources, Processes and Programs, 2004*, 2004.

[CGZ05] Cristiano Calcagno, Philippa Gardner, and Uri Zarfaty. Context Logic and tree update. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 271–282, New York, NY, USA, 2005. ACM Press.

[CGZ06a] Cristiano Calcagno, Philippa Gardner, and Uri Zarfaty. Local reasoning about data update. *Festschrift in honour of Gordon Plotkin, ENTCS 2007*, 2006.

[CGZ06b] Cristiano Calcagno, Philippa Gardner, and Uri Zarfaty. Separation Logic, Ambient Logic and Context Logic: parametric inexpressivity results. Unpublished, 2006.

[CGZ07] Cristiano Calcagno, Philippa Gardner, and Uri Zarfaty. Context logic as modal logic: completeness and parametric inexpressivity. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 123–134, New York, NY, USA, 2007. ACM Press.

[Cho56] Noam Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2:113–124, 1956.

[COY07] Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. Local action and abstract separation logic. In *LICS '07*, pages 366–378, Washington, DC, USA, 2007. IEEE Computer Society.

[CT01] Witold Charatonik and Jean-Marc Talbot. The decidability of model checking mobile ambients. In *CSL '01: Proceedings of the 15th International Workshop on Computer Science Logic*, pages 339–354, London, UK, 2001. Springer-Verlag.

[CYO01] Cristiano Calcagno, Hongseok Yang, and Peter W. O'Hearn. Computability and complexity results for a spatial assertion language for data structures. In Ramesh Hariharan, Madhavan Mukund, and V. Vinay, editors, *FSTTCS*, volume 2245 of *Lecture Notes in Computer Science*, pages 108–119. Springer, 2001.

[da 10] Pedro da Rocha Pinto. Reasoning about concurrent indexes. Master's thesis, Imperial College London, 2010.

[dE99] Willem-Paul de Roever and Kai Engelhardt. *Data Refinement: Model-Oriented Proof Methods and Their Comparison*. Cambridge University Press, Cambridge, UK, 1999.

[DFPV09] Mike Dodds, Xinyu Feng, Matthew Parkinson, and Viktor Vafeiadis. Deny-guarantee reasoning. In *ESOP*, 2009.

[DGG04] Anuj Dawar, Philippa Gardner, and Giorgio Ghelli. Adjunct elimination through games in static Ambient Logic. In K. Lodaya and M. Mahajan, editors, *FSTTCS 2004*, volume 3328 of *LNCS*, Heidelberg, 2004. Springer.

[DJP11] Mike Dodds, Suresh Jagannathan, and Matthew J Parkinson. Modular reasoning for deterministic parallelism. In *POPL*, 2011.

[DLM04] Silvano Dal Zilio, Denis Lugiez, and Charles Meyssonnier. A logic you can count on. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 135–146, New York, NY, USA, 2004. ACM Press.

[dRPDYGW11] Pedro da Rocha Pinto, Thomas Dinsdale-Young, Philippa Gardner, and Mark Wheelhouse. Abstract reasoining for concurrent indexes. In *Verification of Concurrent Data-Structures*, 2011.

[DY06] Thomas Dinsdale-Young. Adjunct elimination in Context Logic. Master's thesis, Imperial College London, 2006.

[DYDG+10] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In Theo D'Hondt, editor, *ECOOP*, volume 6183 of *Lecture Notes in Computer Science*, pages 504–528. Springer, 2010.

[DYGW10a] Thomas Dinsdale-Young, Philippa Gardner, and Mark Wheelhouse. Abstraction and refinement for local reasoning. In Gary Leavens, Peter OHearn, and Sriram Rajamani, editors, *Verified Software: Theories, Tools, Experiments*, volume 6217 of *Lecture Notes in Computer Science*, pages 199–215. Springer Berlin / Heidelberg, 2010.

[DYGW10b] Thomas Dinsdale-Young, Philippa Gardner, and Mark Wheelhouse. Abstraction and refinement for local reasoning. Technical report, Imperial College London, 2010. `http://www.doc.ic.ac.uk/~td202/papers/alrfull.pdf`.

[Elg61] Calvin C. Elgot. Decision problems of finite automata design and related arithmetics. *Transactions of the American Mathematical Society*, 98(1):21–51, 1961.

[ÉW03] Zoltán Ésik and Pascal Weil. On logically defined recognizable tree languages. In Paritosh K. Pandya and Jaikumar Radhakrishnan, editors, *FSTTCS 2003: Foundations of Software*

Technology and Theoretical Computer Science, volume 2914 of *Lecture Notes in Computer Science*, pages 195–207. Springer Berlin / Heidelberg, 2003.

[FPS07] J. Nathan Foster, Benjamin C. Pierce, and Alan Schmitt. A logic your typechecker can count on: Unordered tree types in practice. In *Workshop on Programming Language Technologies for XML (PLAN-X), informal proceedings*, January 2007.

[GBC+07] Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. Local reasoning for storable locks and threads. In *APLAS*, 2007.

[GS97] Ferenc Gécseg and Magnus Steinby. Tree languages. In *Handbook of formal languages, vol. 3: beyond words*, pages 1–68. Springer-Verlag New York, Inc., New York, NY, USA, 1997.

[GSWZ08] Philippa A. Gardner, Gareth D. Smith, Mark J. Wheelhouse, and Uri D. Zarfaty. Local hoare reasoning about dom. In *PODS '08*, pages 261–270, New York, NY, USA, 2008. ACM.

[GW09] Philippa Gardner and Mark J. Wheelhouse. Small specifications for tree update. In Cosimo Laneve and Jianwen Su, editors, *WS-FM*, volume 6194 of *Lecture Notes in Computer Science*, pages 178–195. Springer, 2009.

[Hag04] Matthew Hague. Static checkers for tree structures and heaps. Master's thesis, Imperial College London, 2004.

[HAN08] Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. Oracle semantics for concurrent separation logic. In *ESOP*, 2008.

[Heu91] Uschi Heuter. First-order properties of trees, star-free expressions, and aperiodicity. *Informatique théorique et applications*, 25(2):125–145, 1991.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[HS08] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Elsevier, 2008.

[HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *TOPLAS*, 12(3):463–492, 1990.

[IO01] Samin S. Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *POPL 2001*, New York, 2001. ACM Press.

[Jon81] C. B. Jones. *Development Methods for Computer Programs including a Notion of Interference*. PhD thesis, Oxford University, June 1981. Printed as: Programming Research Group, Technical Monograph 25.

[Kle56] S. Kleene. Representation of events in nerve nets and finite automata. In C. Shannon and J. McCarthy, editors, *Automata Studies*, pages 3–42. Princeton University Press, Princeton, N.J., 1956.

[Loz03] E. Lozes. Adjuncts elimination in the static Ambient Logic. In F. Corradini and U. Nestmann, editors, *EXPRESS 2003*, volume 96 of *ENTCS*, Amsterdam, 2003. Elsevier.

[Loz05] Étienne Lozes. Elimination of spatial connectives in static spatial logics. *Theoretical Computer Science*, 330(3):475–499, February 2005.

[MP71] Robert McNaughton and Seymour A. Papert. *Counter-Free Automata (M.I.T. research monograph no. 65)*. The MIT Press, 1971.

[Mur95] M. Murata. Forest-regular languages and tree-regular languages, 1995.

[O'H04] Peter W. O'Hearn. Resources, concurrency and local reasoning. In *Proc. 15th Int'l Conf. on Concurrency Theory (CONCUR'04)*, volume 3170 of *LNCS*, pages 49–67. Springer, September 2004.

[OYR04] Peter W. O'Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. In *Proc. 31th ACM Symp. on Principles of Prog. Lang.*, pages 268–280. ACM Press, January 2004.

[Par05] Matthew Parkinson. *Local Reasoning for Java*. PhD thesis, University of Cambridge, November 2005.

[PB05] M. J. Parkinson and G. M. Bierman. Separation logic and abstraction. In *POPL*, pages 247–258, 2005.

[PQ68] C. Pair and A. Quere. Définition et étude des bilangages réguliers. *Information and Control*, 13:565–593, 1968. (in French).

[PT93] Andreas Potthoff and Wolfgang Thomas. Regular tree languages without unary symbols are star-free. In *FCT '93: Proceedings of the 9th International Symposium on Fundamentals of Computation Theory*, pages 396–405, London, UK, 1993. Springer-Verlag.

[Rey02] John C. Reynolds. Separation Logic: A logic for shared mutable data structures. In *LICS '02: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.

[RG09] Mohammad Raza and Philippa Gardner. Footprints in local reasoning. *Logical Methods in Computer Science*, 5(2), 2009.

[RS59] M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2):114–125, 1959.

[Sch65] M. P. Schützenberger. On finite monoids having only trivial subgroups. *Information and Control*, 8:190–194, April 1965.

[Tak75] Masako Takahashi. Generalizations of regular sets and their application to a study of context-free langauges. *Information and Control*, 27:1–36, 1975.

[Tar55] Alfred Tarski. A Lattice-Theoretical Fixpoint Theorem and Its Applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[Tho84] Wolfgang Thomas. Logical aspects in the study of tree languages. In *Proc. of the conference on Ninth colloquium on trees in algebra and programming*, pages 31–49, New York, NY, USA, 1984. Cambridge University Press.

[Tho97] W. Thomas. Languages, automata, and logic. In *Handbook of formal languages, vol. 3: beyond words*, pages 389–455. Springer-Verlag New York, Inc., New York, NY, USA, 1997.

[Tra50] B. A. Trakhtenbrot. Impossibility of an algorithm for the decision problem on finite classes. *Doklady Akademii Nauk SSSR*, 70:509–572, 1950. (in Russian).

[Vaf07] Viktor Vafeiadis. *Modular Fine-Grained Concurrency Verification*. PhD thesis, University of Cambridge, July 2007.

[VP07] Viktor Vafeiadis and Matthew Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*, pages 256–271, 2007.

[WDP10] John Wickerson, Mike Dodds, and Matthew Parkinson. Explicit Stabilisation for Modular Rely-Guarantee Reasoning. In Andrew D. Gordon, editor, *19th European Symposium on Programming (ESOP 2010), Paphos, Cyprus*, volume 6012 of *Lecture Notes in Computer Science*, pages 611–630. Springer-Verlag, mar 2010.

[Yu97] Sheng Yu. Regular languages. In *Handbook of formal languages, vol. 1: word, language, grammar*, pages 41–110. Springer-Verlag New York, Inc., New York, NY, USA, 1997.

[Zar07] Uri Zarfaty. *Context Logic and Tree Update*. PhD thesis, Imperial College London, 2007.