

# ADVANCES IN SEPARATION LOGIC

A STUDY OF LOGICS FOR  
REASONING ABOUT STATEFUL PROGRAMS

NOAH TORP-SMITH

DISSERTATION  
PRESENTED TO THE FACULTY  
OF THE IT UNIVERSITY OF COPENHAGEN  
IN PARTIAL FULFILMENT OF  
THE REQUIREMENTS FOR THE DEGREE OF  
DOCTOR OF PHILOSOPHY

## Abstract

Reasoning about programs that involve shared imperative data structures has proven to be notoriously difficult over the years, mainly due to problems related to aliasing. A recent approach to these problem was presented by O’Hearn and Reynolds; they devised a program logic in the style of Hoare, called *separation logic*, and that is the main interest of this thesis. Here is a brief overview of the main contributions.

Since separation logic is a relatively new logic, we should be able to “benchmark” it on realistic examples. We show that separation logic is indeed useful by specifying and proving correctness of Cheney’s copying garbage collector, a program which uses sophisticated pointer manipulations. The main contribution of the work is an illustration of how additions of sets and relations to the languages of expressions and assertions can be used in combination with separation logic to specify and prove an isomorphism between the states before and after execution of the algorithm.

The assertion language of traditional separation logic is a variant of first order propositional BI. We make a straightforward extension of this logic to *higher-order predicate BI*, and demonstrate how it can be used in separation logic. In particular, we show that the extension can be applied to model data abstraction and polymorphism for a programming language with heap manipulating constructs and calls to simple procedures. Moreover, we present a model for this new logic (called *higher order separation logic*), using a general class of models called *BI hyperdoctrines*.

The programming language of standard separation logic is a simple extension of Hoare’s **while**-language with basic heap manipulating constructs. The success of separation logic relies heavily on its ability to support *local reasoning* via the so-called *frame rule*. We extend the scope of separation logic by devising a *separation-logic type system* for a version of idealized algol with heap manipulating constructs, allowing for local reasoning about higher-order programs which manipulate heaps. Moreover, we present a categorical model in which one can soundly and coherently model the programming language in a way that is adequate with respect to the standard semantics of idealized algol. We also take some initial steps towards a *parametric* model for separation-logic typing.

In Hoare’s work on *refinement* for imperative programs one starts with an abstract specification of a data type and derives a concrete representation. This method assumes a static separation of variables between representations of the data type; however, the introduction of pointers into refinement breaks these assumptions. We introduce a model that brings ideas from separation logic into refinement, and prove an abstraction theorem for so-called *separation contexts* which, intuitively, are client programs that are well-behaved with respect to resources.



## Acknowledgments

First and foremost, I thank my supervisor Lars Birkedal for introducing me to the craft of research and for participating actively in maturing both me and my research. He has spent many hours on me, and I feel obliged by the amount of time I have been given of such a busy schedule as his.

I thank Peter O’Hearn for hosting my visit at Queen Mary, University of London in the Spring of 2003. He helped me with practical matters and my research, and we had a lot of fun both during and after my visit. Also thanks to the East London Massive: Josh Berdine, Cristiano Calcagno, Richard Bornat, Philippa Gardner, and in particular Ivana Mijajlović, for good friendship and helpful discussions.

Hongseok Yang and John Reynolds have given most valuable input which has been crucial over the years. I owe my gratitude to them for the inspiration and good times, we have had.

Moreover, I have had helpful discussions with other people, including Uday Reddy, Greg Morrisett, David Naumann, Nick Benton, Thomas Hildebrandt, Carsten Butz, Rasmus Møgelberg, Martin Elsman, Peter Sestoft, Andrzej Wasowski, Fritz Henglein, Sebastian Skalberg, Tobias Nipkow, Henning Niss, Andrzej Filinski, Matthew Parkinson, and Limin Jia.

The Department of Theoretical Computer Science has provided a great working environment, and I have had many good discussions with my colleagues here. In particular, thanks to Søren Debois and Inge Li Gørtz for proof-reading parts of the thesis and to Bodil Biering for valuable collaboration and for being an outstanding office-mate! ☺

The anonymous referees on the parts of this thesis that have been submitted for publication have given detailed comments and valuable input.

On the personal side, there are many people whose friendships have been invaluable through the years. To avoid forgetting any (you know who you are!), I refrain from mentioning those here.

The work presented herein has been partially supported by the Danish Technical Research Council Grant 56-00-0309.



# Contents

<b>I Preliminaries</b>	<b>1</b>
<b>1 Introduction</b>	<b>3</b>
1.1 Introduction to Separation Logic . . . . .	3
1.2 Contributions of this Dissertation . . . . .	7
1.3 Related Work . . . . .	20
1.4 Included Papers . . . . .	22
1.5 Conclusion . . . . .	23
1.6 Future Work . . . . .	24
<b>II Publications and Manuscripts</b>	<b>27</b>
<b>2 Local Reasoning about a Copying Garbage Collector</b>	<b>29</b>
2.1 Introduction . . . . .	29
2.2 An Introduction to Separation Logic . . . . .	32
2.3 Syntax and Semantics . . . . .	33
2.4 Programming Language . . . . .	44
2.5 The Garbage Collection Algorithm . . . . .	46
2.6 Proofs . . . . .	54
2.7 Conceptual Remarks . . . . .	66
2.8 Related Work . . . . .	67
2.9 Conclusion and Future Work . . . . .	69
2.10 Appendix 2 . . . . .	71
<b>3 BI Hyperdoctrines, Higher-order Separation Logic, and Abstraction</b>	<b>83</b>
3.1 Introduction . . . . .	83
3.2 Syntax and Semantics . . . . .	85
3.3 Program Logic . . . . .	92
3.4 Examples . . . . .	97
3.5 General Semantics for Higher-order BI . . . . .	108
3.6 Other Applications of Higher-order BI . . . . .	115
3.7 Related Work . . . . .	118
3.8 Conclusion . . . . .	119
<b>4 Semantics of Separation-logic Typing and Higher-Order Frame Rules</b>	<b>125</b>
4.1 Introduction . . . . .	125
4.2 Storage Model and Assertion Language . . . . .	127
4.3 Programming Language . . . . .	128

4.4	Semantics . . . . .	131
4.5	Related Work . . . . .	139
4.6	Conclusion and Future Directions . . . . .	140
<b>5</b>	<b>Refinement and Separation Contexts</b>	<b>143</b>
5.1	Introduction . . . . .	143
5.2	Basic Ideas . . . . .	144
5.3	Preliminary Definitions . . . . .	145
5.4	Unary Separation Contexts . . . . .	148
5.5	Refinement and Separation . . . . .	149
<b>6</b>	<b>Towards a Parametric Model for Separation Logic Typing</b>	<b>155</b>
6.1	Introduction . . . . .	155
6.2	Synopsis of the Categorical Model . . . . .	156
6.3	Developing a Parametric Model . . . . .	157
6.4	Storage Model and Assertions . . . . .	171
6.5	Programming Languages . . . . .	172
6.6	Conclusion and Problems . . . . .	176

**Part I**

**Preliminaries**



# Chapter 1

## Introduction

This dissertation comprises five papers related to separation logic, and this introduction, which defines various basic concepts and briefly presents the rest of the dissertation. The introduction is organized as follows. First, we give an introduction in Section 1.1 to separation logic which is the foundation for the work presented herein. In Section 1.2 we motivate and briefly describe the contributions of this dissertation. We discuss related work in Section 1.3 and briefly introduce the papers included in the dissertation in Section 1.4. Finally, we conclude and give pointers to future work in Sections 1.5 and 1.6, respectively.

Although we give a short review, we assume that the reader is familiar with basic Hoare logic [50], for example as described in standard textbooks [108, 112, 71, 132]. Likewise, basic category theory, for example as presented in the technical report [128], is a prerequisite for parts of this dissertation.

### 1.1 Introduction to Separation Logic

In this Section, we briefly introduce separation logic; it is a slightly extended version of the description of separation logic from Chapter 2 in this dissertation. For a survey on separation logic, see Reynolds' introductory paper [115]. Separation logic extends Hoare logic, so we begin with a review of that; this will also fix some notation needed later.

Assume a countable set  $\{x, y, \dots\}$  of variables, and define the *expression language* of Hoare logic by the grammar

$$E ::= x \mid n \mid E + E \mid E - E \mid E \times E,$$

where  $n$  can be any integer. The *assertion language* of Hoare logic is defined by

$$P ::= \top \mid E = E \mid E \leq E \mid P \wedge P \mid \neg P \mid P \rightarrow P \mid \forall x. P.$$

The set  $\text{FV}(E)$  of free variables of an expression  $E$  is simply the set of variables occurring in  $E$ , and for an assertion,  $\text{FV}(P)$  is the set of variables in  $P$  that are not bound by a quantifier. A *stack* (sometimes called a *store* in the literature) is a finite assignment of integers to variables. The semantics  $\llbracket E \rrbracket$  of an expression  $E$  is a map from stacks  $s$  with  $\text{FV}(E) \subseteq \text{dom}(s)$  to values, which is defined in the obvious way. For example, given a stack  $s$ , the semantics  $\llbracket x + 4 \rrbracket s$  of the expression  $x + 4$  is the integer  $s(x) + 4$ . Similarly, the semantics  $\llbracket P \rrbracket$  of an assertion  $P$  is a map from stacks  $s$  with  $\text{FV}(P) \subseteq \text{dom}(s)$  to the set  $\{\mathbf{true}, \mathbf{false}\}$ , defined in the obvious way.

The *simple while-language* of Hoare logic is defined by the grammar

$$C ::= \mathbf{skip} \mid x := E \mid C; C \mid \mathbf{if} B \mathbf{then} C \mathbf{else} C \mathbf{fi} \mid \mathbf{while} B \mathbf{do} C \mathbf{od}, \quad (1.1)$$

where  $B$  ranges over the boolean expressions

$$B ::= E = E \mid E \leq E \mid \neg B.$$

The set  $\text{Mod}(C)$  for a command  $C$  consists of those variables that are assigned in  $C$ . The operational semantics of the programming language is a partial judgment  $(C, s) \rightsquigarrow s'$ , possibly taking a *configuration*  $(C, s)$  to a stack  $s'$ , defined by structural induction on  $C$  in a standard way. For instance, if  $\llbracket E \rrbracket s = 15$  and  $(C, s) \rightsquigarrow s'$ , the operational semantics yields

$$(\mathbf{if} E = 15 \mathbf{then} C \mathbf{else} C' \mathbf{fi}, s) \rightsquigarrow s'.$$

A *partial correctness specification*, or a Hoare triple, is of the form

$$\{P\} C \{Q\},$$

and is defined to be *valid*, if for all  $s$  with  $\text{Mod}(C) \cup \text{FV}(P, Q) \subseteq \text{dom}(s)$ ,  $\llbracket P \rrbracket s$  implies  $((C, s) \rightsquigarrow s' \Rightarrow \llbracket Q \rrbracket s')$ , or less formally: If  $P$  holds in a stack  $s$  and if  $C$  terminates starting from  $s$ , then  $Q$  holds in the resulting stack. Hoare logic includes a set of inference rules for deriving valid Hoare triples, and is used to statically prove properties of simple imperative programs.

Hence, in traditional Hoare logic, a *state* simply consists of a stack. Correspondingly, the assertion language only describes properties of expressions which depend on variables, the simple **while** programming language only modifies variables, and program control depends on the values stored in variables only. In this dissertation, we consider a more realistic notion of a state: states in the sense of Hoare logic are extended with *heaps*, which are finite maps from a predefined set of *locations* (sometimes called *pointers*) to values. In our description, locations are simply integers. Hence a state in this introduction is a pair  $(s, h)$  where  $s$  is a finite map from variables to integers and  $h$  is a finite map from integers to integers. Stacks and heaps are abstractions of components in real computers: since compilation of programs usually result in a map from variables in the program to registers, stacks model the register bank in a CPU, whereas heaps model the computer's memory, which is usually an integer-indexed family of storage cells that hold integer values. Given this storage model, we extend the assertion and programming languages to include descriptions of heaps and operations that manipulate heaps. The language of expressions is the same as in traditional Hoare logic.

We consider a programming language which extends the simple **while-language** from standard Hoare logic with basic operations on heaps: allocation of new cells, mutation, lookup, and disposal of heap cells. Formally, the language (1.1) is extended by

$$C ::= \dots \mid x := \mathbf{cons}(E) \mid [E] := E' \mid x := [E] \mid \mathbf{dispose}(E). \quad (1.2)$$

The effect of the non-deterministic command  $x := \mathbf{cons}(E)$  is that a new cell is allocated on the heap and initialized with the value  $E$ , and the address of the new cell is stored in the variable  $x$ . The operational semantics  $\rightsquigarrow$  now takes configurations  $(C, s, h)$  to states  $(s', h')$ , or to *wrong*, which is the state resulting from an attempt to look up, mutate, or dispose a heap cell which is not in the domain of the current heap. A configuration  $(C, s, h)$  is called *safe* if  $(C, s, h) \not\rightsquigarrow \text{wrong}$ .

A naïve approach to incorporating this new storage model and programming language into Hoare logic here is not without complications, as we illustrate here. In traditional Hoare logic, we have the *rule of constancy*:

$$\frac{\{P\} C \{Q\}}{\{P \wedge P'\} C \{Q \wedge P'\}} \text{FV}(P') \cap \text{Mod}(C) = \emptyset, \quad (1.3)$$

where  $P, P', Q$  are assertions,  $\text{FV}(P)$  are the free variables of  $P$ , and  $\text{Mod}(C)$  is the set of variables modified by the program  $C$ . This rule states that if an assertion  $P'$  is “syntactically disjoint” from a command (as specified in the side condition), then from a specification  $\{P\} C \{Q\}$ , we can infer  $\{P \wedge P'\} C \{Q \wedge P'\}$ . This rule is useful, since it allows a simple kind of *local reasoning*: when inferring a “large” specification involving many conjuncts and variables for a program fragment, e. g., a branch of an **if**-statement, we can proceed by showing a specification involving only the conjuncts which contain variables modified by the fragment. The remaining conjuncts can then be added as an invariant of the specification as described in (1.3).

When heaps are included in the storage model, however, things become more complicated. Without giving a formal definition here, we introduce the basic assertion  $E_1 \hookrightarrow E_2$  to express that the domain of the heap contains the value denoted by  $E_1$ , and the contents of the corresponding heap cell is the value denoted by  $E_2$  — this is sometimes pronounced “ $E_1$  points to  $E_2$ ”. Then the specification

$$\{x \hookrightarrow 3\} [x] := 4 \{x \hookrightarrow 4\}$$

is certainly valid. Using the rule of constancy, we get

$$\frac{\{x \hookrightarrow 3\} [x] := 4 \{x \hookrightarrow 4\}}{\{x \hookrightarrow 3 \wedge y \hookrightarrow 3\} [x] := 4 \{x \hookrightarrow 4 \wedge y \hookrightarrow 3\}},$$

since the variable  $y$  is not modified by the command  $[x] := 4$ . However, consider a state where  $x$  and  $y$  are both mapped to the same location which is in the domain of the heap, where the contents is the integer 3. The assertion  $x \hookrightarrow 3 \wedge y \hookrightarrow 3$  holds in this state, but the state after execution of  $[x] := 4$  does *not* validate  $x \hookrightarrow 4 \wedge y \hookrightarrow 3$ . Hence, the rule of constancy is not sound in this setting, and the simple kind of local reasoning outlined before is not applicable. This problem is known as *aliasing*: the two different variables  $x$  and  $y$  give access to the same unit of storage (a heap cell), and this breaks the disjointness properties that the side condition in (1.3) guaranteed in the simpler setting with no heaps. An attempt to get around this problem is via so-called *non-interference* predicates of which we give a simple example here. One has

$$\frac{\{x \hookrightarrow 3\} [x] := 4 \{x \hookrightarrow 4\}}{\{x \hookrightarrow 3 \wedge y \hookrightarrow 3 \wedge x \neq y\} [x] := 4 \{x \hookrightarrow 4 \wedge y \hookrightarrow 3 \wedge x \neq y\}},$$

since  $x \neq y$  ensures that  $x$  and  $y$  are not aliases; this means that the assignment to the heap cell with address  $x$  does not affect the truth of the conjunct  $y \hookrightarrow 3$ . This way of getting around the problem, however, scales poorly in general. When verifying a specification of a program involving many variables, the number of required inequalities grows rapidly, making verification intractable. There have been attempts of constructing more sophisticated non-interference predicates than inequalities to enable local reasoning [32, 26] in the presence of pointers.

*Separation logic* [113, 59, 115] was introduced by O’Hearn and Reynolds in the late 1990es. It addresses the problem of aliasing by incorporating noninterference in the heap directly in the assertion language. The assertion language from Hoare logic is extended with

$$P ::= \dots \mid \text{emp} \mid E_1 \mapsto E_2 \mid P * P.$$

We explain these assertion forms informally. Formal definitions can be found in the papers included in this dissertation. The assertion  $\text{emp}$  states that the domain of the current heap is empty, whereas  $E_1 \mapsto E_2$  asserts that the domain of the current heap has precisely one element, namely the value denoted by  $E_1$ , and the contents of the corresponding heap cell is the value denoted by  $E_2$ . The assertion  $P * P'$  is the *separating conjunction* of  $P$  and  $P'$ , and it holds if the domain of the current heap can be split in two disjoint parts such that the the corresponding subheaps validate  $P$  and  $P'$ , respectively. Notice the *tight* interpretation of the assertion  $E_1 \mapsto E_2$ . It not only says that  $E_1$  points to  $E_2$ , but also that  $E_1$  is the only cell in the heap. Descriptions of larger heaps can be constructed using the separating conjunction, for example, the assertion  $(x \mapsto 4) * (x + 8 \mapsto 42)$  describes a heap with two cells. The assertion form  $E_1 \hookrightarrow E_2$  from before is simply a shorthand for  $(E_1 \mapsto E_2) * \top$ . Sometimes, we overload the points-to notation and write

$$E \mapsto E_0, \dots, E_n$$

as a shorthand for

$$(E \mapsto E_0) * (E + 1 \mapsto E_1) * \dots * (E + n \mapsto E_n).$$

The way in which these new assertion forms enable local reasoning is captured by the *frame rule*:

$$\frac{\{P\} C \{Q\}}{\{P * P'\} C \{Q * P'\}} \text{FV}(P') \cap \text{Mod}(C) = \emptyset. \quad (1.4)$$

Returning to the example from before, the following is an instance of this rule:

$$\frac{\{x \hookrightarrow 3\} [x] := 4 \{x \hookrightarrow 4\}}{\{x \hookrightarrow 3 * y \hookrightarrow 3\} [x] := 4 \{x \hookrightarrow 4 * y \hookrightarrow 3\}}.$$

Here, the conclusion is valid because the separating conjunction implicitly states that  $x$  and  $y$  denote distinct locations — if  $x$  and  $y$  denoted the same location, it would not be possible to split the heap into disjoint subheaps such that  $x \hookrightarrow 3$  holds in one and  $y \hookrightarrow 3$  in the other.

Soundness of the frame rule is shown in detail in Yang’s dissertation [134]. The proof relies on two semantic locality properties that are enjoyed by all programs generated by the grammar (1.2). They state that programs that can execute in a heap can behave essentially the same way when run in a larger heap. The properties are listed below, and uses the following operations on heaps: the notation  $h \# h'$  indicates that the domains of  $h$  and  $h'$  are disjoint, and only when this is the case,  $h \cdot h'$  is defined to be  $h \cup h'$ .

**Safety Monotonicity** If  $C, s, h$  is safe, then for all  $h'$  with  $h \# h'$ ,  $C, s, h \cdot h'$  is safe.

**The Frame Property** For all  $h, h_0, h'_1$ , if  $h \# h_0$ ,  $C, s, h$  is safe, and  $C, s, (h \cdot h_0) \rightsquigarrow s', h'_1$ , then there is an  $h'$  with  $h'_1 = h_0 \cdot h'$  and  $C, s, h \rightsquigarrow s', h'$ .

The fact that these properties hold for all commands can be verified by structural induction, and they make it straightforward to prove the frame rule.

The frame rule entails that *local reasoning at the level of heaps* is sound; we shall see later that this is indeed very useful when verifying programs involving many complex pointer manipulations. Note that if the assertion  $P'$  is restricted to be *pure* (i.e., independent of the heap), the rule of constancy is valid, even in the setting that involves heaps. Hence, both local reasoning in the traditional sense for pure predicates and local reasoning for general predicates in the sense captured by the frame rule, are sound. It is often possible to decide purity syntactically. For instance, an assertion is pure if it only contains equations and conjunctions.

**Notation and Terminology:** To avoid confusion, some remarks regarding notation and terminology are appropriate. We mostly use capital letters for syntactical entities ( $P, P', Q, \dots$  for assertions,  $C, C', \dots$  for programs), and lowercase letters for their denotations ( $p, q, \dots, c, c', \dots$ ). Since this dissertation is a collection of papers published at different fora and with different co-authors, however, the notation in the papers is not completely consistent with this principle.

The assertion language in separation logic is an instance of propositional BI, the logic of Bunched Implications [102]. There has sometimes been confusion about the meaning of the phrase “separation logic”. In this dissertation, “separation logic” is used as a name for all of the program logic, not just the assertion language which is part of it. The latter is referred to as “the assertion language of separation logic” or “propositional BI”.

## 1.2 Contributions of this Dissertation

In this section we motivate and describe the contributions of this dissertation. In particular, we discuss certain shortcomings and questions that relate to the description of separation logic from Section 1.1, and make clear how the work presented in the included papers of this dissertation answers them. Moreover, we describe how the dissertation extends the scope of other previous works by combining them with ideas from separation logic. Details can be found in the papers included in the dissertation. It is worth emphasizing that this is not an attempt to give an account of all challenges regarding separation logic; there are clearly more problems that are not described in this section. Some of these are mentioned briefly in Section 1.6. In this thesis, we do not touch upon issues like logical completeness or complexity. These matters are analyzed in the papers [34, 15, 16] by Calcagno and others.

### 1.2.1 Proof of a Garbage Collector

As mentioned, separation logic was introduced in the late 1990'ies by Reynolds and O'Hearn. Being such a fairly novel logic, it is desirable to demonstrate that it is indeed useful, and the most convincing way to do this is via empirical tests. Separation logic is presented as a logic for reasoning about programs that explicitly manipulate pointers, so proving correctness of a program that involves many nontrivial pointer manipulations will provide a good benchmark for separation logic. Garbage collectors are good examples of such programs, since their inherent task is to perform low-level operations that manipulate possibly complex shared mutable data structures, access to which can be shared by many entry-points.

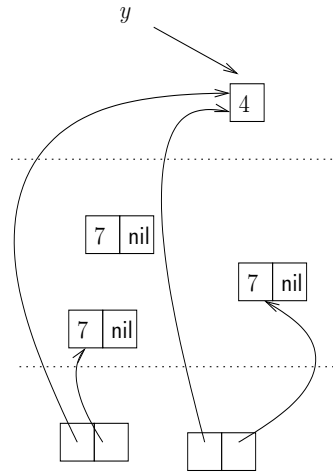


Figure 1.1: A sample partition of a heap

In his dissertation [134], Yang proves the correctness of Schorr-Waite’s marking algorithm, which is an optimized version of a depth-first traversal of a graph that can be used as part of a memory management system. We prove the correctness of Cheney’s stop-and-copy garbage collection algorithm [35] which moves the reachable data using a breadth-first algorithm in such a way that the structure is preserved and the heap is defragmented after execution, thus providing an example of a correctness proof of a non-trivial algorithm in separation logic.

The proof introduces two novel techniques. First, the addition of the notion of finite sets of pointers and relations of expressions and assertions, and secondly, and more importantly, the *iterated separating conjunction*. We introduce the assertion form

$$\forall_* x \in m. A,$$

which asserts that the domain of the current heap can be split disjointly into as many parts as there are pointers in the set denoted  $m$ , and for each of the corresponding sub-heaps, the assertion  $A$  holds.

The following simple example clarifies how  $\forall_*$  can be used to describe heaps. Consider the heap depicted in Figure 1.1. The heap cells here can be split into three disjoint sets in each of which all the cells have similar properties. This partition is illustrated with dotted lines in the figure. The first set simply consists of the cell pointed to by  $y$ , whereas the second set (called  $m_1$ ) consists of three cells which clearly have a similar property. The cells in the last group  $m_2$  have the property that their first fields contain the pointer  $y$ , and the second fields all contain a pointer to a cell in  $m_1$ . Hence, a description of this heap is given by the assertion

$$\begin{aligned} \exists m_1, m_2. ((y \mapsto 4) * \\ (\forall_* x \in m_1. x \mapsto 7, \text{nil}) * \\ (\forall_* x \in m_2. (\exists z. z \in m_1 \wedge x \mapsto y, z))). \end{aligned}$$

At any state of execution of the garbage collection algorithm, the pointers involved in the algorithm can be split into disjoint sets as in the example above, and thus the iterated separating conjunction can be used to obtain an assertion which describes the heap, in a way similar to the example above.

The extensions of standard separation logic just outlined are used to give a specification of the garbage collector, and prove that our implementation meets this specification. Finally, we prove that the specification implies correctness of the garbage collector, *i.e.*, that the states before and after execution of the algorithm are isomorphic. The proof is formal, in the sense that all steps of the proof are justified by rules that are shown to be sound. Although this has not been done in this thesis, we believe that a proof of this nature could be used to compare separation logic to other formalisms for correctness proofs of pointer-manipulating programs. This is discussed further in Section 1.6.

## 1.2.2 Higher-order Separation Logic

The diversity of the languages of assertions used in different expositions of separation logic in the literature is an issue which deserves attention. Several authors apply separation logic to prove properties of programs involving pointers, and often, the language of assertions is tailored to the specific program at hand. For example, Yang extends the language of expressions with trees and the assertion language with assertions about these in his proof of the Schorr-Waite graph algorithm [134], and the assertion language used for the proof of Cheney’s copying garbage collector in the paper [23] is an extension of the language of assertions described in Section 1.1 with finite sets and relations. Other extensions use directed acyclic graphs and sequences of integers [27, 113]. In the respective settings, these extensions make the logics and the proofs easier to understand. However, they raise the fundamental question whether there is a general logic of which all these dialects are an instance, or whether one has to come up with a new “dialect” of separation logic for each program one wishes to verify. Furthermore, if such a logic exists, what new insights or reasoning principles can be obtained from it, if any?

We propose an extension of the assertion language that is higher-order, has a basic type of propositions (so that, for example, it is possible to quantify over all propositions), and contains all of intuitionistic higher-order logic as a sub-logic. It turns out that it is relatively straightforward to do so, but it raises other immediate questions. For example, how do we interpret the new logic, *i.e.*, what are *models* of this higher-order logic? Further, what are examples of useful applications of the higher-order logic? In what follows, we briefly describe the answers that the work presented in this dissertation provides for these questions.

### Higher-Order BI

The assertion language used in separation logic is a variant of propositional first-order BI [83, 59]. In his monograph [102], Pym proposes a notion of higher-order predicate BI, which uses bunches at the level of contexts. We propose another notion of higher-order predicate BI, with no bunch-structure on contexts; this makes it simpler and easier to understand. Our notion is a simple extension of higher-order intuitionistic logic with the commutative and associative connective  $*$ , its unit  $I$ , and its adjoint connective  $\multimap$ . We believe that our notion of higher-order predicate BI and the class of models provided is the right one for separation logic. Pym aimed to model multiplicative quantifiers, whereas the assertion language of separation logic only uses additive quantifiers, just as our notion of higher-order predicate BI. Moreover, we show that the standard semantics for separation logic is an instance of the semantics provided by our class of models.

To model higher-order BI, it is tempting to extend standard models of higher-order classical logic, and combine this with ideas known from models of first-order BI [104].

An example of the latter class of models exploits the doubly closed structure on a certain topos of presheaves [104]. It is standard to interpret higher-order intuitionistic and classical logic in topoi [65], so the obvious attempt is to demand some extra structure on a topos, and model higher-order BI there. We shall see shortly that such an attempt will fail.

We propose a straightforward extension of Lawvere’s notion of *hyperdoctrine* [66], which, informally, is the minimal solution to a model of higher-order intuitionistic logic, in the sense that it satisfies exactly the requirements needed to model that logic. The extension results in a structure called a *BI-hyperdoctrine*, and this is a general model for our version of higher-order predicate BI (and a minimal solution in the sense mentioned above). It is quite easy to see that our notion of BI hyperdoctrines yields a sound and complete class of models for higher-order predicate BI. Moreover, the previously hinted attempt to model higher-order BI by requiring extra structure on topoi is doomed to fail because if a topos can be shown to be a BI-hyperdoctrine, then the BI structure and the usual Cartesian product collapse on each subobject lattice, making the model trivial. Put briefly, the reason that BI hyperdoctrines work (in contrast to topoi) is that predicates are modeled by something more general than subobject lattices.

We demonstrate that our notion of a BI-hyperdoctrine is general and useful, in the sense that earlier models of BI can be seen as instances of BI-hyperdoctrines. In particular, the standard storage model of BI used in separation logic is an instance of a BI hyperdoctrine via a straightforward correspondence. More details can be found in the technical report [21], included in this dissertation, and in the master’s thesis by Biering [19].

## Applications of Higher-Order BI

In his introductory paper [115], Reynolds identifies several special classes of assertions, and gives special axiom schemata that apply to these classes of assertions. These properties are used in several settings; for example, the notion of *precise* assertions plays a critical role in proving soundness of the hypothetical frame rule in the paper [88], and the notion of *pure* predicates is important in our proof of a copying garbage collector [23]. The definition of these classes of assertions are all semantic — for example, an assertion is pure if its truth is independent of the heap. We show how some of these classes can be characterized *in the logic*, using the quantification over all propositions that is provided by our higher-order predicate logic.

Moreover, we show how quantification over propositions can be used in program proofs. Specifically, we demonstrate how one can use existential quantification over propositions to give sound reasoning principles for data abstraction, and how universal quantification can be used to reason about parametricity. The first of these points is explained briefly here; more details can be found in the paper [21], which is included in this dissertation. Referring to the slogan that “abstract type have existential type” [73], we give an *abstract function definition rule*, and prove it sound. A simplified version of the rule is

$$\frac{\begin{array}{c} \vdash \hat{P}:\tau \\ \vdash \{P_1[\hat{P}/x]\} C_1 \{Q_1[\hat{P}/x]\} \\ \exists x:\tau. (\{P_1\}k_1\{Q_1\}) \vdash \{P\}C\{Q\} \end{array}}{\vdash \{P\} \mathbf{let} k_1(\bar{x}_1) = C_1 \mathbf{in} C \mathbf{end} \{Q\}} \quad x \notin \text{FV}(P, Q) \cup \text{Mod}(C). \quad (1.5)$$

This rule can be used to show a specification for a client program  $C$  which uses an im-

plementation  $C_1$  of a module operation, called via the function name  $k_1$ . The intention is that the implementation  $C_1$  may use an internal representation  $\hat{P}$ , which the client program is unaware of and unable to exploit. When proving the specification for the client program, only the abstract specification of the module is used, and the rule (1.5) makes it possible to go from this “abstract proof” to a proof of the program that uses the concrete implementation.<sup>1</sup> As an example, two different implementations of an abstract priority queue, using a sorted singly-linked list and an unsorted doubly-linked list, respectively, are given. We give examples of client programs that use these implementations and illustrate that the proofs of these client programs are independent of which implementation is used.

### 1.2.3 Separation Logic for Higher-order Programming Languages

Despite the success of separation logic, it is an evident limitation that the programming language is somewhat simple-minded. In separation logic as it is described in the introductory paper [115] and in Section 1.1, the programming language is the simple **while** language of Hoare logic, extended with with basic heap operations. This has been a good starting point for separation logic because the difficulties of aliasing that separation logic is designed to deal with arise already in this setting. However, whether the ideas from separation logic apply to more complex programming languages is a commonsensical question. In particular, it is interesting to ask whether it is possible to use separation logic for a higher-order programming language? As mentioned in Section 1.1, the most significant reason for the success of separation logic is the frame rule, which facilitates local reasoning. When separation logic is extended to higher-order programming languages, it is therefore important that a rule resembling the frame rule is valid for higher-order programs.

The first step in this direction is taken in the paper [88], where O’Hearn *et al.* show soundness of a frame rule for programs that call simple functions. This rule is called the *hypothetical frame rule*, and a simplified version of it is

$$\frac{\begin{array}{c} \vdash \{P_1 * P_0\} C_1 \{Q_1 * P_0\} \\ \{P_1\} k_1 \{Q_1\} \vdash \{P\} C \{Q\} \end{array}}{\vdash \{P * P_0\} \mathbf{let } k_1 = C_1 \mathbf{ in } C \mathbf{ end } \{Q * P_0\}} \quad (1.6)$$

subject to certain variable conditions, or *modifies clauses*, which we shall not elaborate on here. The intention is that  $C$  is a client program which can call the procedure named  $k_1$ , implemented by the program  $C_1$ , which uses some private state  $P_0$ . The rule states that when using  $k_1$ , then from a specification which uses the interface specification  $\{P_1\} k_1 \{Q_1\}$  of the procedure, a specification for the program where the client uses the actual implementation (and hence implicitly the hidden resource), can be inferred. Note that the resource invariant is ignored in the interface specification, whereas it may be used in the proof of the specification of the implementation  $C_1$ . Hence, this rule allows local reasoning for programs which calls simple procedures, while at the same time disallowing them to be aware of the internal resources used by implementations of the module used by the client program.

Already at this stage, some issues deserve attention. First, the authors of the paper [88] noted that, to get a system in which both the rule of conjunction and the hypo-

<sup>1</sup>This is what is meant by “data abstraction” in this dissertation. Since we do not prove any explicit contextual equivalence results, one might argue that this should really be called “compositionality”.

thetical frame rule are sound, the rule must demand that some of the assertions in the rule be precise [115] (see the specific restrictions in the paper [88]). Secondly, the amount of bookkeeping needed just to state the modifies clauses, needed for soundness of the rule, is considerable.

It is a reasonable goal to take this approach further; given that the frame rule for second-order programs as in the hypothetical frame rule is sound, it seems feasible that frame rules for higher-order programming languages should be valid. The question remains: how are such rules formulated, what are necessary conditions, and how are the rules proved sound?

We propose a separation-logic type system for a version of idealized algol [109] which includes higher order procedures and heap manipulations. We briefly outline the ideas of the work here. First, the problems of modifies clauses mentioned above are avoided by considering a language with *immutable* variables only. Since the focus of separation logic is mainly about data stored in the heap, this is a reasonable first step which preserves the interesting difficulties related to aliasing. Besides this, there are programming languages used in practice, e. g. ML, which also have this restriction.

More specifically, the programming language is a typed call-by-name  $\lambda$ -calculus with fixed points and constructs for branching, sequencing and heap operations. The language is adapted for *separation logic typing*; the basic intuition being that a simple command  $C$  which satisfies the triple  $\{P\}C\{Q\}$  in standard separation logic has the type  $\{P\} - \{Q\}$  in our system, so that a type for a term can be read as a specification under a straightforward Curry-Howard correspondence. Hence, the base types of the calculus are *triple types*  $\{P\} - \{Q\}$ , where  $P$  and  $Q$  are assertions in the usual sense. The type system also includes invariant extension, dependent types, and function types. Intuitively, if  $\theta$  is a type, the invariant extension  $\theta \otimes P$  is the result of using the frame rule to extend the type  $\theta$  with the invariant  $P$ ; under the correspondence just mentioned, this amounts to applying a frame rule to the specification corresponding to  $\theta$ . Under the specification-reading of types, the dependent type  $\Pi_i \theta$  corresponds to the specification given by universally quantifying  $i$  in the specification corresponding to  $\theta$ .

The programming language has pre-terms given by

$$\begin{aligned} M ::= & x \mid \lambda x:\theta. M \mid MM \mid \lambda i. M \mid ME \mid \text{fix } M \mid \text{ifz } E M M \\ & \mid M; M \mid \text{let } i = \text{new in } M \mid \text{let } i = [E] \text{ in } M \mid [E] := E \mid \text{free}(E). \end{aligned} \quad (1.7)$$

Note that all commands have “command type”, so in contrast to idealized algol, integer expressions are “second class”; the terms in the grammar (1.7) do not have integer type. Notice further that variables are immutable as mentioned before, so only heap cells can be modified after initialization. There are two forms of abstraction and applications; one for general terms, and the other for integer variables.

The typing rules are defined by a judgment  $\Gamma \vdash_{\Delta} M : \theta$ , the definition of which is mostly standard. Function types are used for the general abstraction, and dependent product types for abstraction over integer variables. For the imperative constructs, the typing rules are all imported from separation logic using the Curry-Howard correspondence mentioned before. For example, the typing rule for heap assignment is

$$\frac{}{\Gamma \vdash_{\Delta} [E] := E' : \{E \mapsto -\} - \{E \mapsto E'\}},$$

which exactly corresponds to the rule for heap assignment in separation logic. The higher-order frame rules come from the usual subsumption rule and the subtyping relation  $\preceq_{\Delta}$ , which is the usual structural subtyping relation [97], extended with rules that

are specific to separation-logic typing. These include the rule of consequence:  $\{P'\} - \{Q'\} \preceq_{\Delta} \{P\} - \{Q\}$  when  $P$  implies  $P'$  and  $Q'$  implies  $Q$ , and the generalized frame rule:  $\theta \preceq_{\Delta} \theta \otimes P$ . The name of this last rule comes from the rules regarding  $\otimes$  which include

$$\begin{aligned} & (\{P_0\} - \{Q_0\}) \otimes P \simeq \{P_0 * P\} - \{Q_0 * P\} \text{ and} \\ & (\theta \rightarrow \theta') \otimes P \simeq (\theta \otimes P) \rightarrow (\theta' \otimes P). \end{aligned}$$

This gives rise to many interesting higher-order frame rules; for instance, the hypothetical frame rule (1.6) mentioned before can be derived. In our type system, the preconditions read

$$\begin{aligned} & \vdash C_1 : \{P_1 * P_0\} - \{Q_1 * P_0\} \text{ and} \\ & k_1 : \{P_1\} - \{Q_1\} \vdash C : \{P\} - \{Q\}. \end{aligned}$$

Using the standard rule for  $\lambda$ -abstraction, the latter can be rewritten as

$$\vdash (\lambda k_1. C) : (\{P_1\} - \{Q_1\}) \rightarrow (\{P\} - \{Q\}), \quad (1.8)$$

and since **let**  $k_1 = C_1$  **in**  $C$  **end** is syntactic sugar for  $(\lambda k_1. C)C_1$ , if we can infer

$$\vdash (\lambda k_1. C) : (\{P_1 * P_0\} - \{Q_1 * P_0\}) \rightarrow (\{P * P_0\} - \{Q * P_0\})$$

from (1.8), the desired result follows from the standard typing rule for  $\lambda$ -application. This can be inferred from subtyping:

$$\begin{aligned} & (\{P_1\} - \{Q_1\}) \rightarrow (\{P\} - \{Q\}) \\ & \preceq_{\Delta} (\{P_1\} - \{Q_1\}) \rightarrow (\{P\} - \{Q\}) \otimes P_0 \\ & \preceq_{\Delta} (\{P_1\} - \{Q_1\}) \otimes P_0 \rightarrow (\{P\} - \{Q\}) \otimes P_0 \\ & \preceq_{\Delta} (\{P_1 * P_0\} - \{Q_1 * P_0\}) \rightarrow (\{P * P_0\} - \{Q * P_0\}). \end{aligned}$$

The main challenge is to give a categorical model for this type system which captures the standard semantics for separation logic and at the same time is adequate with respect to the standard semantics of idealized algol. We present such a model in this dissertation. Many aspects of it are standard [61]; for example, function types and dependent types are interpreted in the usual way by exponentials and dependent products. This is possible since our categorical model is a family fibration where each fiber is a cartesian closed category. The intuition behind the interpretation of the base types  $\{P\} - \{Q\}$ , which is the main non-standard feature of the interpretation, is as follows.

Consider the type  $\{P\} - \{Q\}$ , where  $P$  and  $Q$  are sets of assertions, and let the sets  $p$  and  $q$  of heaps be the denotations of  $P$  and  $Q$ , respectively. The interpretation should reflect properties of terms of this type, so we analyze these here. First, these terms are (non-deterministic) commands in the usual sense, so they can be viewed as maps from  $Heap$  to  $\mathcal{P}(Heap \cup \{wrong\})$ . Let  $comm$  be the subset of these maps that possess the properties safety monotonicity and the frame property mentioned in Section 1.1 – this is a pointed cpo, ordered by pointwise inclusion. Elements of the type  $\{P\} - \{Q\}$  are elements of  $comm$  and they “go from  $P$  to  $Q$ ”, in the sense that if  $c$  is an element of the type, then

$$\forall h \in p. c(h) \subseteq q. \quad (1.9)$$

This captures the semantics of standard separation logic. Further, for all extensions of heaps in  $p$ , clients should not be able to distinguish between elements of the type. Therefore, call two elements  $c, c'$  of  $comm$  equivalent if they have the property (1.9) and

$$\forall h \in p * true. c(h) = c'(h).$$

This defines a partial equivalence relation (per) on  $\text{comm}$ . By definition, clients cannot distinguish between elements that are related by this per.

Finally, our semantics should validate higher-order frame rules. Intuitively, this means that for each assertion  $P_0$  (with the predicate  $p_0$  as denotation), elements of the interpretation of  $\{P\} - \{Q\}$  should also have the properties of the elements of type  $\{P * P_0\} - \{Q * P_0\}$ , which defines another per on  $\text{comm}$ . Consequently, the interpretation of  $\{P\} - \{Q\}$  is a *family* of pers, indexed by predicates. The per  $R(p_0)$  has as domain elements in  $\text{comm}$  with the property that

$$\forall h \in p * p_0. c(c) \subseteq q * p_0,$$

and two elements  $c, c'$  in this domain are equivalent if and only if

$$\forall h \in p * p_0 * \text{true}. c(h) = c'(h).$$

More generally, the programming language is interpreted in the category  $\mathcal{C}$ , which has

**Objects** Pairs  $(A, R)$ , where  $A$  is a pointed cpo, and  $R$  is a family of admissible pers on  $A$ , indexed by predicates, such that for all predicates  $p, q$ ,

$$R(p) \subseteq R(p * q).$$

**Morphisms**  $f : (A, R) \rightarrow (B, S)$  are continuous functions  $f : A \rightarrow B$  with the property that for all predicates  $p$ ,

$$a[R(p)]a' \text{ implies } f(a)[S(p)]f(a').$$

It is not hard to see that this category is cartesian closed. The exponential  $(A, R) \Rightarrow (B, S)$  has the exponential of  $A$  and  $B$  as the cpo-part, and two functions in this space are related if they take related arguments to related results for all the relations  $R(p)$  and  $S(p)$ .

The programming language is interpreted in a standard way, namely by induction on the typing derivation of a term. The interpretation of the  $\lambda$ -calculus part of the language follows the usual lines, so we interpret types as objects in  $\mathcal{C}$ , subtyping judgments  $\theta \preceq_{\Delta} \theta'$  as morphisms  $\llbracket \theta \rrbracket \rightarrow \llbracket \theta' \rrbracket$ , and terms as morphisms. Most of this is done in the standard way, but again the interpretation of the special imperative terms of the programming language needs to be specified. As it happens, the subtyping judgments are simply interpreted by identities, and the morphisms corresponding to the imperative constructs are defined straightforwardly. More details can be found in the paper [25] which is included in this dissertation.

The semantics captures the standard semantics of separation logic, and we show that our interpretation is adequate with respect to the standard semantics of idealized algol [89], in the sense that the forgetful functor preserves the interpretations of terms. Finally, we show that our semantics is coherent. Due to the subtyping relation, there can be more than one typing derivation of a term, and these derivations might result in different denotations. Our coherence result shows that morphisms resulting from different typing derivations of the same term are always equivalent in the sense that they take related arguments to related results. Note that this is formulated differently in the paper, where this equivalence is captured by interpreting the terms of the programming language in a category which has the same objects as  $\mathcal{C}$ , and the morphisms are obtained by equating those morphisms in  $\mathcal{C}$  that are equivalent in the sense just mentioned.

Note that, already for the second-order frame rule (1.6), the authors of the paper [88] noted that the rule of conjunctions is not compatible with unrestricted frame rules of higher order. They proposed to restrict the hypothetical frame rule to a certain class of assertions, which is called *precise*. In our work, we do not have the preciseness requirement; we get away with this because our system does not include the conjunction rule. Presumably, if we restrict to precise predicates, as in [88], we could allow the rule of conjunctions on base types, but the details have yet to be worked out. Difficulties lie within the fact that the rule of conjunction expresses an interplay between the logic of assertions and that of specifications. We discuss this further in Section 1.6.

### 1.2.4 Refinement and Data Abstraction

In the classic work on refinement, by Wirth [133] and Hoare [52], the idea is to develop programs starting from abstract representations of data types, and prove properties of client programs that use these abstract representations. Then, from a refinement of these abstract data types to concrete implementations, one can infer properties about the client program which uses the concrete implementations. The idea is sometimes referred to as *data abstraction*, inasmuch as the abstract data type can be said to be an abstract representation of a concrete implementation of the data type. In his treatment of refinement [52], Hoare uses a statically scoped separation between the client’s data and the data of the data type, simply by requiring disjointness of the sets of variables “owned” by the client and the data type, respectively. In later work [49], Hoare, He, and Sanders prove refinement results via *simulations* relative to relations. The idea is to exhibit a relation between the abstract and concrete representations of a data type and prove that for all the operations on the data type, the concrete implementation simulates the abstract, *i.e.*, related states are taken to related states by the two implementations of operations. In the presence of pointers, the aliasing problem arises again, since although a client and a data type only have access to disjoint sets of variables, they might have access to overlapping parts of the heap, and so the required disjointness properties fail. Separation logic expresses disjointness of heap storage via the  $*$  connective, so it is reasonable to ask what new insights can be gained from combining the ideas of separation logic with earlier work on refinement.

We give a partial answer to this question in this dissertation. In particular, we provide an analysis from the point of view of the client programs that uses abstract and concrete versions of data types. Let us consider an interesting example which illustrates the need for separation of the parts of memory used by a client program and the implementation of a data type.

In standard implementations [64] of `malloc` and `free` (which are similar to our `cons` and `dispose`), the heap cells “owned” (or “managed”) by the memory management module are stored in a singly-linked *free-list*. Calls to the memory management operations simply result in pointer swings such that cells are appended to or taken out of the list. Now, consider the program

$$x := \mathbf{cons}(); [x] := 7; y := [x]; [x] := 8; \mathbf{dispose}(x); [x] := y.$$

If `cons` and `dispose` are implemented as outlined above, the last two commands here will append the cell pointed to by  $x$  in the free-list and then change the contents to 7. However, if the memory management system is implemented in some other way, the last heap assignment might cause a program crash. The problem is that the program has

a bad behavior, and should not be allowed – a program should not assign to heap cells that it has disposed.

Motivated by this example, we introduce the notion of a *separation context* which, intuitively speaking, is a client program that does not interfere with the internal heap storage of the modules it uses. We introduce a configuration *av* (for “access violation”) to the operational semantics to capture the result of a client program that reads from or modifies a part of the heap which is owned by a module. Hence, a client program is a separation context if it does not access violate. Several examples (and non-examples) of separation contexts are presented which, among other things, illustrate the difference between a memory fault (when a program accesses a heap cell which is not in the domain of the heap) and access violation. Moreover, we prove a simulation theorem which, states that if two client programs are equal except that one calls an operation  $\text{oper}_1$  of a module  $M_1$  and the other calls an operation  $\text{oper}_2$  of the module  $M_2$ , and if the module  $M_2$  is a refinement of  $M_1$ , in the sense outlined above, then the second client program is a refinement of the first one. Finally, we give examples of relations between implementations of modules that are preserved by the corresponding implementations of module operations. Thus, the relations are indeed refinement relations.

Note that the work presented here is purely *semantic*, there is no inference system to derive refinements or separation contexts for modules, and our sample refinements are proved by giving explicit relations and showing that the operations simulate each other with respect to these, in the sense outlined above. It is on the schedule for future work to investigate the possibilities of a logic which captures the results outlined here. This is elaborated upon in Section 1.6.

### 1.2.5 A Parametric Model for Separation-logic Typing

As mentioned, there is no syntax to capture the work described in Section 1.2.4. Benton [13] and Yang [135] present a notation for imperative programs that preserve relations, making it possible to prove relationships between programs. Benton’s system does not consider pointers, so we focus on Yang’s relational separation logic. The central notion is that of a *quadruple*

$$\{R\} \begin{array}{c} C_1 \\ C_2 \end{array} \{S\}, \quad (1.10)$$

where  $R$  and  $S$  denote relations and  $C_1$  and  $C_2$  are programs; there is a grammar for relations, which is omitted here. The informal meaning of (1.10) is that if states are related by  $R$ , then the programs  $C_1, C_2$  either both diverge or take these states to states that are related by  $S$ . Yang presents proof rules for deriving valid quadruples, and proves them sound using operational semantics. The programming language is that of standard separation logic, so just like standard separation logic was extended to a higher-order programming language, it is interesting to extend the scope of relational separation logic to a more advanced programming language.

Our approach is to extend the model for separation-logic typing to handle *quadruple types* rather than the triple types of separation-logic typing. Whereas the terms in separation-logic typing can be said to preserve predicates, terms in quadruple-typing should preserve *relations*. Hence the goal is to extend the model described in Section 1.2.3 to give a relational interpretation of types, and show that typed terms preserve these interpretations, in the style of earlier work on logical relations for typed  $\lambda$ -calculi [101, 71].

We pursue this goal by presenting a *semantic* study of the properties of such a model.

In other words, we define a category in which a relational interpretation can be given, and show results that correspond to, e. g., the fundamental theorem of logical relations [101] and the simulation theorem mentioned in Section 1.2.4. This category can briefly be described as follows.

The interpretation of types in the model for separation-logic typing takes place in a category of families of pers on pcpo that are indexed by predicates on heaps. This can be seen as a category of functors from a category of predicates to a category of pers on pcpo

$$\mathcal{E}_v \xrightarrow{F_v} \mathcal{D}_v . \quad (1.11)$$

Following the lines of earlier work [116, 74, 86, 41], we use functors between *reflexive graphs* to give relational interpretations of types. A reflexive graph is a standard way to capture the concept of relations in categories; briefly it is a structure consisting of two categories and three functors like this:

$$\begin{array}{ccc} & \mathcal{C}_e & \\ \delta_0 \left( \begin{array}{c} \uparrow \\ Id \\ \downarrow \end{array} \right) \delta_1 & & \\ & \mathcal{C}_v & \end{array}$$

The intention is that sets live in  $\mathcal{C}_v$ , whereas relations live in  $\mathcal{C}_e$ . The functors represent the diagonal relation and projections. The categories in (1.11) are extended to reflexive graphs, and the functors to pairs of functors like in the diagram

$$\begin{array}{ccc} \mathcal{E}_e & \xrightarrow{F_e} & \mathcal{D}_e \\ \delta_0 \left( \begin{array}{c} \uparrow \\ Id \\ \downarrow \end{array} \right) \delta_1 & & \delta'_0 \left( \begin{array}{c} \uparrow \\ Id' \\ \downarrow \end{array} \right) \delta'_1 \\ \mathcal{E}_v & \xrightarrow{F_v} & \mathcal{D}_v \end{array} .$$

Rather than going into details, we mention that for simplicity, general monoids are used on the left side of this diagram, instead of the specific monoid of predicates (with the monoidal operation  $*$ ) used in the model for separation-logic typing. The category  $\mathcal{D}_e$  has “relations between pers” as objects. This amounts to relations that are saturated with respect to two pers [8, 1].

As mentioned, several results are shown in this functor category, but there are problems having to do with the non-deterministic nature of **cons**. This makes the work incomplete, but it is enclosed in this dissertation anyway, since there are many results which are useful and show that the work leads towards a parametric model for separation-logic typing. Also, the manuscript, in which this work is presented, includes proofs of results that are more general than certain important lemmas in the conference paper [25]. The latter paper is included in this thesis, but does not include proofs of the lemmas in question.

## 1.2.6 Summary of Data Abstractions

As can be seen from the introduction of the various papers included in this thesis, separation logic and the extensions described herein offer different ways to reason about data abstraction. We give a systematic overview of the different approaches here. Put very briefly, the work on higher-order separation logic extends the *assertion language* of

standard separation logic, but keeps the programming language simple. In contrast, the work on separation logic-typing described in Section 1.2.3 extends the *programming language* of standard separation logic to higher-order, whereas the assertion language of the program logic is kept simple. Both of these formalisms are designed to prove properties of a single program. To study the relationship between two programs, the work on separation contexts is an initial study of refinement in separation logic, where only first-order programs are analyzed, and the work on quadruple-typing is a first step towards extending this to higher-order programs. Of course, it will be interesting to pursue *combinations* of some of these lines of research. This is discussed further in Section 1.6.

The work on separation contexts is a semantical study, and it does not offer a logic. Further, it is a limitation that the client programming language must be deterministic, except through module operations, and the work does not describe the situation for higher-order programs or modules. On the other hand, we have illustrated that both cross-boundary pointers and ownership transfer fit nicely into the study.

The work on higher-order separation logic and the ways it offers to reason about data abstraction via quantification over propositions is certainly interesting. Firstly, the categorical model we use is a very simple extension of a concept that has been known for a long time, and it is easy to see that the standard pointer model of separation logic is an instance of this general framework. Moreover, we have demonstrated that our framework can handle non-trivial examples: in our paper on higher-order separation logic, we demonstrate how one can reason about two different implementations of a priority queue, and how a client program using these implementations is independent of the implementation used. In other unpublished work, we also demonstrate that the connection pool example of Parkinson's thesis [92] fits into our framework, and since his calculus is quite similar to ours, we should be able to deal with the same examples as he does. Our framework can also handle cross-boundary pointers and ownership transfer of heap cells, but we have not yet fully worked out how to treat other Java-like features like inheritance, cooperative sharing, *etc.* This is discussed further in Section 1.6. We have moreover illustrated how we can use universal quantification over propositions to reason about parametricity on quite simple examples, but we need more examples to give an assessment of its impact. For this, we should extend our work to deal with a programming language of at least third order, so that we could, for example, reason about the **map** function for lists.

Our work on using quadruple typing for proving relationships between programs is, as already noted, at its early stages. So far, we have been unable to find a solution to the problem arising from nondeterministic allocation, and we do not have any examples that demonstrate the usefulness of quadruples – although we believe it should be possible to fit Yang's example in the paper [135], where he shows that the Schorr-Waite is equivalent to a depth-first traversal, into our framework. Furthermore, we have the same problem that Yang pointed in in the paper [135], namely that to use our formalism, the programs one considers, must have similar control structures. On the positive side, the formalism is compositional in the same sense as Hoare logic, since it has rules for atomic program constructs, and also, it uses an extension of the categorical model we use to model separation logic typing.

There is a, so far quite vague, connection between the abstract function definition rule and quadruples. In the paper on higher-order separation logic included in this thesis, we have demonstrated how we can derive representation independence at the meta-level for two implementations of an abstract module. This means that there is a relationship

between the two implementations, and so a certain quadruple presumably holds for them. The details, however, are far from completely worked out, but we have elaborated on this in Section 1.6.

Our work on separation-logic typing has answered an important scientific problem that arose after the hypothetical frame rule was presented, inasmuch as we have extended the scope of separation logic to higher-order programs. We do not have many examples of derivations in our type theory at this point, but examples in the paper [88] on the hypothetical frame rule show that ownership transfer can be dealt with. To get around problems of modifies clauses that are quite complicated already for the hypothetical frame rule, we restrict to immutable variables in our system. As mentioned, this is not an unreasonable restriction, since, e. g., ML is also designed in this way. The categorical model we have provided for the type system is quite elegant and combines basic intuition about the interpretation of terms in standard separation logic with previous work on semantics for algol and other typed  $\lambda$ -calculi. As described below and in work by Parkinson and Bierman [91, 92], the work on higher-order frame rules only deals with static modularity, whereas we get dynamic modularity from higher-order separation logic.

To further illustrate the difference between the work on higher-order separation logic and that on separation logic-typing, certain differences between the abstract function definition rule (AFD) mentioned in Section 1.2.2 and the hypothetical frame rule (HFR) from Section 1.2.3, are discussed here. First, recall these rules. A simplified version of AFD is

$$\frac{\begin{array}{c} \vdash \hat{P}:\tau \\ \vdash \{P_1[\hat{P}/x]\} C_1 \{Q_1[\hat{P}/x]\} \\ \exists x:\tau. (\{P_1\}k_1\{Q_1\}) \vdash \{P\}C\{Q\} \end{array}}{\vdash \{P\} \mathbf{let} k_1(\bar{x}_1) = C_1 \mathbf{in} C \mathbf{end} \{Q\}} \quad x \notin \text{FV}(P, C, Q), \quad (1.12)$$

and HFR is

$$\frac{\begin{array}{c} \vdash \{P_1 * P_0\} C_1 \{Q_1 * P_0\} \\ \{P_1\}k_1\{Q_1\} \vdash \{P\} C \{Q\} \end{array}}{\vdash \{P * P_0\} \mathbf{let} k_1 = C_1 \mathbf{in} C \{Q * P_0\}} \quad , \quad (1.13)$$

These rules look similar, and they are both used to reason about a program  $C$  that uses an abstract name  $k_1$  to call a concrete implementation  $C_1$  of a module operation. However, there are important differences. First, the hypothetical frame rule is the simplest of the rules obtained from the work described in Section 1.2.3; the generalized frame rule gives many other interesting frame rules. Second, the proof of the specification for the client program  $C$  in AFD can use the “abstract predicate” which is called  $x$  in the rule in its proof (in both rules, the proof of the concrete implementation  $C_1$  may use the “resource invariant”, called  $P_0$  in HFR and  $\hat{P}$  in AFD). When using the hypothetical frame rule, it is only possible to use the interface specification of the module; there is no “abstract name” for the module. As explained in the recent paper [91] and in Section 3.4.2 in Chapter 3 of this dissertation, this means that the modularity one obtains from HFR is static, whereas the modularity accomplished from AFD is dynamic. In practice, this means that client programs can only use one instance of a data type in a proof using HFR, while using AFD makes it possible to prove programs that use multiple instances of a given data type. When programming with lists, for example, it is often necessary to have more instances, so here it is clearly advantageous to use the framework described in Section 1.2.2.

### 1.3 Related Work

In this section, we give pointers to the most important related works, and mention a few references that are not already mentioned in the included papers.

Several proofs of correctness of garbage collectors have been published, starting almost 30 years ago [39, 12, 100]. However, many of the implementations there have merely been abstract graph algorithms, in contrast to our implementation, which is quite close to a “real” implementation. Moreover, most proofs have been more informal than ours, and Rusinoff [117] demonstrated that the informal approach can be perilous by showing that the proofs in [12] and [100] are fallacious. As mentioned, Yang specifies and proves correct an implementation of the graph-marking algorithm by Schorr, Waite, and Deutsch in his dissertation [134], and that to our knowledge is the only other proof of a garbage collector which uses local reasoning at the level of heaps. Note that our proof focuses only on a garbage collector, whereas other published papers [42, 130, 46] show safety of entire run-time systems, including a memory management system. The works just mentioned, however, do not use local reasoning, and as mentioned, they show type safety, whereas our work proves correctness of a garbage collector (*i.e.*, the existence of an isomorphism between the heaps before and after execution of the garbage collector). It also deserves mentioning that our garbage collector is a simple stop-and-copy one, whereas the works mentioned above study more complex (e. g., concurrent, generational) garbage collectors.

As mentioned, the notion of a BI hyperdoctrine stems from Lawvere’s notion of a hyperdoctrine [66], which is a categorical structure tailored to model first-order intuitionistic predicate logic with equality. These structures are also analyzed in [99], and they are extended to BI hyperdoctrines in Biering’s master’s thesis [19].

Models of propositional BI were published in [104], exploiting the doubly-closed structure of  $\text{Sub}(1)$  in certain categories of presheaves. Pym [102, 103] presents a different notion of predicate BI than ours, and gives models for it, in the style of Kripke. The claim that our notion of higher-order predicate BI is the right one for separation logic is justified by the fact that the standard semantics for separation logic is an instance of our class of models. To our knowledge, our studies of higher-order predicate BI is the first that yields applications for separation logic, in the form of reasoning principles for data abstraction.

The literature on reasoning principles for data abstraction is quite rich, and was initiated by the works of Parnas [93, 94, 95] and Hoare [52] who, however, considered a setting without pointers. The first works on data abstraction and modularity which use separation logic were published in [88] and [91] and as mentioned before, the logic of Parkinson and Bierman provides dynamic modularity, whereas that of O’Hearn *et al.* only deals with static modularity. Our notion of data abstraction is similar to that of Parkinson and Bierman [91, 92], but our approach follows more standard lines than the abstract predicates introduced in that work, since we directly use existential quantification to model abstraction, in the style of Plotkin and Mitchell [73]. This was also done by Reddy [105] for a higher-order programming language, but in a setting without pointers. It should be noted, however, that Parkinson deals with issues like inheritance and cooperative sharing in his work [92], which we do not.

As mentioned in Section 1.1, the separating conjunction and the frame rule are the most significant reasons for the success of separation logic. It was the soundness proof of the second-order frame rule [88] which spawned our work on a general higher-order

frame rule.

There have been numerous studies of the semantics of idealized algol [89, 109, 106, 86, 134] and we follow the lines of those works by parameterizing the semantics by the shape of the stack. However, our semantics is the first to use indexing by invariant predicates over heaps, and that gives us a more fine-grained control over heap usage. Other type systems track state change for typed assembly languages [4, 5, 77, 124] and give sound reasoning principles for proving safety for programs. Our type system allows proofs of more refined properties, and treats higher-order programs. The downside of our approach is that no type-inference algorithm exists, not even for the first-order fragment of our language (general type inference is easily seen to be undecidable).

We already mentioned earlier work on data abstraction which clearly relate to our work on refinement, but which does not consider pointers. Other authors aim for abstraction results in the presence of pointers, for example Hogg’s work on Islands [54], Clarke’s work on ownership types [36, 30], and the work of Banerjee and Naumann [10] on confinement. These works deal with substantial fragments of Java and treat issues like inheritance, behavioral subtyping, etc., that do not arise in our simpler setting. However, they do not allow pointers that point across “boundaries” between client heap storage and the internals of modules, nor with pointers whose ownership transfers between modules and clients. In the extended version [11] of Banerjee and Naumann’s conference paper [10], outwards pointers are allowed to exist and even be dereferenced and mutated, and in their conference paper [9], representation independence is proved using predicate based ownership, heap encapsulation and control of callbacks (via a specific annotation discipline which uses special auxiliary fields) – and both of these papers treat considerable fragments of Java, like the papers mentioned above. The reasoning principles provided by separation logic allows pointers to change ownership and point across boundaries, as long as they are not dereferenced while they are not in scope. It is clearly desirable to explore the possibilities of extending our work to deal with more realistic fragments of Java. This is discussed further in Section 1.6.

As mentioned, we lack an inference system to prove relationships between programs (e. g., refinements); Yang’s relational separation logic [135] and Benton’s relational Hoare logic [13] are first steps towards such systems. Yang’s system does not consider the viewpoint of *clients* using related programs, but has a frame rule for quadruples. A hypothetical frame rule for quadruples would capture many of the properties of the analysis in [70], which is included in the dissertation.

Our paper [25] on separation-logic typing is the main inspiration for our work towards a parametric model. The idea of using reflexive graphs to prove results of parametricity has been used before [86, 40, 41, 106, 74].

Honda’s group in London has worked on reasoning principles for equivalence of imperative programs [17, 56]. In a recent paper [57], they present a Hoare-logic for a higher-order imperative programming language, but without pointers. In the paper [18], this work is extended to treat aliasing. The aim of that work is to prove observational equivalence; although intriguing and clearly relevant for our work on representation independence, we do not attempt to prove such properties. Another difference is that they do not use a separating connective (like  $*$ ), but use predicate logic with equality to keep track of aliasing; this makes local reasoning harder. Benton and Leperchey use ideas from separation logic and Reddy and Yang’s work on “disjoint relations” [106] to present a nominal semantics for storage [14] in which they model MILLer, a higher-order call-by-value programming language with dynamically allocated mutable references. They use

properties of the denotational model to prove interesting contextual equivalences, for example those of Meyer and Sieber [69]. Naumann presents a theory of data refinement with which one can prove equivalences of higher-order programs in the paper [79]. He uses a semantics based on predicate transformers which is based on an earlier semantics [78] for a higher-order imperative language with record subtyping.

The paper [17] has an elaborated overview of related work. Most of the references accounted for there are also relevant in the context of this dissertation. Although the work in the report [56] does not deal with pointers and the issues of aliasing, the related work section there also has many references relevant to this dissertation.

## 1.4 Included Papers

Apart from this introduction, this dissertation consists of five papers. Here is a brief introduction to each of the papers.

**Paper 1:** L. Birkedal, N. Torp-Smith, and J. C. Reynolds. Local Reasoning about a Copying Garbage Collector. *ACM TOPLAS*, 2005. To appear (Accepted modulo revisions).

This paper is an extended version of the paper with the same name [23] which appeared in POPL'04. We give an extension of separation logic with finite sets and relations, and use the notion of the iterated separating conjunction to give a correctness proof in separation logic of the classical stop-and-copy garbage collection algorithm by Cheney [35].

**Paper 2:** B. Biering, L. Birkedal, and N. Torp-Smith. BI Hyperdoctrines, Higher-Order Separation Logic, and Abstraction. Technical report ITU-TR-2005-69, IT University of Copenhagen.

This paper is an extension of the ESOP'05 paper [20]. We give a correspondence between separation logic and our notion of *predicate* BI. We also introduce the notion of a BI-hyperdoctrine and show that it soundly models intuitionistic and classical first- and higher-order predicate BI, and show that separation logic may easily be extended to *higher-order*. Moreover, we present examples showing that this extension has applications in program proving. In particular, we present an example illustrating how the extension facilitates reasoning about data abstraction in the presence of pointers.

**Paper 3:** L. Birkedal, N. Torp-Smith, and H. Yang. Semantics of Separation-logic Typing and Higher-order Frame Rules. In *Proceedings of the Twentieth Annual IEEE Symposium on Logic in Computer Science (LICS 2005)*, pages 260–269. Chicago, IL, June 2005.

We give a semantics for programs that can be typed in a type system based on separation logic for a programming language with higher types: idealized algol extended with heaps, but with immutable stack variables. This leads to simple rules for deriving higher-order frame rules, and thus allows for local reasoning for higher-order programs. The semantics is shown to be coherent and has a close correspondence with the standard interpretation of idealized algol.

**Paper 4:** I. Mijajlović, N. Torp-Smith, and P. W. O'Hearn. Refinement and Separation Contexts. In *Proceedings of the Twenty-fourth International Conference on Foundations*

of *Software Technology and Theoretical Computer Science (FSTTCS'04)*, pages 421–433. Chennai, India, December 2004.

This paper presents the first steps towards bringing the ideas from separation logic into refinement. We introduce a model, though not yet a logic, which ensures static separation between client programs and modules. The central notion is that of a *separation context*, which, informally, is a client program that does not dereference pointers that belong to a module. We show an abstraction theorem which implies that if a client program is a separation context for an abstract data type, then it is a separation context for all its refinements.

**Paper 5:** N. Torp-Smith, L. Birkedal, and H. Yang. Towards a Parametric Model for Separation-logic Typing. Unpublished manuscript. July 2005.

We present an extension of the categorical model presented in Paper 3 mentioned above, which is intended to be a first step, towards a *parametric* model for separation-logic typing. We show many properties of this model, including a semantic version of an abstraction theorem which illustrates how one can obtain a *quadruple type* of a pair of terms from a triple typing of a term in the sense of Paper 3. However, the work is not completed, since some obstacles with non-determinism of heap allocation need to be resolved.

The papers can all be read independently of each other, but they require a basic familiarity with separation logic, for example as provided in Section 1.1. The only exception is Paper 5, for which the material presented in Paper 3 is a prerequisite. As mentioned earlier, Paper 3, which is a conference paper has proofs of certain lemmas left out. Proofs of more general results than these can be found in Paper 5.

## 1.5 Conclusion

This dissertation presents several advances in the new field of separation logic, and at the same time reveals directions for future work, as discussed in Section 1.6.

As mentioned, we believe that the method used for specifying and proving correct the copying garbage collector in Paper 1 using the iterated separating conjunction is generally applicable in program proving, since it captures the way the task of proving correctness of a program is often approached: we derive an invariant and a specification of the program from a partition of the heap based on a snapshot of execution, using the iterated separating conjunction. The work in Paper 1, however, does not offer a really insightful assessment of separation logic compared to other formalisms for showing properties of programs. This is discussed further in Section 1.6.

The notion of BI-hyperdoctrines and the semantics for higher-order predicate BI it induces has paved the way for interesting research. We have demonstrated that the higher-order logic and its quantification over propositions yields reasoning principles for data abstraction and parametricity in the presence of pointers. We believe the reasoning principles for data abstraction is a first step towards applying higher-order predicate BI to give semantics and reasoning principles for programming languages like e. g., Java, that are used in practice. We discuss some challenges in this regard in Section 1.6.

Once we got the idea of the Curry-Howard correspondence with which we view specifications for programs as types of those programs, the separation-logic type system presented in Paper 3 is quite intuitive. At the same time it captures all the interesting

frame rules we can think of for the simple programming language in that paper (but if further features are added to the language, then presumably there will be other frame rules to consider), hence allowing local reasoning for many higher-order programs. This indicates that this way of bringing the ideas of separation logic into work on higher-order programming languages is the right one. It will be interesting to work towards a parametric model for separation-logic typing and to combine ideas of separation-logic typing with the work on higher-order separation logic, e. g., to move closer to recent research on parametricity.

In conclusion, the work presented in this dissertation demonstrates that separation logic has indeed matured considerably over recent years, and its scope has widened significantly. Consequently, we believe that separation logic will be the foundation for a lot of research on establishing properties of systems involving shared access to mutable data structures.

## 1.6 Future Work

As Reynolds points out [115], “it can be discouraging to go back and read the “future directions” sections of old papers, and to realize how few of the future directions have been successfully pursued”. In spite of this, some interesting directions for future work are supplied here.

As mentioned, the iterated separating conjunction  $\forall_*$  is used crucially in our proof of Cheney’s garbage collection algorithm, and we claim that our methodology of using a snapshot of execution together with  $\forall_*$  is applicable in many settings. It will be interesting to verify this claim, for example by verifying complete run-time systems or more complicated garbage collection algorithms. Using higher-order predicate BI as assertion language might yield simpler proofs than the one presented for our garbage collector in Paper 1. It will also be interesting to formalize our proof of Cheney’s garbage collector, or even a proof of an entire run-time system, and use this along the lines of Necula’s proof carrying code [80]. Nipkow’s group in Munich have worked towards a formalization of separation logic [131] in the proof assistant Isabelle [82], and Lia and Walker also work a logical framework for automated reasoning about pointer programs [62].

Unfortunately, the work in Paper 1 does not offer an in-depth assessment of separation logic compared to other formalisms for showing properties of pointer-manipulating programs. There are several ways in which one could proceed from the work in that paper. Firstly, we could apply some of the other advances in this thesis to the work – the similarities between the two parts of the **while**-loop begs for a procedural abstraction like the one offered by higher-order separation logic. Second, our proof of Cheney’s algorithm could be used to compare separation logic to other formalisms if we carry out a proof of the algorithm in those formalisms. Finally, we could employ separation logic on some of the previous examples in the literature, e. g., a concurrent garbage collector, and compare the resulting proof to other proofs. This would give a better possibility for assessing separation logic.

As mentioned, it is desirable to combine the ideas of data abstraction with separation logic typing, to obtain reasoning principles for data abstraction in higher-order programs. The approach would be to enrich the assertion language of separation logic typing with variables of type `Prop`, and use existential quantification to model data abstraction, as described in Paper 2. However, it is a problem that the model we described in Section 1.2.3 is cartesian closed and has fixed points; this means that it cannot be co-

complete unless it is trivial, as shown by Huwig and Poigne [58]. Either we have to change our model fundamentally, or model existential quantification by something else than a colimit. We plan to work along the lines of the paper [22], which gives a model for polymorphic intuitionistic/linear lambda calculus, to solve this problem. We already know how to use a separated sum to model a sum type over the type of propositions, and presumably, this can be used to model the existentials, but the details are not worked out at this point.

The work in Paper 5 is unfinished and needs more work. The goal is a parametric model and a type system which captures this parametricity. This will extend Yang's relational separation logic [135] to a higher-order programming language and give proof techniques to show relationships between higher-order programs. Presumably, this would supersede the work in Paper 4.

As illustrated in Paper 2, the abstract function definition rule can be used to show independence of abstract modules *at the meta-level* (i.e., that paper does not give a logic to directly infer relationships between programs). It will be interesting to analyze how this kind of representation independence relates to the properties obtained from using quadruples or quadruple types. We suspect that from the preconditions in the abstract function definition rule (1.5), one can derive the quadruple

$$\left\{ \begin{array}{l} (P) \\ (P) \end{array} \right\} \text{ let } k_1 = C_1 \text{ in } C \text{ end} \quad \left\{ \begin{array}{l} (Q) \\ (Q) \end{array} \right\},$$

where we have used the notation from Paper 5. However, the details have yet to be worked out, and it is not at all clear what the situation is at higher order.

As can be seen from Paper 2 and Paper 3, we use quite different models for our logic of assertions (modeled in BI-hyperdoctrines) and our logic of specifications (modeled in a fibered ccc). One issue that should be noted is that our model for the assertion language supports classical reasoning (since powersets of a preordered commutative monoid is a boolean BI-algebra), whereas classical reasoning is not sound in general for our logic of specifications. We omitted the rule of conjunction from our type system in Paper 3, both because we do not have intersection types, and because this allowed us to get around the anomaly with the hypothetical frame rule and the conjunction rule discovered by Reynolds [88]. If we had intersection types  $\theta \& \theta'$  in our system, we could consider the following formulation of the rule of conjunctions:

$$\frac{\Gamma \vdash_{\Delta} M : (\{P\} - \{Q\}) \& (\{P'\} - \{Q'\})}{\Gamma \vdash_{\Delta} M : \{P \wedge P'\} - \{Q \wedge Q'\}}.$$

This rule expresses an interplay between the logic of assertions and that of specifications. Presumably this rule would be admissible at base types, if the predicates are restricted to be precise. However, it raises the question of what a similar rule might look like at higher order, and what are rules for quantifications? Furthermore, what if we move to quadruple types? To answer this question, we need a better understanding of the interplay between the logics just mentioned and the structures we use to model them.

The programming language for which we show data abstraction results in Paper 2 is rather simple. Parkinson and Bierman give proof rules for a programming language MJ (for Middleweight Java) which has object-oriented features such as classes, objects, and inheritance. It is of course an interesting problem to extend our model to deal with a more realistic programming language which includes some or all of these features.

Parkinson [92] uses an indexing by classes to build so-called “abstract predicate families”. Possibly this can be mimicked via a functor category which has as domain the preorder of classes, and it is likely that this will ensure reasoning principles for behavioral subtyping and thus give reasoning principles for dynamic dispatch, but this needs to be worked out. To deal with read sharing, Parkinson changes the storage model in the style of [29], so that heaps map locations to pairs of values and *permissions*. As a first step in this direction, it is easy to see that this set of heaps is also a partial commutative monoid, so much of the work we have done in Paper 2 can be mimicked without complications (although one has to define new atomic predicates which include permissions, and their meanings).

Finally, we need to work on understanding the relationship between the work of Honda *et al.* [56, 17] and our work, to work towards results such as contextual and observational equivalence in separation logic.

## **Part II**

# **Publications and Manuscripts**



## Chapter 2

# Local Reasoning about a Copying Garbage Collector

### Abstract

We present a programming language, model, and logic appropriate for implementing and reasoning about a memory management system. We then state semantically what is meant by correctness of a copying garbage collector, and employ a variant of the novel separation logics to formally specify partial correctness of Cheney’s copying garbage collector in our program logic. Finally, we prove that our implementation of Cheney’s algorithm meets its specification, using the logic we have given, and auxiliary variables.

### Preface

This chapter is a reprint of the journal paper [24] which, in turn, is an extended version of the conference paper [23]. It was co-authored with Lars Birkedal from IT University of Copenhagen and John C. Reynolds from Carnegie Mellon University.

## 2.1 Introduction

Formal reasoning about programs that manipulate imperative data structures involving pointers has proven to be very difficult, mainly due to a lack of reasoning principles that are adequate and simple at the same time. Recently, Reynolds, O’Hearn, and others have suggested *separation logic* as a tool for reasoning about programs involving pointers; see [115] for a survey and historical remarks. In his dissertation, Yang showed that separation logic is a promising direction by giving an elegant proof of the non-trivial Schorr-Waite graph marking algorithm [134]. One of the key features making separation logic a promising tool is that it supports *local reasoning*: when specifying and reasoning about program fragments involving pointers, one may restrict attention to the “footprint” of the programs, that is, to just that part of memory that the program fragments read from or write to.

The aim of this paper is to further explore the idea of local reasoning and its realization in separation logic. To this end we prove correctness of Cheney’s copying garbage collector [35] via local reasoning in separation logic. There are several reasons to focus on Cheney’s algorithm:

- The algorithm involves interesting imperative manipulations of data; in particular, it not only updates an existing data structure as the Schorr-Waite algorithm does, but *relocates* the structure. Moreover, it simultaneously treats the same data as a set of records linked by pointers and as an array of records.

- Cheney’s algorithm copies any kind of data, including cyclic data structures.

- Variants of the algorithm are used in practice, e.g., in runtime systems for implementations of functional programming languages.

There are two other motivating factors that we call attention to. First, our analysis answers a question in the literature and thus paves the way for important future work that so far has been out of reach: In [33], local reasoning and separation logic for a garbage collected language is analyzed. An underlying garbage collector is presumed in the operational semantics of the language, inasmuch as a *partial pruning* and  $\alpha$ -renaming (i.e., relocation) of the current state is allowed at any time during execution of a program. In [33] it is not mentioned *how* this pruning and renaming should be done, let alone proven that it is done correctly. A remark at the end of the paper expresses the desirability of such a proof — we provide one here. The analyses in [33] and the present paper are at two distinct levels: the former is at the level of a user language using a runtime system (a garbage collector), the latter is at the level of a runtime system providing operations for the user language (memory allocation and garbage collection). We believe these analyses pave the way for an investigation of the correctness of *combinations* of user level programs and runtime systems. We present some preliminary ideas in this direction in Sec. 2.9.

The additional motivating factor is that our analysis of garbage collection should be of use in connection with foundational proof-carrying code [6] and typed assembly language [77]. In these settings, a memory allocation (but no deallocation) construct is part of the instruction set and a memory management system is implicitly assumed. We believe that our correctness proof can contribute to mimicking the work of [77] in a more realistic setting, for more machine-like assembly languages.

### 2.1.1 Contributions and Methodology

In Sec. 2.3 we present our storage model and the syntax and semantics of assertions. As usual in separation logic, a *state* consists of a *stack* and a *heap*, where a *stack* is a finite map from variables to values, and a *heap* is a finite map from locations to integers. A new feature is that our values include finite sets and relations of pointers and integers, which are used to give the semantics of assertions and of auxiliary variables [90]. Our storage model is very concrete and close to real machines; it treats *locations* as multiples of four (A datum other than a location can be encoded as a multiple of four that is not a multiple of eight. More precisely, record fields containing such a number will not be altered by the algorithm). This is similar to what is often used in real implementations of runtime systems for compilers. For simplicity we assume that heaps consist only of **cons**-cells, aligned such that the first field is always on a location divisible by eight; hence *pointers* (to **cons**-cells) are multiples of eight. These assumptions and definitions induce a semantic notion of what it means for a heap to be a garbage collected version of another. This is presented in Section 2.3.2. Our definitions are based on the analysis in [33] (already referred to above) and thus involve generalizations of pruning and  $\alpha$ -renaming of program states.

Our assertion language, presented in Sec. 2.3.4, is an extension of separation logic [115] with new assertion forms for finite sets and relations. These are crucially used to express part of the specification of Cheney’s algorithm; in particular the existence of an isomorphism between pointers to old cells and pointers to copies of the old cells – which exist at different points in time (before and after execution of the algorithm) – is established using the new assertion forms. We believe the methodology of using sets and relations can be used more widely, to specify and prove correct other programs involving shared or cyclic structures. Indeed a somewhat similar approach is being used by Richard Bor-nat [27] to specify and verify an algorithm for copying directed acyclic graphs.

Moreover, we have extended the iterated separating conjunction [115] of separation logic to arbitrary finite sets. The assertion  $\forall_* x \in m. A$  holds in a state  $(s, h)$ , if  $m$  denotes a finite set  $\{p_1, \dots, p_k\}$  and  $A[p_1/x] * \dots * A[p_k/x]$  holds in  $(s, h)$  (see Fig. 2.1 for a precise definition). As illustrated in Sec. 2.5, one can specify a program by separating the locations it manipulates into disjoint sets with different properties, and then use the iterated separating conjunction, together with expressions for finite sets and relations, to express the particular properties of each set.

In Sec. 2.4 we define the syntax and semantics of the programming language used to implement the garbage collector. It is a simple imperative programming language, with constructs for heap lookup and heap update (but no constructs for allocation or disposal of heap cells). The associated program logic is presented in Sec. 2.4.2. The program logic is mostly standard except for the new rules regarding sets and relations, and it includes the frame rule of separation logic which makes local reasoning possible, as explained in Sec. 2.4.2.

Cheney’s algorithm and the specification of our implementation thereof are presented in Sec. 2.5; the implementation itself is included in Appendix 2.A. In Sec. 2.5.1 we define a semantic condition on programs which implies that a program is a correct garbage collector and show that this condition ensures that the heap after execution of the program is a garbage collected version (in the sense of Sec. 2.3.2) of the heap before execution if the latter satisfies certain requirements mentioned in Sec. 2.3.2.

We present an informal analysis of the algorithm and use it to derive a formal specification of an invariant. At any point of execution, the pointers manipulated by the algorithm can be divided into disjoint sets in which the elements have the same property. Thus it is natural to use the method of sets and relations along with the iterated separating conjunction mentioned before. The sets and relations are also used in another crucial way, namely to record the initial contents of the heap (before garbage collection). This makes it possible to relate the final heap (after garbage collection) to the initial heap and prove that the final heap is a garbage collected version of the initial heap. As mentioned, we use these to give a specification for our implementation formulated in the program logic from Sec. 2.4.2, and before we formally prove that implementation meets the specification (in Sec. 2.6), we show in Sec. 2.5.4 that the resulting specification ensures that the implementation is indeed a correct garbage collector.

We emphasize the following point. Cheney’s algorithm assumes two contiguous “semi-heaps” of equal size, OLD and NEW, and works by copying all *reachable* data from OLD into NEW. One of the reasons for the popularity of Cheney’s algorithm (and variants thereof) is that it runs in time proportional to the size of the reachable data; it never touches unreachable cells. This fact is reflected directly in our specification of the algorithm, which refers to the reachable part of OLD only. It is in the spirit of local reasoning to have such a direct correspondence between the intuitive understanding of an algo-

rithm and its formal specification.

## 2.2 An Introduction to Separation Logic

In this section, we give a brief introduction to separation logic. Formal definitions of the concepts used here will not be given, since it would clutter the presentation, and since we will extend traditional separation logic with a few constructs later. For formal definitions, we refer to Sections 2.3 and 2.4.

Separation logic is an extension of traditional Hoare logic [50]. The simple **while**-language is extended with commands for manipulating imperative data structures, stored in a *heap*, and if “dangling” pointers are dereferenced, the semantics for the language will get “stuck”, or “abort”. Accordingly, the assertion language is extended with basic predicates concerning the heap, and two new connectives: the *separating conjunction*  $*$  and the *separating implication*  $\multimap$  (in this paper we will not use the separating implication).

We have specifications  $\{A\} C \{B\}$ , stating that in any state in which  $A$  holds, no execution of  $C$  will abort, and if the execution terminates in a final state, then  $B$  will hold in that state. As a consequence, we have the slogan

*“Well-specified programs do not go wrong.”*

for separation logic.

### 2.2.1 An Example

As mentioned in Sec. 2.1, separation logic provides reasoning principles for proving programs that manipulate shared mutable data structures. We will demonstrate the advantage of separation logic by an example.

In traditional Hoare logic (without shared data structures), one has the *rule of constancy*.

$$\frac{\{A\} C \{A'\}}{\{A \wedge B\} C \{A' \wedge B\}} \text{Mod}(C) \cap \text{FV}(B) = \emptyset$$

This rule has been useful, since it has allowed reasoning about only the parts of the store that are modified by the program fragment. In the presence of aliasing, however, the rule of constancy is *not* sound, as can be seen from the following counterexample.

$$\frac{\{x \leftrightarrow 3\} [x] := 4 \{x \leftrightarrow 4\}}{\{x \leftrightarrow 3 \wedge y \leftrightarrow 3\} [x] := 4 \{x \leftrightarrow 4 \wedge y \leftrightarrow 3\}} ,$$

where the effect of  $[x] := 4$  is that 4 is stored in the heap at the address denoted by  $x$ . The problem here is that  $x$  and  $y$  might be references to the same heap cell, and if we change the value stored in this heap cell, the assertion about what  $y$  points to does not remain true. In Hoare logic, we would have to add a premise like  $x \neq y$ , or some other non-interference predicate, for the conclusion to hold. This may seem like a small thing to do, but if there are many variables in play, the induced number of non-interference predicates in the involved assertions would quickly become intractable. Further, if a program deals with more realistic data structures, the assertions which prevent sharing also become unacceptably complex, as can be seen from the examples in [115]. In contrast, the assertion

$$x \leftrightarrow 3 * y \leftrightarrow 3 \tag{2.1}$$

in separation logic *implicitly* states that  $x$  and  $y$  are disjoint, since the two assertions  $x \hookrightarrow 3$  and  $y \hookrightarrow 3$  must hold in *disjoint* parts of the heap in order for (2.1) to hold. Therefore, the derivation

$$\frac{\{x \hookrightarrow 3\} [x] := 4 \{x \hookrightarrow 4\}}{\{x \hookrightarrow 3 * y \hookrightarrow 3\} [x] := 4 \{x \hookrightarrow 4 * y \hookrightarrow 3\}}$$

is valid in separation logic. In fact, it is an instance of the important *frame rule*:

$$\frac{\{A\} C \{A'\}}{\{A * B\} C \{A' * B\}} \text{Mod}(C) \cap \text{FV}(B) = \emptyset$$

This rule allows *local reasoning*. Suppose we have a program with a **while**-loop which performs some manipulations on a data structure stored in the heap. When verifying the program, one has to exhibit an invariant for the **while** loop and prove that it is indeed an invariant. The invariant is typically an assertion about the full data structure, whereas each iteration of the loop manipulates a small portion of this structure only. The Frame Rule allows us to prove the invariant by proving a specification which mentions only the parts of the heap that are actually manipulated in one loop iteration (this has been called the *footprint* of the code fragment by O'Hearn [115]), and then conclude the specification regarding the full structure from this. We will see several applications of the frame rule in subsequent sections. Also, we discuss benefits of separation logic in general terms in Section 2.7.1, after we have shown correctness of our garbage collector.

## 2.3 Syntax and Semantics

In this section we present our storage model, and define a semantic notion of garbage collection. We then proceed with the syntax and semantics of expressions and assertions which are part of our program logic. The basis of the system is the standard separation logic with pointer arithmetic [85], but we extend the expression and assertion languages with finite sets and relations, new basic assertions about these, and an extension of the iterated separating conjunction to arbitrary finite sets.

### 2.3.1 Storage Model

We assume five countably infinite sets  $\text{Var}^{\text{int}}$ ,  $\text{Var}^{\text{fs}}$ ,  $\text{Var}^{\text{frp}}$ ,  $\text{Var}^{\text{fri}}$  of variables, and let  $\text{Var}$  be the disjoint union of these sets. We let metavariables  $x, y, \dots$  range over  $\text{Var}$  and assume a type-function

$$\tau : \text{Var} \rightarrow \text{types}, \text{ where } \text{types} = \{\text{int}, \text{fs}, \text{frp}, \text{fri}\}$$

indicating which type a given variable has. The sets of *locations* and *pointers* are the sets of integers divisible by 4 and 8, respectively. More formally, we define:

Variables	$x, y, \dots$	$\in$	Var
Locations	$l \in Loc$	$\stackrel{\text{def}}{=}$	$\{4n \mid n \in \mathbb{Z}\}$
Pointers	$p \in Ptr$	$\stackrel{\text{def}}{=}$	$\{8n \mid n \in \mathbb{Z}\}$
Finite sets	FS	$\stackrel{\text{def}}{=}$	$\mathcal{P}_{fin}(Ptr)$
Pointer relations	FRP	$\stackrel{\text{def}}{=}$	$\mathcal{P}_{fin}(Ptr \times Ptr)$
Integer relations	FRi	$\stackrel{\text{def}}{=}$	$\mathcal{P}_{fin}(Ptr \times \mathbb{Z})$
Values	$v \in Val$	$\stackrel{\text{def}}{=}$	$\mathbb{Z} \cup FS \cup FRP \cup FRi$
Heaps	$\stackrel{\text{def}}{=}$	$Loc \rightarrow_{fin} \mathbb{Z}$	
Stacks	$\stackrel{\text{def}}{=}$	$\{s : Var \rightarrow_{fin} Val \mid \forall x \in Var. s(x) \in \llbracket \tau(x) \rrbracket\}$	
States	$\stackrel{\text{def}}{=}$	Stacks $\times$ Heaps,	

where  $\llbracket \text{int} \rrbracket = \mathbb{Z}$ ,  $\llbracket \text{fs} \rrbracket = FS$ ,  $\llbracket \text{fri} \rrbracket = FRi$ , and  $\llbracket \text{frp} \rrbracket = FRP$ . We use the notation  $Loc \rightarrow_{fin} \mathbb{Z}$  to denote the set of finite partial functions from  $Loc$  to  $\mathbb{Z}$ . As mentioned in Section 2.1.1, we will assume that data is arranged in **cons** cells that are aligned in such a way that the first field of each cell is located at a location that is divisible by 8, and this is why we define the subset  $Ptr$  of locations which point to such **cons** cells.

The finite sets and both kinds of relations in the table above are extensions of traditional separation logic. In the sense of [108] and [90], variables of these types are used as auxiliary and ghost variables in the garbage collector and the proof of its correctness. This means that they are not necessary for the program to work, but they ease proofs of its properties.

Before we present the expression and assertion languages of our program logic, we define several concepts related to heaps; these are needed for the definition of a correct garbage collector.

### 2.3.2 What is a Garbage Collected Heap?

To implement and reason about a garbage collector, we must make some assumptions about heaps, and the assumptions will be part of our logic. For instance, if a heap contains dangling reachable pointers, then our garbage collector will malfunction, and hence we must make assumptions that prevent this.

In what follows, we therefore define the concept of a “garbage collected heap”, and what it means for a heap to be a garbage collected version of another. First, we give a more precise definition of our requirement about aligned **cons**-cells. Let  $S$  be a finite set of pointers, let  $f_1, f_2 : Ptr \rightarrow \mathbb{Z}$  be finite partial functions from pointers to integers, and let  $h$  be a heap. We then define the semantic condition *pairheap* by

$$\begin{aligned}
 \text{pairheap}(S, f_1, f_2, h) \quad \text{iff} \quad & \text{dom}(f_1) = \text{dom}(f_2) = S \\
 & \text{and } \text{dom}(h) = \{p + k \mid p \in S \text{ and } k \in \{0, 4\}\} \\
 & \text{and } \forall p \in S. h(p) = f_1(p) \text{ and } h(p + 4) = f_2(p).
 \end{aligned}$$

Note that if *pairheap*( $S, f_1, f_2, h$ ) holds, then  $h$  is determined by  $f_1, f_2$ , and  $S$  (and  $S, f_1, f_2$  are determined by  $h$ ).

The concept of garbage collection depends on a given *root set*. Here, we assume for simplicity that there is only one root cell. Thus, given a heap  $h$  and a pointer  $r$ , we call

$(r, h)$  a *rooted heap*. Given such a rooted heap  $(r, h)$ , we formally define the sets  $rp(r, h)$  and  $rl(r, h)$  of pointers and locations that are *reachable* from  $r$  in  $h$  as follows:

$$\begin{aligned} rp_0(r, h) &\stackrel{\text{def}}{=} \{r\} \\ rp_{n+1}(r, h) &\stackrel{\text{def}}{=} \{h(\ell) \mid \ell \in rl_n(r, h) \cap \text{dom}(h) \text{ and } h(\ell) \in \text{Ptr}\} \\ rl_n(r, h) &\stackrel{\text{def}}{=} \{p + k \mid p \in rp_n(r, h) \text{ and } k \in \{0, 4\}\} \\ rp(r, h) &\stackrel{\text{def}}{=} \bigcup_{n=0}^{\infty} rp_n(r, h) \\ rl(r, h) &\stackrel{\text{def}}{=} \bigcup_{n=0}^{\infty} rl_n(r, h). \end{aligned}$$

With these definitions, we can define the condition that “no reachable location dangles”:

$$rl(r, h) \subseteq \text{dom}(h),$$

and that “all locations are reachable”:

$$rl(r, h) \supseteq \text{dom}(h).$$

If both these conditions hold, *i.e.*, if  $rl(r, h) = \text{dom}(h)$ , we say that the rooted heap  $(r, h)$  is *exactly reachable*. The reachable pointers and locations satisfy a monotonicity property:

$$h_0 \subseteq h \text{ implies } rl(r, h_0) \subseteq rl(r, h). \quad (2.2)$$

Furthermore, once there are no dangling reachable locations in a heap, further enlargements of the heap does not increase reachability:

$$h_0 \subseteq h \text{ and } rl(r, h_0) \subseteq \text{dom}(h_0) \text{ implies } rl(r, h_0) = rl(r, h). \quad (2.3)$$

It is not hard to see that if a rooted heap  $(r, h)$  has no dangling reachable locations, then  $h$  contains a unique subheap  $h_0$  such that  $(r, h_0)$  is exactly reachable, and otherwise,  $h$  does not contain a subheap  $h_0$  such that  $(r, h_0)$  is exactly reachable. The subheap  $(r, h_0)$  plays a role similar to that of  $\text{prune}(h)$  from the article [33], but it is only well-behaved if there are no reachable dangling locations in  $(r, h)$ .

For any heap  $h$ , we define

$$\text{pdom}(h) \stackrel{\text{def}}{=} \text{dom}(h) \cap \text{Ptr}.$$

Now, assume  $(r, h)$  is exactly reachable. Then for all pointers  $p$ ,

$$p \in \text{pdom}(h) \text{ iff } p \in \text{dom}(h) \text{ iff } p + 4 \in \text{dom}(h), \quad (2.4)$$

and for all locations  $\ell$ ,

$$\ell \in \text{dom}(h) \text{ and } h(\ell) \in \text{Ptr} \text{ implies } h(\ell) \in \text{pdom}(h), \quad (2.5)$$

and

$$r \in \text{pdom}(h). \quad (2.6)$$

Conversely, if the conditions (2.4) to (2.6) hold for a rooted heap  $(r, h)$ , then one can show

$$rp(r, h) \subseteq \text{pdom}(h) \quad \text{and} \quad rl(r, h) \subseteq \text{dom}(h)$$

by induction on  $n$  to the corresponding formulas where  $rp$  and  $rl$  are subscripted with  $n$ .

The conditions (2.4) to (2.6) are just what is needed to define a *heap morphism* from  $(r, h)$  to an arbitrary rooted heap. Henceforth, assume  $(r, h)$  satisfies (2.4) to (2.6),  $(r', h')$  is an arbitrary rooted heap, and that  $\beta$  is a function from  $\text{pdom}(h)$  to  $\text{pdom}(h')$  such that for all  $p \in \text{pdom}(h)$  and  $k \in \{0, 4\}$ ,

$$h'((\beta(p)) + k) = \beta^*(h(p + k)) \quad (2.7)$$

and

$$r' = \beta(r). \quad (2.8)$$

Here,  $\beta^*$  is the extension of  $\beta$  to  $\mathbb{Z}$  that is the identity on numbers that are not in  $\text{pdom}(h)$ . Then  $\beta$  is called a *heap morphism* from  $(r, h)$  to  $(r', h')$ .

**Lemma 2.1.** *Assume  $(r, h)$  satisfies (2.4) to (2.6),  $(r', h')$  is an arbitrary rooted heap, and  $\beta$  is a heap morphism from  $(r, h)$  to  $(r', h')$ . Then,*

- *for all pointers  $p$ , if  $p \in rp(r, h)$ , then  $\beta(p) \in rp(r', h')$ .*
- *If  $\beta$  is an isomorphism of functions, then for all pointers  $p'$ , if  $p' \in rp(r', h')$ , then  $p' \in \text{pdom}(h')$  and  $\beta^{-1}(p') \in rp(r, h)$ .*
- *If  $\beta$  is an isomorphism of functions and  $(r, h)$  is exactly reachable, then  $(r', h')$  is exactly reachable.*

*Proof.* For the first claim, we prove the slightly stronger statement that for all pointers  $p$ , if  $p \in rp_n(r, h)$ , then  $\beta(p) \in rp_n(r', h')$ . This is done by induction on  $n$ . If  $p \in rp_0(r, h)$ , then  $p = r$  so that  $\beta(p) = \beta(r) = r' \in rp_0(r', h')$ . For the induction step, if  $p \in rp_{n+1}(r, h)$ , there is a pointer  $q \in rp_n(r, h)$  and a  $k \in \{0, 4\}$  with  $p = h(q + k)$ . Moreover,  $p, q \in \text{dom}(h)$ , and by induction,  $\beta(q) \in rp_n(r', h')$ . This implies

$$\beta(p) = \beta(h(q + k)) = \beta^*(h(q + k)) = h'(\beta(q) + k),$$

and thus,  $\beta(p) \in rp_{n+1}(r', h')$ .

The second claim is also proved by induction on  $n$ . More precisely, we show that if  $\beta$  is an isomorphism of functions (*i.e.*, it is a bijection between  $\text{pdom}(h)$  and  $\text{pdom}(h')$ ), then for all natural numbers  $n$  and all pointers  $p'$ , if  $p' \in rp_n(r', h')$ , then  $p' \in \text{pdom}(h')$  and  $\beta^{-1}(p') \in rp_n(r, h)$ . The base case is obvious, and for the induction step, suppose  $p' \in rp_{n+1}(r', h')$ . Then there is a pointer  $q' \in rp_n(r', h')$  and a  $k \in \{0, 4\}$  with  $p' = h'(q' + k)$ . For this  $q'$ , the induction hypothesis yields  $q' \in \text{pdom}(h')$  and  $\beta^{-1}(q') \in rp_n(r, h)$ , and thus

$$p' = h'(q' + k) = h'((\beta(\beta^{-1}(q')) + k)) = \beta^*(h(\beta^{-1}(q') + k)) = \beta(h(\beta^{-1}(q') + k)),$$

where the last step holds since  $p'$  is a pointer. This means that  $p' \in \text{pdom}(h')$ , and

$$\beta^{-1}(p') = \beta^{-1}(\beta(h(\beta^{-1}(q') + k))) = h(\beta^{-1}(q') + k).$$

Now, since  $\beta^{-1}(q') \in rp_n(r, h)$ ,  $\beta^{-1}(p') \in rp_{n+1}(r, h)$ .

Finally, also suppose that  $(r, h)$  is exactly reachable. To show that  $(r', h')$  is exactly reachable, let  $\ell' \in rl(r', h')$ . By definition, there are a pointer  $p' \in rp(r', h')$  and a  $k \in \{0, 4\}$  with  $\ell' = p' + k$ . By the previous proof,  $p' \in \text{pdom}(h')$  and  $\ell' \in \text{dom}(h')$ . Conversely, if  $\ell' \in \text{dom}(h')$ , then  $\ell' = p' + k$  for a  $k \in \{0, 4\}$  and a  $p' \in \text{pdom}(h')$ . Since  $\beta$  is a bijection,  $\beta^{-1}(p') \in \text{pdom}(h)$ , and thus  $\beta^{-1}(p') \in rp(r, h)$ , since  $(r, h)$  is exactly reachable. We can now use the proof of the first item to conclude the desired result.  $\square$

The Lemma justifies the following semantic notion of one heap being a garbage collected version of another. This, of course, is central when verifying correctness of a garbage collector.

**Definition 2.2.** Let  $(r, h), (r', h')$  be rooted heaps, and suppose  $h_0 \subseteq h$  is such that  $(r, h_0)$  is exactly reachable. If there is a heap morphism  $\beta$  from  $(r, h_0)$  to  $(r', h'_0)$  for some subheap  $h'_0 \subseteq h'$ , then  $(r', h')$  is called a *garbage collected version* of  $(r, h)$ .

This notion of garbage collection, of course, depends on the fact that we collect **cons** cells only, and it does not imply that  $h$  and  $h'$  need have the same size. Indeed, a memory management system may shrink the heap after garbage collection, to diminish the heap usage of a program, or it may allocate more space for a program, to reduce the required number of rounds of garbage collection.

With this notion of garbage collection, we also note that the identity is an example of a heap morphism, and the trivial program **skip** is thus an example of a correct garbage collector. However, our interest lies with non-trivial garbage collectors.

Our notion of garbage collection is also quite conservative. Indeed, it has been noted in the literature [3, 76] that if the programming language that uses the garbage collector has a sufficiently rich type system, there can be a significant difference between the data that is reachable and data that can be reclaimed without affecting the program. This is because some objects in the heap might be reachable, but we might be able to infer information about reachable objects in the heap at run-time, (e.g. via type reconstruction [2]) which implies that these objects are not necessary for execution of the rest of the program. In this case we say that these objects are reachable, but not *live*, and it would, of course, give better memory usage to collect objects that are “dead” in this sense. Using region inference [127, 125] it is also possible to statically infer that some data is dead, although it is reachable, and combinations of region inference and garbage collection exist in current run-time systems for ML [126, 47]. It might even be possible to employ such techniques as hash-consing [43, 7] or other dynamic updates that affect data representation [122]. However, these possibilities are not relevant to the goal of this paper, which is a specification and proof of the Cheney algorithm in separation logic.

### 2.3.3 Expressions

We define the syntax and semantics for expressions of each of the types `int`, `fs`, `frp`, and `fri`. For expressions of type `int` we just present the syntax; the semantics is straightforward. For expressions of the remaining types, we just present the semantics as the syntax will be evident from the presentation of the semantics. In general, the semantics for an expression  $E$  (of each of the types just mentioned) is formally a map from stack  $s$  with  $FV(E) \subseteq \text{dom}(s)$  to values of suitable kinds, *i.e.*, the semantics of an expression depends on a stack.

Expressions of type `int` are defined by the following grammar:

$$e ::= n \mid x^{\text{int}} \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 \times e_2 \mid e \bmod j \\ \mid e_1 \leq e_2 \mid e_1 = e_2 \mid \neg e \mid e_1 \wedge e_2 \mid \#m^{\text{fs}},$$

where  $n \in \mathbb{Z}$  and  $j \in \mathbb{N} \setminus \{0\}$ . The semantics of  $\#m^{\text{fs}}$  is the number of elements in the finite set of pointers denoted by  $m^{\text{fs}}$  (see below). In order to avoid introducing an explicit type of boolean values, we use a standard encoding of truth values, where 0 denotes “false”, and all other integers denote “true”. Although the superscript that indicates the

type is only meant to indicate the type of variables, we sometimes use a superscript to indicate the type of composite expressions. Most frequently, however, we will omit the superscripts, even on variables, if it causes no confusion.

We use  $m$  to range over expressions of type  $fs$ . The semantics of an expression of type  $fs$  is a set of pointers. For instance, the expression  $Itv(e_1, e_2)$  denotes the set of pointers in the half-open interval from  $e_1$  to  $e_2$ . In general, the semantics of an expression  $m$  of type  $fs$  is a map from stacks  $s$  with  $FV(m) \subseteq \text{dom}(s)$  to the set  $FS$  of finite sets of pointers, and it is given by

$$\begin{aligned} \llbracket \emptyset^{fs} \rrbracket s &= \emptyset \\ \llbracket x^{fs} \rrbracket s &= s(x) \\ \llbracket \{e\} \rrbracket s &= \{\llbracket e \rrbracket s\} \cap Ptr \\ \llbracket Itv(e_1^{int}, e_2^{int}) \rrbracket s &= \{p \in Ptr \mid \llbracket e_1 \rrbracket s \leq p \wedge p < \llbracket e_2 \rrbracket s\} \\ \llbracket m_1^{fs} \cup m_2^{fs} \rrbracket s &= \llbracket m_1 \rrbracket s \cup \llbracket m_2 \rrbracket s \\ \llbracket m_1^{fs} \setminus m_2^{fs} \rrbracket s &= \llbracket m_1 \rrbracket s \setminus \llbracket m_2 \rrbracket s \end{aligned}$$

We can now formally define the semantics of expressions of form  $\#m^{fs}$ :

$$\begin{aligned} \llbracket \#m^{fs} \rrbracket s &= k, \text{ where } \llbracket m \rrbracket s = \{p_1, \dots, p_k\} \\ &\text{(note that } k \text{ may be 0).} \end{aligned}$$

We use  $r$  to range over expressions of type  $frp$ . The semantics of an expression  $r$  of type  $frp$  is a map from stacks  $s$  with  $FV(r) \subseteq \text{dom}(s)$  to the set  $FRP$  of finite relations on pointers, and it is given by the following clauses.

$$\begin{aligned} \llbracket \emptyset^{frp} \rrbracket s &= \emptyset \\ \llbracket x^{frp} \rrbracket s &= s(x) \\ \llbracket r^\dagger \rrbracket s &= \{(p', p) \mid (p, p') \in \llbracket r \rrbracket s\} \\ \llbracket r_1^{frp} \circ r_2^{frp} \rrbracket s &= \{(p, p'') \mid \exists p'. (p, p') \in \llbracket r_2 \rrbracket s \wedge (p', p'') \in \llbracket r_1 \rrbracket s\} \\ \llbracket r^{frp} \cup \{(e_1^{int}, e_2^{int})\} \rrbracket s &= \llbracket r \rrbracket s \cup (\{\llbracket e_1 \rrbracket s, \llbracket e_2 \rrbracket s\} \cap Ptr \times Ptr) \\ \llbracket r^{frp} \setminus \{(e_1^{int}, e_2^{int})\} \rrbracket s &= \llbracket r \rrbracket s \setminus (\{\llbracket e_1 \rrbracket s, \llbracket e_2 \rrbracket s\}) \end{aligned}$$

Note that we use  $r^\dagger$  for the inverse of the relation  $r$ .

To conclude our semantics for expressions, we give the semantics for expressions of type  $fri$ . We use  $\rho$  to range over such expressions. The  $\odot$  operator will be used to model the structure-preserving property of a garbage collector, inasmuch as it extends a relation with the identity on non-pointers before composing it with another relation (cf. the definition of  $\beta^*$  from Sec. 2.3.2). Hence, the semantics of an expression  $\rho$  of type  $fri$  is a map from stacks  $s$  with  $FV(\rho) \subseteq \text{dom}(s)$  to the set  $FRi$  relations between pointers and integers.

$$\begin{aligned} \llbracket x^{fri} \rrbracket s &= s(x) \\ \llbracket \rho^{fri} \circ r^{frp} \rrbracket s &= \{(p, n) \mid \exists p' \in Ptr. (p, p') \in \llbracket r \rrbracket s \wedge (p', n) \in \llbracket \rho \rrbracket s\} \\ \llbracket r^{frp} \odot \rho^{fri} \rrbracket s &= \{(p, n) \mid ((p, n) \in \llbracket \rho \rrbracket s \wedge n \notin Ptr) \vee \\ &\quad (\exists p' \in Ptr. (p, p') \in \llbracket \rho \rrbracket s \wedge (p', n) \in \llbracket r \rrbracket s)\} \end{aligned}$$

Note the connection between the  $(-)^*$ -construct and the semantics of the special relation composition  $\odot$ : if  $\varphi$  and  $\psi$  are functions denoting the relations  $r$  and  $\rho$ , respectively, then  $\varphi^* \circ \psi$  is the denotation of  $r \odot \rho$ .

Substitution is defined in a standard way; there are no binders in the expression language. The following substitution lemma is easily proved by induction on expressions.

**Lemma 2.3.** *Let  $s$  be a stack, and let  $\delta, \delta' \in \text{Types}$ . Then, for all expressions  $e^\delta, e'^{\delta'}$  of type  $\delta$  and  $\delta'$  respectively, and for all variables  $x^{\delta'}$  of type  $\delta'$ ,*

$$\llbracket [e'/x] \rrbracket s = \llbracket [e] \rrbracket (s[x \mapsto \llbracket [e'] \rrbracket s]),$$

where  $s[x \mapsto v]$  is the function that is like  $s$ , but with  $x$  mapped to  $v$ .

We use  $\equiv$  to denote syntactic equality between expressions, and we sometimes write  $e_1 = e_2$  to denote that  $\llbracket [e_1] \rrbracket s = \llbracket [e_2] \rrbracket s$ , for all stacks  $s$  with  $\text{FV}(e_1) \cup \text{FV}(e_2) \subseteq \text{dom}(s)$ .

### 2.3.4 Assertions

Our assertion language is a variant of that of separation logic [115] with assertion forms for finite sets and relations. We first present the syntax of assertions and give informal explanations of the most interesting assertion forms. Then we give the formal semantics and present some useful inference rules.

We use  $A, B$ , and  $D$  to range over assertions, which are generated by the following grammar:

$A, B, D ::=$	$e_1 \leq e_2$	$e_1 = e_2$
	$\neg e$	$\top$
	$\mathbf{F}$	$\neg A$
	$A \rightarrow B$	$A \wedge B$
	$A \vee B$	$\forall x^\delta. A$
	$\exists x^\delta. A$	
	$\text{emp}$	$e_1 \mapsto e_2$
	$A * B$	$A \multimap B$
	$\forall_* x^{\text{int}} \in m. A$	
	$e \in m$	$m_1 \perp m_2$
	$m_1 = m_2$	$m_1 \subseteq m_2$
	$(e_1, e_2) \in \rho$	$(e_1, e_2) \in r$
	$\text{Ptr}(e)$	$\text{PtrRg}(\rho, m)$
	$\text{Tfun}(r, m)$	$\text{Tfun}(\rho, m)$
	$\text{iso}(r, m_1, m_2)$	
	$\text{Reachable}(\rho_1, \rho_2, m, e),$	

where  $\delta$  ranges over Types. We have grouped the assertion forms by horizontal lines. The assertion forms above the first horizontal line are the usual assertion forms from classical logic with equality.

The second group are assertions that describe the heap. Apart from the iterated separating conjunction, these are all part of standard separation logic. The assertion  $\text{emp}$  states that the heap is empty, and  $e_1 \mapsto e_2$  states that there is precisely one location in the domain of the heap.  $A * B$  means that  $A$  and  $B$  hold in *disjoint* subheaps of the current heap, and  $A \multimap B$  means that for all heaps  $h'$  that are disjoint from the current heap  $h$  and in which  $A$  hold, the combination of the extension and the current heap will satisfy  $B$ . Finally,  $\forall_*$  is an *iterated separating conjunction*. Informally, if  $s, h \Vdash \forall_* x \in m. A$ , and if  $\llbracket [m] \rrbracket s = \{p_1, \dots, p_k\}$ , then  $h$  can be split into disjoint heaps  $h = h_1 \cdot \dots \cdot h_k$  with  $s, h_i \Vdash A[p_i/x]$ . This assertion form plays a crucial role in our specification of our garbage collector, as can be seen in Sections 2.5.2 and 2.5.3. Also note that the sets involved in the assertion form might change throughout execution of a program; thus this assertion form describes a “dynamic” heap. An example of this can be found in Section 2.6.1.

The next group contains assertion forms that are related to our extension of standard separation logic. They are assertions about sets and relations, and most of them are self-explanatory. The assertion  $\text{PtrRg}(\rho, m)$  says that any pointer which is a second component in any pair in the relation denoted by  $\rho$  is in the set denoted by  $m$ ;  $\text{Tfun}$  says that a relation is a total function on a set, and  $\text{iso}$  says that a relation is a bijection between two sets. Finally, the assertion  $\text{Reachable}$  says that an exactly reachable heap is described by two (functional) relations on a set, and a “root pointer”.

The set  $\text{FV}(A)$  of free variables for an assertion  $A$  is defined as usual. Note that  $x$  (and not  $m$ ) is bound in  $\forall_* x \in m. A$ . Substitution  $A[e/x]$  of the expression  $e$  for the variable  $x$  in the assertion  $A$  is defined in the standard way. We sometimes write  $A(x)$  to indicate that the variable  $x$  may occur free in  $A$ .

The semantics for propositions is formally given by a judgment of the form

$$(s, h) \Vdash A,$$

the intended meaning of which is that the proposition  $A$  holds in the state  $(s, h)$ . We require  $\text{FV}(A) \subseteq \text{dom}(s)$ . Before the definition of this judgment, we need to introduce a partial commutative monoid structure on the set of heaps: Write  $h_1 \# h_2$  to indicate  $\text{dom}(h_1) \cap \text{dom}(h_2) = \emptyset$  (we call such heaps *disjoint*), and if  $h_1 \# h_2$ , we define the combined heap  $h_1 \cdot h_2$  by

$$n \mapsto \begin{cases} h_1(n) & \text{if } n \in \text{dom}(h_1) \\ h_2(n) & \text{if } n \in \text{dom}(h_2) \end{cases}.$$

The semantics is given in Fig. 2.1. We let  $b$  range over the boolean expressions  $e_1 \leq e_2, e_1 = e_2, \neg e$ , and  $\delta$  ranges over Types.

Note that the semantics is classical for the standard first-order logic fragment, and that the parameters for the assertion form  $\text{Reachable}$  determine the heap that is required to exist unambiguously.

As for expressions, we have the expected substitution lemma for assertions.

**Lemma 2.4.** *Let  $e$  be an expression of type  $\delta$ , let  $x$  be a variable of type  $\delta$ , and let  $A$  be an assertion. Then, for all states  $s, h$ ,*

$$(s, h) \Vdash A[e/x] \quad \text{iff} \quad (s[x \mapsto \llbracket e \rrbracket s], h) \Vdash A$$

**Definition 2.5.** We call an assertion  $A$  *valid* if, for all states  $(s, h)$  with  $\text{FV}(A) \subseteq \text{dom}(s)$ ,  $(s, h) \Vdash A$ . We use  $\Rightarrow$  to denote semantic validity, i.e.,  $A \Rightarrow B$  if  $(s, h) \Vdash A$  implies  $(s, h) \Vdash B$ , for all states  $(s, h)$ .

The following shorthand notations are standard in separation logic, and we shall also use them in this paper.

$$\begin{aligned} e \mapsto e_1, e_2 &\stackrel{\text{def}}{=} (e \mapsto e_1) * (e + 4 \mapsto e_2) \\ e_1 \hookrightarrow e_2 &\stackrel{\text{def}}{=} e_1 \mapsto e_2 * \top \\ e \hookleftarrow e_1, e_2 &\stackrel{\text{def}}{=} e \mapsto e_1, e_2 * \top \\ e \mapsto - &\stackrel{\text{def}}{=} \exists x^{\text{int}}. e \mapsto x \\ e \mapsto -, - &\stackrel{\text{def}}{=} \exists x^{\text{int}}, y^{\text{int}}. e \mapsto x, y \end{aligned}$$

These notations make sense for all locations, but we shall only use them when  $e$  denotes a pointer. We also write  $e_1 \neq e_2$  for  $\neg(e_1 = e_2)$ .

There are certain special classes of assertions [134],[115], which we will use later. These are defined here.

$A$	$(s, h) \Vdash A$ if and only if
$b$	$\llbracket b \rrbracket s \neq 0$
$\top$	$\top$
$\text{F}$	$\text{F}$
$\neg A$	$s, h \not\Vdash A$
$A \rightarrow B$	$s, h \Vdash A$ implies $s, h \Vdash B$
$A \wedge B$	$s, h \Vdash A$ and $s, h \Vdash B$
$A \vee B$	$s, h \Vdash A$ or $s, h \Vdash B$
$\forall x^\delta. A$	for all $v \in \llbracket \delta \rrbracket, s[x \mapsto v], h \Vdash A$
$\exists x^\delta. A$	for some $v \in \llbracket \delta \rrbracket, s[x \mapsto v], h \Vdash A$
$\text{emp}$	$\text{dom}(h) = \emptyset$
$e_1 \mapsto e_2$	$h = \{(\llbracket e_1 \rrbracket s, \llbracket e_2 \rrbracket s)\}$
$A * B$	there are heaps $h_1, h_2$ such that $h_1 \# h_2$ , $h_1 \cdot h_2 = h, s, h_1 \Vdash A$ , and $s, h_2 \Vdash B$
$A \multimap B$	$s, h \cdot h' \Vdash B$ for all $h'$ with $h \# h'$ and $s, h' \Vdash A$
$\forall_* p \in m. A$	$\begin{cases} s, h \Vdash A[p_1/p] * \dots * A[p_k/p] & \text{if } \llbracket m \rrbracket s = \{p_1, \dots, p_k\} \\ s, h \Vdash \text{emp} & \text{if } \llbracket m \rrbracket s = \emptyset \end{cases}$
$e \in m$	$\llbracket e \rrbracket s \in \llbracket m \rrbracket s$
$m \perp m'$	$\llbracket m \rrbracket s \cap \llbracket m' \rrbracket s = \emptyset$
$m_1 = m_2$	$\llbracket m_1 \rrbracket s = \llbracket m_2 \rrbracket s$
$m_1 \subseteq m_2$	$\llbracket m_1 \rrbracket s \subseteq \llbracket m_2 \rrbracket s$
$(e_1, e_2) \in r$	$(\llbracket e_1 \rrbracket s, \llbracket e_2 \rrbracket s) \in \llbracket r \rrbracket s$
$(e_1, e_2) \in \rho$	$(\llbracket e_1 \rrbracket s, \llbracket e_2 \rrbracket s) \in \llbracket \rho \rrbracket s$
$\text{Ptr}(e)$	$\llbracket e \rrbracket s \in \text{Ptr}$
$\text{PtrRg}(\rho, m)$	$\forall (p, q) \in \llbracket \rho \rrbracket s. q \in \text{Ptr} \Rightarrow q \in \llbracket m \rrbracket s$
$\text{Tfun}(r, m)$	$\forall p \in \llbracket m \rrbracket s. \exists! n \in \mathbb{Z}. (p, n) \in \llbracket r \rrbracket s$
$\text{Tfun}(\rho, m)$	$\forall p \in \llbracket m \rrbracket s. \exists! n \in \mathbb{Z}. (p, n) \in \llbracket \rho \rrbracket s$
$\text{iso}(r, m_1, m_2)$	$\forall p_1 \in M_1. \exists! p_2 \in M_2. (p_1, p_2) \in \varphi \wedge$ $\forall p_2 \in M_2. \exists! p_1 \in M_1. (p_1, p_2) \in \varphi \wedge$ $\forall (p_1, p_2) \in \varphi. p_1 \in M_1 \wedge p_2 \in M_2,$ where $M_1 = \llbracket m_1 \rrbracket s, M_2 = \llbracket m_2 \rrbracket s, \varphi = \llbracket r \rrbracket s$
$\text{Reachable}(\rho, \rho', m, e)$	$s, h \Vdash \text{Tfun}(\rho, m) \wedge \text{Tfun}(\rho', m)$ and $\exists h'. (\llbracket e \rrbracket s, h')$ is exactly reachable, and $\text{pairheap}(\llbracket m \rrbracket s, \llbracket \rho \rrbracket s, \llbracket \rho' \rrbracket s, h')$

Figure 2.1: Semantics of Assertions

**Definition 2.6.**

- An assertion  $A$  is called *pure* if its validity does not depend on the heap, i.e., if  $(s, h) \Vdash A$  if and only if  $(s, h') \Vdash A$ , for all stacks  $s$  and heaps  $h, h'$ .
- We call an assertion  $A$  *intuitionistic* if, for all stacks  $s$  and heaps  $h, h'$ ,

$$(s, h) \Vdash A \text{ and } h \subseteq h' \text{ imply } (s, h') \Vdash A.$$

Here,  $\subseteq$  is just set-theoretic inclusion of graphs.

**Remark 2.7.** Special rules apply for certain of the newly introduced classes of assertions. Here are some of these rules.

- When  $A$  is a pure assertion, the rule

$$A \wedge (B * C) \leftrightarrow (A \wedge B) * C \quad (2.9)$$

is valid for any assertions  $B$  and  $C$ .

- Pure assertions are intuitionistic.
- Syntactically, an assertion is pure, if it does not contain any occurrences of  $\text{emp}$ ,  $\forall_*$ ,  $\mapsto$ , or the shorthand notation  $\leftrightarrow$ .

**2.3.5 Some Useful Rules**

A number of axiom schemata are used in proofs later. We believe they could be part of a small theory of finite sets and relations that could be used in proofs of other programs where the goal is to establish an isomorphism between data structures before and after execution. The schemata most relevant to separation logic are listed below, and more can be found in the survey paper [115]. Beside these, there are a number of obvious rules regarding sets and relations which can easily be verified semantically. For completeness, these are listed in Appendix 2.B.

Rules for  $\forall_*$ :

$$(\forall_* x \in m. A) \wedge m = m' \rightarrow \forall_* x \in m'. A \quad (2.10)$$

$$m = \emptyset \rightarrow ((\forall_* x \in m. A) \leftrightarrow \text{emp}) \quad (2.11)$$

$$\begin{aligned} (\forall_* x \in m. x \mapsto - \wedge A) \wedge y \in m &\rightarrow \\ (\forall_* x \in m. x \mapsto - \wedge A) \wedge (y \leftrightarrow -) &\quad (2.12) \end{aligned}$$

When  $m$  and  $m'$  are disjoint,

$$(\forall_* x \in m. A) * (\forall_* x \in m'. A) \leftrightarrow (\forall_* x \in m \cup m'. A) \quad (2.13)$$

As a special case, we get

$$\begin{aligned} e \in m &\rightarrow \\ ((\forall_* x \in m. A) \leftrightarrow ((\forall_* x \in (m \setminus \{e\}). A) * A[e/x])) &\quad (2.14) \end{aligned}$$

General / Structural rules: When  $x \notin \text{FV}(e)$ ,

$$\begin{aligned} (e \leftrightarrow e') \wedge ((\exists x. e \mapsto x \wedge A) * B) &\leftrightarrow \\ (e \mapsto e' \wedge A[e'/x]) * B &\quad (2.15) \end{aligned}$$

If  $B$  is pure and  $B'$  is intuitionistic,

$$\frac{A \wedge B \rightarrow B'}{(A * A') \wedge B \rightarrow B'}. \quad (2.16)$$

We also use the commutativity and associativity of the separating conjunction (the same rules for the traditional conjunction are also used, but we refrain from stating them here):

$$A * B \leftrightarrow B * A \quad \text{and} \quad A * (B * C) \leftrightarrow (A * B) * C \quad (2.17)$$

**Theorem 2.8.** *The rules (2.10) - (2.17), and the rules (2.B1) - (2.B50) from Appendix 2.B are all valid.*

We shall use the commutativity and associativity rules implicitly in proofs, and we shall often just say “by purity”, when we apply the distributive law (2.9) for pure assertions from Remark 2.7.

The following lemma is useful when reasoning about assertions involving  $\forall_*$ .

**Lemma 2.9.** *Suppose  $A, B, s$ , and  $h$  are such that  $(s, h) \Vdash \forall_* x \in m. A$ , and  $\forall x'. x' \in m \wedge A[x'/x] \rightarrow B[x'/x]$  is valid. Then  $(s, h) \Vdash \forall_* x \in m. B$ .*

*Proof.* We do a case analysis on the cardinality of  $\llbracket m \rrbracket s$ . If  $\llbracket m \rrbracket s = \emptyset$ , then

$$(s, h) \Vdash \forall_* x \in m. A \iff (s, h) \Vdash \text{emp} \iff (s, h) \Vdash \forall_* x \in m. B$$

If  $\llbracket m \rrbracket s = \{p_1, \dots, p_k\}$ , we have

$$(s, h) \Vdash A[p_1/x] * \dots * A[p_k/x],$$

so there is a partition  $h = h_1 \dots h_k$  such that for  $i = 1 \dots k$ ,  $s, h_i \Vdash A[p_i/x]$ . Since we have  $s, h_i \Vdash A[p_i/x] \wedge p_i \in m$ , we get  $(s, h_i) \Vdash B[p_i/x]$ , and this means that

$$(s, h) \Vdash \forall_* x \in m. B,$$

as desired.  $\square$

This shows that we can exploit the information about  $x' \in m$  to do “implication under  $\forall_*$ ”. Another useful rule comes from the following lemma for which we omit the proof.

**Lemma 2.10.** *If  $D$  is a pure assertion, and if  $D \wedge A \rightarrow A'$  and  $D \wedge B \rightarrow B'$  are valid, then  $D \wedge (A * B) \rightarrow D \wedge (A' * B')$  is valid.*

By induction, this means that  $D \wedge (A_1 * \dots * A_k) \rightarrow (A'_1 * \dots * A'_k)$  can be inferred from  $D \wedge A_1 \rightarrow A'_1$ , and  $\dots$ , and  $D \wedge A_k \rightarrow A'_k$ .

Note that Lemma 2.10 is not valid if  $D$  is not pure — as a counterexample, set  $A$  to  $4 \mapsto 2$ ,  $B$  to  $8 \mapsto 3$ ,  $A'$  to  $4 \mapsto 10$ ,  $B'$  to  $8 \mapsto 17$ , and  $D$  to  $(4 \mapsto 2 * 8 \mapsto 3)$ .

As an example of a rule that can be derived from the rules above, we get the following from (2.15) and purity. Here,  $A$  must be a pure assertion; otherwise the second step below is not valid.

$$\begin{aligned} & (e \leftrightarrow e') \wedge ((\exists x. (e \mapsto x \wedge A)) * B) \\ & \Downarrow \\ & (e \mapsto e' \wedge A[e'/x]) * B \\ & \Downarrow \\ & ((e \mapsto e') * B) \wedge A[e'/x] \\ & \Downarrow \\ & A[e'/x] \end{aligned} \quad (2.18)$$

In addition to the rules above, we have the standard rules of classical logic. We will sometimes implicitly substitute equals for equals; for example, we will for example infer  $e \in m_2$  from  $e \in m_1 \wedge m_1 = m_2$ . Also, the most basic arithmetic will be performed implicitly, so we will for example infer  $x - 8 \leq y$  from  $x \leq y$ .

## 2.4 Programming Language

In this section we first define the syntax and semantics of the programming language used for implementing our garbage collector. Next, we use the assertion language from Sec. 2.3.4 to give a program logic in the style of Hoare for the language. This enables us to specify and prove correct the collector in Sec. 2.5 and 2.6.

### 2.4.1 Syntax and Semantics

**Definition 2.11.** The syntax of the implementation language is given by the following grammar:

$$C ::= \mathbf{skip} \mid x^{\text{int}} := e \mid x^{\text{fs}} := m \mid x^{\text{frp}} := r \mid x^{\text{int}} := [e] \mid [e] := e \\ \mid C; C \mid \mathbf{while} \ e \ \mathbf{do} \ C \ \mathbf{od} \mid \mathbf{if} \ e \ \mathbf{then} \ C \ \mathbf{else} \ C \ \mathbf{fi}$$

Note that there are no constructs for allocating or deallocating locations on the heap. In our implementation and specification of Cheney's algorithm we simply assume that the domain of the heap contains the necessary locations. It would be straightforward to add these features to the language, implement a version of the algorithm that allocates the necessary space before collection and disposes garbage afterwards, and modify our proof of its correctness. We discuss this further in Section 2.5.

The operational semantics is given by a (small-step) relation  $\rightsquigarrow$  on *configurations* (ranged over by  $K$ ). Configurations are either of the form  $s, h$  (these are called *terminal*) or of the form  $C, s, h$  (these are called *non-terminal*).

**Definition 2.12.** The relation  $\rightsquigarrow$  on configurations is defined by the following inference rules:

$$\frac{}{\mathbf{skip}, s, h \rightsquigarrow s, h} \quad \frac{[[e]]s = n}{x^{\text{int}} := e, s, h \rightsquigarrow s[x \mapsto n], h} \\ \frac{[[m]]s = M}{x^{\text{fs}} := m, s, h \rightsquigarrow s[x \mapsto M], h} \quad \frac{[[r]]s = \varphi}{x^{\text{frp}} := r, s, h \rightsquigarrow s[x \mapsto \varphi], h} \\ \frac{[[e]]s = p \quad p \in \text{dom}(h) \quad h(p) = n}{x^{\text{int}} := [e], s, h \rightsquigarrow s[x \mapsto n], h} \quad \frac{[[e_1]]s = p \quad [[e_2]]s = n \quad p \in \text{dom}(h)}{[e_1] := e_2, s, h \rightsquigarrow s, h[p \mapsto n]} \\ \frac{C_1, s, h \rightsquigarrow C', s', h'}{C_1; C_2, s, h \rightsquigarrow C'; C_2, s', h'} \quad \frac{C_1, s, h \rightsquigarrow s', h'}{C_1; C_2, s, h \rightsquigarrow C_2, s', h'} \\ \frac{[[e]]s = 0}{\mathbf{while} \ e \ \mathbf{do} \ C \ \mathbf{od}, s, h \rightsquigarrow s, h} \quad \frac{[[e]]s \neq 0 \quad C; \mathbf{while} \ e \ \mathbf{do} \ C \ \mathbf{od}, s, h \rightsquigarrow K}{\mathbf{while} \ e \ \mathbf{do} \ C \ \mathbf{od}, s, h \rightsquigarrow K} \\ \frac{[[e]]s = 0 \quad C_2, s, h \rightsquigarrow K}{\mathbf{if} \ b \ \mathbf{then} \ C_1 \ \mathbf{else} \ C_2 \ \mathbf{fi}, s, h \rightsquigarrow K} \quad \frac{[[e]]s \neq 0 \quad C_1, s, h \rightsquigarrow K}{\mathbf{if} \ b \ \mathbf{then} \ C_1 \ \mathbf{else} \ C_2 \ \mathbf{fi}, s, h \rightsquigarrow K}$$

The semantics is easily seen to be deterministic. We introduce the following terminology.

**Definition 2.13.** We say that

- $C, s, h$  is *stuck* if there is no configuration  $K$  such that  $C, s, h \rightsquigarrow K$ .
- $C, s, h$  *goes wrong* if there is a non-terminal configuration  $K$  with  $C, s, h \rightsquigarrow^* K$  and  $K$  is stuck.
- $C, s, h$  *terminates normally* if there is a terminal configuration  $s', h'$  such that  $C, s, h \rightsquigarrow^* s', h'$ .

Other published definitions of the programming language used in separation logic use a special configuration called “abort” or “fault” instead of the concept of “stuck”; we are able to avoid this complication because we have restricted the programming language to a deterministic sublanguage.

As is standard, we define  $\text{Mod}(C)$  to be the set of variables that are modified by the command  $C$ , i.e., those that occur on the left hand side of the forms  $x^\delta := v$  and  $x^{\text{int}} := [e]$  (but *not*  $[x] := e$ ). The set  $\text{FV}(C)$  for a command is just the set of variables that occur in  $C$ .

## 2.4.2 Partial Correctness Specifications and Program Logic

**Definition 2.14.** Let  $A$  and  $B$  be assertions, and let  $C$  be a command. The *partial correctness specification* (pcs)  $\{A\} C \{B\}$  is said to hold if, for all states  $(s, h)$  with  $\text{FV}(A, C, B) \subseteq \text{dom}(s)$ ,  $(s, h) \Vdash A$  implies

- $C, s, h$  does not go wrong, and
- if  $C, s, h \rightsquigarrow^* s', h'$ , then  $(s', h') \Vdash B$ .

We refer to  $A$  and  $B$  as the *pre-* and *postcondition* of the specification, respectively.

We present a set of proof rules that are sound with respect to Def. 2.14. First, we give rules regarding the different constructs in the programming language and then we give some structural rules.

Rule for **skip**:  $\{A\} \text{skip} \{A\}$

Rules for assignment

$$\begin{aligned} & \{B[e/x]\} x^{\text{int}} := e \{B\} \\ & \{B[m/x]\} x^{\text{fs}} := m \{B\} \\ & \{B[r/x]\} x^{\text{frp}} := r \{B\} \end{aligned} \quad (2.19)$$

Rules for heap lookup. When  $x \notin \text{FV}(e', A)$  and  $y \notin \text{FV}(e)$ ,

$$\{(\exists y. e \mapsto y \wedge A) \wedge x = e'\} x := [e] \{e[e'/x] \mapsto x \wedge A[x/y]\} \quad (2.20)$$

When  $x \notin \text{FV}(e, A)$  and  $y \notin \text{FV}(e)$ ,

$$\{\exists y. e \mapsto y \wedge A\} x := [e] \{e \mapsto x \wedge A[x/y]\} \quad (2.21)$$

$$\text{Rule for heap update:} \quad \{e_1 \mapsto -\} [e_1] := e_2 \{e_1 \mapsto e_2\} \quad (2.22)$$

Rule for sequencing

$$\frac{\{A\} C_1 \{B'\} \quad \{B'\} C_2 \{B\}}{\{A\} C_1; C_2 \{B\}}$$

Rule for conditionals

$$\frac{\{A \wedge b\} C_1 \{B\} \quad \{A \wedge \neg b\} C_2 \{B\}}{\{A\} \text{ if } b \text{ then } C_1 \text{ else } C_2 \text{ fi } \{B\}}$$

Rule for **while** loops

$$\frac{\{A \wedge b\} C \{A\}}{\{A\} \text{ while } b \text{ do } C \text{ od } \{A \wedge \neg b\}}$$

Structural Rules:

Rule of consequence

$$\frac{A \Rightarrow A' \quad \{A'\} C \{B'\} \quad B' \Rightarrow B}{\{A\} C \{B\}}$$

Rule of Conjunction

$$\frac{\{A\} C \{A'\} \quad \{B\} C \{B'\}}{\{A \wedge B\} C \{A' \wedge B'\}}$$

$\forall$  introduction rule

$$\frac{\{A\} C \{B\}}{\{\forall x. A\} C \{\forall x. B\}} \quad x \notin \text{Modifies}(C)$$

The Frame Rule

$$\frac{\{A\} C \{B\}}{\{A * A'\} C \{B * A'\}} \quad \text{Modifies}(C) \cap \text{FV}(A') = \emptyset$$

The frame rule makes local reasoning possible: suppose the assertion  $A * A'$  describes a state in which we are to execute  $C$ , but that the “footprint” of  $C$ , i.e., those locations read or written by  $C$ , is described by  $A$  and  $B$ . Then from a local specification  $\{A\} C \{B\}$  for  $C$ , involving only this footprint, one can infer a global specification  $\{A * A'\} C \{B * A'\}$ , which also mentions locations not in the footprint of  $C$ . It is simpler to state and reason about local specifications, and the frame rule says that it is adequate to do so. We shall see several applications of the frame rule later.

We have the expected soundness result.

**Theorem 2.15.** *If a specification  $\{A\} C \{B\}$  is derivable by the rules in this section, then  $\{A\} C \{B\}$  holds.*

The proof of this theorem follows from proofs in earlier literature on separation logic [134, 115].

## 2.5 The Garbage Collection Algorithm

We implement and reason about *Cheney’s Algorithm* [35]. The implementation of the algorithm and the associated memory allocator is given in Appendix 2.A. It assumes two disjoint contiguous “semi-heaps” which have as domains the intervals  $Itv(\text{startOld}, \text{endOld})$  and  $Itv(\text{startNew}, \text{endNew})$  of equal size. We use the abbreviations

$$\text{OLD} \equiv Itv(\text{startOld}, \text{endOld}) \text{ and } \text{NEW} \equiv Itv(\text{startNew}, \text{endNew})$$

for these two intervals in the rest of the paper. The memory allocator attempts to allocate a **cons**-cell in OLD; if there is no space available in OLD, the garbage collector copies all cells in OLD reachable from root into NEW, and then the allocation resumes in NEW. The

garbage collector  $GC^*$  is delimited by comments in the code in Appendix 2.A. Notice that the algorithm is aware of the locations of the reachable cells only. In the spirit of local reasoning, our specification will therefore only involve the reachable pointers in OLD, called ALIVE, and not the remaining (unreachable) part of OLD. The implementation starts by initializing the variables `offset`, `scan`, `free`, and `maxFree`, according to which of the semi-heaps mentioned above that contains the data to be copied. We consider the case where `offset` is initialized to `startNew`.

The set  $\{\varphi, \text{FWD}, \text{UFWD}\}$  is an *auxiliary variable set* for the implementation, in the sense of [90, 108]. Thus, assignments to these variables are not necessary for the program to work; rather they ease the job of proving properties about the program – hence these assignments need not be executed. We have marked assignments to these variables with vertical bars in the margin of the code. We could have chosen to existentially quantify these variables and omit them from the program, but the reasoning becomes clearer when the program modifies the auxiliary variables explicitly.

Had we chosen to include allocation and disposal of blocks of memory in our programming language, a different implementation might have allocated a new semi-heap initially, then performed the garbage collection, and finally disposed the old semi-heap after collection. We stick to the current implementation, since this other implementation would not add any features of significant interest to our proof.

In the rest of this section, we first present a semantic condition in Sec. 2.5.1 which ensures that our program is a correct garbage collector. This condition is tightly connected to Cheney’s algorithm (and our implementation of it), inasmuch as it explicitly mentions the input and output variables such as `offset` and `scan`, and even auxiliary variables such as  $\varphi$ , `head`, and `tail`. We then present the precondition for the algorithm and the invariant for the **while** loop in the implementation, both formulated in the assertion language from Sec. 2.3.4. Before we formally prove that our implementation meets the corresponding specification in Sec. 2.6, we show in Sec. 2.5.4 that the specification entails that our implementation meets the requirement from Sec. 2.5.1.

### 2.5.1 Semantic Specification of a Copying Garbage Collector

The semantic definition of a garbage collector presented here is quite specific to our implementation, since it mentions the variables we use. It can be generalized by quantifying over these, but it is clearer to use the same variables as in the implementation. Thus, we have the following definition:

**Definition 2.16.** We call a command  $GC^*$  a *correct copying garbage collector* provided that if

- (a)  $\text{hejsa } \llbracket \text{head} \rrbracket s, \llbracket \text{tail} \rrbracket s$  are total functions on  $\llbracket \text{RCH} \rrbracket s$ , and  $\llbracket \text{offset} \rrbracket s, \llbracket \text{maxFree} \rrbracket s$  are pointers,
- (b) the heaps  $h_{rch}, h_{new}, h_{extra}$  are disjoint,
- (c) the rooted heap  $(\llbracket \text{root} \rrbracket, h_{rch})$  is exactly reachable,
- (d)  $\text{pairheap}(\llbracket \text{RCH} \rrbracket s, \llbracket \text{head} \rrbracket s, \llbracket \text{tail} \rrbracket s, h_{rch})$ ,
- (e)  $\text{dom}(h_{new}) = \llbracket \text{NEW} \rrbracket s$ ,
- (f)  $\#\llbracket \text{RCH} \rrbracket s \leq \#\llbracket \text{NEW} \rrbracket s$ , and

(g)  $GC^*, s, h_{rch} \cdot h_{new} \cdot h_{extra} \rightsquigarrow^* s', h'$ ,

then there exists a disjoint split  $h' = h'_{rch} \cdot h'_{fin} \cdot h'_{free}$  such that

(a') the heaps  $h'_{rch}, h'_{fin}, h'_{free}$ , and  $h_{extra}$  are pairwise disjoint,

(b')  $\text{dom}(h'_{rch}) = \text{dom}(h_{rch})$ ,

(c')  $\text{dom}(h'_{fin}) = \llbracket \text{FIN} \rrbracket s'$ ,

(d')  $\text{dom}(h'_{fin} \cdot h'_{free}) = \text{dom}(h_{new})$ , and

(e')  $\llbracket \varphi \rrbracket s'$  is a heap morphism from  $(\llbracket \text{root} \rrbracket s, h_{rch})$  to  $(\llbracket \text{offset} \rrbracket s', h'_{fin})$  that is an isomorphism of functions.

Clearly, if  $(\llbracket \text{root} \rrbracket s, h_{rch} \cdot h_{new} \cdot h_{extra})$  contains an exactly reachable subheap, then that must be  $(\llbracket \text{root} \rrbracket s, h_{rch})$ , and  $(\llbracket \text{offset} \rrbracket s', h'_{fin})$  will be an isomorphic exactly reachable subheap, due to Lemma 2.1. Thus, this condition clearly ensures that  $GC^*$  works as we would expect of a garbage collector on a heap with no dangling reachable locations, *i.e.*, it makes  $(\llbracket \text{offset} \rrbracket s', h')$  a garbage collected version of  $(\llbracket \text{root} \rrbracket s, h_{rch} \cdot h_{new} \cdot h_{extra})$  in the sense of Definition 2.2 (but it does not say what  $GC^*$  does if this is not the case). Furthermore, the definition specifies properties specific to a *copying* garbage collector; for example, it assumes that there is enough new space in the heap to make a new copy of the reachable data.

In the rest of this section, we first present a specification for our implementation and then show that this specification is strong enough to infer that our implementation of Cheney's algorithm meets the requirements in Def. 2.16.

## 2.5.2 The Precondition

Before execution of  $GC^*$ , we assume that an assertion  $\text{InitAss}$  holds.  $\text{InitAss}$  can be split into two parts:

$$\text{InitAss} \equiv \text{I}_c \wedge \text{I}_h,$$

where  $\text{I}_c$  is pure and  $\text{I}_h$  describes the heap unambiguously. These assertions are given by

$$\begin{aligned} \text{I}_c \equiv & \text{RCH} \perp \text{NEW} \wedge \#\text{RCH} \leq \#\text{NEW} \wedge \text{root} \in \text{ALIVE} \wedge \\ & \text{PtrRg}(\text{head}, \text{ALIVE}) \wedge \text{PtrRg}(\text{tail}, \text{ALIVE}) \wedge \text{Tfun}(\text{head}, \text{ALIVE}) \wedge \text{Tfun}(\text{tail}, \text{ALIVE}) \wedge \\ & \text{Reachable}(\text{head}, \text{tail}, \text{ALIVE}, \text{root}) \wedge \text{Ptr}(\text{offset}) \wedge \text{Ptr}(\text{maxFree}) \end{aligned}$$

and

$$\begin{aligned} \text{I}_h \equiv & (\forall_* x \in \text{ALIVE}. ((\exists y. (x, y) \in \text{head} \wedge x \mapsto y) * (\exists y'. (x, y') \in \text{tail} \wedge x + 4 \mapsto y'))) * \\ & (\forall_* x \in \text{NEW}. x \mapsto -, -). \end{aligned}$$

We use the convention that variables in upper case sans serif (like OLD) always have type `fs`. The variables `head` and `tail` are of type `fri`, and the rest of the variables are of type `int`.

Note that  $\text{I}_c$  is “constant” in the sense that it trivially remains true throughout execution of  $GC^*$ , since it is pure and only contains variables that are not modified. This means that it is also part of the invariant of the **while** loop.

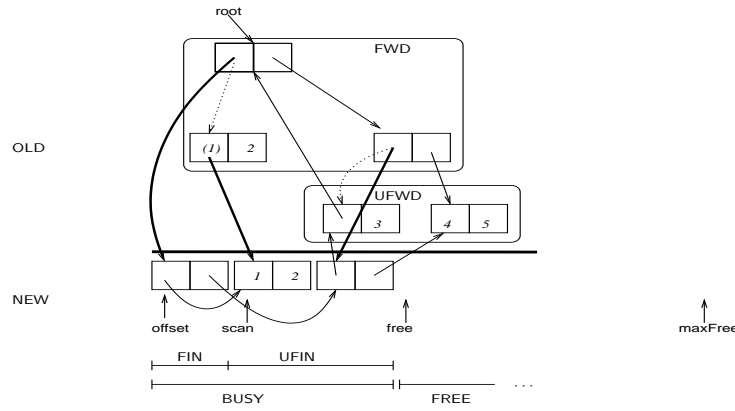


Figure 2.2: A state of execution

We describe part of these assertions in detail. For  $l_c$ , note in particular the conjunct

$$\text{Reachable}(\text{head}, \text{tail}, \text{ALIVE}, \text{root}) \quad (2.23)$$

expressing that  $\text{head}$ ,  $\text{tail}$ ,  $\text{ALIVE}$ , and  $\text{root}$  describe an exactly reachable subheap. We shall see that  $\text{ALIVE}$  is the set of initially reachable pointers from  $\text{root}$ . Since the variables  $\text{head}$ ,  $\text{tail}$ ,  $\text{root}$ , and  $\text{RCH}$  are not modified by the algorithm, this holds at any step of execution. The rest of  $l_c$  simply records basic facts about the relationship between the various sets and pointers. To understand  $l_h$ , suppose  $x$  denotes a pointer and consider the assertion

$$(\exists y. (x, y) \in \text{head} \wedge x \mapsto y) * (\exists y'. (x, y') \in \text{tail} \wedge x + 4 \mapsto y').$$

It asserts that there are values  $v, v'$  that are related to (the denotation of)  $x$  by the relations  $\text{head}$  and  $\text{tail}$ , and at the same time,  $x$  and  $x + 4$  point to these values in the heap. In this way,  $\text{head}$  and  $\text{tail}$  “record” the contents of the cell pointed to by  $x$ . By the definition of  $\forall_*$ , the assertion

$$((\forall_* x \in \text{ALIVE}. ((\exists y. (x, y) \in \text{head} \wedge x \mapsto y) * (\exists y'. (x, y') \in \text{tail} \wedge x + 4 \mapsto y'))))$$

therefore asserts that  $\text{head}$  and  $\text{tail}$  record the contents of all cells pointed to be pointers in  $\text{ALIVE}$ , (and those pointers are in the domain of the heap). It is easy to see from (2.23) and this assertion that  $\text{RCH}$  is the domain of an exactly reachable subheap of the initial heap. Also,  $l_h$  asserts that the set  $\text{NEW}$  is in the domain of the heap, so we can safely place copies of cells from  $\text{ALIVE}$  here.

### 2.5.3 The Invariant

To exhibit an invariant of the **while**-loop, we consider Fig. 2.2, which is a snapshot of a state during execution. Only the reachable cells in  $\text{OLD}$  (the part of the heap above the bold horizontal line) are shown. Three of the cells in  $\text{ALIVE}$  have been modified at this stage: their first fields have been updated with *forwarding pointers*; these appear bolder in the figure. The original contents of these first fields are indicated with dotted lines and parenthesized numbers. The pointers in  $\text{ALIVE}$  naturally divide into two sets:

- $\text{UFWD}$ : The pointers in  $\text{ALIVE}$  that point to cells not yet modified by the algorithm.

- FWD: The pointers in ALIVE that point to cells that have their first fields overwritten with a pointer in NEW.

The algorithm proceeds by traversing all the cells in between the scan and free pointers, that is, scan always points to the next cell to be traversed. A cell is traversed by *scanning* its two fields. If the field being scanned contains a non-pointer, the traversal simply proceeds to the next field or cell; if the field being scanned contains a pointer  $p$  in UFWD, the cell pointed to by  $p$  is copied and a forwarding pointer is placed in the original field; if the field being scanned contains a pointer in FWD, then the cell pointed to has already been copied and we simply update the scanned field to point to the copy. We use the auxiliary variables  $\varphi$  (which is of type `frp`), FWD, and UFWD to keep track of the forwarding pointers, and to record the reachable cells that have been already copied into NEW. When a cell is copied from RCH to NEW, the corresponding pointer is moved from UFWD to FWD, and  $\varphi$  is updated. To simplify certain aspects of the proof, each execution of the body of the **while** command uses the subprograms `ScanCar` and `ScanCdr` to scan the two fields of a cell that lie at the locations `scan` and `scan + 4`. Thus the actual value of scan always addresses the first field of a cell, and is therefore a pointer.

The pointers in NEW divide into the following three sets:

- FIN (which is an abbreviation of  $Itv(\text{offset}, \text{scan})$ ): The pointers in NEW that have been scanned. These are not modified further by the algorithm.
- UFIN (which is an abbreviation of  $Itv(\text{scan}, \text{free})$ ): The pointers in NEW that have not been scanned. These point to the original contents of cells pointed to by pointers in ALIVE.
- FREE (which is an abbreviation of  $Itv(\text{free}, \text{maxFree})$ ): The pointers in NEW that are available for allocation.

The five sets are illustrated in Fig. 2.2. Note that FIN, UFIN and FREE are intervals, whereas this is not the case for FWD and UFWD in general. Also, observe that  $\varphi$  is a one-to-one correspondence between the pointers in FWD and those in  $\text{BUSY} \equiv \text{FIN} \cup \text{UFIN} = Itv(\text{offset}, \text{free})$ . This bijection will turn out to be the heap morphism we are looking for.

The invariant of the algorithm has a pure and an impure part; the latter describes the heap. The pure part is

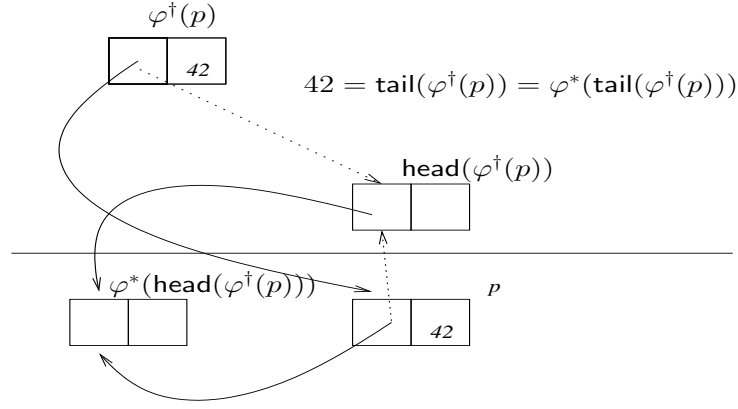
$$\begin{aligned} I_{\text{pure}} \equiv & I_c \wedge (\text{root}, \text{offset}) \in \varphi \wedge (\text{root} \in \text{FWD}) \wedge \\ & \text{iso}(\varphi, \text{FWD}, \text{BUSY}) \wedge (\text{ALIVE} = \text{FWD} \cup \text{UFWD}) \wedge \\ & \text{scan} \leq \text{free} \wedge \text{offset} \leq \text{scan} \wedge \text{Ptr}(\text{free}) \wedge \text{Ptr}(\text{scan}) \end{aligned}$$

Note in particular the conjunct  $\text{iso}(\varphi, \text{FWD}, \text{BUSY})$  expressing that  $\varphi$  is a bijection. The rest of  $I_{\text{pure}}$  simply records basic facts about the relationship between the various sets and pointers.

We now describe the impure part of the invariant; we use the partitioning of pointers into sets from before.

The cells pointed to by pointers in UFWD have not been modified by the algorithm; hence they are described by head and tail in the same way as the pointers in RCH in `InitAss`. We thus define

$$\begin{aligned} A_{\text{UFWD}} \equiv & \forall_* x \in \text{UFWD}. \\ & ((\exists y. (x, y) \in \text{head} \wedge x \mapsto y) * (\exists y'. (x, y') \in \text{tail} \wedge x + 4 \mapsto y')). \end{aligned}$$

Figure 2.3: The situation for a pointer  $p$  in FIN

Each of the cells pointed to by a pointer in FWD has a forwarding pointer in its first field. Recalling that  $\varphi$  records the forwarding pointers, we define

$$A_{\text{FWD}} \equiv \forall_* x \in \text{FWD}. (\exists y. (x, y) \in \varphi \wedge x \mapsto y, -).$$

A cell pointed to by a pointer in UFIN contains the original contents of a cell pointed to by a pointer in FWD. The latter pointer is recorded by the inverse of  $\varphi$ , and hence we define

$$A_{\text{UFIN}} \equiv \forall_* x \in \text{UFIN}. ((\exists y. (x, y) \in \text{head} \circ \varphi^\dagger \wedge x \mapsto y) * (\exists y'. (x, y') \in \text{tail} \circ \varphi^\dagger \wedge x + 4 \mapsto y')).$$

The cells in FIN have been scanned. The case-distinction between pointers and non-pointers during scanning is captured by the operator  $\odot$ . We define

$$A_{\text{FIN}} \equiv \forall_* x \in \text{FIN}. ((\exists y. (x, y) \in \varphi \odot (\text{head} \circ \varphi^\dagger) \wedge x \mapsto y) * (\exists y'. (x, y') \in \varphi \odot (\text{tail} \circ \varphi^\dagger) \wedge x + 4 \mapsto y')).$$

To understand  $A_{\text{UFIN}}$  and  $A_{\text{FIN}}$ , it is helpful to consider Fig. 2.3, in which we use a functional notation for the functional relations  $\text{head}$ ,  $\text{tail}$ , and  $\varphi$ . The pointer  $p \in \text{FIN}$  is the address of the rightmost bottom cell. Before  $p$  was scanned, it held the original contents of a cell pointed to by a pointer  $q \in \text{ALIVE}$ . After that cell was copied, it had its first field overwritten with the forwarding pointer  $p$ ; this is recorded by  $\varphi$ , hence  $(q, p) \in \varphi$ . The original contents of the cell pointed to by  $q$  is recorded by  $\text{head}$  and  $\text{tail}$ , so letting  $q'$  denote the address of the rightmost upper cell in Fig. 2.3, we have  $(q, q') \in \text{head}$ , hence  $(p, q') \in \text{head} \circ \varphi^\dagger$ . Before the field pointed to by  $p$  was scanned, it had  $q'$  in its first field. Now, by scanning the first field in the cell pointed to by  $p$ , we copy the cell pointed to by  $q'$  (if necessary), and update the field we are scanning to point to the address of the copy of that cell. Denoting the address of the copy by  $q''$ , we then have  $(q', q'') \in \varphi$ , by the definition of  $\varphi$ , and hence,  $(p, q'') \in \varphi \odot (\text{head} \circ \varphi^\dagger)$ .

For the pointers in FREE, we need only know that they are in the domain of the heap, to allow us to safely copy cells into FREE. We therefore define

$$A_{\text{FREE}} \equiv \forall_* x \in \text{FREE}. x \mapsto -, -.$$

	Unchanging	Changing
Abbreviations	NEW $\equiv Itv(\text{startNew}, \text{endNew})$	FIN $\equiv Itv(\text{offset}, \text{scan})$ UFIN $\equiv Itv(\text{scan}, \text{free})$ FREE $\equiv Itv(\text{free}, \text{maxFree})$ BUSY $\equiv Itv(\text{offset}, \text{free})$
Auxiliary	ALIVE head tail	FWD UFWD $\varphi$
Program	startOld endOld startNew endNew	root scan free offset maxFree

Table 2.1: Variables involved in the proof of the garbage collector.

In summary, the invariant of the algorithm is

$$I \equiv I_{\text{pure}} \wedge (A_{\text{UFWD}} * A_{\text{FWD}} * A_{\text{FIN}} * A_{\text{UFIN}} * A_{\text{FREE}}),$$

and hence, we aim to prove the specification

$$\{\text{InitAss}\} \text{GC}^* \{I \wedge \text{scan} = \text{free}\}. \quad (2.24)$$

Notice that the sets UFWD, FWD, etc. change throughout execution of the algorithm. Therefore, each of the corresponding assertions  $A_{\text{UFWD}}, A_{\text{FWD}}, \dots$  describe *dynamic* portions of the heap.

We prove that the specification (2.24) holds in Section 2.6, but prior to that, we prove that this implies correctness of the algorithm in Section 2.5.4.

For the assertions  $A_{\_}$  defined above, we will abuse notation slightly. It will sometimes be practical to consider an iterated separating conjunction over one of the above mentioned sets, except for one element, which needs to be considered separately. Therefore, for example, we will let  $A_{\text{FWD}-x}$  denote the following assertion (compare with the definition of  $A_{\text{FWD}}$ )

$$\forall *z \in (\text{FWD} \setminus \{x\}). (\exists y. (z, y) \in \varphi \wedge z \mapsto y, -).$$

To get a better overview, we list the different variables (along with what they abbreviate) that are involved in the program and the proof in Table 2.1. Recall that auxiliary variables are not needed for the program to work, but are included to ease the proof of the program. Auxiliary variables that are not modified by programs are called “ghost variables” or “logical variables” in the literature [108, 44].

### 2.5.4 Sufficiency of the Specification

Before we formally show that our implementation meets the specification (2.24), we show that the specification entails that the implementation meets the requirements from Def. 2.16.

First, since heaps are finite, there is a variable-free assertion  $A_{h_0}$  which is satisfied only by  $h_0$ , for every heap  $h_0$ . For instance, if  $h$  is the two-element heap  $[0 : 4 \mid 8 : 67]$ ,

then  $A_i$  might be  $0 \mapsto 4 * 8 \mapsto 67$ . For any heap  $h_{extra}$ , the following specification therefore follows from (2.24) and the frame rule.

$$\{\text{InitAss} * A_{h_{extra}}\} GC^* \{(I \wedge \text{scan} = \text{free}) * A_{h_{extra}}\}. \quad (2.25)$$

Thus, to show that our specification implies that our implementation meets the requirements of Def. 2.16, it suffices to show that if the conditions (a) to (f) in Def. 2.16 are met, then

$$(s, h_{rch} \cdot h_{new} \cdot h_{extra}) \Vdash \text{InitAss} * A_{h_{extra}'}$$

and if  $GC^*, s, h_{rch} \cdot h_{new} \cdot h_{extra} \rightsquigarrow^* s', h'$  and

$$(s', h') \Vdash I * A_{h_{extra}} \wedge \text{scan} = \text{free},$$

then conditions (a') to (e') from Def. 2.16 are met. The first of these claims is relatively straightforward, so we focus on the latter. For brevity, we first set  $h = h_{rch} \cdot h_{new} \cdot h_{extra}$ .

Thus, assume  $GC^*, s, h \rightsquigarrow^* s', h'$  for a state  $(s, h)$  in which the conditions (a) to (f) hold, and that the specification (2.25) holds. We then need to show that the conditions (a') to (e') from Def. 2.16 hold. The desired split of  $h'$  is the obvious one, so we only focus on showing the condition (e').

In our proof, we occasionally use names of variables instead of their denotations. Also, since head, tail, and  $\varphi$  denote functional relations (this clearly follows from I), we use functional notation for them, taking care to ensure definedness. For example, we write “there exists a  $q$  such that  $\varphi(p) = q$ ” instead of “there exists a  $q$  such that  $(p, q) \in \varphi$ ”.

First note that the variables RCH, head, and tail are not modified by  $GC^*$ , so the values of these before and after execution of the algorithm are equal (for example,  $\llbracket \text{RCH} \rrbracket s = \llbracket \text{RCH} \rrbracket s'$ ). The following lemma is important to establish that (the denotation of)  $\varphi$  is indeed a bijection that has the right domain and codomain.

**Lemma 2.17.** *Under the assumptions above,*

$$\llbracket \text{RCH} \rrbracket s = \llbracket \text{RCH} \rrbracket s' = \llbracket \text{FWD} \rrbracket s'.$$

*Proof.* The invariant I contains the assertion  $\text{Reachable}(\text{head}, \text{tail}, \text{RCH}, \text{root})$ , and therefore there is a heap  $h_0$  such that  $(\llbracket \text{root} \rrbracket s', h_0)$  is exactly reachable and  $\text{pairheap}(\text{RCH}, \text{head}, \text{tail}, h_0)$ . As mentioned in Section 2.3.2,  $h_0$  is determined by the values of RCH, head, and tail. These variables are not altered by the algorithm, so  $h_0$  is the same heap that is determined by  $\text{Reachable}(\text{head}, \text{tail}, \text{RCH}, \text{root})$  in the state  $(s, h)$ , which implies  $h_0 = h_{rch}$ . Hence  $(\llbracket \text{root} \rrbracket s', h_{rch})$  is exactly reachable. Let  $p$  be a pointer in  $\llbracket \text{RCH} \rrbracket s' = \text{pdom}(h_{rch})$ . This implies  $p \in \text{rp}(\llbracket \text{root} \rrbracket s', h_{rch})$ , and we show by induction on  $n$ , that  $p \in \text{rp}_n(\llbracket \text{root} \rrbracket s', h_{rch})$  implies  $p \in \llbracket \text{FWD} \rrbracket s'$ .

If  $n = 0$ , then  $p = \text{root}$ , and  $\text{root} \in \text{FWD}$  is part of I.

Assume, for the induction step, that  $\text{rp}_n(\llbracket \text{root} \rrbracket s', h_{rch}) \subseteq \llbracket \text{FWD} \rrbracket s'$ , and let  $p \in \text{rp}_{n+1}(\llbracket \text{root} \rrbracket s', h_{rch})$ . We need to show that  $p \in \llbracket \text{FWD} \rrbracket s'$ . By assumption, there is  $\ell \in \text{rl}_n(\llbracket \text{root} \rrbracket s', h_{rch})$  with  $h_{rch}(\ell) = p$ . By definition,  $\ell = p_0 + k$  for some  $p_0 \in \text{rp}_n(\llbracket \text{root} \rrbracket s', h_{rch})$  and  $k \in \{0, 4\}$ . By the induction hypothesis,  $p_0 \in s'(\text{FWD})$ . We argue by cases on whether  $k$  is 0 or 4.

If  $k = 0$ , then  $h_{rch}(p_0) = p$ . Because we have  $\text{pairheap}(\text{RCH}, \text{head}, \text{tail}, h_{rch})$ , this implies  $\text{head}(p_0) = p$ . Since  $p_0$  is in FWD, I implies that there is a  $q_0$  such that  $(p_0, q_0) \in \varphi$  (i.e.,  $p_0 = \varphi^{-1}(q_0)$ ), and this means that  $q_0$  is in FIN. The iterated separating conjunction over the set FIN that is part of I implies that there is a  $q_1$  such that  $q_0$  points to  $q_1$ , and

$$\begin{aligned}
q_1 &= \varphi^*(\text{head} \circ \varphi^{-1}(q_0)) \\
&= \varphi^*(\text{head}(p_0)) \\
&= \varphi^*(p) \\
&= \varphi(p),
\end{aligned}$$

since  $p$  is a pointer. This means that  $p$  is in the domain of  $\varphi$ , and thus  $p \in \text{FWD}$ .

If  $k = 4$ , we use the same argument, but with  $\text{tail}$  instead of  $\text{head}$ . For example,  $p = h_{rch}(p_0 + 4) = \text{tail}(p_0)$ .

This completes the proof of Lemma 2.17.  $\square$

**Lemma 2.18.** *The equation*

$$h'_{fin}((\varphi(p)) + k) = \varphi^*(h_{rch}(p + k))$$

holds for all pointers  $p \in \text{dom}(h_{rch}) = s'(\text{RCH}) = s'(\text{FWD})$  and  $k \in \{0, 4\}$ .

*Proof.* Let  $p$  be such a pointer. We show the desired equation by cases on whether  $k$  is 0 or 4.

If  $k = 0$ , we show  $h'_{fin}(\varphi(p)) = \varphi^*(h_{rch}(p))$ . Note first that  $\text{head}(p) = h_{rch}(p)$  because of  $\text{pairheap}(\text{RCH}, \text{head}, \text{tail}, h_{rch})$ .

The fact that  $p \in s'(\text{FWD})$  implies (by I), that there is a  $q \in \text{FIN}$  with  $q = \varphi(p)$ . For this  $q$ , there is a  $q_0$  such that  $q$  points to  $q_0$ , so that  $q_0 = h'_{fin}(q) = h'_{fin}(\varphi(p))$ , and

$$\begin{aligned}
q_0 &= \varphi^*(\text{head} \circ (\varphi^{-1}(q))) \\
&= \varphi^*(\text{head} \circ (\varphi^{-1}(\varphi(p)))) \\
&= \varphi^*(\text{head}(p)) \\
&= \varphi^*(h_{rch}(p)),
\end{aligned}$$

as desired.

If  $k = 4$ , we show  $h'_{fin}(\varphi(p) + 4) = \varphi^*(h_{rch}(p + 4))$ . Again, since  $p \in \text{FWD}$ , there is a  $q \in \text{FIN}$  such that  $q = \varphi(p)$ . For this  $q$ , there is a  $q_1$  such that  $h'_{fin}$  maps the location  $q + 4$  to  $q_1$  (i.e.,  $h'_{fin}(\varphi(p) + 4) = q_1$ ). At the same time,

$$\begin{aligned}
q_1 &= \varphi^*(\text{tail} \circ \varphi^{-1}(q)) \\
&= \varphi^*(\text{tail} \circ \varphi^{-1}(\varphi(p))) \\
&= \varphi^*(\text{tail}(p)) \\
&= \varphi^*(h_{rch}(p)),
\end{aligned}$$

as desired. This completes the proof of Lemma 2.18.  $\square$

Now, since I implies that  $\varphi$  is a bijection between  $\llbracket \text{FWD} \rrbracket s' = \llbracket \text{RCH} \rrbracket s' = \llbracket \text{RCH} \rrbracket s$  and  $\llbracket \text{FIN} \rrbracket s' = \llbracket \text{BUSY} \rrbracket s'$ , and the equation in Lemma 2.18 holds, we only need to show that  $\varphi(\llbracket \text{root} \rrbracket s) = \llbracket \text{offset} \rrbracket s'$  to conclude that the condition (e') holds for the state  $(s', h')$  after execution of  $\text{GC}^*$ . But this holds since  $(\text{root}, \text{offset}) \in \varphi$  is part of I and  $\text{root}$  is not modified by  $\text{GC}^*$ .

## 2.6 Proofs

In this section, we show the two specifications:

$$\frac{\{ \text{InitAss} \}}{\text{INIT}^*} \quad \text{and} \quad \frac{\{ I \wedge \neg(\text{scan} = \text{free}) \}}{\text{BODY}},$$

$$\frac{}{\{ I \}}$$

where  $\text{INIT}^*$  is the code before the **while** loop, and  $\text{BODY}$  is the body of the loop. The assertions  $\text{InitAss}$  and  $I$  are those formulated in Sections 2.5.2 and 2.5.3.

Both of these proofs use *local reasoning*. Recall that the idea is to infer a local specification for program fragments that manipulate the heap. These local specifications mention exactly the parts of the heap that are manipulated by the fragments, and the frame rule is then applied to obtain a global specification. Without the separating conjunction  $*$  and the frame rule, we would have to assert complicated non-interference claims each time we manipulate the heap. This is discussed further in Section 2.7.1.

### 2.6.1 Establishing the Invariant

We show that  $\text{INIT}^*$  establishes  $I$  when run in a state satisfying the precondition  $\text{InitAss}$  from Section 2.5.2. Therefore, let  $\text{INIT}$  and  $\text{INIT}^*$  be the code fragments

$$\begin{aligned} \text{INIT} \equiv & t_1 := [\text{root}]; \\ & t_2 := [\text{root} + 4]; \\ & [\text{free}] := t_1; \\ & [\text{free} + 4] := t_2; \\ & [\text{root}] := \text{free} \end{aligned}$$

and

$$\begin{aligned} \text{INIT}^* \equiv & \text{scan} := \text{offset}; \\ & \text{free} := \text{offset}; \\ & \text{FWD} := \emptyset; \\ & \text{UFWD} := \text{ALIVE}; \\ & \varphi := \emptyset; \\ & \text{INIT}; \\ & \text{FORW} := \text{FORW} \cup \{\text{root}\}; \\ & \text{UNFORW} := \text{UNFORW} \setminus \{\text{root}\}; \\ & \varphi := \varphi \cup \{(\text{root}, \text{free})\}; \\ & \text{free} := \text{free} + 8. \end{aligned}$$

As mentioned, we use local reasoning and thus we first infer a local specification for  $\text{INIT}$ , and then use this local specification and the frame rule to obtain a global specification for  $\text{INIT}^*$ .

The local specification for  $\text{INIT}$  below only mentions the locations that are read or manipulated by  $\text{INIT}$ . We use the notation  $\{A\} \Rightarrow \{B\}$  for applications of the rule of

consequence, *i.e.*, to denote that  $A \rightarrow B$  is valid.

$$\begin{array}{l}
\left\{ \begin{array}{l} (\exists y. (\text{root}, y) \in \text{head} \wedge \text{root} \mapsto y) * (\exists y'. (\text{root}, y') \in \text{tail} \wedge \text{root} + 4 \mapsto y') * \\ (\text{free} \mapsto -, -) \end{array} \right\} \\
t_1 := [\text{root}] \\
\left\{ \begin{array}{l} ((\text{root}, t_1) \in \text{head} \wedge \text{root} \mapsto t_1) * (\exists y'. (\text{root}, y') \in \text{tail} \wedge \text{root} + 4 \mapsto y') * \\ (\text{free} \mapsto -, -) \end{array} \right\} \\
t_2 := [\text{root} + 4] \\
\left\{ \begin{array}{l} ((\text{root}, t_1) \in \text{head} \wedge \text{root} \mapsto t_1) * ((\text{root}, t_2) \in \text{tail} \wedge \text{root} + 4 \mapsto t_2) * \\ (\text{free} \mapsto -, -) \end{array} \right\} \\
[\text{free}] := t_1 \\
[\text{free} + 4] := t_2 \\
\left\{ \begin{array}{l} ((\text{root}, t_1) \in \text{head} \wedge \text{root} \mapsto t_1) * ((\text{root}, t_2) \in \text{tail} \wedge \text{root} + 4 \mapsto t_2) * \\ (\text{free} \mapsto t_1, t_2) \end{array} \right\} \\
\Downarrow \\
\left\{ \begin{array}{l} (\text{root}, t_1) \in \text{head} \wedge (\text{root}, t_2) \in \text{tail} \wedge ((\text{root} \mapsto t_1) * (\text{root} + 4 \mapsto t_2) * \\ (\text{free} \mapsto t_1, t_2)) \end{array} \right\} \\
[\text{root}] := \text{free} \\
\left\{ \begin{array}{l} (\text{root}, t_1) \in \text{head} \wedge (\text{root}, t_2) \in \text{tail} \wedge ((\text{root} \mapsto \text{free}, -) * \\ (\text{free} \mapsto t_1, t_2)) \end{array} \right\}
\end{array}$$

We explain the first of these specifications in detail. First, the specification

$$\begin{array}{l}
\{(\exists y. (\text{root}, y) \in \text{head} \wedge \text{root} \mapsto y)\} \\
t_1 := [\text{root}] \\
\{((\text{root}, t_1) \in \text{head} \wedge \text{root} \mapsto t_1)\}
\end{array}$$

is valid by the rule (2.21) for heap lookup. The first specification above is then valid by the frame rule. The second specification is valid by the same argument. For the third specification, we use the frame rule again, along with two applications of the rule (2.22) for heap update. The implication follows from Remark 2.7 (we use purity of the assertions  $(\text{root}, t_1) \in \text{head}$  and  $(\text{root}, t_2) \in \text{tail}$ ), and for the last specification, we use the rule (2.22) for update, the frame rule, and an obvious rule for existentials, to “forget” an occurrence of  $t_2$ .

From the local specification for INIT above, we infer the following specification for INIT\* using the frame rule. We have labeled the steps with numbers and emphasized changes from the preceding stage in the assertions by underlining, to ease the reading. For brevity, we use  $\varphi'$  instead of  $\varphi \setminus \{(\text{root}, \text{free} - 8)\}$  in the last step below.

$$\begin{array}{l}
\{\text{InitAss}\} \\
(1) \quad \text{scan} := \text{offset}; \text{free} := \text{offset}; \text{FWD} := \emptyset; \text{UFWD} := \text{ALIVE}; \varphi := \emptyset; \\
\left. \begin{array}{l}
I_c \wedge \text{scan} = \text{offset} \wedge \text{free} = \text{offset} \wedge \text{FWD} = \emptyset \wedge \varphi = \emptyset \wedge \text{UFWD} = \text{ALIVE} \wedge \\
((\forall_* x \in \text{ALIVE}. ((\exists y. (x, y) \in \text{head} \wedge x \mapsto y) * (\exists y'. (x, y') \in \text{tail} \wedge x + 4 \mapsto y'))) * \\
(\forall_* x \in \text{NEW}. x \mapsto -, -))
\end{array} \right\} \\
(2) \downarrow \\
\left. \begin{array}{l}
I_c \wedge \text{iso}(\varphi, \text{FWD}, \text{BUSY}) \wedge \text{FWD} \cup \text{UFWD} = \text{ALIVE} \wedge \text{root} \in \text{UFWD} \wedge \\
\text{scan} = \text{offset} \wedge \text{free} = \text{offset} \wedge \text{FWD} = \emptyset \wedge \varphi = \emptyset \wedge \text{UFWD} = \text{ALIVE} \wedge \\
(\underline{A_{\text{UFWD}} * A_{\text{FWD}} * A_{\text{UFIN}} * A_{\text{FIN}} * A_{\text{FREE}}})
\end{array} \right\} \\
(3) \downarrow \\
\left. \begin{array}{l}
I_c \wedge \text{iso}(\varphi, \text{FWD}, \text{BUSY}) \wedge \text{FWD} \cup \text{UFWD} = \text{ALIVE} \wedge \text{root} \in \text{UFWD} \wedge \\
\text{scan} = \text{offset} \wedge \text{free} = \text{offset} \wedge \text{FWD} = \emptyset \wedge \varphi = \emptyset \wedge \text{UFWD} = \text{ALIVE} \wedge \\
((\underline{A_{\text{UFWD}-\text{root}} * A_{\text{FWD}} * A_{\text{UFIN}} * A_{\text{FIN}} * A_{\text{FREE}-\text{free}}}) * \\
((\exists y. (\text{root}, y) \in \text{head} \wedge \text{root} \mapsto y) * (\exists y'. (\text{root}, y') \in \text{tail} \wedge \text{root} + 4 \mapsto y') * \\
(\text{free} \mapsto -, -)))
\end{array} \right\} \\
(4) \quad \text{INIT} \\
\left. \begin{array}{l}
I_c \wedge \text{iso}(\varphi, \text{FWD}, \text{BUSY}) \wedge \text{FWD} \cup \text{UFWD} = \text{ALIVE} \wedge \text{root} \in \text{UFWD} \wedge \\
\text{scan} = \text{offset} \wedge \text{free} = \text{offset} \wedge \text{FWD} = \emptyset \wedge \varphi = \emptyset \wedge \text{UFWD} = \text{ALIVE} \wedge \\
((\underline{A_{\text{UFWD}-\text{root}} * A_{\text{FWD}} * A_{\text{UFIN}} * A_{\text{FIN}} * A_{\text{FREE}-\text{free}}}) * \\
(((\text{root} \mapsto \text{free}, -) * (\text{free} \mapsto t_1, t_2)) \wedge (\text{root}, t_1) \in \text{head} \wedge (\text{root}, t_2) \in \text{tail}))
\end{array} \right\} \\
(5) \downarrow \\
\left. \begin{array}{l}
I_c \wedge \text{iso}(\varphi, \text{FWD}, \text{BUSY}) \wedge \text{FWD} \cup \text{UFWD} = \text{ALIVE} \wedge \text{root} \in \text{UFWD} \wedge \\
\text{scan} = \text{offset} \wedge \text{free} = \text{offset} \wedge \text{FWD} = \emptyset \wedge \varphi = \emptyset \wedge \text{UFWD} = \text{ALIVE} \wedge \\
(\text{root}, t_1) \in \text{head} \wedge (\text{root}, t_2) \in \text{tail} \wedge \neg(\text{root} \in \text{FWD}) \wedge \neg((\text{root}, \text{free}) \in \varphi) \wedge \\
((\underline{A_{\text{UFWD}-\text{root}} * A_{\text{FWD}} * A_{\text{UFIN}} * A_{\text{FIN}} * A_{\text{FREE}-\text{free}}}) * \\
((\text{root} \mapsto \text{free}, -) * (\text{free} \mapsto t_1, t_2)))
\end{array} \right\} \\
(6) \quad \begin{array}{l}
\text{FWD} := \text{FWD} \cup \{\text{root}\}; \text{UFWD} := \text{UFWD} \setminus \{\text{root}\}; \\
\varphi := \varphi \cup \{(\text{root}, \text{free})\}; \text{free} := \text{free} + 8
\end{array} \\
\left. \begin{array}{l}
I_c \wedge \text{iso}(\varphi', \text{FWD} \setminus \{\text{root}\}, \text{BUSY} \setminus \{\text{free} - 8\}) \wedge \\
(\underline{\text{FWD} \setminus \{\text{root}\} \cup (\text{UFWD} \cup \{\text{root}\}) = \text{ALIVE}}) \wedge \\
(\underline{\text{root} \in \text{FWD}}) \wedge \neg(\text{root} \in \text{UFWD}) \wedge (\text{root}, \text{free} - 8) \in \varphi \wedge \\
\text{scan} = \text{offset} \wedge \text{offset} = \text{free} - 8 \wedge (\text{root}, t_1) \in \text{head} \wedge (\text{root}, t_2) \in \text{tail} \wedge \\
((\forall_* x \in ((\underline{\text{UFWD} \cup \{\text{root}\}}) \setminus \{\text{root}\}). ((\exists y. (x, y) \in \text{head} \wedge x \mapsto y) * \\
(\exists y'. (x, y') \in \text{tail} \wedge x + 4 \mapsto y'))) * \\
(\forall_* x \in (\underline{\text{FWD}} \setminus \{\text{root}\}). (\exists y. (x, y) \in \varphi' \wedge x \mapsto y, -)) * \\
(\forall_* x \in \text{FIN}. ((\exists y. (x, y) \in \varphi' \odot (\text{head} \circ (\varphi')^\dagger) \wedge x \mapsto y) * \\
(\exists y'. (x, y') \in \varphi' \odot (\text{tail} \circ (\varphi')^\dagger) \wedge x + 4 \mapsto y'))) * \\
(\forall_* x \in (\underline{\text{UFIN}} \setminus \{\text{free} - 8\}). ((\exists y. (x, y) \in \text{head} \circ (\varphi')^\dagger \wedge x \mapsto y) * \\
(\exists y'. (x, y') \in \text{tail} \circ (\varphi')^\dagger \wedge x + 4 \mapsto y'))) * \\
(\forall_* x \in ((\underline{\text{FREE}} \cup \{\text{free} - 8\}) \setminus \{\text{free} - 8\}). x \mapsto -, -) * \\
(\text{root} \mapsto (\text{free} - 8), -) * ((\text{free} - 8) \mapsto t_1, t_2))
\end{array} \right\}
\end{array}$$

In this derivation, the first step uses Hoare's rule for assignment several times. The second step uses the rule (2.11) for  $\forall_*$  (to conclude  $A_{\text{FWD}}$ ,  $A_{\text{FIN}}$ , and  $A_{\text{UFIN}}$ ) and (2.10) (for  $A_{\text{UFWD}}$  and  $A_{\text{FREE}}$ ). This step also uses the rules (2.B1), (2.B50), (2.B24), and (2.B11). The third step uses the rule (2.14) and the fact that  $\text{free} \in \text{FREE}$  by definition, and we use the frame rule and purity along with our local specification from before to take the fourth step. The fifth step is a consequence of purity and the rules (2.B22), (2.B23), and the last step follows from Hoare's rule for assignment.

We must show that the invariant  $I$  follows from the conclusion in this derivation.

We make an informal argument of this here, but a completely formal argument may be found in Appendix 2.C.

For the pure parts of  $l$ , the only thing to notice is that since  $\varphi \setminus \{(\text{root}, \text{free} - 8)\}$  is a bijection, we can conclude that when we add the pair  $(\text{root}, \text{free} - 8)$  to it and also add the two pointers to the relevant sets, it is still a bijection.

For the impure part of  $l$ , we argue that each of the iterated separating conjunctions can be inferred from the corresponding parts of the conclusion above.

The iterated separating conjunction  $A_{\text{UFWD}}$  over  $\text{UFWD}$  can be inferred from the conclusion because  $\neg(\text{root} \in \text{UFWD})$  implies that

$$(\text{UFWD} \cup \{\text{root}\}) \setminus \{\text{root}\} = \text{UFWD}.$$

The same argument can be used for  $A_{\text{FREE}}$ . The assertion  $A_{\text{FWD}}$  can be inferred from the iterated separating conjunction over  $\text{FWD} \setminus \{\text{root}\}$ , and  $\text{root} \mapsto (\text{free} - 8), -$  in the conclusion above, since  $\text{root} \in \text{FWD}$  and we can infer the assertion

$$(\exists y. (\text{root}, y) \in \varphi \wedge \text{root} \mapsto y, -)$$

for  $\text{root}$ , and therefore add  $\text{root}$  to the set  $\text{FWD} \setminus \{\text{root}\}$  over which we quantify in the iterated separating conjunction in the conclusion above. For  $A_{\text{FIN}}$  we can use the conjunct  $\text{scan} = \text{free}$  to infer that both of the assertions about  $\text{FIN}$  are equivalent to  $\text{emp}$ .

For  $A_{\text{UFIN}}$ , we can use the parts of the conclusion above that involve  $\varphi$ ,  $\text{head}$ , and  $\text{tail}$  along with what we know about  $\text{free} - 8$ , to infer

$$\begin{aligned} & (\exists y. (\text{free} - 8) \mapsto y \wedge (\text{free} - 8, y) \in \text{head} \circ \varphi^\dagger) * \\ & (\exists y'. ((\text{free} - 8) + 4) \mapsto y' \wedge (\text{free} - 8, y') \in \text{tail} \circ \varphi^\dagger), \end{aligned}$$

and then add  $\text{free} - 8$  to  $\text{UFIN} \setminus \{\text{free} - 8\}$ , to obtain the desired iterated separating conjunction.

This establishes that running  $\text{INIT}^*$  starting from a state satisfying the assertion  $\text{InitAss}$  from Section 2.5.2 terminates in a state that satisfies  $l$ , as desired.

## 2.6.2 Maintaining the Invariant

We have shown that  $l$  is established by the initializing code. The next step is to show that  $l$  is indeed an invariant, *i.e.*, that the specification

$$\begin{aligned} & \{l \wedge \text{scan} \neq \text{free}\} \\ & \text{BODY} \\ & \{l\} \end{aligned}$$

holds, where  $\text{BODY}$  is the body of the **while** loop. Note that  $\text{BODY}$  consists of two similar parts  $\text{ScanCar}$  and  $\text{ScanCdr}$ , one for each field of the cell pointed to by  $\text{scan}$ , they are marked in the code with comments. Between these halves, that cell is in a “mixed state”: the first field of it is finished, whereas the other is about to be scanned. The aim is thus to show that the following specifications hold.

$$\begin{aligned} & \{l \wedge \text{scan} \neq \text{free}\} \\ & \text{ScanCar;} \\ & \{l'\} \\ & \text{ScanCdr;} \\ & \text{scan} := \text{scan} + 8 \\ & \{l\}, \end{aligned} \tag{2.26}$$

where  $l'$  is an assertion which holds in the intermediate state where  $\text{scan}$  is “halfway between UFIN and FIN”:

$$\begin{aligned}
l' \equiv & \\
& l_{\text{pure}} \wedge \text{scan} \neq \text{free} \wedge \\
& (\text{A}_{\text{UFW D}} * \text{A}_{\text{FWD}} * \text{A}_{\text{FIN}} * \text{A}_{\text{UFIN}-\text{scan}} * \text{A}_{\text{FREE}} * \\
& (\exists y. (\text{scan}, y) \in \varphi \odot (\text{head} \circ \varphi^\dagger) \wedge \text{scan} \mapsto y) * \\
& (\exists y'. (\text{scan}, y') \in \text{head} \circ \varphi^\dagger \wedge \text{scan} + 4 \mapsto y'))
\end{aligned}$$

We focus on showing the first of the involved specifications,  $\{l\} \text{ScanCar} \{l'\}$ . The proof of the other is analogous and not described in all details. First, we describe  $\text{ScanCar}$  informally. It “scans” the first field in the cell pointed to by  $\text{scan}$ , and there are three branches according to the value  $a$  in it (and maybe the place it points to):

1. If  $a$  is a non-pointer, nothing happens.
2. If  $a$  is a pointer, we branch according to the value  $b = [a]$  of the first field of the cell pointed to by  $a$ .
  - (a) If  $b$  is a forwarding pointer, i.e., a pointer in  $\text{NEW}$ , we just update  $[\text{scan}]$  to  $b$ .
  - (b) If  $b$  is not a forwarding pointer, we copy the cell and update  $[a]$  to a forwarding pointer, and we also update  $[\text{scan}]$  to point to the new copy.

The effect of the command  $a := [\text{scan}]$  is formalized in this specification.

$$\begin{aligned}
& \{l \wedge \text{scan} \neq \text{free}\} \\
& \Downarrow \\
& \left\{ \begin{array}{l} l_{\text{pure}} \wedge \text{scan} \neq \text{free} \wedge \\ ((\text{A}_{\text{UFW D}} * \text{A}_{\text{FWD}} * \text{A}_{\text{FIN}} * \text{A}_{\text{UFIN}-\text{scan}} * \text{A}_{\text{FREE}}) * \\ (\exists y. (\text{scan}, y) \in \text{head} \circ \varphi^\dagger \wedge \text{scan} \mapsto y) * (\exists y'. (\text{scan}, y') \in \text{tail} \circ \varphi^\dagger \wedge \text{scan} + 4 \mapsto y')) \end{array} \right\} \\
& a := [\text{scan}] \\
& \left\{ \begin{array}{l} l_{\text{pure}} \wedge \text{scan} \neq \text{free} \wedge \\ ((\text{A}_{\text{UFW D}} * \text{A}_{\text{FWD}} * \text{A}_{\text{FIN}} * \text{A}_{\text{UFIN}-\text{scan}} * \text{A}_{\text{FREE}}) * \\ ((\text{scan}, a) \in \text{head} \circ \varphi^\dagger \wedge \text{scan} \mapsto a) * (\exists y'. (\text{scan}, y') \in \text{tail} \circ \varphi^\dagger \wedge \text{scan} + 4 \mapsto y')) \end{array} \right\} \\
& \Downarrow \\
& \left\{ \begin{array}{l} l_{\text{pure}} \wedge \text{scan} \neq \text{free} \wedge (\text{scan}, a) \in \text{head} \circ \varphi^\dagger \wedge \\ ((\text{A}_{\text{UFW D}} * \text{A}_{\text{FWD}} * \text{A}_{\text{FIN}} * \text{A}_{\text{UFIN}-\text{scan}} * \text{A}_{\text{FREE}}) * \\ (\text{scan} \mapsto a) * (\exists y'. (\text{scan}, y') \in \text{tail} \circ \varphi^\dagger \wedge \text{scan} + 4 \mapsto y')) \end{array} \right\}.
\end{aligned}$$

The first step here is due to (2.14), the specification step uses the rule (2.21) for lookup, the frame rule and purity; the last rewriting uses purity.

Henceforth, we let  $l_a$  be the conclusion in the derivation above.

According to the rule of conditionals, there are two specifications to be shown, according to the outer **if**-branch in  $\text{ScanCar}$ . The first of these is

$$\begin{aligned}
& \{l_a \wedge \neg(a \bmod 8 = 0)\} \\
& \Downarrow \\
& \{l_a \wedge \neg\text{Ptr}(a)\} \\
& \quad \mathbf{skip} \\
& \{l'\}
\end{aligned} \tag{2.27}$$

This specification is shown in Sec. 2.6.2. The second specification we have to show for the outer **if**-branch contains an inner **if**-branch, so it too splits into two specifications. Before writing these down, we formalize the effect of the command  $b := [a]$ . We have

$$\begin{aligned}
& \{l_a \wedge a \bmod 8 = 0\} \\
(1) \quad & \Downarrow \\
& \{l_a \wedge \text{Ptr}(a)\} \\
(2) \quad & \Downarrow \\
& \{l_a \wedge \text{Ptr}(a) \wedge a \in \text{ALIVE}\} \\
(3) \quad & \Downarrow \\
& \{l_a \wedge \text{Ptr}(a) \wedge (a \in \text{FWD} \vee a \in \text{UFWD})\} \\
(4) \quad & \Downarrow \\
& \{l_a \wedge \text{Ptr}(a) \wedge (a \leftrightarrow - \vee a \leftrightarrow -)\} \\
(5) \quad & \Downarrow \\
& \{l_a \wedge \text{Ptr}(a) \wedge (a \leftrightarrow -)\} \\
& \quad b := [a] \\
& \{l_a \wedge \text{Ptr}(a) \wedge (a \leftrightarrow b)\}
\end{aligned}$$

The first of the implications uses (2.B16), and the second follows from  $(\text{scan}, x) \in \text{head} \circ \varphi^\dagger \wedge \text{PtrRg}(\text{head}, \text{ALIVE})$ , (2.B40), and (2.B41). The third follows from (2.B44), and the fourth from (2.12). Finally the specification is an instance of the rule for lookup and the frame rule, since  $b$  does not occur free in  $l_a \wedge \text{Ptr}(a)$ .

According to the rule of conditionals, there are two more specifications to show to conclude the desired specification  $\{l \wedge \neg(\text{scan} = \text{free})\} \text{ScanCar} \{l'\}$ . They are

$$\begin{aligned}
& \{l_a \wedge \text{Ptr}(a) \wedge (a \leftrightarrow b) \wedge b \bmod 8 = 0 \wedge \text{offset} \leq b \leq \text{maxFree}\} \\
& \Downarrow \\
& \{l_a \wedge \text{Ptr}(a) \wedge (a \leftrightarrow b) \wedge b \in \text{NEW}\} \\
& \quad [\text{scan}] := b \\
& \{l'\}
\end{aligned} \tag{2.28}$$

and

$$\begin{aligned}
& \{l_a \wedge \text{Ptr}(a) \wedge (a \leftrightarrow b) \wedge \neg(b \bmod 8 = 0 \wedge \text{offset} \leq b \leq \text{maxFree})\} \\
& \Downarrow \\
& \{l_a \wedge \text{Ptr}(a) \wedge (a \leftrightarrow b) \wedge \neg(b \in \text{NEW})\} \\
& \quad \text{CopyCell}^* \\
& \{l'\},
\end{aligned} \tag{2.29}$$

where

$$\begin{aligned}
\text{CopyCell}^* \quad & \equiv \quad t_1 := [a]; \\
& \quad t_2 := [a + 4]; \\
& \quad [\text{free}] := t_1; \\
& \quad [\text{free} + 4] := t_2; \\
& \quad [a] := \text{free}; \\
& \quad [\text{scan}] := \text{free}; \\
& \quad \text{FWD} := \text{FWD} \cup \{a\}; \\
& \quad \text{UFWD} := \text{UFWD} \setminus \{a\}; \\
& \quad \varphi := \varphi \cup \{(a, \text{free})\}; \\
& \quad \text{free} := \text{free} + 8
\end{aligned}$$

These specifications are shown in Sec. 2.6.2 and Sec. 2.6.2, respectively. But first, we note a lemma for later use.

**Lemma 2.19.**  $I_{\text{pure}}$  implies  $\text{free} \leq \text{maxFree}$ .

*Proof.* By the rule (2.B13) for intervals with a common start-point, it suffices to show that  $\#\text{BUSY} \leq \#\text{NEW}$ . But this follows from

$$\#\text{BUSY} = \#\text{FWD} \leq \#\text{ALIVE} \leq \#\text{NEW}$$

The first equality follows from  $\text{iso}(\varphi, \text{FWD}, \text{BUSY})$  and (2.B32), whereas the first inequality follows from (2.B11), (2.B46), and  $\text{FWD} \cup \text{UFWD} = \text{ALIVE}$ . The last inequality is part of  $I_{\text{pure}}$ .  $\square$

### If Nothing Happens

We show that the specification (2.27) holds. According to the rule for **skip** and the rule of consequence, that amounts to

**Lemma 2.20.** *The assertion*

$$A \equiv I_a \wedge \neg \text{Ptr}(a)$$

*implies*  $I'$ .

*Proof.* We have

$$\begin{aligned} & I_{\text{pure}} \wedge \text{scan} \neq \text{free} \wedge \neg \text{Ptr}(a) \wedge (\text{scan}, a) \in \text{head} \circ \varphi^\dagger \wedge \\ & ((\text{AUFWD} * \text{AFWD} * \text{AFIN} * \text{AUFIN}_{\text{-scan}} * \text{AFREE}) * \\ & (\text{scan} \mapsto a) * (\exists y'. (\text{scan}, y') \in \text{tail} \circ \varphi^\dagger \wedge \text{scan} + 4 \mapsto y')) \\ & \Downarrow \\ & I_{\text{pure}} \wedge \text{scan} \neq \text{free} \wedge \neg \text{Ptr}(a) \wedge \underline{(\text{scan}, a) \in \varphi \odot (\text{head} \circ \varphi^\dagger)} \wedge \\ & ((\text{AUFWD} * \text{AFWD} * \text{AFIN} * \text{AUFIN}_{\text{-scan}} * \text{AFREE}) * \\ & (\text{scan} \mapsto a) * (\exists y'. (\text{scan}, y') \in \text{tail} \circ \varphi^\dagger \wedge \text{scan} + 4 \mapsto y')) \\ & \Downarrow \\ & I_{\text{pure}} \wedge \text{scan} \neq \text{free} \wedge \neg \text{Ptr}(a) \wedge \\ & ((\text{AUFWD} * \text{AFWD} * \text{AFIN} * \text{AUFIN}_{\text{-scan}} * \text{AFREE}) * \\ & \underline{((\text{scan}, a) \in \varphi \odot (\text{head} \circ \varphi^\dagger) \wedge \text{scan} \mapsto a)} * \\ & (\exists y'. (\text{scan}, y') \in \text{tail} \circ \varphi^\dagger \wedge \text{scan} + 4 \mapsto y')) \\ & \Downarrow \\ & I' \end{aligned}$$

The first implication follows from the fact that  $A$  implies  $(\text{scan}, a) \in \varphi \odot (\text{head} \circ \varphi^\dagger)$  by (2.B34). The second implication follows from purity.  $\square$

This implies that the specification (2.27) holds.

### If We do not Copy

We show the specification (2.28) in this section. The proof goes as follows: first, we show that the precondition implies  $a \in \text{FWD}$ , and use this to infer  $(\text{scan}, b) \in \varphi \odot (\text{head} \circ \varphi^\dagger)$ . Then we use a local specification to infer the desired global specification.

**Lemma 2.21.** *The assertion*

$$A \equiv I_a \wedge \text{Ptr}(a) \wedge (a \hookrightarrow b) \wedge b \in \text{NEW}$$

*implies*  $a \in \text{FWD}$ .

*Proof.* We give an informal argument here, and we refer to Appendix 2.D for a more formal proof. We have  $a \in \text{ALIVE}$  which is equal to the union of FWD and UFWD, so it suffices to assume  $a \in \text{UFWD}$  and derive a contradiction. If  $a \in \text{UFWD}$ , we know that the assertion in the body of  $A_{\text{UFWD}}$  holds for  $a$ . This implies that whatever  $a$  points to is related to  $a$  by head, and since any pointer in the range of head is ALIVE, which is disjoint from NEW, we get the desired contradiction from  $a \hookrightarrow b$  and  $b \in \text{NEW}$ .  $\square$

**Lemma 2.22.** *The assertion  $A$  from Lemma 2.21 implies  $(\text{scan}, b) \in \varphi \odot (\text{head} \circ \varphi^\dagger)$ .*

*Proof.* We use Lemma 2.21 and show  $A \wedge a \in \text{FWD} \rightarrow (\text{scan}, b) \in \varphi \odot (\text{head} \circ \varphi^\dagger)$ . By (2.B35),

$$(\text{scan}, a) \in \text{head} \circ \varphi^\dagger \wedge A \wedge (a, b) \in \varphi \rightarrow (\text{scan}, b) \in \varphi \odot (\text{head} \circ \varphi^\dagger),$$

so it suffices to show

$$A \wedge a \in \text{FWD} \rightarrow (a, b) \in \varphi.$$

Like before, we use (2.16) and show

$$A_{\text{FWD}} \wedge a \hookrightarrow b \wedge a \in \text{FWD} \rightarrow (a, b) \in \varphi.$$

We have

$$\begin{aligned} & A_{\text{FWD}} \wedge a \hookrightarrow b \wedge a \in \text{FWD} \\ & \Downarrow \\ & (\underline{A_{\text{FWD}-a} * (\exists y. (a, y) \in \varphi \wedge a \mapsto y, -)}) \wedge a \hookrightarrow b \\ & \Downarrow \\ & \underline{A_{\text{FWD}-a} * ((a, b) \in \varphi \wedge a \mapsto b, -)} \\ & \Downarrow \\ & (a, b) \in \varphi \end{aligned}$$

In this derivation, we have first used (2.14), then (2.18), and finally purity.  $\square$

We turn to the local specification for this branch of the program. Again, it only mentions the footprint of the branch:

$$\begin{aligned} & \{\text{scan} \mapsto - \wedge (\text{scan}, b) \in \varphi \odot (\text{head} \circ \varphi^\dagger)\} \\ & \quad [\text{scan}] := b \\ & \{\text{scan} \mapsto b \wedge (\text{scan}, b) \in \varphi \odot (\text{head} \circ \varphi^\dagger)\} \\ & \Downarrow \\ & \{\exists y. (\text{scan}, y) \in \varphi \odot (\text{head} \circ \varphi^\dagger) \wedge \text{scan} \mapsto y\} \end{aligned} \tag{2.30}$$

The first step follows from the rule (2.22) for heap update and the rule of conjunction.

We can now show a global specification for  $[\text{scan}] := b$ .

$$\begin{aligned}
& \{I_a \wedge \text{Ptr}(a) \wedge (a \leftrightarrow b) \wedge b \in \text{NEW}\} \\
& \Downarrow \\
& \{I_a \wedge \underline{(\text{scan}, b) \in \varphi \odot (\text{head} \circ \varphi^\dagger)}\} \\
& \Downarrow \\
& \left\{ \begin{array}{l} I_{\text{pure}} \wedge \text{scan} \neq \text{free} \wedge \\ ((A_{\text{UFW}} * A_{\text{FWD}} * A_{\text{FIN}} * A_{\text{UFIN}}\text{-scan} * A_{\text{FREE}}) * \\ (\text{scan} \mapsto a \wedge \underline{(\text{scan}, b) \in \varphi \odot (\text{head} \circ \varphi^\dagger)}) * \\ (\exists y'. (\text{scan}, y') \in \text{tail} \circ \varphi^\dagger \wedge \text{scan} + 4 \mapsto y')) \end{array} \right\} \quad (2.31) \\
& [\text{scan}] := b \\
& \left\{ \begin{array}{l} I_{\text{pure}} \wedge \text{scan} \neq \text{free} \wedge \\ ((A_{\text{UFW}} * A_{\text{FWD}} * A_{\text{FIN}} * A_{\text{UFIN}}\text{-scan} * A_{\text{FREE}}) * \\ (\underline{\text{scan} \mapsto b} \wedge (\text{scan}, b) \in \varphi \odot (\text{head} \circ \varphi^\dagger)) * \\ (\exists y'. (\text{scan}, y') \in \text{tail} \circ \varphi^\dagger \wedge \text{scan} + 4 \mapsto y')) \end{array} \right\} \\
& \Downarrow \\
& I'
\end{aligned}$$

For the first implication, we use Lemma 2.22, and for the second, we use purity. The specification step follows from our local specification (2.30), the frame rule, and purity. This proves the desired global specification (2.28).

**Remark 2.23.** If we did not have the separating conjunction  $*$ , the specification step in (2.31) would require us to ensure that the assignment to the heap cell  $[\text{scan}]$  does not affect any of the assertions  $A_\_$ , via non-interference predicates stating that, e.g.,  $\text{scan}$  is not in any of the sets involved in the specification.

We are now ready to address the specification (2.29) for the most complicated branch of  $\text{ScanCar}$ . The code resembles  $\text{INIT}^*$ , hence the proof of its specification will be similar to the proof in Section 2.6.1.

### If We Copy

We show that the specification (2.29) is derivable. To this end,  $\text{CopyCell}^*$  is split into two parts:

$$\begin{array}{ll}
\text{CopyCell} \equiv & t_1 := [a]; & \text{Increment} \equiv & \text{FWD} := \text{FWD} \cup \{a\}; \\
& t_2 := [a + 4]; & & \text{UFW} := \text{UFW} \setminus \{a\}; \\
& [\text{free}] := t_1; & & \varphi := \varphi \cup \{(a, \text{free})\}; \\
& [\text{free} + 4] := t_2; & & \text{free} := \text{free} + 8; \\
& [a] := \text{free}; & & \\
& [\text{scan}] := \text{free}; & & 
\end{array}$$

We first show that in this case,  $a \in \text{UFW}$ . Then we derive a local specification for  $\text{CopyCell}$ , which leads to the desired global specification for  $\text{CopyCell}^*$ .

**Lemma 2.24.** *The assertion*

$$A \equiv I_a \wedge \text{Ptr}(a) \wedge (a \leftrightarrow b) \wedge \neg(b \in \text{NEW})$$

*implies*  $a \in \text{UFW} \wedge \neg(a \in \text{FWD})$ .

*Proof.* We give an informal proof here; a completely formal proof is given in Appendix 2.E. The proof goes by contradiction (as the proof of Lemma 2.21); this time we assume  $a \in \text{FWD}$  and derive a contradiction. As in the proof just mentioned, if  $a$  is in FWD, then whatever  $a$  points to is related to  $a$  in  $\varphi$ , which is a bijection which has BUSY as its codomain. This contradicts that  $a$  points to  $b$  which is not in NEW.  $\square$

We turn to the local specification for CopyCell. As usual, it only involves the footprint of the program fragment.

$$\begin{aligned}
& \left\{ \begin{array}{l} (\exists y. (a, y) \in \text{head} \wedge a \mapsto y) * (\exists y'. (a, y') \in \text{tail} \wedge a + 4 \mapsto y') * \\ (\text{scan} \mapsto -) * (\text{free} \mapsto -, -) \end{array} \right\} \\
& t_1 := [a] \\
& \left\{ \begin{array}{l} ((a, t_1) \in \text{head} \wedge a \mapsto t_1) * (\exists y. (a, y') \in \text{tail} \wedge a + 4 \mapsto y') * \\ (\text{scan} \mapsto -) * (\text{free} \mapsto -, -) \end{array} \right\} \\
& t_2 := [a + 4] \\
& \left\{ \begin{array}{l} ((a, t_1) \in \text{head} \wedge a \mapsto t_1) * ((a, t_2) \in \text{tail} \wedge a + 4 \mapsto t_2) * \\ (\text{scan} \mapsto -) * (\text{free} \mapsto -, -) \end{array} \right\} \\
& [\text{free}] := t_1 \\
& \left\{ \begin{array}{l} ((a, t_1) \in \text{head} \wedge a \mapsto t_1) * ((a, t_2) \in \text{tail} \wedge a + 4 \mapsto t_2) * \\ (\text{scan} \mapsto -) * (\text{free} \mapsto \underline{t_1}, -) \end{array} \right\} \\
& [\text{free} + 4] := t_2 \\
& \left\{ \begin{array}{l} ((a, t_1) \in \text{head} \wedge a \mapsto t_1) * ((a, t_2) \in \text{tail} \wedge a + 4 \mapsto t_2) * \\ (\text{scan} \mapsto -) * (\text{free} \mapsto t_1, \underline{t_2}) \end{array} \right\} \tag{2.32} \\
& \Downarrow \\
& \left\{ \begin{array}{l} ((a \mapsto t_1) * (a + 4 \mapsto t_2) * (\text{scan} \mapsto -) * (\text{free} \mapsto t_1, t_2)) \wedge \\ \underline{(a, t_1) \in \text{head} \wedge (a, t_2) \in \text{tail}} \end{array} \right\} \\
& [a] := \text{free} \\
& \left\{ \begin{array}{l} ((a \mapsto \text{free}) * (a + 4 \mapsto t_2) * (\text{scan} \mapsto -) * (\text{free} \mapsto t_1, t_2)) \wedge \\ (a, t_1) \in \text{head} \wedge (a, t_2) \in \text{tail} \end{array} \right\} \\
& [\text{scan}] := \text{free} \\
& \left\{ \begin{array}{l} ((a \mapsto \text{free}) * (a + 4 \mapsto t_2) * (\underline{\text{scan} \mapsto \text{free}}) * (\text{free} \mapsto t_1, t_2)) \wedge \\ (a, t_1) \in \text{head} \wedge (a, t_2) \in \text{tail} \end{array} \right\} \\
& \Downarrow \\
& \left\{ \begin{array}{l} (\underline{(a \mapsto \text{free}, -)} * (\text{scan} \mapsto \text{free}) * (\text{free} \mapsto t_1, t_2)) \wedge \\ (a, t_1) \in \text{head} \wedge (a, t_2) \in \text{tail} \end{array} \right\}
\end{aligned}$$

The implication in the middle of this derivation is due to pureness, and the rest of the steps use the rules for lookup (2.21) and update (2.22), along with the frame rule.

We now infer the specification for CopyCell\* via the local specification (2.32). In this derivation, we write  $\varphi'$  instead of  $\varphi \setminus \{(a, \text{free} - 8)\}$  for brevity.

$$\begin{aligned}
& \{I_a \wedge \text{Ptr}(a) \wedge (a \leftrightarrow b) \wedge \neg(b \in \text{NEW})\} \\
& \Downarrow \\
& \{I_a \wedge \text{Ptr}(a) \wedge a \in \text{UFWD} \wedge \neg(a \in \text{FWD})\} \\
& \Downarrow \\
& \left\{ \begin{array}{l}
I_{\text{pure}} \wedge \text{scan} \neq \text{free} \wedge (\text{scan}, a) \in \text{head} \circ \varphi^\dagger \wedge \text{Ptr}(a) \wedge a \in \text{UFWD} \wedge \\
\neg(a \in \text{FWD}) \wedge \forall x. \neg((a, x) \in \varphi) \wedge \\
((\text{A}_{\text{UFWD}-a} * \text{A}_{\text{FWD}} * \text{A}_{\text{FIN}} * \text{A}_{\text{UFIN}-\text{scan}} * \text{A}_{\text{FREE}-\text{free}}) * \\
(\exists y'. (\text{scan}, y') \in \text{tail} \circ \varphi^\dagger \wedge \text{scan} + 4 \mapsto y')) * \\
(\exists y. (a, y) \in \text{head} \wedge a \mapsto y) * (\exists y'. (a, y') \in \text{tail} \wedge a + 4 \mapsto y')) * \\
(\text{scan} \mapsto -) * (\text{free} \mapsto -, -)
\end{array} \right\} \\
& \text{CopyCell} \\
& \left\{ \begin{array}{l}
I_{\text{pure}} \wedge \text{scan} \neq \text{free} \wedge (\text{scan}, a) \in \text{head} \circ \varphi^\dagger \wedge \text{Ptr}(a) \wedge a \in \text{UFWD} \wedge \\
\neg(a \in \text{FWD}) \wedge \forall x. \neg((a, x) \in \varphi) \wedge \\
((\text{A}_{\text{UFWD}-a} * \text{A}_{\text{FWD}} * \text{A}_{\text{FIN}} * \text{A}_{\text{UFIN}-\text{scan}} * \text{A}_{\text{FREE}-\text{free}}) * \\
(\exists y'. (\text{scan}, y') \in \text{tail} \circ \varphi^\dagger \wedge \text{scan} + 4 \mapsto y')) * \\
((a \mapsto \text{free}, -) * (\text{scan} \mapsto \text{free}) * (\text{free} \mapsto t_1, t_2)) \wedge (a, t_1) \in \text{head} \wedge (a, t_2) \in \text{tail}
\end{array} \right\} \\
& \Downarrow \\
& \left\{ \begin{array}{l}
I_{\text{pure}} \wedge \text{scan} \neq \text{free} \wedge (\text{scan}, a) \in \text{head} \circ \varphi^\dagger \wedge \text{Ptr}(a) \wedge a \in \text{UFWD} \wedge \\
\neg(a \in \text{FWD}) \wedge \neg((a, \text{free}) \in \varphi) \wedge (a, t_1) \in \text{head} \wedge (a, t_2) \in \text{tail} \wedge \\
((\text{A}_{\text{UFWD}-a} * \text{A}_{\text{FWD}} * \text{A}_{\text{FIN}} * \text{A}_{\text{UFIN}-\text{scan}} * \text{A}_{\text{FREE}-\text{free}}) * \\
(\exists y'. (\text{scan}, y') \in \text{tail} \circ \varphi^\dagger \wedge \text{scan} + 4 \mapsto y')) * \\
((a \mapsto \text{free}, -) * (\text{scan} \mapsto \text{free}) * (\text{free} \mapsto t_1, t_2))
\end{array} \right\} \\
& \text{FWD} := \text{FWD} \cup \{a\}; \text{UFWD} := \text{UFWD} \setminus \{a\}; \\
& \varphi := \varphi \cup \{(a, \text{free})\}; \text{free} := \text{free} + 8 \\
& \left\{ \begin{array}{l}
I_c \wedge \text{root} \in \text{FWD} \setminus \{a\} \wedge \\
\text{iso}(\varphi', \text{FWD} \setminus \{a\}, \text{BUSY} \setminus \{\text{free} - 8\}) \wedge \\
(\text{ALIVE} = (\text{FWD} \setminus \{a\}) \cup (\text{UFWD} \cup \{a\})) \wedge \\
\text{scan} \leq \text{free} - 8 \wedge \text{offset} \leq \text{scan} \wedge \text{Ptr}(\text{free} - 8) \wedge \text{Ptr}(\text{scan}) \wedge \\
\text{scan} \neq \text{free} - 8 \wedge (\text{scan}, a) \in \text{head} \circ (\varphi')^\dagger \wedge \text{Ptr}(a) \wedge \neg(a \in \text{UFWD}) \wedge \\
a \in \text{FWD} \wedge (a, t_1) \in \text{head} \wedge (a, t_2) \in \text{tail} \wedge \\
((\forall *x \in ((\text{UFWD} \cup \{a\}) \setminus \{a\}). ((\exists y. (x, y) \in \text{head} \wedge x \mapsto y) * \\
(\exists y'. (x, y') \in \text{tail} \wedge x + 4 \mapsto y')))) * \\
(\forall *x \in (\text{FWD} \setminus \{a\}). (\exists y. (x, y) \in \varphi' \wedge x \mapsto y, -)) * (a \mapsto (\text{free} - 8), -) * \\
(\forall *x \in \text{FIN}. ((\exists y. (x, y) \in \varphi' \odot (\text{head} \circ (\varphi')^\dagger) \wedge x \mapsto y) * \\
(\exists y'. (x, y') \in \varphi' \odot (\text{tail} \circ (\varphi')^\dagger) \wedge x + 4 \mapsto y')) * \\
(\forall *x \in (\text{UFIN} \setminus \{\text{free} - 8\}). ((\exists y. (x, y) \in \text{head} \circ (\varphi')^\dagger \wedge x \mapsto y) * \\
(\exists y'. (x, y') \in \text{tail} \circ (\varphi')^\dagger \wedge x + 4 \mapsto y')) * ((\text{free} - 8) \mapsto t_1, t_2)) * \\
(\forall *x \in ((\text{FREE} \cup \{\text{free} - 8\}) \setminus \{\text{free} - 8\}). x \mapsto -, -))
\end{array} \right\}
\end{aligned}$$

The first implication follows from Lemma 2.24. The second follows from (2.14) and (2.B33). The global specification for CopyCell follows from our local specification (2.32), purity, and the frame rule. The implication immediately thereafter is a consequence of purity. The specification for the three auxiliary variables and the update of free follows from Hoare's rule for assignment and obvious rules for intervals.

**Remark 2.25.** Notice the crucial use of local reasoning in this derivation. If we did not have the separating conjunction, then for each of the updates of the heap in CopyCell in this global specification, we would have to make sure that the location we update does not interfere with each of the assertions  $\text{A}_{\text{UFWD}-a}$ ,  $\text{A}_{\text{FWD}}$ , etc. Essentially the same remark can be made about the proof in Section 2.6.1 for INIT.

Like in Section 2.6.1, we must now show that  $I'$  follows from the conclusion in the derivation above. The proof of this, however, is for the most part completely analogous to the proof there (if one replaces  $\text{root}$  by  $a$ ). Therefore, we omit it here; the diligent reader may find a proof in Appendix 2.F.

We therefore conclude that the first part of the specification (2.26) for the **while**-loop holds. The treatment of the other half will not be as detailed as this one, since the proofs are completely analogous for the most part. However, the specification for  $\text{scan} := \text{scan} + 8$  needs an argument.

### After ScanCdr

We show that the invariant  $I$  is established after running  $\text{ScanCdr}; \text{scan} := \text{scan} + 8$  in a state in which  $I'$  holds. We omit the detailed proof for  $\text{ScanCdr}$ , since it is analogous to that of  $\text{ScanCar}$ . One can obtain the specification

$$\left\{ \begin{array}{l} \{I'\} \\ \text{ScanCdr} \\ \left( \begin{array}{l} I_{\text{pure}} \wedge \text{scan} \neq \text{free} \wedge \\ ((A_{\text{UFW}} * A_{\text{FWD}} * A_{\text{FIN}} * A_{\text{UFIN}-\text{scan}} * A_{\text{FREE}}) * \\ (\exists y. (\text{scan}, y) \in \varphi \odot (\text{head} \circ \varphi^\dagger) \wedge \text{scan} \mapsto y) * \\ (\exists y'. (\text{scan}, y') \in \varphi \odot (\text{tail} \circ \varphi^\dagger) \wedge \text{scan} + 4 \mapsto y')) \end{array} \right) \end{array} \right\}$$

Letting  $A$  be the conclusion in the above specification, we must show that

$$\left\{ \begin{array}{l} \{A\} \\ \text{scan} := \text{scan} + 8 \\ \{I\} \end{array} \right\} \quad (2.33)$$

holds. Intuitively, this specification holds because by increasing  $\text{scan}$ , we move the border between the intervals  $\text{FIN}$  and  $\text{UFIN}$ , and so the cell that has been scanned by the current iteration of the **while** loop moves from  $\text{UFIN}$  to  $\text{FIN}$ . There is a formal derivation of (2.33) in Appendix 2.G.

This means that the specification (2.26) holds, as desired, and hence we can conclude

**Theorem 2.26.** *The implementation  $GC^*$  of Cheney's algorithm in Appendix 2.A is a correct copying garbage collector in the sense of Definition 2.16.*

## 2.7 Conceptual Remarks

In this section, we discuss our proof at the meta-level. In particular, we illustrate what we gained from using separation logic, and in particular local reasoning. We also discuss our extensions of standard separation logic.

### 2.7.1 Benefits of Local Reasoning

Given that one of the aims of this paper is to demonstrate the power of separation logic, it is natural to ask to what extent local reasoning helped in the proof, and what we could have done without it.

A “brute force” way of answering this question would be to present a proof of our implementation that does not use local reasoning, and then compare the two proofs. That task, however, would be too tedious. Instead, we argue that proofs of “nontrivial” pointer-manipulating programs in general tend to be more complicated than proofs of the same programs in separation logic. We have also made remarks in the proof where we outline how local reasoning helped us (cf. Remarks 2.23 and 2.25).

A semantic analysis of pointer manipulations that does not use separation logic may be found in the papers [32] and [26]. In the first-mentioned paper, it is shown how to reason about programs that manipulate a list stored in the heap. The idea is to treat the list as a sequence of locations, and when a location is updated, the precondition is that the updated location is disjoint from the list. An important difference from our work is that lists are inductively defined; the structure we garbage collect can be cyclic, and thus cannot be defined inductively. In [26], proofs of several pointer manipulating programs are outlined. Some of these programs manipulate structures in the heap that are not inductively defined. In these cases, the approach is to determine the set of locations that is involved in the representation of a structure, and for each heap update, one has to ensure that the updated location is disjoint from this set, using non-interference predicates.

In contrast, local reasoning takes advantage of the  $*$  connective and the frame rule to state the required non-interference *implicitly*. The proof of the critical operations, namely the heap updates, are simple and do not require any non-interference predicates.

### 2.7.2 Remarks on Our Extension of Separation Logic

As mentioned, our proof uses an extension of standard separation logic with finite sets, relations, paths, and the iterated separating conjunction  $\forall_*$ . It is therefore appropriate to discuss the applicability of these extensions of standard separation logic in other settings.

As mentioned, we believe that the approach with sets and relations is applicable in other settings where the goal is to establish the existence of a relation between the heap before and after execution of a program, in particular if the structure represented in the heap is not definable by induction. The approach would be similar to that used here: given a snapshot of execution like that of Fig. 2.2, it might be possible to divide the heap into disjoint portions where the locations in each portion have a certain property. Given such a partition, one can then use the  $\forall_*$  connective to give an unambiguous description of the heap and use this in a specification, as we have done in our specification and proof.

On the other hand, it is quite possible that some of our extensions are of limited use in other settings. We use paths and the reachability predicates to express what is reachable before execution of our garbage collector. In a list reversal program, for example, this would be explicitly assumed in the specification. Also the  $\text{PtrRg}$  is used to take a crucial step in the proof of Lemma 2.21, but it is not straightforward to think of other proofs of programs where such a predicate would be crucial.

## 2.8 Related Work

There have been several proposals for ways of using types to manage the problem of reasoning about programs that manipulate imperative data structures [38, 121, 4, 96]. They are based on the idea that well-typed programs do not go wrong, but they are not aimed at giving proofs of *correctness*. In the work [38] on capabilities, traditional region calculus [127] is extended with an annotation of a capability to each region, and this gives

criteria to decide when it is safe to deallocate a region. In the setting of alias types [121], a static notion of constraint is used to describe the shape of the heap, and this is used to decide when it is safe to execute a program. In the work [4] on hierarchical storage, ideas from BI [102] and region calculi are used to give a type system with structure on locations. In [96] Petersen et. al. propose to use a type theory based on ordered linear logic as a foundation for defining how data is laid out in memory. The type theory in [96] builds upon a concrete allocation model such as the one provided by Cheney's copying garbage collector.

The first attempt of a formal correctness proof of a garbage collector was published in [39], where the problem "was selected as one of the most challenging – and hopefully, most instructive! – problems". The proof given there is informal and merely gives an idea of how to obtain a formal proof. Other informal proofs were published in [12] and [100]. The fact that a "mechanically verifiable proof would need all kinds of trivial invariants" was used to justify the informal approach. Russinoff [117] explored how great a detail that was needed for a formal proof and demonstrated that the proofs in [12] and [100] are fallacious. Wadler [129] gave an analysis and gave (semantic) proofs of complexity properties of a realtime garbage collection system, and moreover, there have been several formal verifications of correctness proofs of *abstract* versions of mark-and-sweep garbage collectors, using several different techniques [117, 60, 48, 37]. Note that many of the garbage collectors mentioned in this paragraph are *concurrent* and as such, more complicated to prove. Recent progress [84] paves the way for verifying concurrent garbage collectors in separation logic. A comparison between logical frameworks for verifying proofs has been performed by Burdy [31].

In their work on a type preserving garbage collector, Wang and Appel [130] transform well-typed programs into a form where they call a function, which acts as a garbage collector for the program. This function is designed such that it is well-typed in the target language, and thus is safe to execute. The approach of Wang and Appel guarantees safety, but not correctness of the garbage collector, and there is no treatment of cyclic data structures, since the user language does not create cyclic data structures. Monnier and Shao [75] combine ideas from region calculi and alias types in their work on typed regions and propose a programming language with a type system expressive enough to type a garbage collector, which is type preserving, generational, and handles cyclic data structures. As mentioned in Section 2.9, it is on the schedule for future work to extend our reasoning principles to a complete runtime system, and not only a garbage collector. Fluet and Wang [42] have implemented a safe runtime system for Scheme in Cyclone [63], including a copying garbage collector which is also based on Cheney's algorithm. Since Cyclone does not allow address arithmetic, they use a linked list to keep track of their queue in the NEW-space, whereas our implementation language uses address arithmetic and exploits the contiguous space NEW to implement the "implicit queue" that is used in the breadth-first search of Cheney's algorithm. Further, the type system of Cyclone guarantees memory safety, whereas our proof implies the existence of an isomorphism between the heaps before and after execution of the garbage collector. Another implementation of a complete runtime system for Scheme is the VLISP project, an overview of which can be found in [46] and [45]. As part of that project, a proof that a garbage collector implemented on a Garbage Collected Stored Bytecode Machine, establishes a "state correspondence" between the states before and after execution is included in a technical report [123]. However, this proof is carried out at a completely semantical level, using the operational semantics and the above-mentioned state correspondences,

which are similar to simulation relations. As mentioned in Section 2.3.2, we simply assume that there is only one cell in the root set before we garbage collect, and we have thus glossed over the issues of correctly constructing and keeping track of the root set. In both the implementation from [42] and the VLISP project, it is shown that the root sets are correctly maintained.

Recently, there has been a lot of work on Proof Carrying Code [81], [80]. The basic idea of a code producer submitting a proof of safety along with a program could, of course, be transferred to low-level programming languages, like the one used with separation logic. Nipkow's research group in Munich has developed a framework for formally verifying programs in traditional Hoare logic with arrays [68], and an extension to separation logic is at its early stages [131]. Also, Berdine *et al.* are working on Smallfoot [16], a model checker for low-level programs that uses ideas from separation logic. Once more developed, these would allow one to verify correctness proof mechanically and perhaps to ship the proof of a garbage collector along with proofs of programs using the garbage collector.

## 2.9 Conclusion and Future Work

We have specified and proved correct Cheney's copying garbage collector using local reasoning in an extension of standard separation logic. The specification and the proof are manageable because of local reasoning and we conclude that the idea of local reasoning scales well to such challenging algorithms.

We have extended separation logic with sets and relations, generalized the iterated separating conjunction and shown how these features can be used to specify naturally and prove correct an algorithm involving movement of cyclic data structures. We believe the methods used herein are of wider use and future work should include further experimentation with other subtle algorithms, such as those analyzed in [28] (and also, Bornat's methods might be applicable to Cheney's algorithm).

One the goals of this paper was to prove the simple variant of Cheney's collector, but it is natural to ask whether the approach of this work scales to more complex systems where the collected data have more complex types or where the collector is of a different type than stop-and-copy. We do not have a proof of such a collector, but we believe that an extension of the methodology presented here will serve as a basis for proofs of such algorithms. For example, in a more complex type system, the definition of a heap morphism needs to be refined, and presumably this will induce new notions in the logic.

One could argue that it is a weakness of separation logic that we had to extend it with the above mentioned new constructs, since it would be worrying if one would have to extend the logic for every new major proof. However, as explained in the recent article [20], one can see these extensions as simple *definitional* extensions of *higher-order* separation logic. That is, there is a single logic in which one can define, e.g., the finite sets and relations and then prove in the logic (rather than semantically) that properties such as those in Appendix 2.B hold.

Future work also includes studying how to specify and prove correct combinations of user level programs and runtime systems, as mentioned in the introduction. In his work on Foundational Proof Carrying Code [6], Appel suggests compiling high-level languages into the Typed Assembly Language [77]. Our work offers an alternative to this. We suggest compiling types from high-level languages into *garbage insensitive predicates*, in the sense of [33], and using our memory allocator and garbage collector as an

implementation of the `malloc` operation of TAL. By the nature of garbage insensitive predicates, we would have  $\{P\} GC \{P\}$  for these predicates, for any correct garbage collector  $GC$ , and thus predicates resulting from type-safety guarantees would be preserved by the garbage collector, as desired.

### **Acknowledgments**

The authors wish to thank Peter O'Hearn, Hongseok Yang, Richard Bornat, Cristiano Calcagno, Henning Niss, Martin Elsmann, and Mads Tofte for insightful discussions. We also thank the anonymous referees for their useful and detailed comments.

## 2.10 Appendix 2

### 2.A Implementation of Cheney's Algorithm

```

alloc(l, n1, n2) {
  if (free < maxFree)
    [free] := n1;
    [free + 4] := n2;
    free := free + 8;
    l := free - 8;
  else
    if (offset = startOld) then
      offset := startNew;
      maxFree := endNew;
    else
      offset := startOld;
      maxFree := endOld;
    fi;
  // Garbage Collection starts
  scan := offset;
  free := offset;
  FWD := ∅;
  UFWD := RCH;
  φ := ∅;
  t1 := [root];
  t2 := [root + 4];
  [free] := t1;
  [free + 4] := t2;
  [root] := free;
  FWD := FWD ∪ {root};
  UFWD := UFWD \ {root};
  φ := φ ∪ {(root, free)};
  free := free + 8;
  while ¬(scan = free)
    // ScanCar begins
    a := [scan];
    if (a mod 8 = 0)
      b := [a];
      if (b mod 8 = 0 ∧
          offset ≤ b ∧
          b ≤ maxFree)
        [scan] := b;
      else
        // CopyCell* begins
        t1 := [a];
        t2 := [a + 4];
        [free] := t1;
        [free + 4] := t2;
        [scan] := free;
        FWD := FWD ∪ {a};
        UFWD := UFWD \ {a};
        φ := φ ∪ {(a, free)};
        free := free + 8;
      fi;
    else skip
    fi;
  // ScanCdr ends
  scan := scan + 8;
od;
// Garbage Collection ends
root := offset;
alloc(l, n1, n2)
fi
}

```

## 2.B Set-Theoretic Rules

### Elementary rules

$$e \geq e' \rightarrow Itv(e, e') = \emptyset \quad (2.B1)$$

$$e \in m \rightarrow (m \setminus \{e\}) \cup \{e\} = m \quad (2.B2)$$

$$\neg(e \in m) \rightarrow (m \cup \{e\}) \setminus \{e\} = m \quad (2.B3)$$

$$e \in (m \setminus \{e'\}) \rightarrow e \in m \quad (2.B4)$$

$$\text{Ptr}(e) \rightarrow e \in m \cup \{e\} \quad (2.B5)$$

$$(e, e') \in r \setminus \{(e_1, e_2)\} \rightarrow (e, e') \in r \quad (2.B6)$$

$$\text{Ptr}(e) \wedge \text{Ptr}(e') \rightarrow (e, e') \in r \cup \{(e, e')\} \quad (2.B7)$$

$$\neg(e - 8 \in Itv(e, e')) \quad (2.B8)$$

$$e \leq e' - 8 \wedge \text{Ptr}(e') \rightarrow e' - 8 \in Itv(e, e') \quad (2.B9)$$

$$e \leq e' \wedge e' \leq e'' - 8 \wedge \text{Ptr}(e) \wedge \text{Ptr}(e') \wedge \text{Ptr}(e'') \rightarrow e' \in Itv(e, e'') \quad (2.B10)$$

$$m_1 \subseteq m_2 \rightarrow \#m_1 \leq \#m_2 \quad (2.B11)$$

$$e \in m \rightarrow \text{Ptr}(e) \quad (2.B12)$$

$$m_1 = Itv(e, e_1) \wedge m_2 = Itv(e, e_2) \wedge \#m_1 \leq \#m_2 \wedge \text{Ptr}(e_1) \wedge \text{Ptr}(e_2) \quad (2.B13)$$

$$\rightarrow e_1 \leq e_2$$

$$m_1 \subseteq m_2 \wedge m_2 \subseteq m_1 \rightarrow m_1 = m_2 \quad (2.B14)$$

$$\text{iso}(r, m_1, m_2) \wedge (e, e') \in \rho \circ r \circ r^\dagger \rightarrow (e, e') \in \rho \quad (2.B15)$$

$$e \bmod 8 = 0 \rightarrow \text{Ptr}(e) \quad (2.B16)$$

$$\text{Ptr}(e) \rightarrow \text{Ptr}(e - 8) \wedge \text{Ptr}(e + 8) \quad (2.B17)$$

$$(e, e') \in r \rightarrow (e', e) \in r^\dagger \quad (2.B18)$$

$$(e, e') \in r_1 \wedge (e', e'') \in r_2 \rightarrow (e, e'') \in r_2 \circ r_1 \quad (2.B19)$$

$$(e, e') \in (r_1 \circ r_2) \circ r_3 \leftrightarrow (e, e') \in r_1 \circ (r_2 \circ r_3) \quad (2.B20)$$

$$e \in Itv(e_1, e_2) \wedge e_2 \leq e_3 \rightarrow e \in Itv(e_1, e_3) \quad (2.B21)$$

$$m = \emptyset \rightarrow \forall x. \neg(x \in m) \quad (2.B22)$$

$$r = \emptyset \rightarrow \forall x, y. \neg((x, y) \in r) \quad (2.B23)$$

### Rules for iso:

$$\text{iso}(\emptyset, \emptyset, \emptyset) \quad (2.B24)$$

$$\text{Ptr}(e_1) \wedge \text{Ptr}(e_2) \wedge \neg(e_1 \in m_1) \wedge \neg(e_2 \in m_2) \wedge \text{iso}(r, m_1, m_2) \rightarrow \text{iso}(r \cup \{(e_1, e_2)\}, m_1 \cup \{e_1\}, m_2 \cup \{e_2\}) \quad (2.B25)$$

$$(e_1 \in m_1) \wedge (e_2 \in m_2) \wedge (e_1, e_2) \in r \wedge \text{iso}(r \setminus \{(e_1, e_2)\}, m_1 \setminus \{e_1\}, m_2 \setminus \{e_2\}) \rightarrow \text{iso}(r, m_1, m_2) \quad (2.B26)$$

$$\text{iso}(r, m_1, m_2) \wedge (e_1, e_2) \in r \rightarrow e_1 \in m_1 \wedge e_2 \in m_2 \quad (2.B27)$$

$$\text{iso}(r, m_1, m_2) \rightarrow \text{iso}(r^\dagger, m_2, m_1) \quad (2.B28)$$

$$\text{iso}(r, m_1, m_2) \wedge e \in m_1 \rightarrow \exists x. (e, x) \in r \wedge x \in m_2 \quad (2.B29)$$

$$\text{Tfun}(\rho, m_2) \wedge \text{iso}(r, m_1, m_2) \rightarrow \text{Tfun}(\rho \circ r, m_1) \quad (2.B30)$$

$$\text{iso}(r, m_1, m_2) \rightarrow \text{Tfun}(r, m_1) \quad (2.B31)$$

$$\text{iso}(r, m_1, m_2) \rightarrow \#m_1 = \#m_2 \quad (2.B32)$$

$$\text{iso}(r, m_1, m_2) \wedge \neg(e \in m_1) \rightarrow \forall x^{\text{int}}. \neg((e, x) \in r) \quad (2.B33)$$

### Rules for $\odot$

$$(e, e') \in \rho \wedge \neg \text{Ptr}(e') \rightarrow \forall r^{\text{frp}}. (e, e') \in r \odot \rho \quad (2.B34)$$

$$(e, e') \in \rho \wedge \text{Ptr}(e') \wedge (e', e'') \in r \rightarrow (e, e'') \in r \odot \rho \quad (2.B35)$$

$$\text{Tfun}(\rho, m) \wedge (e', e) \in \rho \wedge \text{Ptr}(e) \wedge e' \in m \wedge (e', e'') \in r \odot \rho \rightarrow (e, e'') \in r \quad (2.B36)$$

$$(e, e') \in r \odot \rho \rightarrow ((e, e') \in \rho \wedge \neg \text{Ptr}(e')) \vee (\exists x. \text{Ptr}(x) \wedge (e, x) \in \rho \wedge (x, e') \in r) \quad (2.B37)$$

$$(e, e') \in r \odot \rho \wedge (e, e'') \in r \odot \rho \wedge \text{Tfun}(r, m) \wedge \text{Tfun}(\rho, m') \wedge e \in m' \wedge ((\exists z. (e, z) \in \rho \wedge z \in m) \vee \neg \text{Ptr}(e')) \rightarrow e' = e'' \quad (2.B38)$$

$$(e, e') \in r \odot (\rho \circ \rho') \leftrightarrow (e, e') \in (r \odot \rho) \circ \rho' \quad (2.B39)$$

### Rules for PtrRg

$$\text{PtrRg}(r, m) \wedge \text{Ptr}(e) \wedge (e', e) \in r \rightarrow e \in m \quad (2.B40)$$

$$\text{PtrRg}(r, m) \rightarrow \text{PtrRg}(r \circ \rho, m) \quad (2.B41)$$

### Rules for $\subseteq$ and $\perp$

$$m \cup \emptyset = m \quad (2.B42)$$

$$m_1 \cup m_2 = m_2 \cup m_1 \quad (2.B43)$$

$$e \in m_1 \cup m_2 \rightarrow (e \in m_1 \vee e \in m_2) \quad (2.B44)$$

$$e \in m_1 \cup m_2 \wedge \neg(e \in m_1) \rightarrow e \in m_2 \quad (2.B45)$$

$$m_1 \cup m_2 = m \rightarrow m_1 \subseteq m \wedge m_2 \subseteq m \quad (2.B46)$$

$$m_1 \perp m_2 \rightarrow \forall x. \neg(x \in m_1 \wedge x \in m_2) \quad (2.B47)$$

$$m_1 \cup m_2 = m \wedge e \in m_1 \rightarrow (m_1 \setminus \{e\}) \cup (m_2 \cup \{e\}) = m \quad (2.B48)$$

$$(m_1 \setminus \{e\}) \cup (m_2 \cup \{e\}) = m \wedge e \in m_1 \rightarrow m_1 \cup m_2 = m$$

$$(\forall x. (x \in m_1) \rightarrow (x \in m_2)) \leftrightarrow m_1 \subseteq m_2 \quad (2.B49)$$

$$m_1 \perp m_2 \wedge m'_1 \subseteq m_1 \rightarrow m'_1 \perp m_2 \quad (2.B50)$$

## 2.C Formal Proof of INIT\*

We need to show that  $\text{I}$  follows from the conclusion of the derivation in Section 2.6.1.

$\text{I}$  can be viewed as a conjunction  $\text{I} \equiv \text{l}_1 \wedge \cdots \wedge \text{l}_k$  of assertions, where some of the  $\text{l}_i$  are pure and one is not pure; let us say that  $\text{l}_k$  is the impure part of  $\text{I}$ , and write  $\text{l}_k$  on the form  $A_1 * \cdots * A_m$ . Similarly, the conclusion in the derivation has the form  $\text{l}'_1 \wedge \cdots \wedge \text{l}'_{k'}$  where  $\text{l}'_i$  are pure for  $i \in \{1, \dots, k' - 1\}$  and where  $\text{l}'_{k'} \equiv A'_1 * \cdots * A'_m$ .

We show that each of  $\text{l}_1, \dots, \text{l}_{k-1}$  follow from  $\text{l}'_1 \wedge \cdots \wedge \text{l}'_{k'-1}$  and each of the  $A_i$  from  $\text{l}'_1 \wedge \cdots \wedge \text{l}'_{k'-1} \wedge A'_i$ ; this is sufficient by Lemma 2.10. We start by proving the pure conjuncts of  $\text{I}$  from the conclusion in the derivation in Section 2.6.1.

- $\text{iso}(\varphi, \text{FWD}, \text{BUSY})$ . This follows from

$$\text{iso}(\varphi \setminus \{(\text{root}, \text{free} - 8)\}, \text{FWD} \setminus \{\text{root}\}, \text{BUSY} \setminus \{(\text{free} - 8)\}) \wedge \text{root} \in \text{FWD} \wedge (\text{free} - 8) \in \text{BUSY} \wedge (\text{root}, \text{free} - 8) \in \varphi \wedge \text{Ptr}(\text{free} - 8),$$

(2.B26), and (2.B9).

- $\text{ALIVE} = \text{FWD} \cup \text{UFWD}$  follows from

$$(\text{FWD} \setminus \{\text{root}\}) \cup (\text{UFWD} \cup \{\text{root}\}) = \text{ALIVE} \wedge \text{root} \in \text{FWD}$$

and (2.B48).

- $I_c \wedge (\text{ALIVE} \perp \text{NEW}) \wedge \#\text{ALIVE} \leq \#\text{NEW} \wedge \text{root} \in \text{FWD} \wedge \text{offset} \leq \text{scan}$  is part of the conclusion in Section 2.6.1.
- $\text{scan} \leq \text{free}$ . Follows from  $\text{scan} = \text{offset} \wedge \text{free} - 8 = \text{offset}$ .
- $\text{Ptr}(\text{scan}) \wedge \text{Ptr}(\text{free})$ . Follows from (2.B17) and  $\text{Ptr}(\text{offset}) \wedge \text{scan} = \text{offset} \wedge \text{Ptr}(\text{free} - 8)$ .
- $(\text{root}, \text{offset}) \in \varphi$  follows from  $(\text{root}, \text{free} - 8) \in \varphi$  and  $\text{offset} = \text{free} - 8$ .

For the impure parts, the argument is a bit more complicated. We deal with each of the parts in the iterated separating conjunction  $(A_{\text{UFWD}} * A_{\text{FWD}} * A_{\text{FIN}} * A_{\text{UFIN}} * A_{\text{FREE}})$  separately.

For UFWD, we have

$$\begin{aligned} & \neg(\text{root} \in \text{UFWD}) \wedge \\ & (\forall_* x \in ((\text{UFWD} \cup \{\text{root}\}) \setminus \{\text{root}\}). \\ & \quad ((\exists y. (x, y) \in \text{head} \wedge x \mapsto y) * (\exists y'. (x, y') \in \text{tail} \wedge x + 4 \mapsto y'))) \\ & \Downarrow \\ & \frac{((\text{UFWD} \cup \{\text{root}\}) \setminus \{\text{root}\}) = \text{UFWD} \wedge}{(\forall_* x \in ((\text{UFWD} \cup \{\text{root}\}) \setminus \{\text{root}\}).} \\ & \quad ((\exists y. (x, y) \in \text{head} \wedge x \mapsto y) * (\exists y'. (x, y') \in \text{tail} \wedge x + 4 \mapsto y'))) \\ & \Downarrow \\ & (\forall_* x \in \underline{\text{UFWD}}. ((\exists y. (x, y) \in \text{head} \wedge x \mapsto y) * (\exists y'. (x, y') \in \text{tail} \wedge x + 4 \mapsto y'))) \\ & \equiv \\ & A_{\text{UFWD}}, \end{aligned}$$

where the two implications follow from (2.B3) and (2.10), respectively.

For FWD,

$$\begin{aligned}
& (\text{root}, \text{free} - 8) \in \varphi \wedge \text{root} \in \text{FWD} \wedge \\
& ((\forall_* x \in (\text{FWD} \setminus \{\text{root}\}). (\exists y. (x, y) \in \varphi \setminus \{(\text{root}, \text{free} - 8)\} \wedge p \mapsto y, -)) * \\
& (\text{root} \mapsto \text{free} - 8, -)) \\
& \Downarrow \\
& \text{root} \in \text{FWD} \wedge \\
& ((\forall_* x \in (\text{FWD} \setminus \{\text{root}\}). (\exists y. (x, y) \in \varphi \wedge x \mapsto y, -)) * \\
& (\text{root} \mapsto \text{free} - 8, - \wedge (\text{root}, \text{free} - 8) \in \varphi)) \\
& \Downarrow \\
& \text{root} \in \text{FWD} \wedge \\
& ((\forall_* x \in (\text{FWD} \setminus \{\text{root}\}). (\exists y. (x, y) \in \varphi \wedge x \mapsto y, -)) * \\
& (\exists y. \text{root} \mapsto y, - \wedge (\text{root}, y) \in \varphi)) \\
& \Downarrow \\
& \forall_* x \in \underline{\text{FWD}}. (\exists y. (x, y) \in \varphi \wedge x \mapsto y, -) \\
& \parallel \\
& A_{\text{FWD}}
\end{aligned}$$

The first implication follows from the rule for pure assertions in Remark 2.7, from (2.B6), and Lemma 2.9. The third implication is an instance of (2.14).

For FIN, it is easiest to note that the condition  $\text{scan} = \text{offset}$  in the conclusion of the derivation above makes both of the assertions about FIN equivalent to  $\text{emp}$ .

The following derivation takes care of FREE (we implicitly use (2.B8)):

$$\begin{aligned}
& \neg(\text{free} - 8 \in \text{FREE}) \wedge \\
& \forall_* x \in ((\text{FREE} \cup \{(\text{free} - 8)\}) \setminus \{(\text{free} - 8)\}). x. \rightarrow -, - \\
& \Downarrow \\
& \underline{\text{FREE} = (\text{FREE} \cup \{(\text{free} - 8)\}) \setminus \{(\text{free} - 8)\}} \wedge \\
& \forall_* x \in ((\text{FREE} \cup \{(\text{free} - 8)\}) \setminus \{(\text{free} - 8)\}). x. \rightarrow -, - \\
& \Downarrow \\
& \forall_* x \in \underline{\text{FREE}}. x \mapsto -, - \\
& \parallel \\
& A_{\text{FREE}}
\end{aligned}$$

The first implication here is an instance of (2.B3), and the second implication follows from (2.10).

Finally, for UFIN, we use (2.B9) and get

$$\begin{aligned}
& (\text{root}, \text{free} - 8) \in \varphi \wedge (\text{root}, t_1) \in \text{head} \wedge (\text{root}, t_2) \in \text{tail} \wedge \\
& \text{Ptr}(\text{free} - 8) \wedge (\text{free} - 8) \in \text{UFIN} \wedge \\
& ((\forall_* x \in (\text{UFIN} \setminus \{\text{free} - 8\}). ((\exists y. (x, y) \in \text{head} \circ (\varphi \setminus \{(\text{root}, \text{free} - 8)\})^\dagger \wedge x \mapsto y) * \\
& \quad (\exists y'. (x, y') \in \text{tail} \circ (\varphi \setminus \{(\text{root}, \text{free} - 8)\})^\dagger \wedge x + 4 \mapsto y')) * \\
& \quad (\text{free} - 8 \mapsto t_1, t_2)) \\
(1) \Downarrow & \\
& (\text{free} - 8, \text{root}) \in \varphi^\dagger \wedge (\text{root}, t_1) \in \text{head} \wedge (\text{root}, t_2) \in \text{tail} \wedge \\
& \text{Ptr}(\text{free} - 8) \wedge (\text{free} - 8) \in \text{UFIN} \wedge \\
& ((\forall_* x \in (\text{UFIN} \setminus \{\text{free} - 8\}). ((\exists y. (x, y) \in \text{head} \circ \underline{\varphi^\dagger} \wedge x \mapsto y) * \\
& \quad (\exists y'. (x, y') \in \text{tail} \circ \underline{\varphi^\dagger} \wedge x + 4 \mapsto y')) * \\
& \quad (\text{free} - 8 \mapsto t_1) * ((\text{free} - 8) + 4 \mapsto t_2)) \\
(2) \Downarrow & \\
& (\text{free} - 8, t_1) \in \text{head} \circ \varphi^\dagger \wedge (\text{free} - 8, t_2) \in \text{tail} \circ \varphi^\dagger \wedge (\text{free} - 8) \in \text{UFIN} \wedge \\
& ((\forall_* x \in (\text{UFIN} \setminus \{\text{free} - 8\}). ((\exists y. (x, y) \in \text{head} \circ \varphi^\dagger \wedge x \mapsto y) * \\
& \quad (\exists y'. (x, y') \in \text{tail} \circ \varphi^\dagger \wedge x + 4 \mapsto y')) * \\
& \quad (\text{free} - 8 \mapsto t_1) * ((\text{free} - 8) + 4 \mapsto t_2)) \\
(3) \Downarrow & \\
& (\text{free} - 8) \in \text{UFIN} \wedge \\
& ((\forall_* x \in (\text{UFIN} \setminus \{\text{free} - 8\}). ((\exists y. (x, y) \in \text{head} \circ \varphi^\dagger \wedge x \mapsto y) * \\
& \quad (\exists y'. (x, y') \in \text{tail} \circ \varphi^\dagger \wedge x + 4 \mapsto y')) * \\
& \quad (\text{free} - 8 \mapsto t_1 \wedge (\text{free} - 8, t_1) \in \text{head} \circ \varphi^\dagger) * \\
& \quad ((\text{free} - 8) + 4 \mapsto t_2 \wedge \underline{(\text{free} - 8, t_2) \in \text{tail} \circ \varphi^\dagger})) \\
(4) \Downarrow & \\
& (\text{free} - 8) \in \text{UFIN} \wedge \\
& ((\forall_* x \in (\text{UFIN} \setminus \{\text{free} - 8\}). ((\exists y. (x, y) \in \text{head} \circ \varphi^\dagger \wedge x \mapsto y) * \\
& \quad (\exists y'. (x, y') \in \text{tail} \circ \varphi^\dagger \wedge x + 4 \mapsto y')) * \\
& \quad (\exists y. \text{free} - 8 \mapsto y \wedge (\text{free} - 8, y) \in \text{head} \circ \varphi^\dagger) * \\
& \quad \underline{(\exists y'. (\text{free} - 8) + 4 \mapsto y' \wedge (\text{free} - 8, y') \in \text{tail} \circ \varphi^\dagger)}) \\
(5) \Downarrow & \\
& \forall_* x \in \underline{\text{UFIN}}. ((\exists y. (x, y) \in \text{head} \circ \varphi^\dagger \wedge x \mapsto y) * \\
& \quad (\exists y'. (x, y') \in \text{tail} \circ \varphi^\dagger \wedge x + 4 \mapsto y')) \\
& \parallel \\
& A_{\text{UFIN}}
\end{aligned}$$

Here, the first implication uses (2.B6), (2.B18), and Lemma 2.9. The second follows from (2.B19), and the third implication follows from purity. Finally, the last implication is an instance of the rule (2.14).

## 2.D Formal Proof of Lemma 2.21

The assertion  $(\text{scan}, a) \in \text{head} \circ \varphi^\dagger \wedge \text{Ptr}(a) \wedge \text{PtrRg}(\text{head}, \text{ALIVE})$  implies  $a \in \text{ALIVE}$  by (2.B41) and (2.B40), so  $A$  implies  $a \in \text{ALIVE}$ . By (2.B45),

$$a \in \text{ALIVE} \wedge \text{FWD} \cup \text{UFW} = \text{ALIVE} \wedge \neg(a \in \text{UFW}) \rightarrow a \in \text{FWD},$$

so we assume  $a \in \text{UFWD}$  and derive a contradiction, i.e., we show  $A \wedge (a \in \text{UFWD}) \rightarrow F$ . By (2.16), it suffices to show (since  $F$  is a intuitionistic assertion)

$$\begin{aligned} & A_{\text{UFWD}} \wedge (a \leftrightarrow b) \wedge \text{PtrRg}(\text{head}, \text{ALIVE}) \wedge \text{Ptr}(b) \wedge \\ & \text{ALIVE} \perp \text{NEW} \wedge b \in \text{NEW} \wedge a \in \text{UFWD} \rightarrow F \end{aligned}$$

We have

$$\begin{aligned} & A_{\text{UFWD}} \wedge a \leftrightarrow b \wedge \text{PtrRg}(\text{head}, \text{ALIVE}) \wedge \text{Ptr}(b) \wedge \\ & \text{ALIVE} \perp \text{NEW} \wedge b \in \text{NEW} \wedge a \in \text{UFWD} \\ (1) \downarrow & \\ & \frac{(A_{\text{UFWD}-a} * (\exists y. (a, y) \in \text{head} \wedge a \mapsto y) * (\exists y'. (a, y') \in \text{tail} \wedge a + 4 \mapsto y')) \wedge}{a \leftrightarrow b \wedge \text{PtrRg}(\text{head}, \text{ALIVE}) \wedge \text{Ptr}(b) \wedge \text{ALIVE} \perp \text{NEW} \wedge b \in \text{NEW}} \\ (2) \downarrow & \\ & \frac{(A_{\text{UFWD}-a} * (a + 4 \mapsto -) * (\exists y. (a, y) \in \text{head} \wedge a \mapsto y)) \wedge}{a \leftrightarrow b \wedge \text{PtrRg}(\text{head}, \text{ALIVE}) \wedge \text{Ptr}(b) \wedge \text{ALIVE} \perp \text{NEW} \wedge b \in \text{NEW}} \\ (3) \downarrow & \\ & \frac{(A_{\text{UFWD}-a} * (a + 4 \mapsto -) * ((a, b) \in \text{head} \wedge a \mapsto b)) \wedge}{\text{PtrRg}(\text{head}, \text{ALIVE}) \wedge \text{Ptr}(b) \wedge \text{ALIVE} \perp \text{NEW} \wedge b \in \text{NEW}} \\ (4) \downarrow & \\ & \frac{(A_{\text{UFWD}-a} * (a + 4 \mapsto -) * (a \mapsto b)) \wedge}{(a, b) \in \text{head} \wedge \text{PtrRg}(\text{head}, \text{ALIVE}) \wedge \text{Ptr}(b) \wedge \text{ALIVE} \perp \text{NEW} \wedge b \in \text{NEW}} \\ (5) \downarrow & \\ & \underline{b \in \text{ALIVE}} \wedge \text{ALIVE} \perp \text{NEW} \wedge b \in \text{NEW} \\ (6) \downarrow & \\ & F \end{aligned}$$

The first of these implications follows from (2.14), the third follows from (2.15), the fourth from purity, the fifth from (2.B40), and the last follows from (2.B47).  $\square$

## 2.E Formal Proof of Lemma 2.24

First,

$$\begin{aligned} & I_a \wedge \text{Ptr}(a) \wedge (a \leftrightarrow b) \wedge \neg(b \in \text{NEW}) \\ \downarrow & \\ & (\text{scan}, a) \in \text{head} \circ \varphi^\dagger \wedge \text{PtrRg}(\text{head}, \text{ALIVE}) \wedge \\ & \text{Ptr}(a) \wedge \text{FWD} \cup \text{UFWD} = \text{ALIVE} \\ \downarrow & \\ & \underline{a \in \text{ALIVE}} \wedge \text{FWD} \cup \text{UFWD} = \text{ALIVE} \\ \downarrow & \\ & a \in \text{FWD} \vee a \in \text{UFWD} \end{aligned}$$

The second implication follows from (2.B40) and (2.B41), and the third is by (2.B44). So, as in the proof of Lemma 2.21, we assume  $a \in \text{FWD}$  and derive a contradiction, i.e., we show  $(a \in \text{FWD}) \wedge A \rightarrow F$ . By (2.16) and Lemma 2.19, the following derivation

establishes this.

$$\begin{aligned}
& A_{\text{FWD}} \wedge a \hookrightarrow b \wedge \text{iso}(\varphi, \text{FWD}, \text{BUSY}) \wedge \neg(b \in \text{NEW}) \wedge \text{free} \leq \text{maxFree} \wedge a \in \text{FWD} \\
(1) \downarrow & (A_{\text{FWD}-a} * (\exists y. (a, y) \in \varphi \wedge a \mapsto y, -)) \wedge \\
& a \hookrightarrow b \wedge \text{iso}(\varphi, \text{FWD}, \text{BUSY}) \wedge \neg(b \in \text{NEW}) \wedge \text{free} \leq \text{maxFree} \\
(2) \downarrow & (A_{\text{FWD}-a} * (a + 4 \mapsto -) * (\exists y. (a, y) \in \varphi \wedge a \mapsto y)) \wedge \\
& a \hookrightarrow b \wedge \text{iso}(\varphi, \text{FWD}, \text{BUSY}) \wedge \neg(b \in \text{NEW}) \wedge \text{free} \leq \text{maxFree} \\
(3) \downarrow & (A_{\text{FWD}-a} * (a + 4 \mapsto -) * ((a, b) \in \varphi \wedge a \mapsto b)) \wedge \\
& \text{iso}(\varphi, \text{FWD}, \text{BUSY}) \wedge \neg(b \in \text{NEW}) \wedge \text{free} \leq \text{maxFree} \\
(4) \downarrow & (A_{\text{FWD}-a} * (a + 4 \mapsto -) * (a \mapsto b)) \wedge \\
& (a, b) \in \varphi \wedge \text{iso}(\varphi, \text{FWD}, \text{BUSY}) \wedge \neg(b \in \text{NEW}) \wedge \text{free} \leq \text{maxFree} \\
(5) \downarrow & (a, b) \in \varphi \wedge \text{iso}(\varphi, \text{FWD}, \text{BUSY}) \wedge \neg(b \in \text{NEW}) \wedge \text{free} \leq \text{maxFree} \\
(6) \downarrow & b \in \text{BUSY} \wedge \neg(b \in \text{NEW}) \wedge \text{free} \leq \text{maxFree} \\
(7) \downarrow & b \in \text{Itv}(\text{offset}, \text{free}) \wedge \neg(b \in \text{Itv}(\text{offset}, \text{maxFree})) \wedge \text{free} \leq \text{maxFree} \\
(8) \downarrow & \text{F}
\end{aligned}$$

The first of these implications follows from (2.14), the second is a matter of notation. The third implication is an instance of (2.15), and the next comes from purity. The sixth implication in the derivation follows from (2.B27), and the last is by (2.B21). This shows the lemma.  $\square$

## 2.F Last Step of Specification for CopyCell\*

We need to show that  $I'$  follows from the conclusion of the specification for CopyCell\* from Section 2.6.2. As mentioned, this proof is similar to the proof in Section 2.6.1. In particular, the pure part of  $I'$  follows from the pure part of the conclusion above by the same argument as in Section 2.6.1, and the same argument goes for the separating conjunction over the sets FWD, UFWD, and FREE, and for the location scan + 4. Thus, if we let  $B$  be the pure part of the conclusion of the global specification in Section 2.6.2, what is left to show is that

$$\begin{aligned}
& B \wedge (((\text{free} - 8) \mapsto t_1, t_2) * (\text{scan} \mapsto \text{free} - 8) * \\
& (\forall_* x \in ((\text{UFIN} \setminus \{\text{scan}\}) \setminus \{\text{free} - 8\})). \\
& ((\exists y. (x, y) \in \text{head} \circ (\varphi \setminus \{(a, \text{free} - 8)\})^\dagger \wedge x \mapsto y) * \\
& (\exists y'. (x, y') \in \text{tail} \circ (\varphi \setminus \{(a, \text{free} - 8)\})^\dagger \wedge x + 4 \mapsto y')))
\end{aligned} \tag{2.F1}$$

implies

$$\begin{aligned}
& (\forall_* x \in (\text{UFIN} \setminus \{\text{scan}\})). \\
& ((\exists y. (x, y) \in \text{head} \circ \varphi^\dagger \wedge x \mapsto y) * \\
& (\exists y'. (x, y') \in \text{tail} \circ \varphi^\dagger \wedge x + 4 \mapsto y'))) * \\
& (\exists y. (\text{scan}, y) \in \varphi \odot (\text{head} \circ \varphi^\dagger) \wedge \text{scan} \mapsto y),
\end{aligned} \tag{2.F2}$$

and that

$$\begin{aligned}
& B \wedge \\
& (\forall_* x \in \text{FIN}. \\
& ((\exists y. (x, y) \in \varphi \setminus \{(a, \text{free} - 8)\} \odot (\text{head} \circ (\varphi \setminus \{(a, \text{free} - 8)\})^\dagger) \wedge x \mapsto y) * \\
& (\exists y'. (x, y') \in (\varphi \setminus \{(a, \text{free} - 8)\}) \odot (\text{tail} \circ (\varphi \setminus \{(a, \text{free} - 8)\})^\dagger) \wedge x + 4 \mapsto y'))))
\end{aligned}$$

implies

$$\begin{aligned}
& B \wedge \\
& (\forall_* x \in \text{FIN}. ((\exists y. (x, y) \in \varphi \odot (\text{head} \circ \varphi^\dagger) \wedge x \mapsto y) * \\
& (\exists y'. (x, y') \in \varphi \odot (\text{tail} \circ \varphi^\dagger) \wedge x + 4 \mapsto y'))).
\end{aligned}$$

The last of these follows from (2.B6) and Lemma 2.9. For the first, we have

$$\begin{aligned}
& (a, t_1) \in \text{head} \wedge (a, t_2) \in \text{tail} \wedge (a, \text{free} - 8) \in \varphi \wedge \\
& (\text{scan}, a) \in \text{head} \circ (\varphi \setminus \{(a, \text{free} - 8)\})^\dagger \wedge \text{Ptr}(a) \wedge \\
& ((\forall_* x \in ((\text{UFIN} \setminus \{\text{scan}\}) \setminus \{\text{free} - 8\}). \\
& ((\exists y. (x, y) \in \text{head} \circ (\varphi \setminus \{(a, \text{free} - 8)\})^\dagger \wedge x \mapsto y) * \\
& (\exists y'. (x, y') \in \text{tail} \circ (\varphi \setminus \{(a, \text{free} - 8)\})^\dagger) \wedge x + 4 \mapsto y')) * \\
& (\text{scan} \mapsto \text{free} - 8) * (\text{free} - 8 \mapsto t_1, t_2)) \\
(1) \downarrow & \\
& \frac{(\text{free} - 8, t_1) \in \text{head} \circ \varphi^\dagger \wedge (\text{free} - 8, t_2) \in \text{tail} \circ \varphi^\dagger \wedge \\
& (\text{scan}, \text{free} - 8) \in \varphi \odot (\text{head} \circ \varphi^\dagger) \wedge \\
& (\forall_* x \in ((\text{UFIN} \setminus \{\text{scan}\}) \setminus \{\text{free} - 8\}). \\
& ((\exists y. (x, y) \in \text{head} \circ \varphi^\dagger \wedge x \mapsto y) * \\
& (\exists y'. (x, y') \in \text{tail} \circ \varphi^\dagger \wedge x + 4 \mapsto y')) * \\
& (\text{scan} \mapsto \text{free} - 8) * ((\text{free} - 8) \mapsto t_1) * ((\text{free} - 8) + 4 \mapsto t_2))}{(2) \downarrow} \\
& \forall_* x \in ((\text{UFIN} \setminus \{\text{scan}\}) \setminus \{\text{free} - 8\}). \\
& ((\exists y. (x, y) \in \text{head} \circ \varphi^\dagger \wedge x \mapsto y) * \\
& (\exists y'. (x, y') \in \text{tail} \circ \varphi^\dagger \wedge x + 4 \mapsto y')) * \\
& (\text{scan} \mapsto \text{free} - 8 \wedge \frac{(\text{scan}, \text{free} - 8) \in \varphi \odot (\text{head} \circ \varphi^\dagger) * \\
& ((\text{free} - 8) \mapsto t_1 \wedge (\text{free} - 8, t_1) \in \text{head} \circ \varphi^\dagger) * \\
& ((\text{free} - 8) + 4 \mapsto t_2 \wedge (\text{free} - 8, t_2) \in \text{tail} \circ \varphi^\dagger)}{(3) \downarrow} \\
& \forall_* x \in ((\text{UFIN} \setminus \{\text{scan}\}) \setminus \{\text{free} - 8\}). \\
& ((\exists y. (x, y) \in \text{head} \circ \varphi^\dagger \wedge x \mapsto y) * \\
& (\exists y'. (x, y') \in \text{tail} \circ \varphi^\dagger \wedge x + 4 \mapsto y')) * \\
& \frac{(\exists y. \text{scan} \mapsto y \wedge (\text{scan}, y) \in \varphi \odot (\text{head} \circ \varphi^\dagger)) * \\
& (\exists y. (\text{free} - 8) \mapsto y \wedge (\text{free} - 8, y) \in \text{head} \circ \varphi^\dagger) * \\
& (\exists y'. (\text{free} - 8) + 4 \mapsto y' \wedge (\text{free} - 8, y') \in \text{tail} \circ \varphi^\dagger)}{(4) \downarrow} \\
& \forall_* x \in (\text{UFIN} \setminus \{\text{scan}\}). \\
& ((\exists y. (x, y) \in \text{head} \circ \varphi^\dagger \wedge x \mapsto y) * \\
& (\exists y'. (x, y') \in \text{tail} \circ \varphi^\dagger \wedge x + 4 \mapsto y')) * \\
& (\exists y. \text{scan} \mapsto y \wedge (\text{scan}, y) \in \varphi \odot (\text{head} \circ \varphi^\dagger))
\end{aligned}$$

The first implication follows from (2.B6), ordinary composition of relations (2.B19), the rule (2.B35) for the special relation composition  $\odot$ , and from Lemma 2.9. The second implication follows from purity, and the last implication from (2.14). We have thus obtained (2.F1)  $\Rightarrow$  (2.F2).

## 2.G Formal Derivation of (2.33)

By the rule for assignment and obvious rules for intervals, we get

$$\begin{array}{l}
\{A\} \\
\Downarrow \\
\left\{ \begin{array}{l}
I_{\text{pure}} \wedge \text{scan} \neq \text{free} \wedge \\
((A_{\text{UFWD}} * A_{\text{FWD}} * A_{\text{FREE}}) * \\
(\forall_* x \in \text{FIN}. ((\exists y. (x, y) \in \varphi \odot (\text{head} \circ \varphi^\dagger) \wedge x \mapsto y) * \\
(\exists y'. (x, y') \in \varphi \odot (\text{tail} \circ \varphi^\dagger) \wedge x + 4 \mapsto y')) * \\
(\forall_* x \in (\text{UFIN} \setminus \{\text{scan}\}). ((\exists y. (x, y) \in \text{head} \circ \varphi^\dagger \wedge x \mapsto y) * \\
(\exists y'. (x, y') \in \text{tail} \circ \varphi^\dagger \wedge x + 4 \mapsto y')) * \\
(\exists y. (\text{scan}, y) \in \varphi \odot (\text{head} \circ \varphi^\dagger) \wedge \text{scan} \mapsto y) * \\
(\exists y'. (\text{scan}, y') \in \varphi \odot (\text{tail} \circ \varphi^\dagger) \wedge (\text{scan} + 4) \mapsto y'))
\end{array} \right\} \\
\text{scan} := \text{scan} + 8 \\
\left\{ \begin{array}{l}
I_c \wedge \text{ALIVE} \perp \text{NEW} \wedge \#\text{ALIVE} \leq \#\text{NEW} \wedge \text{root} \in \text{FWD} \wedge \\
\text{iso}(\varphi, \text{FWD}, \text{BUSY}) \wedge \text{FWD} \cup \text{UFWD} = \text{ALIVE} \\
\text{scan} - 8 \leq \text{free} \wedge \text{offset} \leq \text{scan} - 8 \wedge \text{Ptr}(\text{free}) \wedge \text{Ptr}(\text{scan} - 8) \wedge \\
((A_{\text{UFWD}} * A_{\text{FWD}} * A_{\text{FREE}}) * \\
(\forall_* x \in (\text{FIN} \setminus \{\text{scan} - 8\}). ((\exists y. (x, y) \in \varphi \odot (\text{head} \circ \varphi^\dagger) \wedge x \mapsto y) * \\
(\exists y'. (x, y') \in \varphi \odot (\text{tail} \circ \varphi^\dagger) \wedge x + 4 \mapsto y')) * \\
(\forall_* x \in ((\text{UFIN} \cup \{\text{scan} - 8\}) \setminus \{\text{scan} - 8\}). \\
((\exists y. (x, y) \in \text{head} \circ \varphi^\dagger \wedge x \mapsto y) * \\
(\exists y'. (x, y') \in \text{tail} \circ \varphi^\dagger \wedge x + 4 \mapsto y')) * \\
(\exists y. (\text{scan} - 8, y) \in \varphi \odot (\text{head} \circ \varphi^\dagger) \wedge (\text{scan} - 8) \mapsto y) * \\
(\exists y'. (\text{scan} - 8, y') \in \varphi \odot (\text{tail} \circ \varphi^\dagger) \wedge ((\text{scan} - 8) + 4) \mapsto y'))
\end{array} \right\} \\
\Downarrow \\
\left\{ \begin{array}{l}
I_c \wedge \text{ALIVE} \perp \text{NEW} \wedge \#\text{ALIVE} \leq \#\text{NEW} \wedge \text{root} \in \text{FWD} \wedge \\
\text{iso}(\varphi, \text{FWD}, \text{BUSY}) \wedge \text{FWD} \cup \text{UFWD} = \text{ALIVE} \\
\text{scan} - 8 \leq \text{free} \wedge \text{offset} \leq \text{scan} - 8 \wedge \text{Ptr}(\text{free}) \wedge \text{Ptr}(\text{scan} - 8) \wedge \\
((A_{\text{UFWD}} * A_{\text{FWD}} * A_{\text{FREE}}) * \\
(\forall_* x \in (\text{FIN} \setminus \{\text{scan} - 8\}). ((\exists y. (x, y) \in \varphi \odot (\text{head} \circ \varphi^\dagger) \wedge x \mapsto y) * \\
(\exists y'. (x, y') \in \varphi \odot (\text{tail} \circ \varphi^\dagger) \wedge x + 4 \mapsto y')) * \\
(\exists y. (\text{scan} - 8, y) \in \varphi \odot (\text{head} \circ \varphi^\dagger) \wedge (\text{scan} - 8) \mapsto y) * \\
(\exists y'. (\text{scan} - 8, y') \in \varphi \odot (\text{tail} \circ \varphi^\dagger) \wedge ((\text{scan} - 8) + 4) \mapsto y')) * \\
(\forall_* x \in ((\text{UFIN} \cup \{\text{scan} - 8\}) \setminus \{\text{scan} - 8\}). \\
((\exists y. (x, y) \in \text{head} \circ \varphi^\dagger \wedge x \mapsto y) * \\
(\exists y'. (x, y') \in \text{tail} \circ \varphi^\dagger \wedge x + 4 \mapsto y'))))
\end{array} \right\}
\end{array}$$

The specification step uses the rule for assignment.

Like in Section 2.6.1 and 2.6.2, we now have to show that the pure part of  $I$  follows from the pure part  $I_p''$  of the conclusion  $I''$  in the derivation above, and that the separating conjunction in  $I$  follows from that of  $I''$  and  $I_p''$ . The only problem in the pure part of  $I$  is to conclude

$$\text{Ptr}(\text{scan}) \wedge \text{scan} \leq \text{free} \wedge \text{offset} \leq \text{scan}$$

But this follows from

$$\text{Ptr}(\text{scan} - 8) \wedge \text{Ptr}(\text{free}) \wedge \text{scan} - 8 \neq \text{free} \wedge \text{scan} - 8 \leq \text{free} \wedge \text{offset} \leq \text{scan} - 8.$$

For the heap-dependent part of  $I$ , we see that  $A_{\text{UFWD}}$ ,  $A_{\text{FWD}}$ , and  $A_{\text{FREE}}$  follow directly from the corresponding parts of  $I''$ . So what is left to show is that

$$I''_p \wedge \\ (\forall_* x \in ((\text{UFIN} \cup \{\text{scan} - 8\}) \setminus \{\text{scan} - 8\}). ((\exists y. (x, y) \in \text{head} \circ \varphi^\dagger \wedge x \mapsto y) * \\ (\exists y'. (x, y') \in \text{tail} \circ \varphi^\dagger) \wedge x + 4 \mapsto y'))$$

implies

$$\forall_* x \in \text{UFIN}. ((\exists y. (x, y) \in \text{head} \circ \varphi^\dagger \wedge x \mapsto y) * \\ (\exists y'. (x, y') \in \text{tail} \circ \varphi^\dagger) \wedge x + 4 \mapsto y'),$$

and that

$$I''_p \wedge \\ ((\forall_* x \in (\text{FIN} \setminus \{\text{scan} - 8\}). ((\exists y. (x, y) \in \varphi \odot (\text{head} \circ \varphi^\dagger) \wedge x \mapsto y) * \\ (\exists y'. (x, y') \in \varphi \odot (\text{tail} \circ \varphi^\dagger) \wedge x + 4 \mapsto y')) * \\ (\exists y. (\text{scan} - 8, y) \in \varphi \odot (\text{head} \circ \varphi^\dagger) \wedge \text{scan} - 8 \mapsto y) * \\ (\exists y'. (\text{scan} - 8, y') \in \varphi \odot (\text{tail} \circ \varphi^\dagger) \wedge \text{scan} - 8 \mapsto y'))$$

implies

$$\forall_* x \in \text{FIN}. ((\exists y. (x, y) \in \varphi \odot (\text{head} \circ \varphi^\dagger) \wedge x \mapsto y) * \\ (\exists y'. (x, y') \in \varphi \odot (\text{tail} \circ \varphi^\dagger) \wedge x + 4 \mapsto y')).$$

For the first of these, the implication follows from (2.10), Lemma 2.9, and (2.B6), since  $\neg((\text{scan} - 8) \in \text{UFIN})$  implies  $(\text{UFIN} \cup \{\text{scan} - 8\}) \setminus \{\text{scan} - 8\} = \text{UFIN}$  (recall  $\text{UFIN} \equiv \text{Itv}(\text{scan}, \text{free})$ ). The second implication follows from Lemma 2.9, (2.B6), and (2.14), since  $\text{scan} - 8 \in \text{FIN} \equiv \text{Itv}(\text{offset}, \text{scan})$ .



## Chapter 3

# BI-Hyperdoctrines, Higher-order Separation Logic, and Abstraction

### Abstract

We present a simple extension of separation logic which makes the specification language *higher-order*, in the sense that quantification over predicates and higher types is possible. The fact that this is a useful extension is illustrated via examples; specifically we demonstrate that existential and universal quantification correspond to abstract data types and parametric data types, respectively. We also illustrate that the semantics we give is an instance of a general notion, namely that of a *BI hyperdoctrine*, of models for higher-order predicate BI.

### Preface

This chapter is a reprint of the technical report [21] which, in turn, is an extended version of the conference paper [20]. It was co-authored with Bodil Biering and Lars Birkedal, both from IT University of Copenhagen.

### 3.1 Introduction

Variants of the recent formalism of *separation logic* [115, 59] have been used to prove correct many interesting algorithms involving pointers, both in sequential and concurrent settings [84, 134, 23]. It is a Hoare-style program logic, and its main advantage over traditional program logics is that it facilitates *local reasoning*.

The force of separation logic comes from both its language of assertions – which is a variant of propositional BI [102] – and its language of specifications, or Hoare triples. In the present paper, we extend both of these. First, we introduce an assertion language which is a variant of *higher-order predicate BI*. The extension from the traditional assertion language of separation logic simply allows function types, has a type `Prop` of proposition, and allows quantification over variables of this type. Thus the assertion language is higher-order, in the usual sense that it allows quantification over predicates. Next, we present a specification logic for a simple second-order programming language in which it is also possible to quantify over variables of any type. We provide models for both the

new assertion language and the specification logic, and provide inference rules for deriving valid specifications. As it turns out, it is technically straightforward to do so; this emphasizes that our notion of higher-order predicate BI is the correct one for separation logic.

Next we consider the effectiveness of higher-order separation logic and argue, with the use of several examples, that it is quite effective. In particular, we show that higher-order separation logic can be used in a natural way to model data abstraction, via existential quantification over predicates corresponding to abstract resource invariants; we do so by means of a detailed example, which involves two implementations of abstract priority queues. This way of reasoning about data abstraction is more natural than the recently suggested abstract predicates of Parkinson and Bierman [91], for providing modular proofs of programs using abstract data types. Moreover, we show that, using universal quantification over predicates, we can prove correct polymorphic operations on polymorphic data types, e.g., reversing a list of elements described by some arbitrary predicate. For this to be useful, however, it is clear that a higher-order programming language would be preferable (such that one could program many more useful polymorphic operations, e.g., the `map` function for lists) — we have chosen to stick with the simpler second-order language here to communicate more easily the ideas of higher-order separation logic.

Having introduced higher-order separation logic and a semantics thereof, we show that our semantics is in fact an instance of a general concept. Part of the pointer model of separation logic, namely that given by heaps (but not stacks) has been related to *propositional* BI [104]. We show how the correspondence may be taken further, in the sense that our notion of *predicate* BI corresponds to all of the pointer model (including stacks). Moreover, we introduce the notion of a *BI hyperdoctrine*, a simple extension of Lawvere’s notion of a hyperdoctrine [66] and show that it soundly models predicate BI. Finally, show that our semantics is an instance of this general semantics.

It should be noted that our notion of higher-order predicate BI differs from that of the book [102, 103], which has a BI structure on contexts. However, we believe our notion of higher-order predicate BI with its class of BI hyperdoctrine models is the right one for separation logic (Pym aimed to model multiplicative quantifiers; separation logic only uses additive quantifiers); the correspondence mentioned above serves to illustrate this claim. Moreover, we present some applications of the extension of the assertion language of separation language to higher-order.

Before proceeding with the technical development we give an intuitive justification of the use of BI hyperdoctrines to model higher-order predicate BI. A powerful way of obtaining models of BI is by means of functor categories (presheaves), using Day’s construction to obtain a doubly-closed structure on the functor category [104]. Such functor categories can be used to model *propositional* BI in two different senses: In the first sense, one models *provability*, entailment between propositions, and it works because the lattice of subobjects of the terminal object in such functor categories form a BI algebra (a doubly cartesian closed preorder). In the second sense, one models *proofs*, and it works because the whole functor category is doubly cartesian closed. Here we seek models of provability of *predicate* BI. Since the considered functor categories are toposes and hence model higher-order predicate logic, one might think that a straightforward extension is possible. But, alas, it is not the case. In general, for this to work, *every* lattice of subobjects (for any object, not only for the terminal object) should be a BI algebra and, moreover, to model substitution correctly, the BI algebra structure should be preserved by pulling

back along any morphism. We show this can only be the case if the BI algebra structure is trivial, that is, coincides with the cartesian structure (see Theorem 3.14). Our theorem holds for any topos, not just for the functor categories just mentioned. Hence we need to consider a wider class of models for predicate BI than just toposes and this justifies the notion of a BI hyperdoctrine. The intuitive reason that BI hyperdoctrines work, is that predicates are not required to be modeled by subobjects, they can be something more general. Another important point of BI hyperdoctrines is that they are easy to come by: given any complete BI algebra  $B$ , there is a canonical BI hyperdoctrine in which predicates are modeled as  $B$ -valued functions; this is explained in detail in Example 3.13.

The rest of the paper is organized as follows. In Section 3.2 we introduce the syntax and semantics of both the assertion language and the specification language of higher-order separation logic. This includes the definition of a simple programming language which has heap manipulating constructs and simple procedures, and its operational semantics. In Section 3.3, we present rules for deriving valid specifications, which constitute a specification logic for our programming language. We explain these rules at an intuitive level and show soundness of them with respect to the semantics from Section 3.2. In Section 3.4 we give several examples which illustrate that higher-order separation logic is indeed useful. In particular, we show how existential quantification may be used to reason about data abstraction inasmuch as it can be used to show representation independence of two implementations of an abstract priority queue. Furthermore, we illustrate how universal quantification of the specification language can be used to model polymorphic types. In Section 3.5, we first recall Lawvere’s notion of a hyperdoctrine [66] and straightforwardly extend it to the notion of BI hyperdoctrines, and we show that this soundly models predicate BI and that our semantics of assertions is an instance of a BI hyperdoctrine. In Section 3.6 we discuss applications of the extension of the assertion language to higher-order, and in particular show how one can use the higher-order logic to give logical characterizations of interesting classes of assertions. In the last sections we give pointers to related and future work, and conclude.

## 3.2 Syntax and Semantics

We present our languages for terms (including propositions), programs, and specifications. The term language is a higher-order typed language with base types for integers and propositions. The programming language is a variant of the usual programming language for separation logic, extended with definitions of and calls to simple procedures. Correspondingly, our language for specifications include assumptions about such procedures, in the style of the hypothetical frame rule [88].

### 3.2.1 Syntax

**Types** are generated by the grammar

$$\tau ::= \text{Int} \mid \text{Prop} \mid \tau \times \tau \mid \tau \rightarrow \tau \mid \dots$$

The “ $\dots$ ” are used to indicate that more base types may be added to the system without complications. For now, the core system is the one indicated.

**Terms** There is a judgment  $\Delta \vdash t:\tau$ , where  $\Delta$  is a list of type assignments  $x:\tau$  to distinct variables, and the judgment states that the free variables of  $t$  are included in  $\Delta$ , and that

the term  $t$  is well-formed and of type  $\tau$  in  $\Delta$ . The judgment is defined by

$$\begin{array}{c}
\vdash n:\text{Int} \\
\Delta, x:\tau \vdash x:\tau \\
\frac{\Delta \vdash t:\text{Int} \quad \Delta \vdash t':\text{Int}}{\Delta \vdash t \otimes t':\text{Int}} \text{ where } \otimes \in \{+, -, \times\} \\
\frac{\Delta \vdash t:\text{Int} \quad \Delta \vdash t':\text{Int}}{\Delta \vdash t \triangleleft t':\text{Prop}} \text{ where } \triangleleft \in \{=, \leq\} \\
\vdash \top:\text{Prop} \\
\vdash \perp:\text{Prop} \\
\vdash \text{emp}:\text{Prop} \\
\frac{\Delta \vdash t:\text{Int} \quad \Delta \vdash t':\text{Int}}{\Delta \vdash t \mapsto t':\text{Prop}} \\
\frac{\Delta \vdash \varphi:\text{Prop} \quad \Delta \vdash \varphi':\text{Prop}}{\Delta \vdash \varphi \diamond \varphi':\text{Prop}} \text{ where } \diamond \in \{\vee, \wedge, \rightarrow, \neg, *\} \\
\frac{\Delta, x:\tau \vdash \varphi:\text{Prop}}{\Delta \vdash \exists x. \varphi:\text{Prop}} \text{ where } \exists \in \{\forall, \exists\} \\
\frac{\Delta, x:\tau \vdash t:\tau'}{\Delta \vdash (\lambda x:\tau. t):\tau \rightarrow \tau'} \\
\frac{\Delta \vdash t:\tau' \rightarrow \tau \quad \Delta \vdash t':\tau'}{\Delta \vdash tt':\tau}
\end{array}$$

Further, weakening and exchange in contexts is allowed. We assume that all the contexts are well-formed, so, for example, in the second rule above, it is implicitly assumed that  $x \notin \Delta$ . As can be seen from this grammar, the meta-variable  $t$  ranges over terms, and  $\varphi$  is mostly used for propositions (terms of type Prop). When propositions are used in program logic, however, we sometimes use  $P, Q, \dots$  to range over propositions, in accordance with traditional separation logic.

**Programming Language** The programming language uses a restricted set of terms of type Int, referred to as *expressions*, and uses *booleans*, which consists of a restricted (heap-independent) set of terms of type Prop.  $E$  and  $B$  range over these, and they are generated by the grammars:

$$\begin{array}{l}
E ::= n \mid x \mid E + E \mid E - E \mid E \times E \mid \text{null} \\
B ::= E = E \mid E \leq E \mid B \wedge B \mid \dots
\end{array}$$

Formally, booleans have type Prop in our system, but we sometimes write  $B : \text{Bool}$  if they can be generated from this grammar.

The syntax of the programming language is given by the following grammar. Here,  $k$  ranges over function names, and  $x$  ranges over program variables.

$$\begin{array}{l}
c ::= \mathbf{skip} \\
| x := k_i(E_1, \dots, E_{m_i}) \\
| \mathbf{newvar} \ x; c \\
| x := E \\
| x := [E] \\
| [E] := E' \\
| x := \mathbf{cons}(E_1, \dots, E_m) \\
| \mathbf{dispose}(E) \\
| \mathbf{if} \ B \ \mathbf{then} \ c \ \mathbf{else} \ c \ \mathbf{fi} \\
| \mathbf{while} \ B \ \mathbf{do} \ c \ \mathbf{od} \\
| c; c \\
| \mathbf{let} \ k_1(x_1, \dots, x_{m_1}) = c_1 \\
\quad \vdots \\
\quad k_n(x_1, \dots, x_{m_n}) = c_n \\
| \mathbf{in} \ c \ \mathbf{end} \\
| \mathbf{return} \ e
\end{array}$$

There are some restrictions on the programs, and a program is called *well-formed* if it meets these restrictions. This could be expressed formally with auxiliary grammars, but we refrain from that here. The restrictions include:

- There is always a **return** at the end of a function body.
- A function name is declared at most once in a **let**.
- There are the right number of parameters in function calls.
- Function bodies do not modify non-local variables (other than *ret*).

**Function Specifications** There is a judgment

$$\Delta \vdash \gamma : \text{FSpec}$$

stating that  $\gamma$  is a well-formed *function specification* in the context  $\Delta$ . Function specifications are used to record assumptions about functions used in programs. The judgment is given by

$$\begin{array}{c}
\frac{\Delta \vdash P : \text{Prop} \quad \Delta \vdash Q : \text{Prop}}{\Delta \vdash \{P\} k \{Q\} : \text{FSpec}} \\
\frac{\Delta \vdash \gamma : \text{FSpec} \quad \Delta \vdash \gamma' : \text{FSpec}}{\Delta \vdash \gamma \wedge \gamma' : \text{FSpec}} \\
\frac{\Delta, x : \tau \vdash \gamma : \text{FSpec}}{\Delta \vdash \lambda x : \tau. \gamma : \text{FSpec}} \text{ where } \lambda \in \{\exists, \forall\}
\end{array}$$

The set of free variables for a function specification is defined as the free variables in the assertions occurring in it.

**Specifications** We introduce syntax for commands and specifications. There is a judgment

$$\Delta; \Pi \vdash c : \text{comm}, \tag{3.1}$$

which asserts that the program  $c$  is well-formed in the context  $\Delta$  and *semantic function environment*  $\Pi$ . A semantic function environment maps function names  $k$  to pairs  $(\bar{x}, c)$ , where  $\bar{x}$  is a vector of integer variables and  $c$  is a command from the programming language. Such an environment is well-formed if the function bodies only modify local variables (and *ret*, by the **return** command):

$$\Pi \text{ ok iff } \forall (x, c) \in \text{cod}(\Pi). \text{Mod}(c) = \emptyset.$$

The definition of the judgment (3.1) is omitted here.

The *specifications* of higher-order separation logic is given by a judgment

$$\Delta; \Pi \vdash \delta: \text{Spec},$$

which asserts that  $\delta$  is a well-formed specification in the context  $\Delta$  and semantic function environment  $\Pi$ . This judgment is given by

$$\frac{\Delta; \Pi \vdash c: \text{comm} \quad \Delta \vdash P: \text{Prop} \quad \Delta \vdash Q: \text{Prop}}{\Delta; \Pi \vdash \{P\} c \{Q\}: \text{Spec}} \\ \frac{\Delta; \Pi \vdash \delta: \text{Spec} \quad \Delta; \Pi \vdash \delta': \text{Spec}}{\Delta; \Pi \vdash \delta \wedge \delta': \text{Spec}} \\ \frac{\Delta, x: \tau; \Pi \vdash \delta: \text{Spec}}{\Delta; \Pi \vdash \lambda x: \tau. \delta: \text{Spec}} \quad \natural \in \{\exists, \forall\}$$

The set  $\text{FV}(\delta)$  of free variables of a specification  $\delta$  is the set of free variables in the assertions and the *modified* variables in the commands occurring in the specification. The set  $\text{Mod}(\delta)$  of modified variables of  $\delta$  is the set of modified variables in the commands occurring in  $\delta$ .

### 3.2.2 Semantics

**Semantics of Types** The semantics of types is a set, and it is given by

$$\begin{aligned} \llbracket \text{Prop} \rrbracket &= \mathcal{P}(H) \\ \llbracket \text{Int} \rrbracket &= \mathbb{Z} \\ \llbracket \tau \times \tau' \rrbracket &= \llbracket \tau \rrbracket \times \llbracket \tau' \rrbracket \\ \llbracket \tau \rightarrow \tau' \rrbracket &= \llbracket \tau \rrbracket \Rightarrow \llbracket \tau' \rrbracket \end{aligned}$$

When more base types are added, one of course has to specify the semantics of them.

**Semantics of Terms** The semantics  $\llbracket \Delta \vdash t: \tau \rrbracket$  is a map

$$\llbracket \Delta \rrbracket \xrightarrow{\llbracket \cdot \rrbracket} \llbracket \tau \rrbracket,$$

where  $\llbracket \Delta \rrbracket$  is the product of the  $\tau_i$  for  $x_i: \tau_i \in \Delta$ . Although elements of  $\llbracket \Delta \rrbracket$  are tuples  $(v_1, \dots, v_n)$ , we treat them as maps from variables in  $\Delta$  to values. Hence, if  $\Delta = x_1: \tau_1, \dots, x_n: \tau_n$  and  $\eta = (v_1, \dots, v_n) \in \llbracket \Delta \rrbracket$ , we write  $\eta(x_i)$  instead of  $\pi_i(\eta)$  for the value  $v_i$ . For this correspondence, we use the following notation. If  $x: \tau \notin \Delta$ ,  $v \in \llbracket \tau \rrbracket$ , and  $\eta = (v_1, \dots, v_n) \in \llbracket \Delta \rrbracket$ , write  $\eta_{[x \rightarrow v]}$  for the tuple  $(v_1, \dots, v_n, v) \in \llbracket \Delta, x: \tau \rrbracket$ . Also, this notation is used for updates in  $\eta$ : if  $x_i: \tau_i \in \Delta$ ,  $v_i \in \llbracket \tau_i \rrbracket$ , and  $\eta = (v_1, \dots, v_n) \in \llbracket \Delta \rrbracket$ , write  $\eta_{[x_i \rightarrow v_i]}$  for the tuple  $(v_1, \dots, v_i, \dots, v_n) \in \llbracket \Delta \rrbracket$ . Finally the notation  $\eta - x$

is used to “remove a component from a tuple”: If  $x_i:\tau_i \in \Delta = x_1:\tau_1, \dots, x_n:\tau_n$  and  $\eta = (v_1, \dots, v_i, \dots, v_n) \in \llbracket \Delta \rrbracket$ , we write  $\eta - x_i$  for the tuple  $(v_1, \dots, v_{i-1}, v_{i+1}, \dots, v_n) \in \llbracket x_1:\tau_1, \dots, x_{i-1}:\tau_{i-1}, x_{i+1}:\tau_{i+1}, \dots, x_n:\tau_n \rrbracket$ .

The semantics is defined in a standard way; the most important clauses are presented here.

$$\begin{aligned} \llbracket \Delta \vdash n:\text{Int} \rrbracket \eta &= n \\ \llbracket \Delta, x:\tau \vdash x:\tau \rrbracket \eta &= \eta(x) \\ \llbracket \Delta \vdash t \triangleleft t':\text{Prop} \rrbracket \eta &= \begin{cases} H & \text{if } \llbracket \Delta \vdash t:\text{Int} \rrbracket \eta \triangleleft \llbracket \Delta \vdash t':\text{Int} \rrbracket \eta \\ \emptyset & \text{if } \neg(\llbracket \Delta \vdash t:\text{Int} \rrbracket \eta \triangleleft \llbracket \Delta \vdash t':\text{Int} \rrbracket \eta) \end{cases}, \text{ where } \triangleleft \in \{=, \leq\} \\ \llbracket \Delta \vdash \text{emp}:\text{Prop} \rrbracket \eta &= \{\emptyset\} \\ \llbracket \Delta \vdash \varphi * \varphi':\text{Prop} \rrbracket \eta &= \left\{ h \mid \begin{array}{l} \exists h_0, h_1. h_0 \# h_1 \wedge h = h_0 \cup h_1 \wedge \\ h_0 \in \llbracket \Delta \vdash \varphi \rrbracket \eta \wedge h_1 \in \llbracket \Delta \vdash \varphi' \rrbracket \eta \end{array} \right\} \\ \llbracket \Delta \vdash \varphi \multimap \varphi':\text{Prop} \rrbracket \eta &= \bigcup W, \text{ with } W * \llbracket \Delta \vdash \varphi:\text{Prop} \rrbracket \eta \subseteq \llbracket \Delta \vdash \varphi':\text{Prop} \rrbracket \eta \\ \llbracket \Delta \vdash \exists x:\tau. \varphi:\text{Prop} \rrbracket \eta &= \bigcup_{v \in \llbracket \tau \rrbracket} \llbracket \Delta, x:\tau \vdash \varphi:\text{Prop} \rrbracket \eta_{[x \rightarrow v]} \\ \llbracket \Delta \vdash \forall x:\tau. \varphi:\text{Prop} \rrbracket \eta &= \bigcap_{v \in \llbracket \tau \rrbracket} \llbracket \Delta, x:\tau \vdash \varphi:\text{Prop} \rrbracket \eta_{[x \rightarrow v]} \end{aligned}$$

This semantics uses the fact that  $\mathcal{P}(H)$  is a boolean BI-algebra [20] to give the semantics of the BI connectives  $(*, \multimap)$  of the logic. See Sec. 3.5.3 for more details.

The expected substitution lemma holds:

**Lemma 3.1.** *Suppose  $\Delta \vdash t:\tau$  and  $\Delta, x:\tau \vdash t':\tau'$ . Then for all  $\eta \in \llbracket \Delta \rrbracket$ ,*

$$\llbracket \Delta \vdash t'[t/x] \rrbracket \eta = \llbracket \Delta, x:\tau \vdash t' \rrbracket \eta_{[x \rightarrow v]},$$

where  $v = \llbracket \Delta \vdash t:\tau \rrbracket \eta$ .

**Remark 3.2.** The traditional way of giving semantics of assertions is via a forcing relation

$$s, h \Vdash \varphi, \tag{3.2}$$

which asserts that the assertion  $\varphi$  holds in the state  $(s, h)$  (where the free variables of  $\varphi$  are included in the domain of the stack  $s$ ). The grammar of assertions and the definition of the forcing relation (3.2) is completely standard, and different variants may be found in numerous papers, e.g., [23, 85, 33]. There is a tight connection between these two forms of semantics, since it is not hard to see that if  $\varphi$  has free variables  $x_1, \dots, x_n$ , and if  $v_1, \dots, v_n$  are values of the corresponding types (in “traditional” separation logic, there is only one type), then

$$\begin{aligned} h \in \llbracket \varphi \rrbracket (v_1, \dots, v_n) \text{ in our semantics} \\ \text{iff} \\ [x_1 \rightarrow v_1, \dots, x_n \rightarrow v_n], h \Vdash \varphi \text{ in the traditional semantics.} \end{aligned}$$

**Operational Semantics of the Programming Language** The operational semantics of the programming language is given by a judgment

$$(\Pi, c, \eta, h) \Downarrow (\eta', h').$$

The proviso here is that  $\eta \in \Delta$  for some  $\Delta$  in which  $\Delta; \Pi \vdash c:\text{comm}$  holds, and it intuitively says that the state  $(\eta, h)$  is transformed to the state  $(\eta', h')$  by the program  $c$ . The

judgment is the same as in the paper [91] and it is given by the clauses in Fig. 3.1. We occasionally use  $\Delta$  for the domain of  $\eta$  in the definition of the judgment. For example, in the second rule (for assignment), the precondition is  $\llbracket \Delta \vdash E:\text{Int} \rrbracket \eta = n$ . What is meant is just that  $E$  is a term of type  $\text{Int}$  in any context containing the variables in  $\eta$ . Furthermore, the notation  $h - \{n\}$  is used to denote the heap which is like  $h$ , but with  $n$  taken out of its domain, and other notation like that for  $\eta$ s to update contents of heap cells.

The configuration  $(\Pi, c, \eta, h)$  is called *safe* if  $(\Pi, c, \eta, h) \not\Downarrow \text{wrong}$ . A configuration may either terminate in a state  $(\eta', h')$ , diverge, or go wrong.

Note that, since this semantics is the same as the operational semantics of the language of Parkinson and Bierman [91], the properties needed to prove the frame rule, namely safety monotonicity and the frame property, are valid for all programs of the language. As a reminder, these properties are repeated here.

**Safety Monotonicity.** For all well-formed semantic function environments  $\Pi$ , programs  $c$ , stacks  $\eta$ , and heaps  $h$ , if  $(\Pi, c, \eta, h)$  is safe, then for all heaps  $h'$  disjoint from  $h$ ,  $(\Pi, c, \eta, h \cup h')$  is also safe.

**The Frame Property.** For all well-formed semantic function environments  $\Pi$ , programs  $c$ , stacks  $\eta$ , and heaps  $h$ , if  $(\Pi, c, \eta, h)$  is safe and  $h'$  is disjoint from  $h$ , then  $(\Pi, c, \eta, h \cup h') \Downarrow (\eta', h'')$ , implies that there is  $h_0$  such that  $h'' = h_0 \cup h'$  and  $(\Pi, c, \eta, h) \Downarrow (\eta', h_0)$ .

### 3.2.3 Program Logic Judgments

A list  $\Gamma$  of function specifications where the function names are distinct, is called an *environment*. We define the judgment

$$\Delta; \Gamma \models \delta: \text{Spec},$$

which states that in the context  $\Delta$ , given the assumptions about functions recorded in  $\Gamma$ , the specification  $\delta$  holds. This judgment is defined in several straightforward steps, and it is basically the same as the corresponding judgment in the paper [91].

First, we give the semantics of a triple, relative to a context and a semantic function environment. The semantics of  $\llbracket \Delta, \Pi \vdash \delta: \text{Spec} \rrbracket$  is a map from  $\llbracket \Delta \rrbracket$  to the domain  $\{\mathbf{true}, \mathbf{false}\}$ , and it is given by (some obvious type annotations are omitted):

$$\begin{aligned} \llbracket \Delta, \Pi \vdash \{P\} c \{Q\} \rrbracket & \text{ iff } \forall h \in \llbracket \Delta \vdash P \rrbracket \eta. \\ & \quad - (\Pi, c, \eta, h) \text{ is safe, and} \\ & \quad - (\Pi, c, \eta, h) \Downarrow (\eta', h') \text{ implies } h' \in \llbracket \Delta \vdash Q \rrbracket \eta' \\ \llbracket \Delta, \Pi \vdash \delta \wedge \delta' \rrbracket \eta & \text{ iff } \llbracket \Delta, \Pi \vdash \delta \rrbracket \eta \text{ and } \llbracket \Delta, \Pi \vdash \delta' \rrbracket \eta \\ \llbracket \Delta, \Pi \vdash \exists x:\tau. \delta \rrbracket \eta & \text{ iff } \llbracket \Delta, \Pi \vdash \delta \rrbracket \eta_{[x \rightarrow v]} \text{ for some } v \in \llbracket \tau \rrbracket \\ \llbracket \Delta, \Pi \vdash \forall x:\tau. \delta \rrbracket \eta & \text{ iff } \llbracket \Delta, \Pi \vdash \delta \rrbracket \eta_{[x \rightarrow v]} \text{ for all } v \in \llbracket \tau \rrbracket. \end{aligned}$$

We call  $\Delta, \Pi \vdash \delta$  *valid* and write  $\Delta, \Pi \models \delta$  iff  $\llbracket \Delta, \Pi \vdash \delta \rrbracket \eta = \mathbf{true}$  for all  $\eta \in \llbracket \Delta \rrbracket$ . There is a substitution lemma for this semantics, which is needed later.

**Lemma 3.3.** *Let  $\delta$  be a specification,  $x:\tau$  a variable, and  $\Delta \vdash t:\tau$  a term. Further, let  $\eta \in \llbracket \Delta \rrbracket$ , and  $\Pi$  be well-formed. Then,*

$$\llbracket \Delta; \Pi \vdash \delta[t/x] \rrbracket \eta \text{ iff } \llbracket \Delta, x:\tau; \Pi \vdash \delta \rrbracket \eta_{[x \rightarrow v]},$$

where  $v = \llbracket \Delta \vdash t:\tau \rrbracket \eta$ .

$$\begin{array}{c}
(\Pi, \mathbf{skip}, \eta, h) \Downarrow (\eta, h) \\
\hline
\llbracket \Delta \vdash E:\text{Int} \rrbracket \eta = n \\
(\Pi, x := E, \eta, h) \Downarrow (\eta_{[x \rightarrow n]}, h) \\
\hline
\llbracket \Delta \vdash E:\text{Int} \rrbracket \eta = n \\
(\Pi, \mathbf{return} E, \eta, h) \Downarrow (\eta_{[\text{ret} \rightarrow n]}, h) \\
\hline
\llbracket \Delta \vdash E:\text{Int} \rrbracket \eta = n \quad n \in \text{dom}(h) \quad h(n) = n' \\
(\Pi, x := [E], \eta, h) \Downarrow (\eta_{[x \rightarrow n']}, h) \\
\hline
\llbracket \Delta \vdash E:\text{Int} \rrbracket \eta = n \quad \llbracket \Delta \vdash E':\text{Int} \rrbracket \eta = n' \quad n \in \text{dom}(h) \\
(\Pi, [E] := E', \eta, h) \Downarrow (\eta, h_{[n \rightarrow n']}) \\
\hline
\llbracket \Delta \vdash E:\text{Int} \rrbracket \eta = n \quad n \in \text{dom}(h) \\
(\Pi, \mathbf{dispose}(E), \eta, h) \Downarrow (\eta, h - \{n\}) \\
\hline
\llbracket \Delta \vdash E:\text{Int} \rrbracket \eta = n \quad n \notin \text{dom}(h) \\
(\Pi, x := [E], \eta, h) \Downarrow \text{wrong} \\
\hline
\llbracket \Delta \vdash E:\text{Int} \rrbracket \eta = n \quad n \notin \text{dom}(h) \\
(\Pi, [E] := E', \eta, h) \Downarrow \text{wrong} \\
\hline
\llbracket \Delta \vdash E:\text{Int} \rrbracket \eta = n \quad n \notin \text{dom}(h) \\
(\Pi, \mathbf{dispose}(E), \eta, h) \Downarrow \text{wrong} \\
\hline
\{n, n+1, \dots, n+m\} \perp \text{dom}(h) \quad (\llbracket \Delta \vdash E_i:\text{Int} \rrbracket \eta = n_i)_{i=0, \dots, m} \\
(\Pi, x := \mathbf{cons}(E_0, \dots, E_m), \eta, h) \Downarrow (\eta_{[x \rightarrow n]}, h_{[n+i \rightarrow n_i]_{i=0, \dots, m}}) \\
\hline
(\Pi, c_1, \eta, h) \Downarrow (\eta', h') \quad (\Pi, c_2, \eta', h') \Downarrow (\eta'', h'') \\
\hline
(\Pi, c_1; c_2, \eta, h) \Downarrow (\eta'', h'') \\
\hline
\llbracket \Delta \vdash B:\text{Bool} \rrbracket \eta = \mathbf{false} \quad (\Pi, c_1, \eta, h) \Downarrow (\eta', h') \\
\hline
(\Pi, \mathbf{if} B \mathbf{then} c_0 \mathbf{else} c_1 \mathbf{fi}) \Downarrow (\eta', h') \\
\hline
\llbracket \Delta \vdash B:\text{Bool} \rrbracket \eta = \mathbf{true} \quad (\Pi, c_0, \eta, h) \Downarrow (\eta', h') \\
\hline
(\Pi, \mathbf{if} B \mathbf{then} c_0 \mathbf{else} c_1 \mathbf{fi}) \Downarrow (\eta', h') \\
\hline
\llbracket \Delta \vdash B:\text{Bool} \rrbracket \eta = \mathbf{false} \\
(\Pi, \mathbf{while} B \mathbf{do} c \mathbf{od}, \eta, h) \Downarrow (\eta, h) \\
\hline
\llbracket \Delta \vdash B:\text{Bool} \rrbracket \eta = \mathbf{true} \quad (\Pi, c; \mathbf{while} B \mathbf{do} c \mathbf{od}, \eta, h) \Downarrow (\eta', h') \\
\hline
(\Pi, \mathbf{while} B \mathbf{do} c \mathbf{od}, \eta, h) \Downarrow (\eta', h') \\
\hline
\Pi(k) = ((x_1, \dots, x_m), c_k) \quad (\Pi, c_k, [x_i \rightarrow n_i], h) \Downarrow (\eta', h') \\
(\llbracket \Delta \vdash E_i:\text{Int} \rrbracket \eta = n_i)_{i=1, \dots, m} \\
\hline
(\Pi, x = k(E_1, \dots, E_m), \eta, h) \Downarrow (\eta_{[x \rightarrow \eta'(\text{ret})]}, h') \\
\hline
(\Pi, c, \eta_{[x \rightarrow \text{null}]}, h) \Downarrow (\eta', h') \quad \eta(x) = v \\
\hline
(\Pi, \mathbf{newvar} x; c, \eta, h) \Downarrow (\eta'_{[x \rightarrow v]}, h') \\
\hline
(\Pi \cup (k_1 \rightarrow ((x_1, \dots, x_{n_1}), c_1), \dots, k_n \rightarrow ((x_1, \dots, x_{n_k}), c_n)), c, \eta, h) \Downarrow (\eta', h') \\
\hline
(\Pi, \mathbf{let} k_1(x_1, \dots, x_{n_1}) = c_1, \dots, k_n(x_1, \dots, x_{n_n}) = c_n \mathbf{in} c, \eta, h) \Downarrow (\eta', h')
\end{array}$$

Figure 3.1: Operational Semantics of the Programming Language

There is a similar semantics for function specifications. This semantics is a map

$$\llbracket \Delta, \Pi \vdash \gamma : \text{FSpec} \rrbracket : \llbracket \Delta \rrbracket \rightarrow \{\mathbf{true}, \mathbf{false}\},$$

and it is given in the same way as the same map for specifications. The only difference is the base case, which is given by

$$\llbracket \Delta; \Pi \vdash \{P\} k \{Q\} \rrbracket \eta \quad \text{iff} \quad \llbracket \Delta'; \Pi \vdash \{P\} c_m \{Q\} \rrbracket \eta \\ \text{where } \Pi(k) = ((x_1, \dots, x_{n_m}), c_m),$$

where  $\Delta'$  is  $\Delta$  with those  $x_i$  added (with type  $\text{Int}$ ) that are not there.

For this semantics, there is a substitution lemma, which resembles Lemma 3.3.

**Lemma 3.4.** *Let  $\gamma$  be a specification,  $x:\tau$  a variable, and  $\Delta \vdash t:\tau$  a term. Further, let  $\eta \in \llbracket \Delta \rrbracket$ , and  $\Pi$  be well-formed. Then,*

$$\llbracket \Delta; \Pi \vdash \gamma[t/x] \rrbracket \eta \quad \text{iff} \quad \llbracket \Delta, x:\tau; \Pi \vdash \gamma \rrbracket \eta_{[x \rightarrow v]},$$

where  $v = \llbracket \Delta \vdash t:\tau \rrbracket \eta$ .

As mentioned, an environment is a list of function specifications. The semantics of an environment is given componentwise:

$$\llbracket \Delta, \Pi \vdash \Gamma \rrbracket \eta \quad \text{iff} \quad \llbracket \Delta, \Pi \vdash \gamma \rrbracket \eta \text{ for all } \gamma \in \Gamma.$$

**Lemma 3.5.** *Let  $\Delta \vdash t:\tau$  be a term,  $\eta \in \llbracket \Delta \rrbracket$ , and  $\Gamma$  an environment. Then,*

$$\llbracket \Delta; \Pi \vdash \Gamma[t/x] \rrbracket \eta \quad \text{iff} \quad \llbracket \Delta, x:\tau; \Pi \vdash \Gamma \rrbracket \eta_{[x \rightarrow v]},$$

where  $v = \llbracket \Delta \vdash t:\tau \rrbracket \eta$ .

Finally, the semantics of specifications, relative to a context and an environment, is defined by

$$\Delta; \Gamma \models \delta \quad \text{iff} \quad \text{for all well-formed } \Pi \text{ and all } \eta \in \llbracket \Delta \rrbracket, \\ \llbracket \Delta; \Pi \vdash \Gamma \rrbracket \eta \text{ implies } \llbracket \Delta; \Pi \vdash \delta \rrbracket \eta.$$

The relevant substitution lemma for this semantics is:

**Lemma 3.6.** *Let  $\Delta \vdash t:\tau$  be a term. Then*

$$\Delta, x:\tau; \Gamma \models \delta \quad \text{implies} \quad \Delta; \Gamma[t/x] \models \delta[t/x].$$

### 3.3 Program Logic

We define a judgment

$$\Delta; \Gamma \vdash \delta,$$

for deriving valid specifications. The complete set of rules is given in Fig. 3.2. We first explain some of the rules at an intuitive level, and then show soundness.

$$\begin{array}{c}
\frac{}{\Delta; \Gamma \vdash \{P\} \mathbf{skip} \{P\}} \quad \frac{}{\Delta; \Gamma \vdash \{P[E/x]\} x := E \{P\}} \quad x \notin \text{FV}(E) \\
\frac{}{\Delta; \Gamma \vdash \{P[E/ret]\} \mathbf{return} E \{P\}} \quad \frac{\{P\} k(\bar{x}) \{Q\} \in \Gamma}{\Delta; \Gamma \vdash \{P[\bar{E}/\bar{x}]\} y = k(\bar{E}) \{Q[\bar{E}, y/\bar{x}, ret]\}} \\
\frac{}{\Delta; \Gamma \vdash \{\mathbf{emp} \wedge x = m\} x := \mathbf{cons}(E_1, \dots, E_n) \{x \mapsto \mathcal{E}_1[m/x], \dots, \mathcal{E}_n[m/x]\}} \\
\frac{}{\Delta; \Gamma \vdash \{E \mapsto -\} \mathbf{dispose}(E) \{\mathbf{emp}\}} \\
\frac{}{\Delta; \Gamma \vdash \{E \mapsto n \wedge x = m\} x := [E] \{E[m/x] \mapsto n \wedge x = n\}} \\
\frac{}{\Delta; \Gamma \vdash \{E \mapsto -\} [E] := E' \{E \mapsto E'\}} \\
\Delta; \Gamma \vdash \{P_1\} c_1 \{Q_1\} \\
\vdots \\
\Delta; \Gamma \vdash \{P_n\} c_n \{Q_n\} \\
\frac{\Delta; \Gamma, \{P_1\} k_1 \{Q_1\}, \dots, \{P_n\} k_n \{Q_n\} \vdash \{P\} c \{Q\}}{\Delta; \Gamma \vdash \{P\} \mathbf{let} k_1(\bar{x}_1) = c_1, \dots, k_n(\bar{x}_n) = c_n \mathbf{in} c \{Q\}} \\
\frac{\Delta; \Gamma \vdash \{P\} c_1 \{P'\} \quad \Delta; \Gamma \vdash \{P'\} c_2 \{Q\}}{\Delta; \Gamma \vdash \{P\} c_1; c_2 \{Q\}} \\
\frac{\Delta, x:\text{Int}; \Gamma \vdash \{P \wedge x = \mathbf{null}\} c \{Q\}}{\Delta; \Gamma \vdash \{P\} \mathbf{newvar} x \mathbf{in} c \mathbf{end} \{Q\}} \quad x \notin \text{FV}(P) \\
\frac{\Delta; \Gamma \vdash \{P \wedge B\} c_1 \{Q\} \quad \Delta; \Gamma \vdash \{P \wedge \neg B\} c_2 \{Q\}}{\{P\} \mathbf{if} B \mathbf{then} c_1 \mathbf{else} c_2 \mathbf{fi} \{Q\}} \\
\frac{}{\Delta; \Gamma \vdash \{P \wedge B\} c \{P\}} \\
\Delta; \Gamma \vdash \{P\} \mathbf{while} B \mathbf{do} c \mathbf{od} \{P \wedge \neg B\} \\
\frac{\Delta; \Gamma \vdash \{P\} c \{Q\}}{\Delta, \Delta'; \Gamma \vdash \{P\} c \{Q\}} \quad \Delta \cap \Delta' = \emptyset \\
\frac{\Delta, \Delta'; \Gamma \vdash \{P\} c \{Q\}}{\Delta; \Gamma \vdash \{P\} c \{Q\}} \quad \Delta' \text{ disjoint from } \Delta, P, c, Q, \Gamma \\
\frac{\Delta \vdash P \Rightarrow P' \quad \Delta; \Gamma \vdash \{P'\} c \{Q'\} \quad \Delta \vdash Q' \Rightarrow Q}{\Delta; \Gamma \vdash \{P\} c \{Q\}} \\
\frac{\Delta, x:\tau; \Gamma, \gamma \vdash \delta}{\Delta; \Gamma, \exists x:\tau. \gamma \vdash \delta} \quad x \notin \text{FV}(\Gamma) \cup \text{Mod}(\delta) \\
\frac{\Delta, x:\tau; \Gamma \vdash \delta}{\Delta; \Gamma \vdash \forall x:\tau. \delta} \quad x \notin \text{FV}(\Gamma) \cup \text{Mod}(\delta) \\
\frac{\Delta; \Gamma \vdash \{P\} c \{Q\}}{\Delta; \Gamma \vdash \{P * P'\} c \{Q * P'\}} \quad \text{Mod}(c) \cap \text{FV}(P') = \emptyset
\end{array}$$

Figure 3.2: Program Logic

### 3.3.1 Informal Explanation of Rules

The first two rules are the usual rules for **skip** and assignment from Hoare logic. The rule for **return** is similar to the rule for assignment, since **return** simply amounts to an assignment to the special variable *ret*.

The rule

$$\frac{\{P\} k(\bar{x}) \{Q\} \in \Gamma}{\Delta; \Gamma \vdash \{P[\bar{E}/\bar{x}]\} y = k(\bar{E}) \{Q[\bar{E}, y/\bar{x}, \text{ret}]\}}$$

for function call says that in order to call a function, the precondition for the function must be satisfied. This precondition is recorded in the environment, along with the corresponding postcondition.

The next four rules which involve the heap-manipulating constructs of the programming language, are the standard rules of separation logic, adapted to our setting. Note that the specifications are “tight” in the sense that they only mention the heap cells that are actually manipulated by the commands. For example, the rule

$$\frac{}{\Delta; \Gamma \vdash \{\text{emp} \wedge x = m\} x := \mathbf{cons}(\bar{E})\{x \mapsto \bar{E}[m/x]\}}$$

for **cons** produces a new cell when run in an empty heap. Note that this does *not* mean that **cons** can only be executed in an empty heap. The last rule of the system,

$$\frac{\Delta; \Gamma \vdash \{P\} c \{Q\}}{\Delta; \Gamma \vdash \{P * P'\} c \{Q * P'\}} \text{Mod}(c) \cap \text{FV}(P') = \emptyset,$$

called *the frame rule*, implies that one can infer a *global* specification from a *local* specification like the one for **cons**. Hence, **cons** can be executed in *any* heap, described by the predicate *P* (in which *x* does not occur freely), by the following instance of the frame rule:

$$\frac{\Delta; \Gamma \vdash \{\text{emp} \wedge x = m\} x := \mathbf{cons}(\bar{E})\{x \mapsto \bar{E}[m/x]\}}{\Delta; \Gamma \vdash \{P \wedge x = m\} x := \mathbf{cons}(\bar{E})\{P * (x \mapsto \bar{E}[m/x])\}}.$$

The rule

$$\frac{\begin{array}{c} \Delta; \Gamma \vdash \{P_1\} c_1 \{Q_1\} \\ \vdots \\ \Delta; \Gamma \vdash \{P_n\} c_n \{Q_n\} \\ \Delta; \Gamma, \{P_1\} k_1 \{Q_1\}, \dots, \{P_n\} k_n \{Q_n\} \vdash \{P\} c \{Q\} \end{array}}{\Delta; \Gamma \vdash \{P\} \mathbf{let} k_1(\bar{x}_1) = c_1, \dots, k_n(\bar{x}_n) = c_n \mathbf{in} c \{Q\}}$$

for function definitions is the usual one from Hoare logic with procedures [51]. The rules for **while** and **if-then-else** are also standard. The next two rules are structural and allow certain straightforward manipulations of contexts. The rule of consequence is standard, and the rules

$$\frac{\Delta, x:\tau; \Gamma, \gamma \vdash \delta}{\Delta; \Gamma, \exists x:\tau. \gamma \vdash \delta} x \notin \text{FV}(\Gamma) \cup \text{Mod}(\delta)$$

$$\frac{\Delta, x:\tau; \Gamma \vdash \delta}{\Delta; \Gamma \vdash \forall x:\tau. \delta} x \notin \text{FV}(\Gamma) \cup \text{Mod}(\delta)$$

are straightforward adaptations of standard rules of higher-order logic. They are used later for reasoning about data abstraction and parametricity. Note that in both of the rules,  $x \notin \text{Mod}(\delta)$ . Often, *x* denotes an “abstract value”, which should not be part of the specification one wants to show in the end, as we shall see later.

### 3.3.2 Soundness

**Theorem 3.7.** *If a specification*

$$\Delta; \Gamma \vdash \delta$$

*can be derived from the rules in Fig. 3.2, then it is valid.*

*Proof.* For each rule of form

$$\frac{\Delta; \Gamma \vdash \delta}{\Delta'; \Gamma' \vdash \delta'} \quad (3.3)$$

this is checked by showing that  $\Delta'; \Gamma' \models \delta'$ , under the assumption  $\Delta; \Gamma \models \delta$ . For axioms of the form

$$\overline{\Delta; \Gamma \vdash \delta'}$$

the proof obligation is to show  $\Delta; \Gamma \models \delta$ .

Consider the rule for **skip**:

$$\overline{\Delta; \Gamma \vdash \{P\} \mathbf{skip} \{P\}}$$

Although trivial, we show soundness of this rule here, to exercise the definitions. Let  $\Pi$  be a well-formed semantic function environment. It suffices to show

$$\llbracket \Delta; \Pi \vdash \{P\} \mathbf{skip} \{P\} \rrbracket \eta$$

for all  $\eta \in \llbracket \Delta \rrbracket$ . Let  $h \in \llbracket P \rrbracket \eta$ . Then,

$$(\Pi, \mathbf{skip}, \eta, h) \Downarrow (\eta, h),$$

and clearly,  $h \in \llbracket P \rrbracket \eta$ , so this rule is sound.

Soundness of the rule for assignment

$$\overline{\Delta; \Gamma \vdash \{P[E/x]\} x := E \{P\}}$$

depends, as usual, on the substitution Lemma 3.1.

The rule for **return**

$$\overline{\Delta; \Gamma \vdash \{P[E/ret]\} \mathbf{return} E \{P\}}$$

is essentially just an instance of the assignment rule.

Now consider the rule for function call:

$$\frac{\{P\} k_i(x_1, \dots, x_{n_i}) \{Q\} \in \Gamma}{\Delta, \Gamma \vdash \{P[E_1/x_1 \dots E_{n_i}/x_{n_i}]\} y = k_i(E_1, \dots, E_{n_i}) \{Q[E_1/x_1 \dots E_{n_i}/x_{n_i}, y/ret]\}}$$

To show soundness, suppose  $\{P\} k_i(x_1, \dots, x_{n_i}) \{Q\} \in \Gamma$ . Let  $\eta \in \llbracket \Delta \rrbracket$ , and let  $\Pi$  be a well-formed semantic function environment with  $\llbracket \Delta; \Pi \models \Gamma \rrbracket \eta$ . In particular,

$$\llbracket \Delta; \Pi \vdash \{P\} k_i(x_1, \dots, x_{n_i}) \{Q\} \rrbracket \eta,$$

so if  $\Pi(k_i) = ((x_1, \dots, x_{n_i}), c_i)$ , then  $\llbracket \Delta; \Pi \vdash \{P\} c_i \{Q\} \rrbracket \eta$ . Now, suppose

$$h \in \llbracket P[E_1/x_1 \dots E_{n_i}/x_{n_i}] \rrbracket \eta = \llbracket P \rrbracket \eta_{[x_1 \rightarrow v_1, \dots, x_{n_i} \rightarrow v_{n_i}]},$$

where  $v_j = \llbracket \Delta \vdash E_j; \text{Int} \rrbracket \eta$  for  $j = 1, \dots, n_i$ , by the substitution lemma. This means that if

$$(\Pi, c_i, \eta_{[x_1 \rightarrow v_1, \dots, x_{n_i} \rightarrow v_{n_i}]}, h) \Downarrow (\eta', h'),$$

then  $h' \in \llbracket Q \rrbracket \eta'$ . Since  $\Pi$  is well-formed,  $c_i$  does not modify any variables, so  $\eta'$  is of the form

$$\eta' = \eta_{[x_1 \rightarrow v_1, \dots, x_{n_i} \rightarrow v_{n_i}, \text{ret} \rightarrow \eta'(\text{ret})]},$$

and by the substitution lemma,  $h' \in \llbracket Q[E_1/x_1 \cdots E_{n_i}/x_{n_i}, \eta'(\text{ret})/\text{ret}] \rrbracket \eta$ . By the operational semantics for function calls,

$$(\Pi, y = k_i(E_1, \dots, E_{n_i}), \eta, h) \Downarrow (\eta[y \rightarrow \eta'(\text{ret})], h'),$$

and thus, the rule holds.

The first rule for existentials is

$$\frac{\Delta, x:\tau; \Gamma, \gamma \vdash \delta}{\Delta; \Gamma, \exists x:\tau. \gamma \vdash \delta} \quad x \notin \text{Mod}(\delta) \cup \text{FV}(\Gamma)$$

Suppose that for all well-formed  $\Pi$  and  $\eta \in \llbracket \Delta, x:\tau \rrbracket$ ,

$$\llbracket \Delta, x:\tau; \Pi \vdash \Gamma, \gamma \rrbracket \eta \text{ implies } \llbracket \Delta, x:\tau; \Pi \vdash \delta \rrbracket \eta,$$

and let  $\llbracket \Delta; \Pi \vdash \Gamma \rrbracket \eta$  and  $\llbracket \Delta; \Pi \vdash \exists x:\tau. \gamma \rrbracket \eta$ . This means  $\llbracket \Delta; \Pi \vdash \gamma \rrbracket \eta_{[x \rightarrow v]}$  for some  $v \in \llbracket \tau \rrbracket$ . Since  $x \notin \text{FV}(\Gamma)$ ,  $\llbracket \Delta; \Pi \vdash \Gamma \rrbracket \eta_{[x \rightarrow v]}$ . This implies  $\llbracket \Delta; \Pi \vdash \delta \rrbracket \eta_{[x \rightarrow v]}$ , and since  $x \notin \text{Mod}(\delta)$ , we have  $\llbracket \Delta; \Pi \vdash \delta \rrbracket \eta$  (an obvious property of specifications that do not contain variables is used here).

The other rule for existentials is

$$\frac{\Delta; \Gamma, \exists x:\tau. \gamma \vdash \delta}{\Delta, x:\tau; \Gamma, \gamma \vdash \delta} \quad x \notin \text{Mod}(\delta) \cup \text{FV}(\Gamma).$$

For soundness, first suppose  $\tau$  is inhabited and that for all well-formed  $\Pi$  and  $\eta \in \llbracket \Delta \rrbracket$ ,

$$\llbracket \Delta; \Pi \vdash \Gamma, \exists x:\tau. \gamma \rrbracket \eta \text{ implies } \llbracket \Delta; \Pi \vdash \delta \rrbracket \eta,$$

and suppose  $\llbracket \Delta, x:\tau; \Pi \vdash \Gamma, \gamma \rrbracket \eta$ . Since  $\tau$  is inhabited, this means

$$\llbracket \Delta, x:\tau; \Pi \vdash \Gamma, \gamma \rrbracket \eta_{[x \rightarrow \eta(x)]},$$

and since  $x \notin \text{FV}(\Gamma)$ , this implies

$$\llbracket \Delta, x:\tau; \Pi \vdash \Gamma, \exists x:\tau. \gamma \rrbracket \eta,$$

and thus,  $\llbracket \Delta; \Pi \vdash \delta \rrbracket \eta$ , as desired. If  $\tau$  is an empty type, one can make an easy case analysis on whether  $x$  occurs in  $\gamma$  or not.

Soundness of the downwards rule for universals is easy. For soundness of the upwards rule:

$$\frac{\Delta; \Gamma \vdash \forall x:\tau. \delta}{\Delta, x:\tau; \Gamma \vdash \delta} \quad x \notin \text{FV}(\Gamma) \cup \text{Mod}(\delta),$$

suppose for all well-formed  $\Pi$  and  $\eta \in \llbracket \Delta \rrbracket$ ,

$$\llbracket \Delta; \Pi \vdash \Gamma \rrbracket \eta \text{ implies } \llbracket \Delta; \Pi \vdash \forall x:\tau. \delta \rrbracket \eta,$$

and let  $\eta' \in \llbracket \Delta, x:\tau \rrbracket$ . Suppose  $\llbracket \Delta, x:\tau; \Pi \vdash \Gamma \rrbracket \eta'$ . Since  $x \notin \text{FV}(\Gamma)$ ,

$$\llbracket \Delta; \Pi \vdash \Gamma \rrbracket \eta' - x,$$

and this implies

$$\llbracket \Delta, x:\tau; \Pi \vdash \delta \rrbracket (\eta' - x)_{[x \rightarrow v]} \text{ for all } v \in \llbracket \tau \rrbracket.$$

This means in particular,

$$\llbracket \Delta, x:\tau; \Pi \vdash \delta \rrbracket \eta'_{[x \rightarrow \eta'(x)]},$$

which shows the desired result.  $\square$

### 3.3.3 A Derived Rule

There is an important rule for abstract function definitions that is derivable from the rules in Fig. 3.2. The rule is

$$\frac{\begin{array}{c} \Delta \vdash \hat{P}:\tau \\ \Delta; \Gamma \vdash \{P_1[\hat{P}/x]\} c_1 \{Q_1[\hat{P}/x]\} \\ \vdots \\ \Delta; \Gamma \vdash \{P_n[\hat{P}/x]\} c_n \{Q_n[\hat{P}/x]\} \\ \Delta; \Gamma, \exists x:\tau. (\{P_1\}k_1\{Q_1\} \wedge \dots \wedge \{P_n\}k_n\{Q_n\}) \vdash \{P\} c \{Q\} \end{array}}{\Delta; \Gamma \vdash \{P\} \text{ let } k_1(\bar{x}_1) = c_1, \dots, k_n(\bar{x}_n) = c_n \text{ in } c \text{ end } \{Q\}} x \notin \text{FV}(\{P\} c \{Q\}). \quad (3.4)$$

We show how this rule can be derived; for simplicity, we assume  $n = 1$  and that there are no parameters. The proof of the more general case is essentially the same as for this case. First, by the function definition rule,

$$\frac{\begin{array}{c} \Delta; \Gamma \vdash \{P_1[\hat{P}/x]\} c_1 \{Q_1[\hat{P}/x]\} \\ \Delta; \Gamma, \{P_1[\hat{P}/x]\} k_1 \{Q_1[\hat{P}/x]\} \vdash \{P\} c \{Q\} \end{array}}{\Delta; \Gamma \vdash \{P\} \text{ let } k_1 = c_1 \text{ in } c \{Q\}}.$$

The rule for existentials gives us

$$\frac{\Delta; \Gamma, \exists x:\tau. \{P_1\} k_1 \{Q_1\} \vdash \{P\} c \{Q\}}{\Delta, x:\tau; \Gamma, \{P_1\} k_1 \{Q_1\} \vdash \{P\} c \{Q\}},$$

so we need to establish

$$\Delta; \Gamma, \{P_1[\hat{P}/x]\} k_1 \{Q_1[\hat{P}/x]\} \vdash \{P\} c \{Q\}$$

given

$$\Delta, x:\tau; \Gamma, \{P_1\} k_1 \{Q_1\} \vdash \{P\} c \{Q\}.$$

But this follows from Lemma 3.6, since  $x$  is not free in  $\{P\} c \{Q\}$ .

## 3.4 Examples

We show an example of data abstraction and how it can be handled using our program logic. The example involves two implementations of a priority queue, and the intention is, of course, that client programs which use these implementations should be unaware of and unable to exploit details of the particular implementation used. Data abstraction is modeled via existential quantification over predicates, corresponding to the slogan “abstract types have existential type” [73].

We first define an abstract priority queue, and use abstract operations in several client programs to demonstrate uses of abstract operations and their specifications. We then show two *implementations* of the abstract module, and prove that these have specifications which make application of the abstract function definition rule (3.4) possible.

### 3.4.1 Reasoning with an Abstract Priority Queue

*Priority queues* are used frequently in programming, for example in scheduling algorithms for processes in operating systems [120]. They consist of pairs  $(p, v)$ , where  $v$  is a stored value, and  $p$  is the *priority* associated with  $v$ . In such a structure, one can then enqueue such pairs and extract an element with the highest priority. For simplicity, assume that the values are integers. Here is a simple grammar for such a structure.

$$Q ::= \varepsilon \mid (p, v) \cup Q.$$

Some operations on such queues are needed. They use the axiom of choice, and are defined by

$$\begin{aligned} \text{MaxPri}(\varepsilon) &= -1 \\ \text{MaxPri}((p, v) \cup Q) &= \text{Max}(p, \text{MaxPri}(Q)) \\ \text{MaxPair}(Q) &= \mathbf{choose}(\{(p, v) \in Q \mid p = \text{MaxPri}(Q)\}). \end{aligned}$$

Note that  $\text{MaxPair}$  is a nondeterministic operation. We assume a base type  $\text{PriQ}$  whose values are priority queues, and an operation  $\text{Set}$  which, given a priority queue, returns the multiset of pairs occurring in it. These types and operations are only used in the logic, *not* in programs. Moreover, type  $\text{PriQ}$  could have been encoded in our higher-order logic, but for simplicity, we just introduce it in the logic here.

We now discuss how to reason about client code which uses an abstract priority queue. First, since client programs cannot modify abstract values, there should be a predicate stating that there is a “handle” to a priority queue. Hence, we introduce the predicate

$$\text{repr}(q, Q),$$

which asserts that the integer denoted by  $q$  is a handle to the priority queue  $Q$  – but it does *not* say anything about how  $Q$  is represented. This will be used as an abstract predicate in our proofs (and thus plays the role of  $x$  in the abstract function definition rule (3.4)). Given this predicate, the following are reasonable specifications for the various operations on a priority queue.

**Creating a Queue** There should be an operation which enables a client program to create a priority queue. Its specification is

$$\{\text{emp}\} \mathbf{createqueue}() \{\text{repr}(ret, \varepsilon)\},$$

which merely states that upon creation of a queue, a handle to an empty priority queue is returned.

**Enqueing** There should be an operation for storing elements in a queue. The specification is

$$\{\text{repr}(q, Q)\} \mathbf{enqueue}(q, (p, v)) \{\text{repr}(q, (p, v) \cup Q)\}.$$

**Dequeing** There should be an operation for dequeing. We make sure not to dequeue from an empty queue via the specification

$$\begin{aligned} &\{\text{repr}(q, Q) \wedge Q \neq \varepsilon\} \\ &\mathbf{dequeue}(q) \\ &\{\exists Q', p, v. \text{repr}(q, Q') \wedge Q = (p, v) \cup Q' \wedge (p, v) = \text{MaxPair}(Q) \wedge ret = v\}. \end{aligned}$$

**Disposing a Queue** When done with a priority queue, we should be able to dispose it. Hence the specification

$$\{\text{repr}(q, Q)\} \mathbf{disposequeue}(q) \{\text{emp}\}.$$

### 3.4.2 Sample Programs

We consider examples of client programs which use priority queues, along with their specifications.

The first program creates a queue, enqueues some elements, and dequeues again. Here,  $Q''$  is used as a shorthand for  $\langle(2, 42), (4, 17)\rangle$ .

$$\begin{array}{l}
 \{\text{emp}\} \\
 \quad q = \mathbf{createqueue}(); \\
 \{\text{repr}(q, \varepsilon)\} \\
 \quad \mathbf{enqueue}(q, (4, 17)); \\
 \{\text{repr}(q, (4, 17) \cup \varepsilon)\} \\
 \quad \mathbf{enqueue}(q, (2, 42)); \\
 \{\text{repr}(q, Q'')\} \\
 \{\text{repr}(q, Q'') \wedge Q'' \neq \varepsilon\} \\
 \quad y := \mathbf{dequeue}(); \\
 \{\exists Q', p, v. \text{repr}(q, Q') \wedge Q'' = Q' \cup (p, v) \wedge (p, v) = \text{MaxElt}(Q'') \wedge y = v\} \\
 \Downarrow \\
 \{\text{repr}(q, \langle(2, 42)\rangle) \wedge y = 17\} \\
 \quad \mathbf{disposequeue}(q) \\
 \{\text{emp} \wedge y = 17\}.
 \end{array}$$

The implication in the middle of this derivation uses the rule of consequence. Henceforth, this kind of implications is used implicitly in proofs. That is, given a nonempty abstract priority queue  $Q$ , if  $\text{MaxPair}(Q) = (p, v)$  and  $Q' = Q \setminus (p, v)$  (such that  $Q = Q' \cup (p, v)$ ), we use the following specification for **dequeue**:

$$\begin{array}{l}
 \{\text{repr}(q, Q) \wedge Q \neq \varepsilon\} \\
 \quad \mathbf{dequeue}() \\
 \{\text{repr}(q, Q') \wedge \text{ret} = v\}.
 \end{array}$$

Here is another basic program (and its specification) which, however, uses two queues. In the third step, the specification for **createqueue** is used along with the frame rule, and the frame rule is used again in all subsequent steps.

```

{emp}
  q1 = createqueue();
{repr(q1, ε)}
  enqueue(q1, (3, 7));
{repr(q1, ⟨(3, 7)⟩)}
  q2 = createqueue();
{repr(q1, ⟨(3, 7)⟩) * repr(q2, ε)}
  enqueue(q1, (7, 3));
{repr(q1, ⟨(3, 7), (7, 3)⟩) * repr(q2, ε)}
  enqueue(q2, (4, 13));
{repr(q1, ⟨(3, 7), (7, 3)⟩) * repr(q2, ⟨(4, 13)⟩)}
  y1 := dequeue(q1);
{repr(q1, ⟨(3, 7)⟩) * repr(q2, ⟨(4, 13)⟩) ∧ y1 = 3}
  y2 := dequeue(q2);
{repr(q1, ⟨(3, 7)⟩) * repr(q2, ε) ∧ y1 = 3 ∧ y2 = 13}
  disposequeue(q1);
  disposequeue(q2);
{emp ∧ y1 = 3 ∧ y2 = 13}

```

This example illustrates that our notion of modularity is not static, as is the case in the paper [88]. In that setting, it is not possible to give arguments to function calls – one has to initialize variables which are laid out in the specification of each module. This makes it impossible to have multiple instances of the same data structure, as noted by Parkinson and Bierman [91].

It is illustrative to demonstrate that erroneous programs cannot have a specification in our system. We show two such examples. In the first example, we call **dequeue** on an empty queue.

```

{emp}
  q = createqueue()
  {repr(q, ε)}
  y = dequeue(q)
  {???)

```

Since the precondition for **dequeue** is that there is a non-empty queue represented, there is no assertion which can be put in place of ??? in this derivation.

In the second erroneous program, we dequeue from a disposed queue.

```

{emp}
  q = createqueue();
{repr(q, ε)}
  enqueue(q, (4, 42));
{repr(q, ⟨4, 42⟩)}
  disposequeue(q);
{emp}
  y = dequeue(q)
  {???)

```

Since the precondition for **dequeue** is that there is a non-empty queue represented (and this is not implied by emp), there is no assertion which can be put in place of ???

in this derivation. In some implementations of a priority queue, the **disposequeue** operation might be a no-op, and thus the **dequeue** would make sense operationally. But allowing this in our system would amount to exposing the implementation to the client program, contradicting the principle of *encapsulation* in data abstraction. This is even more evident in the program

```
{emp}
  q = createqueue();
{repr(q, ε)}
  newvar x; x := [q]
{???}.
```

In most implementations of priority queues,  $q$  denotes a pointer to a data structure in the heap which in some sense “represents” the priority queue  $Q$ , and therefore, it *would* make operational sense to dereference  $q$ . But, the abstract interface to the queue does not mention any heap cells, and therefore, there is no assertion we can fill in for  $???$ . This is fine, since disallowing client programs to dereference handles is in harmony with the principles of data abstraction, which is addressed here.

Note that the priority queues here do not involve any *ownership transfer* [88], since the priority queues hold simple data values (integers) only. We could have implemented queues where ownership of heap cells transfers back and forth between clients and modules. Although this would certainly be interesting, the core ideas of our approach to data abstraction in separation logic are presented via the examples at hand.

### 3.4.3 Implementations of Priority Queues

In this section, we first present two implementations and then discuss how these can be used to reason about data abstraction in our framework. One implementation uses a sorted, singly-linked list, whereas the other uses an unsorted doubly-linked list.

#### Sorted, Singly-linked Lists

Singly-linked lists (or rather, singly-linked list segments) are introduced in Reynolds’ introductory paper on separation logic [115]. The idea is that the predicate

$$\text{slist}(\alpha, i, j)$$

asserts that the finite sequence  $\alpha = (p_0, v_0), \dots, (p_n, v_n)$  of priority / value pairs (Reynolds’ just uses sequences of integers) is represented in the heap in a singly linked list. It is defined by induction on the length of  $\alpha$ .

$$\begin{aligned} \text{slist}(\varepsilon, i, j) &\stackrel{\text{def}}{=} \text{emp} \wedge i = j \\ \text{slist}((p, v) \cdot \alpha, i, j) &\stackrel{\text{def}}{=} \exists k. i \mapsto p, v, k * \text{slist}(\alpha, k, j). \end{aligned}$$

We show programs which implement the abstract priority queue with sorted singly linked lists in Appendix 3.A (we have overloaded the **dispose** operation, to dispose several heap cells). With a sorted list, the only non-trivial operations are to enqueue an element and to dispose a queue (since we do not require the queue to be empty when it is disposed). Here is a brief explanation of each of the programs.

The `slistcreate` program simply initializes an empty slist (it needs only return `null`). The implementation of `slistEnque` first finds the appropriate position in the list to insert the new element to keep the list sorted (this is what happens in the **while** loop), and then inserts the element. Since the list is sorted according to priorities, we always dequeue the first element of the list, so it is easy to implement `slistDeque`. Finally, to dispose a queue amounts to dispose an slist.

One can prove the specifications shown in Figure 3.3 for these implementations, using the proof rules from Section 3.3 plus lemmas from Reynolds' paper [115] about the slist predicate, and some obvious implications regarding sorted sequences. Our proofs use certain assertions and operations on sequences, which are easy to encode in higher-order logic. The predicate  $\text{sorted}(\alpha)$  is true iff the sequence  $\alpha$  is sorted (in decreasing order) according to priorities. For a nonempty sequence  $\alpha = (p_0, v_0), \dots, (p_n, v_n)$ ,  $\text{Max}(\alpha)$  is a pair from  $\alpha$  with the highest priority – note that this is non-deterministic. Finally, for a sequence  $\alpha$ ,  $\text{Set}(\alpha)$  is the multiset of pairs occurring in  $\alpha$ .

$$\begin{array}{l}
\{\text{emp}\} \\
\text{slistcreate} \\
\{\text{slist}(\varepsilon, \text{ret}, \text{null})\} \\
\\
\{\text{slist}(\alpha, q, \text{null}) \wedge \text{sorted}(\alpha)\} \\
\text{slistEnque}(q, (p, v)) \\
\{\exists \alpha'. \text{slist}(\alpha', q, \text{null}) \wedge \text{sorted}(\alpha') \wedge \text{Set}(\alpha') = \text{Set}(\alpha) \uplus (p, v)\} \\
\\
\{\text{slist}(\alpha, q, \text{null}) \wedge \text{sorted}(\alpha) \wedge \alpha \neq \varepsilon\} \\
\text{slistDeque}(q) \\
\left\{ \begin{array}{l} \exists p, v, \alpha'. \text{slist}(\alpha', q, \text{null}) \wedge \text{Set}(\alpha) = \text{Set}(\alpha') \uplus (p, v) \wedge \\ (p, v) = \text{Max}(\alpha) \wedge \text{ret} = v \end{array} \right\} \\
\\
\{\text{slist}(\alpha, q, \text{null})\} \\
\text{slistDispose}(q) \\
\{\text{emp}\}
\end{array}$$

Figure 3.3: Specifications for slist-implementation of priority queues

The hard part is to show the specification for `slistEnque`, which is the most complicated program in this implementation. For the diligent reader who wants to verify the specifications, there are some basic facts and the invariant of the **while** loop of `slistEnque` that was used in our proof in Appendix 3.C.

### Doubly-linked Lists

Doubly linked lists are also introduced by Reynolds [115]. The predicate

$$\text{dlist}(\alpha, i, i', j, j')$$

asserts that the sequence  $\alpha$  is represented in the heap by a doubly linked list segment from  $i$  to  $j'$ . It is defined by induction on the length of  $\alpha$ :

$$\begin{aligned} \text{dlist}(\varepsilon, i, i', j, j') &\stackrel{\text{def}}{=} \text{emp} \wedge i = j \wedge i' = j' \\ \text{dlist}((p, v) \cdot \alpha, i, j, i', j') &\stackrel{\text{def}}{=} \exists k. i \mapsto (p, v, k, i') * \text{dlist}(\alpha, k, i, j, j') \end{aligned}$$

In addition to this implementation, we implement priority queues by doubly linked lists, but in contrast to our implementation with singly linked lists, this implementation does not keep the list sorted. Consequently, the tricky part is to dequeue, whereas enqueueing is easy. The implementations are shown in Appendix 3.B. The handle  $q$  is used to point to a cell containing the values  $i, i', j, j'$  that determine the “boundaries” of the list, for technical reasons.

Here is a brief explanation to each of the programs in this implementation. The `dlistcreate` program just initializes an empty doubly linked list by storing *nulls* in the “handle cell” in the heap. To enqueue, just insert at the front of the doubly linked list, since the list is not kept sorted. The tricky part in this implementation is to dequeue, since we have to find the correct element of the list and take it out of the list. This is done in `dlistDeque`; in the **while** loop we traverse the list and find an element with the highest priority, and then take it out of the list, in a way according to the position of the element in the list.

One can prove the specifications shown in Fig. 3.4 for these implementations of the priority queue operations. The proofs of these specifications use the same assertions and operations regarding sequences as the proofs about the `slist` representation. Most of the proofs are straightforward. As mentioned, the tricky part of this implementation is `dlistDeque`, and the proof of this program is correspondingly the trickiest one. The diligent reader who wishes to verify the specifications in Fig. 3.4 may find hints in Appendix 3.D.

### 3.4.4 Representation Independence

We argue that our system may be used to reason about independence of the representation of data using the examples from Sec. 3.4.3. Intuitively, a client program should not be able to distinguish between the two implementations of a priority queue we have presented; this intuition is justified here.

Consider the two programs

```

c1 ≡
let
  createqueue() = slistcreate
  enqueue( $q, (p, v)$ ) = slistEnque( $q, (p, v)$ )
  dequeue( $q$ ) = slistDeque( $q$ )
  disposequeue( $q$ ) = slistDispose( $q$ )
in c end

```

$$\begin{array}{l}
\{\text{emp}\} \\
\text{dlistcreate} \\
\{\exists i, i', j, j'. q \mapsto i, i', j, j' * \text{dlist}(\varepsilon, i, i', j, j')\} \\
\\
\{\exists i, i', j, j'. q \mapsto i, i', j, j' * \text{dlist}(\alpha, i, i', j, j')\} \\
\text{dlistEnqueue}(q, (p, v)) \\
\{\exists i, i', j, j'. q \mapsto i, i', j, j' * \text{dlist}((p, v) \cdot \alpha, i, i', j, j')\} \\
\\
\{\exists i, i', j, j'. q \mapsto i, i', j, j' * \text{dlist}(\alpha, i, i', j, j') \wedge \alpha \neq \varepsilon\} \\
\text{dlistDeque}(q) \\
\left. \begin{array}{l}
\exists i, i', j, j', p, v, \alpha'. \text{dlist}(\alpha', i, i', j, j') \wedge \text{Set}(\alpha) = \text{Set}(\alpha') \uplus (p, v) \wedge \\
(p, v) = \text{Max}(\alpha) \wedge \text{ret} = v
\end{array} \right\} \\
\\
\{\exists i, i', j, j'. q \mapsto i, i', j, j' * \text{dlist}(\alpha, i, i', j, j')\} \\
\text{dlistDispose}(q) \\
\{\text{emp}\}
\end{array}$$

Figure 3.4: Specifications for dlist-implementation of Priority Queues

and

$$\begin{array}{l}
c_2 \equiv \\
\mathbf{let} \\
\quad \mathbf{createqueue}() = \text{dlistcreate} \\
\quad \mathbf{enqueue}(q, (p, v)) = \text{dlistEnqueue}(q, (p, v)) \\
\quad \mathbf{dequeue}(q) = \text{dlistDeque}(q) \\
\quad \mathbf{disposequeue}(q) = \text{dlistDispose}(q) \\
\mathbf{in } c \mathbf{ end,}
\end{array}$$

where  $c$  is a program which uses priority queues, for example the ones in Section 3.4.2. Intuitively, it should not matter which implementation is used, so that any specification we can show for one program, we should be able to show for the other. Consider the abstract function definition rule from Section 3.3.3, which can be used to verify such programs. The work we have already done can be used to apply this rule.

In the setting with singly linked lists,  $\hat{P}$  in the rule can be instantiated with the predicate  $\hat{P}_{\text{slist}}: (\text{PriQ} \times \text{Int}) \Rightarrow \text{Prop}$  defined by

$$\hat{P}_{\text{slist}} \equiv \lambda(Q, q). \exists \alpha. \text{sorted}(\alpha) \wedge \text{Set}(\alpha) = \text{Set}(Q) \wedge \text{slist}(\alpha, q, \text{null}).$$

Using the specifications mentioned in Section 3.4.3, it is not hard to show the following

specifications.

$$\begin{array}{l} \{\text{emp}\} \\ \text{slistcreate} \\ \{\hat{P}_{\text{slist}}(q, \varepsilon)\} \end{array}$$

$$\begin{array}{l} \{\hat{P}_{\text{slist}}(q, Q)\} \\ \text{slistEnqueue}(q, (p, v)) \\ \{\hat{P}_{\text{slist}}(q, Q \uplus (p, v))\} \end{array}$$

$$\begin{array}{l} \{\hat{P}_{\text{slist}}(q, Q) \wedge Q \neq \varepsilon\} \\ \text{slistDeque}(q) \\ \{\exists Q', p, v. \hat{P}_{\text{slist}}(q, Q') \wedge \text{Set}(Q) = \text{Set}(Q') \uplus (p, v) \wedge (p, v) = \text{MaxElt}(Q) \wedge \text{ret} = v\} \end{array}$$

$$\begin{array}{l} \{\hat{P}_{\text{slist}}(q, Q)\} \\ \text{slistDeque}(q) \\ \{\text{emp}\}. \end{array}$$

Similarly, if we define the predicate  $\hat{P}_{\text{dlist}}: (\text{PriQ} \times \text{Int}) \Rightarrow \text{Prop}$  by

$$\hat{P}_{\text{dlist}}(Q, q) \equiv \lambda(Q, q). \exists i, i', j, j', \alpha. q \mapsto i, i', j, j' * \text{dlist}(\alpha, i, i', j, j') \wedge \text{Set}(\alpha) = \text{Set}(Q),$$

the work in Section 3.4.3 entails the specifications

$$\begin{array}{l} \{\text{emp}\} \\ \text{dlistcreate}() \\ \{\hat{P}_{\text{dlist}}(\varepsilon, q)\} \end{array}$$

$$\begin{array}{l} \{\hat{P}_{\text{dlist}}(Q, q)\} \\ \text{dlistEnqueue}(q, (p, v)) \\ \{\hat{P}_{\text{dlist}}(Q \cup (p, v), q)\} \end{array}$$

$$\begin{array}{l} \{\hat{P}_{\text{dlist}}(q, Q) \wedge Q \neq \varepsilon\} \\ \text{dlistDeque}(q) \\ \{\exists Q', p, v, \alpha. \hat{P}_{\text{dlist}}(q, Q') \wedge Q = Q' \cup (p, v) \wedge (p, v) = \text{MaxElt}(Q) \wedge \text{ret} = v\} \end{array}$$

$$\begin{array}{l} \{\hat{P}_{\text{dlist}}(q, Q)\} \\ \text{dlistDispose}(q) \\ \{\text{emp}\}. \end{array}$$

The proofs of the programs in Section 3.4.2 which use the abstract predicate can be fitted into our framework. For example, the first of those programs has the specification

$$\Delta; \Gamma \vdash \{\text{emp}\} c \{\text{emp} \wedge y = 2\},$$

where  $\Delta$  is the context  $q:\text{Int}, y:\text{Int}, \text{repr}:(\text{PriQ} \times \text{Int}) \Rightarrow \text{Prop}$  and  $\Gamma$  is the environment

containing the abstract functions listed at the end of Section 3.4.1:

$$\begin{aligned}
& \{\text{emp}\} \mathbf{createqueue}() \{\text{repr}(ret, \varepsilon)\}, \\
& \{\text{repr}(q, Q)\} \mathbf{enqueue}(q, (v, p)) \{\text{repr}(q, (v, p) \cup Q)\}, \\
& \{\text{repr}(q, Q) \wedge Q \neq \varepsilon\} \\
& \quad \mathbf{dequeue}(q) \\
& \{\exists Q', p, v. \text{repr}(q, Q') \wedge Q = (p, v) \cup Q' \wedge (p, v) = \text{MaxElt}(Q) \wedge ret = v\}, \\
& \{\text{repr}(q, Q)\} \mathbf{disposequeue}(q) \{\text{emp}\}.
\end{aligned}$$

According to the rule (3.4) in Section 3.3.3, we can then derive the specification

$$q, : \text{Int}, y: \text{Int}; \emptyset \vdash \{\text{emp}\} \tilde{c} \{\text{emp} \wedge y = 2\},$$

where  $\tilde{c}$  is replaced by either  $c_1$  or  $c_2$ . Note that *the proof of the client program is independent of the implementation of the module it uses*. This is the sense in which our system can be used to reason about representation independence.

### 3.4.5 Polymorphic Types via Universal Quantification

We show that universally quantified predicates may be used to prove correct polymorphic operations on polymorphic data types.

The queue module example from the paper [88] is parametric in a predicate  $P$  at the meta-level. In higher-order separation logic, the parametricity may be expressed in the logic. Consider the following version of the parametric list predicate from the paper [88] just mentioned.

$$\text{list}(P, \beta, i) = \begin{cases} i = \text{null} \wedge \text{emp} & \text{if } \beta = \varepsilon \\ \exists j. i \mapsto x, j * P(x) * \text{list}(P, \beta', j) & \text{if } \beta = \langle x \rangle \cdot \beta' \end{cases}$$

The predicate  $P$  is required to hold for each element of the sequence  $\beta$  involved. Different instantiations of  $P$  yield different versions of the list, with different amounts of data stored in the list. If  $P \equiv \text{emp}$ , plain values are stored (and no ownership transfer to the queue module [88]), and if  $P \equiv x \mapsto -, -$ , addresses of cells are stored in the queue (and ownership of the cells is transferred in and out of the queue [88]).

Returning to higher-order separation logic, the definition of list may be formalized with

$$i: \text{Int}, \beta: \text{seqInt}, P: \text{Prop}^{\text{Int}} \vdash \text{list}(P, \beta, i): \text{Prop}$$

where we have used type  $\text{seqInt}$  of sequences of integers, which is easily definable in higher-order separation logic, and the definition of  $\text{list}(P, \beta, i)$  can be given by induction on  $\beta$  in the logic, in the same sense as the  $\text{slist}$  and  $\text{dlist}$  predicates from Section 3.4.3.

Suppose  $\mathbf{listRev}$  is the list reversal program in Fig. 3.5 (taken from the introduction of Reynolds' introductory paper [115]). Then one can easily show the specification

$$\{\text{list}(P, \beta, i)\} \mathbf{listRev} \{\text{list}(P, \beta^\dagger, i)\},$$

where  $\beta^\dagger$  is the reverse of the sequence  $\beta$ . By the introduction rule for universal quantification,

$$\beta: \text{seqInt} \vdash \forall P: \text{Prop}^{\text{Int}}. \{\text{list}(P, \beta, i)\} \mathbf{listRev} \{\text{list}(P, \beta^\dagger, i)\},$$

which expresses that  $\mathbf{listRev}$  is *parametric* in the sense that it, roughly speaking, reverses singly-linked lists of any type.

```

j := ;
while i ≠ do  k := [i + 1];
                [i + 1] := j;
                j := i;
                i := k
od

```

Figure 3.5: The list reversal program **listRev**

Thus we have *one* parametric correctness proof of a specification for **listRev**, which may then be used to prove correct different applications of **listRev** (to lists of different types).

For such parametric operations on polymorphic data types to be more useful, one would of course prefer a higher-order programming language instead of the first-order language considered here. Then one could, e.g., program the usual **map** function on lists, and provide a single parametric correctness proof for it. In future work we show how to make use of higher-order separation logic for a higher-order language, specifically by extending the separation-logic typing discipline for idealized algol recently introduced in joint work with Yang [25].

### 3.4.6 Invariance

We briefly consider an example, suggested to us by John Reynolds, which demonstrates that one may use universal quantification to specify that a command does not modify its input state. Stacks are disregarded here since they are not important for the argument.

Suppose our intention is to specify that some command  $c$  takes any heap  $h$  described by a predicate  $q$ , and produces a heap (assume for simplicity that  $c$  terminates), which is an extension of  $h$ . We might attempt to use a specification of the form:

$$\{q\} c \{q' * q\}. \quad (3.5)$$

This does not work, however, unless  $q$  is *strictly exact* [115] (for example, if  $q$  is the predicate  $\exists\beta:\text{seqInt}. \text{list}(\beta, i)$ , then  $c$  may delete some elements from the list in the input heap  $h$ ).

Instead, we use the specification

$$\forall p:\text{Prop}. \{q \wedge p\} c \{q' * p\}, \quad (3.6)$$

as seen by the following argument. Predicate  $q$  describes a set of heaps  $\llbracket q \rrbracket$ . For each  $h \in \llbracket q \rrbracket$ , let  $p_h = \{h\}$ . Suppose  $c$  terminates in heap  $h'$ . Then  $h' = h_1 * h$ , for some  $h_1$ . That is, the heap  $h$  is *invariant* under the execution of  $c$ , as intended.

Note that (3.6) is stronger than (3.5): by instantiating  $p$  with  $q$  in (3.6) we get (3.5). Thus if we wish to prove (3.5), we may prove something stronger (3.6), which may be easier (c.f., strengthening an induction hypothesis).

This illustrates that we can use universal quantification to express invariance of commands.

### 3.4.7 Predicates via Fixed Points

Recall the `slist` predicate from the priority queue example above. It is required to satisfy the following recursive equation:

$$\text{slist} = \lambda(x, s).(x = \text{null} \wedge \text{emp}) \vee (\exists p, v, k. x \mapsto p, v, k * \text{slist}(k, s)).$$

Solutions to such equations are definable in higher-order separation logic. Indeed, we define both minimal and maximal fixed points for any monotone operator on predicates, using standard encodings of fixed points. To wit, consider for notational simplicity an arbitrary predicate

$$q:\text{Prop} \vdash \varphi(q):\text{Prop}$$

satisfying that  $q$  only occurs positively in  $\varphi$ . Then

$$\mu q.\varphi(q) = \forall q.(\varphi(q) \rightarrow q) \rightarrow q$$

is the least fixed point for  $\varphi$  in the obvious sense that  $\varphi(\mu q.\varphi(q)) \rightarrow \mu q.\varphi(q)$  and  $\forall p.(\varphi(p) \rightarrow p) \rightarrow (\mu q.\varphi(q) \rightarrow p)$  holds in the logic. Likewise,

$$\nu q.\varphi(q) = \exists q.(q \rightarrow \varphi(q)) \wedge q$$

is the maximal fixed point for  $\varphi$ .

## 3.5 General Semantics for Higher-order BI

The semantics from Section 3.2.2 of the assertion language (higher-order predicate BI) is an instance of a general concept, which is introduced in this section.

Part of the pointer model of separation logic, namely that given by heaps (but not stacks), has been related to *propositional* BI, the logic of bunched implications introduced by O’Hearn and Pym [83]. We show that the correspondence may be extended to a precise correspondence between all of the pointer model (including stacks) and our notion of *predicate* BI. We introduce the notion of a *BI hyperdoctrine*, a simple extension of Lawvere’s notion of hyperdoctrine [66], and show that it soundly models predicate BI.

We first introduce Lawvere’s notion of a hyperdoctrine [66] and briefly recall how it can be used to model intuitionistic and classical first- and higher-order predicate logic (see, for example, the handbook chapter [98] and Jacobs’ book [61] for more explanations). We then define the notion of a BI hyperdoctrine, which is a straightforward extension of the standard notion of hyperdoctrine, and explain how it can be used to model predicate BI logic.

### 3.5.1 Hyperdoctrines

A first-order hyperdoctrine is a categorical structure tailored to model first-order predicate logic with equality. The structure has a base category  $\mathcal{C}$  for modeling the types and terms, and a  $\mathcal{C}$ -indexed category  $\mathcal{P}$  for modeling formulas.

**Definition 3.8 (First-order hyperdoctrines).** Let  $\mathcal{C}$  be a category with finite products. A *first-order hyperdoctrine*  $\mathcal{P}$  over  $\mathcal{C}$  is a contravariant functor  $\mathcal{P}:\mathcal{C}^{op} \rightarrow \mathbf{Poset}$  from  $\mathcal{C}$  into the category of partially ordered sets and monotone functions, with the following properties.

1. For each object  $X$ , the partially ordered set  $\mathcal{P}(X)$  is a Heyting algebra.

2. For each morphism  $f : X \rightarrow Y$  in  $\mathcal{C}$ , the monotone function  $\mathcal{P}(f) : \mathcal{P}(Y) \rightarrow \mathcal{P}(X)$  is a Heyting algebra homomorphism.
3. For each diagonal morphism  $\Delta_X : X \rightarrow X \times X$  in  $\mathcal{C}$ , the left adjoint to  $\mathcal{P}(\Delta_X)$  at the top element  $\top \in \mathcal{P}(X)$  exists. In other words, there is an element  $=_X$  of  $\mathcal{P}(X \times X)$  satisfying that for all  $A \in \mathcal{P}(X \times X)$ ,

$$\top \leq \mathcal{P}(\Delta_X)(A) \quad \text{iff} \quad =_X \leq A.$$

4. For each product projection  $\pi : \Gamma \times X \rightarrow \Gamma$  in  $\mathcal{C}$ , the monotone function  $\mathcal{P}(\pi) : \mathcal{P}(\Gamma) \rightarrow \mathcal{P}(\Gamma \times X)$  has both a left adjoint  $(\exists X)_\Gamma$  and a right adjoint  $(\forall X)_\Gamma$ :

$$\begin{aligned} A \leq \mathcal{P}(\pi)(A') & \quad \text{if and only if} \quad (\exists X)_\Gamma(A) \leq A' \\ \mathcal{P}(\pi)(A') \leq A & \quad \text{if and only if} \quad A' \leq (\forall X)_\Gamma(A). \end{aligned}$$

Moreover, these adjoints are natural in  $\Gamma$ , i.e., given  $s : \Gamma \rightarrow \Gamma'$  in  $\mathcal{C}$ ,

$$\begin{array}{ccc} \mathcal{P}(\Gamma' \times X) & \xrightarrow{\mathcal{P}(s \times \text{id}_X)} & \mathcal{P}(\Gamma \times X) \\ (\exists X)_{\Gamma'} \downarrow & & \downarrow (\exists X)_\Gamma \\ \mathcal{P}(\Gamma') & \xrightarrow{\mathcal{P}(s)} & \mathcal{P}(\Gamma) \end{array} \quad \begin{array}{ccc} \mathcal{P}(\Gamma' \times X) & \xrightarrow{\mathcal{P}(s \times \text{id}_X)} & \mathcal{P}(\Gamma \times X) \\ (\forall X)_{\Gamma'} \downarrow & & \downarrow (\forall X)_\Gamma \\ \mathcal{P}(\Gamma') & \xrightarrow{\mathcal{P}(s)} & \mathcal{P}(\Gamma). \end{array}$$

The elements of  $\mathcal{P}(X)$ , where  $X$  ranges over objects of  $\mathcal{C}$ , are referred to as  $\mathcal{P}$ -predicates.

**Interpretation of first-order logic in a first-order hyperdoctrine.** Given a (first-order) signature with types  $X$ , function symbols  $f : X_1, \dots, X_n \rightarrow X$ , and relation symbols  $R \subset X_1, \dots, X_n$ , a *structure* for the signature in a first-order hyperdoctrine  $\mathcal{P}$  over  $\mathcal{C}$  assigns an object  $\llbracket X \rrbracket$  in  $\mathcal{C}$  to each type, a morphism  $\llbracket f \rrbracket : \llbracket X_1 \rrbracket \times \dots \times \llbracket X_n \rrbracket \rightarrow \llbracket X \rrbracket$  to each function symbol, and a  $\mathcal{P}$ -predicate  $\llbracket R \rrbracket \in \mathcal{P}(\llbracket X_1 \rrbracket \times \dots \times \llbracket X_n \rrbracket)$  to each relation symbol. Any term  $t$  over the signature, with free variables in  $\Gamma = \{x_1 : X_1, \dots, x_n : X_n\}$  and of type  $X$  say, is interpreted as a morphism  $\llbracket t \rrbracket : \llbracket \Gamma \rrbracket \rightarrow \llbracket X \rrbracket$ , where  $\llbracket \Gamma \rrbracket = \llbracket X_1 \rrbracket \times \dots \times \llbracket X_n \rrbracket$ , by induction on the structure of  $t$  (in the standard manner in which terms are interpreted in categories).

Each formula  $\varphi$  with free variables in  $\Gamma$  is interpreted as a  $\mathcal{P}$ -predicate  $\llbracket \varphi \rrbracket \in \mathcal{P}(\llbracket \Gamma \rrbracket)$  by induction on the structure of  $\varphi$  using the properties given in Definition 3.8. For atomic formulas  $R(t_1, \dots, t_n)$ , the interpretation is given by  $\mathcal{P}(\langle \llbracket t_1 \rrbracket, \dots, \llbracket t_n \rrbracket \rangle)(\llbracket R \rrbracket)$ . In particular, the atomic formula  $t =_X t'$  is interpreted by the  $\mathcal{P}$ -predicate  $\mathcal{P}(\langle \llbracket t \rrbracket, \llbracket t' \rrbracket \rangle)(=_{\llbracket X \rrbracket})$ . The interpretation of other formulas is defined by structural induction. Assume  $\varphi, \varphi'$  are formulas with free variables in  $\Gamma$  and that  $\psi$  is a formula with free variables in  $\Gamma \cup \{x : X\}$ . Then,

$$\begin{aligned} \llbracket \top \rrbracket &= \top_H & \llbracket \varphi \wedge \varphi' \rrbracket &= \llbracket \varphi \rrbracket \wedge_H \llbracket \varphi' \rrbracket \\ \llbracket \perp \rrbracket &= \perp_H & \llbracket \varphi \vee \varphi' \rrbracket &= \llbracket \varphi \rrbracket \vee_H \llbracket \varphi' \rrbracket \\ & & \llbracket \varphi \rightarrow \varphi' \rrbracket &= \llbracket \varphi \rrbracket \rightarrow_H \llbracket \varphi' \rrbracket \\ \llbracket \forall x : X. \psi \rrbracket &= (\forall \llbracket X \rrbracket)_{\llbracket \Gamma \rrbracket}(\llbracket \psi \rrbracket) \in \mathcal{P}(\llbracket \Gamma \rrbracket) \\ \llbracket \exists x : X. \psi \rrbracket &= (\exists \llbracket X \rrbracket)_{\llbracket \Gamma \rrbracket}(\llbracket \psi \rrbracket) \in \mathcal{P}(\llbracket \Gamma \rrbracket), \end{aligned}$$

where  $\wedge_H, \vee_H$ , etc., is the Heyting algebra structure on  $\mathcal{P}(\llbracket \Gamma \rrbracket)$ .

A formula  $\varphi$  with free variables in  $\Gamma$  is said to be *satisfied* if  $\llbracket \varphi \rrbracket$  is the top element of  $\mathcal{P}(\llbracket \Gamma \rrbracket)$ . This notion of satisfaction is *sound* for intuitionistic predicate logic, in the sense that all provable formulas are satisfied. Moreover, it is *complete* in the sense that a formula is provable if it is satisfied in all structures in first-order hyperdoctrines. A first-order hyperdoctrine  $\mathcal{P}$  is sound for *classical* predicate logic in case all the fibres  $\mathcal{P}(X)$  are Boolean algebras and all the reindexing functions  $\mathcal{P}(f)$  are Boolean algebra homomorphisms.

**Definition 3.9 (Hyperdoctrines).** A (general) *hyperdoctrine* is a first-order hyperdoctrine with the following additional properties:  $\mathcal{C}$  is cartesian closed, and there is a Heyting algebra  $H$  and a natural bijection  $\Theta_X : \text{Obj}(\mathcal{P}(X)) \simeq \mathcal{C}(X, H)$ .

A hyperdoctrine is sound for higher-order intuitionistic predicate logic: the Heyting algebra  $H$  is used to interpret the type  $\text{Prop}$  of propositions and higher types (e.g.,  $\text{Prop}^X$ , the type for predicates over  $X$ ), are interpreted by exponentials in  $\mathcal{C}$ . The natural bijection  $\Theta_X$  is used to interpret substitution of formulas in formulas: Suppose  $\varphi$  is a formula with a free variable  $q$  of type  $\text{Prop}$  and with remaining free variables in  $\Gamma$ , and that  $\psi$  is a formula with free variables in  $\Gamma$ . Then  $\llbracket \psi \rrbracket \in \mathcal{P}(\llbracket \Gamma \rrbracket)$ ,  $\llbracket \varphi \rrbracket \in \mathcal{P}(\llbracket \Gamma \rrbracket \times H)$ , and  $\varphi[\psi/q]$  ( $\varphi$  with  $\psi$  substituted in for  $q$ ) is interpreted by  $\mathcal{P}(\langle \text{id}, \Theta(\llbracket \psi \rrbracket) \rangle)(\llbracket \varphi \rrbracket)$ . For more details see, e.g., the handbook chapter [98].

Again it is the case that a hyperdoctrine  $\mathcal{P}$  is sound for *classical* higher-order predicate logic in case all the fibres  $\mathcal{P}(X)$  are Boolean algebras and all the reindexing functions  $\mathcal{P}(f)$  are Boolean algebra homomorphisms.

**Example 3.10 (Canonical hyperdoctrine over a topos).** Let  $\mathcal{E}$  be a topos. It is well-known that  $\mathcal{E}$  models higher-order predicate logic, by interpreting types as objects in  $\mathcal{E}$ , terms as morphisms in  $\mathcal{E}$  and predicates as subobjects in  $\mathcal{E}$ . The topos  $\mathcal{E}$  induces a canonical  $\mathcal{E}$ -indexed hyperdoctrine  $\text{Sub}_{\mathcal{E}} : \mathcal{E}^{op} \rightarrow \mathbf{Poset}$ , which maps an object  $X$  in  $\mathcal{E}$  to the poset of subobjects of  $X$  in  $\mathcal{E}$  and a morphism  $f : X \rightarrow Y$  to the pullback functor  $f^* : \text{Sub}(Y) \rightarrow \text{Sub}(X)$ . Then the standard interpretation of predicate logic in  $\mathcal{E}$  coincides with the interpretation of predicate logic in the hyperdoctrine  $\text{Sub}_{\mathcal{E}}$ . Compared to the standard interpretation in toposes, however, hyperdoctrines do not require that predicates are always modeled by subobjects but can come from some other universe. This means that hyperdoctrines describe a wider class of models than toposes do.

### 3.5.2 BI Hyperdoctrines

Recall that a Heyting algebra is a bi-cartesian closed partial order, i.e., a partial order, which, when considered as a category, is cartesian closed ( $\top, \wedge, \rightarrow$ ) and has finite coproducts ( $\perp, \vee$ ). Further recall that a *BI algebra* is a Heyting algebra, which has an additional symmetric monoidal closed structure ( $\mathbb{I}, *, \multimap$ ) [102].

We now present a straightforward extension of (first-order) hyperdoctrines, which models first and higher-order predicate BI.

**Definition 3.11 (BI Hyperdoctrines).**

- A first-order hyperdoctrine  $\mathcal{P}$  over  $\mathcal{C}$  is a *first-order BI hyperdoctrine* in case all the fibres  $\mathcal{P}(X)$  are BI algebras and all the reindexing functions  $\mathcal{P}(f)$  are BI algebra homomorphisms.

- A *BI hyperdoctrine* is a first-order BI hyperdoctrine with the additional properties that  $\mathcal{C}$  is cartesian closed, and there is a BI algebra  $B$  and a natural bijection  $\Theta_X : \text{Obj}(\mathcal{P}(X)) \simeq \mathcal{C}(X, B)$ .

*First-order predicate BI* is first-order predicate logic with equality, extended with formulas  $I$ ,  $\varphi * \psi$ ,  $\varphi \multimap \psi$  satisfying the following rules (in any context  $\Gamma$  including the free variables of the formulas):

$$\begin{array}{ccc} (\varphi * \psi) * \theta \vdash_{\Gamma} \varphi * (\psi * \theta) & \varphi * (\psi * \theta) \vdash_{\Gamma} (\varphi * \psi) * \theta & \vdash_{\Gamma} \varphi \leftrightarrow \varphi * I \\ \varphi * \psi \vdash_{\Gamma} \psi * \varphi & \frac{\varphi \vdash_{\Gamma} \psi \quad \theta \vdash_{\Gamma} \omega}{\varphi * \theta \vdash_{\Gamma} \psi * \omega} & \frac{\varphi * \psi \vdash_{\Gamma} \theta}{\varphi \vdash_{\Gamma} \psi \multimap \theta} \end{array}$$

Our notion of predicate BI should not be confused with the one presented in Pym's book [102]; the latter seeks to include a BI structure on contexts but we do not attempt to do that here, since this is not what is used in separation logic. In particular, weakening at the level of variables is always allowed:

$$\frac{\varphi \vdash_{\Gamma} \psi}{\varphi \vdash_{\Gamma \cup \{x:X\}} \psi}.$$

We interpret first-order predicate BI in a first-order BI hyperdoctrine simply by extending the interpretation of first-order logic in first-order hyperdoctrine defined above by:

$$\begin{aligned} \llbracket I \rrbracket &= I_B \\ \llbracket \varphi * \psi \rrbracket &= \llbracket \varphi \rrbracket *_B \llbracket \psi \rrbracket \\ \llbracket \varphi \multimap \psi \rrbracket &= \llbracket \varphi \rrbracket \multimap_B \llbracket \psi \rrbracket, \end{aligned}$$

where  $I_B$ ,  $*_B$  and  $\multimap_B$  is the monoidal closed structure in the BI algebra  $\mathcal{P}(\llbracket \Gamma \rrbracket)$ . We then have:

**Theorem 3.12.** *The interpretation of first-order predicate BI given above is sound and complete.*

Likewise, BI hyperdoctrines form sound and complete models for higher-order predicate BI. Of course, a (first-order) BI hyperdoctrine is sound for classical BI in case all the fibres  $\mathcal{P}(X)$  are Boolean BI algebras and all the reindexing functions  $\mathcal{P}(f)$  are Boolean BI algebra homomorphisms. Here is a canonical example of a BI hyperdoctrine.

**Example 3.13 (BI hyperdoctrine over a complete BI algebra).** Let  $B$  be a complete BI algebra, i.e., it has all joins and meets. It determines a BI hyperdoctrine over the category **Set** as follows. For each set  $X$ , let  $\mathcal{P}(X) = B^X$ , the set of functions from  $X$  to  $B$ , ordered pointwise. Given  $f : X \rightarrow Y$ ,  $\mathcal{P}(f) : B^Y \rightarrow B^X$  is the BI algebra homomorphism given by composition with  $f$ . For example if  $s, t \in \mathcal{P}(Y)$ , i.e.,  $s, t : Y \rightarrow B$ , then  $\mathcal{P}(f)(s) = s \circ f : X \rightarrow B$  and  $s * t$  is defined pointwise as  $(s * t)(y) = s(y) * t(y)$ . Equality predicates  $=_X$  in  $B^{X \times X}$  are defined by

$$=_X(x, x') \stackrel{\text{def}}{=} \begin{cases} \top & \text{if } x = x' \\ \perp & \text{if } x \neq x' \end{cases},$$

where  $\top$  and  $\perp$  are the greatest and least elements of  $B$ , respectively. The quantifiers use set-indexed joins ( $\bigvee$ ) and meets ( $\bigwedge$ ). Specifically, given  $A \in B^{\Gamma \times X}$  one has

$$(\exists X)_{\Gamma}(A) \stackrel{\text{def}}{=} \lambda i \in \Gamma. \bigvee_{x \in X} A(i, x) \qquad (\forall X)_{\Gamma}(A) \stackrel{\text{def}}{=} \lambda i \in \Gamma. \bigwedge_{x \in X} A(i, x)$$

in  $B^{\Gamma}$ . The conditions in Definition 3.9 are trivially satisfied ( $\Theta$  is the identity).

There are plenty of examples of complete BI algebras: for any Grothendieck topos  $\mathcal{E}$  with an additional symmetric monoidal closed structure,  $\text{Sub}_{\mathcal{E}}(1)$  is a complete BI algebra, and for any monoidal category  $\mathcal{C}$  such that the monoid is cover preserving with respect to the Grothendieck topology  $J$ ,  $\text{Sub}_{\text{Sh}(\mathcal{C}, J)}(1)$  is a complete BI algebra [19, 104].

The following theorem shows that to get interesting models of higher-order predicate BI, it does not suffice to consider BI hyperdoctrines arising as the canonical hyperdoctrine over a topos (as in Example 3.10). Indeed this is the reason for introducing the more general BI hyperdoctrines.

**Theorem 3.14.** *Let  $\mathcal{E}$  be a topos and suppose  $\text{Sub}_{\mathcal{E}} : \mathcal{E}^{\text{op}} \rightarrow \mathbf{Poset}$  is a BI hyperdoctrine. Then the BI structure on each lattice  $\text{Sub}_{\mathcal{E}}(X)$  is trivial, i.e., for all  $\varphi, \psi \in \text{Sub}_{\mathcal{E}}(X)$ ,  $\varphi * \psi \leftrightarrow \varphi \wedge \psi$ .*

*Proof.* Let  $\mathcal{E}$  be a topos and suppose  $\text{Sub}_{\mathcal{E}} : \mathcal{E}^{\text{op}} \rightarrow \mathbf{Poset}$  is a BI hyperdoctrine. Let  $X$  be an object of  $\mathcal{E}$  and let  $\varphi, \psi, \psi' \in \text{Sub}_{\mathcal{E}}(X)$ . Furthermore let  $Y$  be the domain of the mono  $\varphi$ , and notice that the lattice  $\text{Sub}_{\mathcal{E}}(Y)$  can be characterized by

$$\text{Sub}_{\mathcal{E}}(Y) = \{\psi \wedge \varphi \mid \psi \in \text{Sub}_{\mathcal{E}}(X)\}. \quad (3.7)$$

Furthermore, notice that the order on  $\text{Sub}_{\mathcal{E}}(Y)$  is inherited from  $\text{Sub}_{\mathcal{E}}(X)$ , i.e.,

$$\text{For all } \chi, \chi' \in \text{Sub}_{\mathcal{E}}(Y), \chi \vdash_Y \chi' \text{ iff } \chi \vdash_X \chi'. \quad (3.8)$$

Since  $\wedge$  is modeled by pullback which by assumption preserves  $*$ , the following equations hold in  $\text{Sub}_{\mathcal{E}}(Y)$  (and therefore also in  $\text{Sub}_{\mathcal{E}}(X)$ ):

$$(\varphi \wedge \psi) *_Y (\varphi \wedge \psi') \leftrightarrow \varphi \wedge (\psi *_X \psi') \quad (3.9)$$

and

$$(\varphi \wedge \psi) \dashv*_Y (\varphi \wedge \psi') \leftrightarrow \varphi \wedge (\psi \dashv*_X \psi'). \quad (3.10)$$

By assumption,  $\text{Sub}_{\mathcal{E}}(Y)$  forms a BI algebra with connectives  $*_Y, \dashv*_Y$  and  $I_Y$ , so using the characterization of subobjects of  $Y$  given in (3.7), yields the following rule for each  $\chi \in \text{Sub}_{\mathcal{E}}(X)$ :

$$\frac{(\varphi \wedge \psi) *_Y (\varphi \wedge \psi') \vdash_Y \chi \wedge \varphi}{\varphi \wedge \psi \vdash_Y (\varphi \wedge \psi') \dashv*_Y (\chi \wedge \varphi)}$$

Using (3.8), (3.9), and (3.10) we deduce

$$\frac{\varphi \wedge (\psi *_X \psi') \vdash_X \chi \wedge \varphi}{\varphi \wedge \psi \vdash_X \varphi \wedge (\psi' \dashv*_X \chi)}$$

for all  $\varphi, \psi, \psi', \chi \in \text{Sub}_{\mathcal{E}}(X)$ , which implies

$$\frac{\frac{\varphi \wedge (\psi *_X \psi') \vdash_X \chi \wedge \varphi}{\varphi \wedge \psi \vdash_X \psi' \dashv*_X \chi}}{(\varphi \wedge \psi) *_X \psi' \vdash_X \chi} \quad (3.11)$$

Inserting  $\varphi \wedge (\psi *_X \psi')$  for  $\chi$  into (3.11) yields

$$\frac{\varphi \wedge (\psi *_X \psi') \vdash_X \varphi \wedge (\psi *_X \psi')}{(\varphi \wedge \psi) *_X \psi' \vdash_X \varphi \wedge (\psi *_X \psi')}. \quad (3.12)$$

Since the entailment above the line in (3.12) always holds,

$$(\varphi \wedge \psi) *_X \psi' \vdash_X \varphi \wedge (\psi *_X \psi').$$

This gives us projections for  $*_X$  by letting  $\psi$  be  $\top$ :

$$(\varphi *_X \psi') \dashv\vdash_X (\varphi \wedge \top) *_X \psi' \vdash_X \varphi \wedge (\top *_X \psi') \vdash_X \varphi.$$

Now, let  $\chi$  be the subobject  $(\varphi \wedge \psi) *_X \psi'$ , then  $\chi \leftrightarrow \chi \wedge \varphi$  due to the projections for  $*_X$ . Using (3.11) downwards-up, gives

$$\frac{(\varphi \wedge \psi) *_X \psi' \vdash_X (\varphi \wedge \psi) *_X \psi'}{\varphi \wedge (\psi *_X \psi') \vdash_X (\varphi \wedge \psi) *_X \psi'} \quad (3.13)$$

By (3.12) and (3.13) we conclude that for all  $\varphi, \psi, \psi' \in \text{Sub}_\varepsilon(X)$ ,

$$\varphi \wedge (\psi *_X \psi') \leftrightarrow (\varphi \wedge \psi) *_X \psi'. \quad (3.14)$$

We already noted the projections for  $*_X$ , so  $\top *_X I_X \vdash_X I_X$ , which entails  $\top \leftrightarrow I_X$ . Let  $\psi$  be  $\top$  in (3.14), then  $\varphi \wedge (\top *_X \psi') \leftrightarrow (\varphi \wedge \top) *_X \psi'$  and so  $\varphi \wedge \psi' \leftrightarrow \varphi *_X \psi'$ , as claimed.  $\square$

In fact, it is possible to make a slight strengthening of Theorem 3.14. We say that a logic has *full subset types* [61] if the following conditions are satisfied.

- For each formula  $\varphi(x_1, \dots, x_n)$ , there is a type  $\{x_1:\tau_1, \dots, x_n:\tau_n \mid \varphi(x_1, \dots, x_n)\}$ .
- For a term  $N$  of type  $\{x_1:\tau_1, \dots, x_n:\tau_n \mid \varphi(x_1, \dots, x_n)\}$ , in a context  $\Gamma$ , there is a term  $o(N)$  of type  $\tau_1 \times \dots \times \tau_n$  in  $\Gamma$ .
- The rule

$$\frac{\Gamma, y:\{x:X \mid \varphi\} \mid \theta[o(y)/x] \vdash \psi[o(y)/x]}{\Gamma, x:X \mid \theta, \varphi \vdash \psi} \quad (3.15)$$

is valid.

One can then show

**Proposition 3.15.** *If our notion of predicate BI has full subset types, then for all formulas  $\varphi, \psi$  in a context  $\Gamma$ ,*

$$\varphi \wedge \psi \dashv\vdash_\Gamma \varphi * \psi.$$

The proof is omitted here; it can be found in the technical report [21]. The following is an easy consequence.

**Corollary 3.16.** *Any BI hyperdoctrine which satisfies the rules for full subset types is trivial.*

The BI hyperdoctrine we use to model separation logic (the standard pointer model) satisfies all of the above except the downward direction of (3.15). When this is the case, we say that the logic has subset types, but not *full* subset types [61].

### 3.5.3 The Pointer Model as a BI Hyperdoctrine

We now show how the semantics of assertions from Section 3.2.2 is an instance of the general framework just described.

Let  $(H_\perp, *)$  be the discretely ordered set of heaps with a bottom element added to represent undefined, and let  $*$  :  $H_\perp \times H_\perp \rightarrow H_\perp$  be the total extension of  $*$  :  $H \times H \rightarrow H$  satisfying  $\perp * h = h * \perp = \perp$ , for all  $h \in H_\perp$ . This defines a partially ordered commutative monoid with the empty heap  $\{\}$  as the unit for  $*$ . The powerset of  $H$ ,  $\mathcal{P}(H)$  (without  $\perp$ ) is a complete Boolean BI algebra, ordered by inclusion and with monoidal closed structure defined by (for  $U, V \in \mathcal{P}(H)$ ):

- $I$  is  $\{\emptyset\}$
- $U * V := \{h * h' \mid h \in U \wedge h' \in V\} \setminus \{\perp\}$
- $U \multimap V := \bigcup \{W \subseteq H \mid (W * U) \subseteq V\}$ .

It can easily be verified directly that this defines a complete Boolean BI algebra; it also follows from more abstract arguments [104, 19].

Let  $S$  be the BI hyperdoctrine induced by the complete Boolean BI algebra  $\mathcal{P}(H)$  as in Example 3.13. To show that the interpretation of separation logic in this BI hyperdoctrine exactly corresponds to the standard pointer model presented above, we spell out the interpretation of separation logic in  $S$ .

A term  $t$  in a context  $\Gamma = \{x_1:\tau_1, \dots, x_n:\tau_n\}$  is interpreted as a morphism between sets. For example,

- $\llbracket x_i:\tau_i \rrbracket = \pi_i$ ,
- $\llbracket n \rrbracket$  is the map  $\llbracket n \rrbracket : \llbracket \Gamma \rrbracket \rightarrow \{*\} \rightarrow \mathbb{Z}$  which sends the unique element of the one-point set  $\{*\}$  to  $n$ ,
- $\llbracket t_0 \pm t_1 \rrbracket = \llbracket t_0 \rrbracket \pm \llbracket t_1 \rrbracket : \llbracket \Gamma \rrbracket \rightarrow \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$ , where  $\llbracket t_i \rrbracket : \llbracket \Gamma \rrbracket \rightarrow \mathbb{Z}$ , for  $i = 0, 1$ .

The interpretation of a formula  $\varphi$  in a context  $\Gamma = \{x_1:\tau_1, \dots, x_n:\tau_n\}$  is defined inductively. Let  $I = \llbracket \Gamma \rrbracket$  and write  $\bar{v}$  for elements of  $I$ . Then  $\varphi$  is interpreted as an element of  $\mathcal{P}(I)$ . The interpretation is given in Fig. 3.6.

Now it is easy to verify by structural induction on formulas  $\varphi$  that the interpretation in the BI hyperdoctrine  $S$  corresponds exactly to the semantics of terms of type Prop in Section 3.2.2, in the sense given by

**Theorem 3.17.** *Let the predicate  $\Delta \vdash \varphi : \text{Prop}$  have free variables in the context  $\Delta = x_1:\tau_1, \dots, x_n:\tau_n$ , and let  $(v_1, \dots, v_n) \in \llbracket \tau_1 \times \dots \times \tau_n \rrbracket$ . Then,*

$$\llbracket \varphi \rrbracket(v_1, \dots, v_n) = \llbracket \varphi \rrbracket_{(x_1 \rightarrow v_1, \dots, x_n \rightarrow v_n)},$$

where the semantics on the left is the one in Fig. 3.6 and the semantics on the right is the semantics from Section 3.2.2.

As a consequence, separation logic is sound for classical first-order BI. The correspondence also shows that it is indeed sensible to extend separation logic to higher-order since the BI hyperdoctrine  $S$  soundly models higher-order BI. Moreover, one can obtain a correspondence like that of Theorem 3.17 for other versions of separation logic.

$$\begin{aligned}
\llbracket t_1 \mapsto t_2 \rrbracket(\bar{v}) &= \{h \mid \text{dom}(h) = \{\llbracket t_1 \rrbracket(\bar{v})\} \text{ and } h(\llbracket t_1 \rrbracket(\bar{v})) = \llbracket t_2 \rrbracket(\bar{v})\} \\
\llbracket t_1 = t_2 \rrbracket(\bar{v}) &= H \text{ if } \llbracket t_1 \rrbracket(\bar{v}) = \llbracket t_2 \rrbracket(\bar{v}), \emptyset \text{ otherwise} \\
\llbracket \top \rrbracket(*) &= H \\
\llbracket \perp \rrbracket(*) &= \emptyset \\
\llbracket \text{emp} \rrbracket(*) &= \{h \mid \text{dom}(h) = \emptyset\} \\
\llbracket \varphi \wedge \psi \rrbracket(\bar{v}) &= \llbracket \varphi \rrbracket(\bar{v}) \cap \llbracket \psi \rrbracket(\bar{v}) \\
\llbracket \varphi \vee \psi \rrbracket(\bar{v}) &= \llbracket \varphi \rrbracket(\bar{v}) \cup \llbracket \psi \rrbracket(\bar{v}) \\
\llbracket \varphi \rightarrow \psi \rrbracket(\bar{v}) &= \{h \mid h \in \llbracket \varphi \rrbracket(\bar{v}) \text{ implies } h \in \llbracket \psi \rrbracket(\bar{v})\} \\
\llbracket \varphi * \psi \rrbracket(\bar{v}) &= \llbracket \varphi \rrbracket(\bar{v}) * \llbracket \psi \rrbracket(\bar{v}) \\
&= \{h_1 * h_2 \mid h_1 \in \llbracket \varphi \rrbracket(\bar{v}) \text{ and } h_2 \in \llbracket \psi \rrbracket(\bar{v})\} \setminus \{\perp\} \\
\llbracket \varphi \multimap \psi \rrbracket(\bar{v}) &= \llbracket \varphi \rrbracket(\bar{v}) \multimap \llbracket \psi \rrbracket(\bar{v}) \\
&= \{h \mid \llbracket \varphi \rrbracket(\bar{v}) * \{h\} \subseteq \llbracket \psi \rrbracket(\bar{v})\} \\
\llbracket \forall x:\tau. \varphi \rrbracket(\bar{v}) &= \bigcap_{v_x \in \llbracket \tau \rrbracket} (\llbracket \varphi \rrbracket(v_x, \bar{v})) \\
\llbracket \exists x:\tau. \varphi \rrbracket(\bar{v}) &= \bigcup_{v_x \in \llbracket \tau \rrbracket} (\llbracket \varphi \rrbracket(v_x, \bar{v}))
\end{aligned}$$

Figure 3.6: Semantics of Assertions in the Hyperdoctrine  $S$ 

**An intuitionistic model.** Consider again the set of heaps  $(H_\perp, *)$  with an added bottom  $\perp$ , as above. The order is now defined by

$$h_1 \sqsupseteq h_2 \quad \text{iff} \quad \text{dom}(h_1) \subseteq \text{dom}(h_2) \text{ and for all } x \in \text{dom}(h_1). h_1(x) = h_2(x).$$

Let  $I$  be the set of sieves on  $H$ , i.e., downwards closed subsets of  $H$ , ordered by inclusion. This is a complete BI algebra, as can be verified directly or by an abstract argument [19, 104]. Now let  $T$  be the BI hyperdoctrine induced by the complete BI algebra  $I$  as in Example 3.13. The interpretation of predicate BI in this BI hyperdoctrine corresponds exactly to the intuitionistic pointer model of separation logic, which is presented using a forcing style semantics in the paper [59].

**The permissions model.** It is possible to fit the permissions model of separation logic presented in the paper [29] into the framework presented here. The main point is that the set of heaps, which in that model map locations to values and permissions, has a binary operation  $*$ , which makes  $(H_\perp, *)$  a partially ordered commutative monoid.

**Remark 3.18.** The correspondences between separation logic and BI hyperdoctrines above illustrate that what matters for the interpretation of separation logic is the choice of BI algebra. Indeed, the main relevance of the topos-theoretic constructions in the article [104] for models of separation logic is that they can be used to construct suitable BI-algebras (as subobject lattices in categories of sheaves).

### 3.6 Other Applications of Higher-order BI

We have shown that it is completely natural and straightforward to interpret first-order predicate BI in first-order BI-hyperdoctrines and that the standard pointer model of separation logic corresponds to a particular case of BI-hyperdoctrine. Based on this correspondence, in this section some further consequences for separation logic are outlined.

### 3.6.1 Formalizing Separation Logic

The strength of separation logic has been demonstrated in numerous papers before. In the early days of separation logic, it was shown that it could handle simple programs for copying trees, deleting lists, etc. The first proof of a more realistic program appeared in Yang’s thesis [134], in which he showed correctness of the Schorr-Waite graph marking algorithm. Later, a proof of correctness of Cheney’s garbage collection algorithm was published in the paper [23], and other examples of correctness proofs of non-trivial algorithms may be found in Bornat’s paper [28]. In all of these papers, different simple extensions of core separation logic were used. For example, Yang used lists and binary trees as parts of his term language, and Birkedal et. al. introduced expression forms for finite sets and relations. It would seem that it is a weakness of separation logic that one has to come up with suitable extensions of it every time one has to prove a new program correct. In particular, it would make machine-verifiable formalizations of such proofs more burdensome and dubious if one would have to alter the underlying logic for every new proof.

The right way to look at these “extensions” is that they are really trivial definitional extensions of one and the same logic, namely the internal logic of the classical BI hyperdoctrine  $S$  presented in Section 3.5. The internal language of a BI hyperdoctrine  $\mathcal{P}$  over  $\mathcal{C}$  is formed as follows: to each object of  $\mathcal{C}$  one associates a type, to each morphism of  $\mathcal{C}$  one associates a function symbol, and to each predicate in  $\mathcal{P}(X)$  one associates a relation symbol. The terms and formulas over this signature (considered as a higher-order signature [61]) form the internal language of the BI hyperdoctrine. There is an obvious structure for this language in  $\mathcal{P}$ .

Let  $2 = \{\perp, \top\}$  be a two-element set (the subobject classifier of  $\text{Set}$ ). There is a canonical map  $\iota : 2 \rightarrow \mathcal{P}(H)$  which maps  $\perp$  to  $\{\}$  (the bottom element of the BI algebra  $\mathcal{P}(H)$ ) and  $\top$  to  $H$  (the top element of  $\mathcal{P}(H)$ ).

**Definition 3.19.** Let  $\varphi$  be an  $S$ -predicate over a set  $X$ , i.e., a function  $\varphi : X \rightarrow \mathcal{P}(H)$ . Call  $\varphi$  *pure* if  $\varphi$  factors through  $\iota$ .

Thus  $\varphi : X \rightarrow \mathcal{P}(H)$  is pure if there exists a map  $\chi_\varphi : X \rightarrow 2$  such that

$$\begin{array}{ccc} X & \xrightarrow{\varphi} & \mathcal{P}(H) \\ & \searrow \chi_\varphi & \nearrow \iota \\ & 2 & \end{array}$$

commutes. This corresponds to the notion of pure predicate traditionally used in separation logic [115].

The sub-logic of pure predicates is simply the standard classical higher-order logic of  $\text{Set}$ , and thus it is sound for classical higher-order logic. Hence one can use classical higher-order logic for defining lists, trees, finite sets and relations in the standard manner using pure predicates and prove the standard properties of these structures, as needed for the proofs presented in the papers referred to above. In particular, notice that recursive definitions of predicates, which in the papers [134, 23, 28] are defined at the meta level, can be defined inside the higher-order logic itself. For machine verification one would thus only need to formalize one and the same logic, namely a sufficient fragment of the internal logic of the BI hyperdoctrine (with obvious syntactic rules for when a formula is pure). The internal logic itself is “too big” (it can have class-many types and

function symbols, e.g.); hence the need for a fragment thereof, say classical higher-order logic with natural numbers.

### 3.6.2 Logical Characterizations of Classes of Assertions

Different classes of assertions, precise, monotone, and pure, are introduced by Reynolds [115], who notices that special axioms for these classes of assertions are valid. Such special axioms are exploited in the proof of Cheney’s garbage collector [23], where pure assertions are moved in and out of the scope of iterated separating conjunctions, and in the paper [88], where properties of precise assertions are crucially applied to verify soundness of the hypothetical frame rule. The different classes of assertions are defined semantically and the special axioms are validated using the semantics. We show how the higher-order features of higher-order separation logic allows a logical characterization of the classes of assertions, and logical proofs of the properties earlier taken as axioms. This is, of course, important for machine verification, since it means that the special classes of assertions and their properties can be expressed *in the logic*.

To simplify notation we just present the characterizations for *closed* assertions, the extension to open assertions is straightforward. Recall that closed assertions are interpreted in  $S$  as functions from  $1$  to  $\mathcal{P}(H)$ , i.e., as subsets of  $H$ .

In the proofs below, we use assertions which describe heaps in a canonical way. Since a heap  $h$  has finite domain, there is a unique (up to permutation) way to write an assertion  $p_h \equiv l_1 \mapsto n_1 * \dots * l_k \mapsto n_k$  such that  $\llbracket p_h \rrbracket = \{h\}$ .

**Precise assertions.** The traditional definition of a precise assertion is semantic, inasmuch as an assertion  $q$  is precise if, and only if, for all states  $(s, h)$ , there is at most one subheap  $h_0$  of  $h$  such that  $(s, h_0) \Vdash q$ . The following proposition logically characterizes closed precise assertions (at the semantic level, this characterization of precise predicates has been mentioned before [87]).

**Proposition 3.20.** *The closed assertion  $q$  is precise if, and only if, the assertion*

$$\forall p_1, p_2 : \text{Prop. } (p_1 * q) \wedge (p_2 * q) \rightarrow (p_1 \wedge p_2) * q \quad (3.16)$$

*is valid in the BI hyperdoctrine  $S$ .*

*Proof.* The “only-if” direction is trivial, so we focus on the other implication. Thus suppose (3.16) holds for  $q$ , and let  $h$  be a heap with two different subheaps  $h_1, h_2$  for which  $h_i \in \llbracket q \rrbracket$ . Let  $p_1, p_2$  be canonical assertions describing the heaps  $h \setminus h_1$  and  $h \setminus h_2$ , respectively. Then  $h \in \llbracket (p_1 * q) \wedge (p_2 * q) \rrbracket$ , so  $h \in \llbracket (p_1 \wedge p_2) * q \rrbracket$ , whence there is a subheap  $h' \subseteq h$  with  $h' \in \llbracket p_1 \wedge p_2 \rrbracket$ . This is a contradiction.  $\square$   $\square$

One can verify properties for precise assertions *in the logic* without using semantical arguments. For example, one can show that  $q_1 * q_2$  is precise if  $q_1$  and  $q_2$  are by the following logical argument: Suppose (3.16) holds for  $q_1, q_2$ . Then,

$$\begin{aligned} & (p_1 * (q_1 * q_2)) \wedge (p_2 * (q_1 * q_2)) \Rightarrow ((p_1 * q_1) * q_2) \wedge ((p_2 * q_1) * q_2) \\ \Rightarrow & ((p_1 * q_1) \wedge (p_2 * q_1)) * q_2 \Rightarrow ((p_1 \wedge p_2) * q_1) * q_2 \\ \Rightarrow & (p_1 \wedge p_2) * (q_1 * q_2), \end{aligned}$$

as desired.

**Monotone assertions.** A closed assertion  $q$  is defined to be *monotone* if, and only if, whenever  $h \in \llbracket q \rrbracket$  then also  $h' \in \llbracket q \rrbracket$ , for all extensions  $h' \supseteq h$ .

**Proposition 3.21.** *The closed assertion  $q$  is monotone if, and only if, the assertion  $\forall p:\text{Prop. } p * q \rightarrow q$  is valid in the BI hyperdoctrine  $S$ .*

This is easily verified, and again, one can show the usual rules for monotone assertions in the logic (without semantical arguments) using this characterization.

**Pure assertions.** Recall from above that an assertion  $q$  is pure iff its interpretation factors through 2. Thus a closed assertion is pure iff its interpretation is either  $\emptyset$  or  $H$ .

**Proposition 3.22.** *The closed assertion  $q$  is pure if, and only if, the assertion*

$$\forall p_1, p_2:\text{Prop. } (q \wedge p_1) * p_2 \leftrightarrow q \wedge (p_1 * p_2) \quad (3.17)$$

*is valid in the BI hyperdoctrine  $S$ .*

*Proof.* Again, the interesting direction here is the “if” implication. Hence, suppose (3.17) holds for the assertion  $q$ , and that  $h \in \llbracket q \rrbracket$ . For any heap  $h_0$ , we must then show that  $h_0 \in \llbracket q \rrbracket$ . This is done via the verification of two claims.

**Fact 1:** For all  $h' \subseteq h$ ,  $h' \in \llbracket q \rrbracket$ . *Proof:* Let  $p_1$  be a canonical description of  $h'$ , and  $p_2$  a canonical description of  $h \setminus h'$ . Then  $h \in \llbracket q \wedge (p_1 * p_2) \rrbracket$ , so  $h \in \llbracket (q \wedge p_1) * p_2 \rrbracket$ . This means there is a split  $h_1 * h_2 = h$  with  $h_1 \in \llbracket q \wedge p_1 \rrbracket$  and  $h_2 \in \llbracket p_2 \rrbracket$ . But then,  $h_2 = h \setminus h'$ , so  $h_1 = h'$ , and thus,  $h' \in \llbracket q \rrbracket$ .

**Fact 2:** For all  $h' \supseteq h$ ,  $h' \in \llbracket q \rrbracket$ . *Proof:* Let  $p_1$  and  $p_2$  be canonical descriptions of  $h$  and  $h' \setminus h$ , respectively. Then,  $h' \in \llbracket (q \wedge p_1) * p_2 \rrbracket$ , so  $h' \in \llbracket q \wedge (p_1 * p_2) \rrbracket$ , and in particular,  $h' \in \llbracket q \rrbracket$ , as desired.

Using Facts 1 and 2, we deduce  $h \in \llbracket q \rrbracket \Rightarrow \emptyset \in \llbracket q \rrbracket \Rightarrow h_0 \in \llbracket q \rrbracket$ .  $\square$

## 3.7 Related Work

There are references to related work throughout the paper. We give pointers to more related work here.

As mentioned, Parkinson and Bierman proposed a system for reasoning about data abstraction [91], in which they introduce a notion of “abstract predicates”. Our approach is more straightforward and natural, as we have already argued. Prior to that, Reddy gave a semantics for objects and classes [105], in which he likewise models data abstraction via existential quantification. The main difference from this work compared to the present is that Reddy considers a programming language without heap manipulating constructs, but which is higher-order. Furthermore, Reddy uses a more sophisticated type system than ours, and types are interpreted relationally. It is ongoing work to give a relational interpretation to a higher-order programming language with heap-manipulating constructs.

Kohei Honda’s group has numerous papers on higher-order imperative languages [18, 57]. The similarities between their work and the present is that both seek to reason about equivalence of programs in programming languages with pointers. Their work is, however, not restricted to our programming language with simple procedures. The main

differences include: (i) Their logic does not include a  $*$  connective; instead they use predicate logic with equality to keep track of aliasing. This makes local reasoning harder. (ii) The interpretation of triples is not “tight” in their work. For example, the triple

$$\{\mathbf{true}\} [x] := 4 \{\mathbf{true}\}$$

is valid in the setting of Honda *et al.*, but not in separation logic. (iii) One of the goal of Honda *et al.*'s work is to show observational equivalence. Although intriguing, we do not aim to answer such questions in the present work. See the long version of the paper [18] for an extensive comparison of their work to separation logic.

### 3.8 Conclusion

We have extended the assertion language and specification language of separation logic to higher-order and presented a model for it. Further, we argued that this is a useful extension. In particular, we showed shown that programs which use abstract data types with with internal (hidden) resources can be proved correct, and illustrated how universal quantification over predicates can be used to reason about polymorphic operations on data types. Further, we introduced the notion of a BI hyperdoctrine and showed that our semantics is an instance of this general concept, inasmuch as our interpretation of predicates coincides with the standard interpretation of predicates in a hyperdoctrine. Finally, we showed that the general concept of hyperdoctrines is needed, since one cannot hope to get interesting models of predicate BI by extending the standard interpretation of predicate logic in toposes.

**Acknowledgments** The authors wish to thank Carsten Butz and the anonymous referees of previous versions of the work in this paper for helpful comments and insights.

## Appendix 3

### 3.A Implementation Using slists

Creating a queue with slists:

```
slistcreate  $\stackrel{\text{def}}{=} \text{return null}$ 
```

Enqueing with slists:

```
slistEnque( $q, (p, v)$ )  $\stackrel{\text{def}}{=} \text{newvar temp, tempPri, temp, found, new;}$   

  ptemp := null;  

  if ( $q = \text{null}$ )  

     $q := \text{cons}(p, v, \text{null})$   

  else  

    tempPri := [ $q$ ];  

    if ( $p \geq \text{tempPri}$ )  

       $q := \text{cons}(p, v, q)$   

    else  

      ptemp :=  $q$ ;  

      temp := [ $q + 2$ ];  

      found := false;  

      while ( $\text{temp} \neq \text{null} \wedge \text{found} = \text{false}$ )  

        tempPri := [ $\text{temp}$ ];  

        if ( $p \geq \text{tempPri}$ )  

          found := true  

        else  

          ptemp := temp;  

          temp := [ $\text{ptemp} + 2$ ]  

      fi  

    od;  

    new :=  $\text{cons}(p, v, \text{temp})$ ;  

    [ $\text{ptemp} + 2$ ] := new  

  fi  

fi
```

Dequeing with slists:

```
slistDeque( $q$ )  $\stackrel{\text{def}}{=} \text{newvar temp, maxVal;}$   

  temp :=  $q$ ;  

  maxVal := [ $q + 1$ ];  

   $q := [q + 2]$ ;  

  dispose(temp, temp + 1, temp + 2);  

  return maxVal
```

Disposing a with slist-queue:

```
slistDispose( $q$ )  $\stackrel{\text{def}}{=} \text{newvar temp;}$   

  while ( $q \neq \text{null}$ ) do  

    temp :=  $q$ ;  

     $q := [q + 2]$ ;  

    dispose(temp, temp + 1, temp + 2)  

  od
```

### 3.B Implementation Using dlists

Creating a queue with dlists:

```
dlistcreate  $\stackrel{\text{def}}{=}
q := \mathbf{cons}(null, null, null, null);
\mathbf{return} q$ 
```

Enqueing with dlists:

```
dlistEnque(q)  $\stackrel{\text{def}}{=}
\mathbf{newvar} i, i', j, j', \mathbf{temp};
i := [q]; i' := [q + 1]; j := [q + 2]; j' := [q + 3];
\mathbf{if} (i = j)
  i := \mathbf{cons}(p, v, j, i');
  j' := i
\mathbf{else}
  \mathbf{temp} := \mathbf{cons}(p, v, i, i');
  [i + 3] := \mathbf{temp};
  i := \mathbf{temp}
\mathbf{fi};
[q] := i; [q + 1] := i'; [q + 2] := j; [q + 3] := j'$ 
```

Dequeuing with dlists

```

dlistDeque( $q$ )  $\stackrel{\text{def}}{=}$ 
  newvar  $i, i', j, j', \text{max}, \text{maxP}, \text{maxVal}, \text{temp}, \text{tempP}, \text{temp}_0, \text{temp}_1$ ;
   $i := [q]; i' := [q + 1]; j := [q + 2]; j' := [q + 3];$ 
   $\text{max} := i;$ 
   $\text{maxP} := [i];$ 
   $\text{temp} := [i + 2];$ 
  while ( $\text{temp} \neq j$ ) do
     $\text{tempP} := [\text{temp}];$ 
    if ( $\text{tempP} > \text{maxP}$ )
       $\text{max} := \text{tempP};$ 
       $\text{maxP} := \text{tempP}$ 
    else skip
    fi;
     $\text{temp} := [\text{temp} + 2]$ 
  od;
   $\text{maxVal} := [\text{max} + 1];$ 
  if ( $\text{max} = i$ )
    if ( $\text{max} = j'$ )
      dispose( $\text{max}$ );
       $i := j; i' := j'$ 
    else
       $\text{temp} := i;$ 
       $i := [i + 2];$ 
       $[i + 3] := i;$ 
      dispose( $\text{max}, \text{max} + 1, \text{max} + 2, \text{max} + 3$ )
    fi
  else
    if ( $\text{max} = j'$ )
       $j' := [j' + 3];$ 
       $[j' + 2] := j;$ 
      dispose( $\text{max}, \text{max} + 1, \text{max} + 2, \text{max} + 3$ )
    else
       $\text{temp}_0 := [\text{max} + 2];$ 
       $\text{temp}_1 := [\text{max} + 3];$ 
       $[\text{temp}_0 + 3] := \text{temp}_1;$ 
       $[\text{temp}_1 + 2] := \text{temp}_0;$ 
      dispose( $\text{max}, \text{max} + 1, \text{max} + 2, \text{max} + 3$ )
    fi;
  fi;
   $[q] := i; [q + 1] := i'; [q + 2] := j; [q + 3] := j'$ 
  return  $\text{maxVal}$ 

```

Disposing with dlist-queue:

```

dlistDispose( $q$ )  $\stackrel{\text{def}}{=}$ 
  newvar  $i, j, \text{temp};$ 
   $i := [q]; j := [q + 2];$ 
  while ( $i \neq j$ )
     $\text{temp} := i;$ 
     $i := [\text{temp} + 2];$ 
    dispose( $\text{temp}, \text{temp} + 1, \text{temp} + 2, \text{temp} + 3$ )
  od;
  dispose( $q, q + 1, q + 2, q + 3$ )

```

### 3.C Hints for Proof of slistEnque

An invariant of the **while** loop in slistEnque is

$$\begin{aligned} & \exists \beta, \beta', p', v'. \\ & \text{slist}(\beta, i, \text{ptemp}) * \text{ptemp} \mapsto (p', v', \text{temp}) * \text{slist}(\beta', \text{temp}, j) \wedge \\ & ((\text{found} \wedge \text{sorted}(\beta \cdot (p', v') \cdot (p, v) \cdot \beta')) \vee p' > p) \wedge \\ & \alpha = \beta \cdot (p', v') \cdot \beta' \wedge \text{sorted}(\alpha) \end{aligned}$$

Here are some properties about the slist predicate that we used in the proof.

$$\begin{aligned} & \text{slist}(\alpha, i, \text{null}) \wedge i \neq j \Rightarrow \exists \alpha', p, v, k. \alpha = (p, v) \cdot \alpha' \wedge i \mapsto (p, v, k) * \text{slist}(\alpha', k, j) \\ & k \mapsto (p, v, i) * \text{slist}(\alpha, i, j) \Rightarrow \text{slist}((p, v) \cdot \alpha, k, j) \\ & \text{slist}(\alpha, i, k) * k \mapsto (p, v, j) \Rightarrow \text{slist}(\alpha \cdot (p, v), i, j) \end{aligned}$$

### 3.D Hints for Proof of dlistDeque

An invariant of the **while** loop in this program is

$$\begin{aligned} & \exists i, i', j, j'. q \mapsto (i, i', j, j') * \\ & (\exists \beta, \beta', \beta'', \text{ptemp}, \text{pmax}. \\ & \quad \text{dlist}(\beta, i, i', \text{max}, \text{pmax}) * \\ & \quad \text{dlist}(\beta', \text{max}, \text{pmax}, \text{temp}, \text{ptemp}) * \\ & \quad \text{dlist}(\beta'', \text{temp}, \text{ptemp}, j, j') \wedge \\ & \quad \alpha = \beta \cdot \beta' \cdot \beta'' \wedge \beta' \neq \varepsilon \wedge \text{maxP} = \text{Max}(\beta \cdot \beta') = \text{head}(\beta'')). \end{aligned}$$

The cell containing the “boundary values” for the *slist* need not appear in most of the proof but can be “framed in” via the frame rule.

The following basic properties about the dlist predicate have been used in our proof of the dlist implementation of priority queues. Some of these are also listed in [115].

$$\begin{aligned} & \text{dlist}(\alpha, i, i', j, j') \wedge i \neq j \Rightarrow \text{dlist}(\alpha, i, i', j, j') \wedge \alpha \neq \varepsilon \\ & \text{dlist}(\alpha, i, i', j, j') \wedge \alpha \neq \varepsilon \Rightarrow \\ & \quad \exists p, v, k, \alpha'. \alpha = (p, v) \cdot \alpha' \wedge i \mapsto (p, v, k, i') * \text{dlist}(\alpha', k, i, j, j') \\ & i = i' \wedge j = j' \wedge \text{emp} \Rightarrow \text{dlist}(\varepsilon, i, i', j, j') \\ & \text{dlist}(\alpha, i, i', k, k') * \text{dlist}(\alpha', k, k', j, j') \Rightarrow \text{dlist}(\alpha \cdot \alpha', i, i', j, j') \end{aligned}$$



## Chapter 4

# Semantics of Separation-logic Typing and Higher-Order Frame Rules

### Abstract

We show how to give a coherent semantics to programs that are well-specified in a version of separation logic for a language with higher types: idealized algol extended with heaps (but with immutable stack variables). In particular, we provide simple sound rules for deriving higher-order frame rules, allowing for local reasoning.

### Preface

This chapter is a reprint of the conference paper [25], which I co-authored with Lars Birkedal from IT University of Copenhagen and Hongseok Yang from ERC-ACI, Seoul National University.

## 4.1 Introduction

Separation logic [115, 113, 59, 106, 85, 23] is a Hoare-style program logic, and variants of it have been applied to prove correct interesting pointer algorithms such as copying a dag, disposing a graph, the Schorr-Waite graph algorithm, and Cheney's copying garbage collector. The main advantage of separation logic compared to ordinary Hoare logic is that it facilitates *local reasoning*, formalized via the so-called *frame rule* using a connective called *separating conjunction*. The development of separation logic has mostly focused on *low-level* languages with heaps and pointers, although in recent work [88] it was shown how to extend separation logic to a language with a simple kind of procedures, and a second-order frame rule was proved sound.

Our aim here is to extend the study of separation logic to *high-level* languages, in particular to *higher-order* languages, in such a way that a wide collection of frame rules are sound, thus allowing for local reasoning in the presence of higher-order procedures. For concreteness, we choose to focus on the language of idealized algol extended with heaps and pointers and we develop a semantics for this language in which all commands and procedures are appropriately local. Our approach is to refine the type system of idealized algol extended with heaps, essentially by making specifications be types, and give semantics to well-specified programs. Thus we develop a *separation-logic type system*

for idealized algol extended with heaps. It is a dependent type theory and the types include Hoare triples, rules corresponding to the rules of separation logic, and subtyping rules formalizing higher-order versions of the frame rule of separation logic.

Our type system is related to modern proposals for type systems for low-level imperative languages, such as TAL [77], in that types may express state changes (since they include forms of Hoare triples as types). The type system for TAL was proved sound using an operational semantics. We provide a soundness proof of our type system using a denotational semantics which we, moreover, formally relate to the standard semantics for idealized algol [89, 109]. The denotational semantics of a well-typed program is given by induction on its typing derivation and the relation to the standard semantics for idealized algol is then used to prove that the semantics is *coherent* (i.e., is independent of the chosen typing derivation). We should perhaps stress that soundness is not a trivial issue: Reynolds has shown [88] that already the soundness of the second-order frame rule is tricky, by proving that if a proof system contains the second-order frame rule and the conjunction rule, together with the ordinary frame rule and rule of Consequence, then the system becomes inconsistent. The semantics of our system proves that if we drop the conjunction rule, then we get soundness of all higher-order frame rules, including the second-order one.

In idealized algol, variables are allocated on a stack and they are mutable (i.e., one can assign to variables). We only consider *immutable* variables (as in the ML programming language) for simplicity. The reason for this choice is that all mutation then takes place in the heap and thus we need not bother with so-called *modifies clauses* on frame rules, which become complicated to state already for the second-order frame rule [88].

We now give an intuitive overview of the technical development. Recall that the standard semantics of idealized algol is given using the category CPO of pointed complete partial orders and continuous functions. Thus types are interpreted as pointed complete partial orders and terms (programs) are interpreted as continuous functions. The semantics of our refined type system is given by refining the standard semantics. A type  $\theta$  in our refined type system specifies which elements of the “underlying” type in the standard semantics satisfy the specification corresponding to  $\theta$  and are appropriately local (to ensure soundness of the frame rules), that is, it “extracts” those elements. Moreover, the semantics also equates elements, which cannot be distinguished by clients, that is, it quotients some of the extracted elements. Corresponding to these two aspects of the semantics we introduce two categories,  $\mathcal{C}$  and  $\mathcal{D}$ , where  $\mathcal{C}$  just contains the extracted elements and  $\mathcal{D}$  is a quotient of  $\mathcal{C}$ . Thus there is a faithful functor from  $\mathcal{C}$  to CPO and a full functor from  $\mathcal{C}$  to  $\mathcal{D}$ . We show that the categories  $\mathcal{C}$  and  $\mathcal{D}$  are cartesian closed and have additional structure to interpret the higher-order frame rules, and that the mentioned functors preserve all this structure. The semantics of our type system is then given in the category  $\mathcal{D}$  and the functors relating  $\mathcal{C}$ ,  $\mathcal{D}$ , and CPO are then used to prove coherence of the semantics. In fact, as mentioned above, our type system is a *dependent* type theory, with dependent product type  $\Pi_i \theta$  intuitively corresponding to the specification given by universally quantifying  $i$  in the specification corresponding to  $\theta$  (the usual Curry-Howard correspondence). For this reason the semantics is really not given in  $\mathcal{D}$  but rather in the family fibration  $Fam(\mathcal{D}) \rightarrow \mathbf{Set}$  over  $\mathcal{D}$ .

The remainder of this paper is organized as follows. In Section 4.2, we define the storage model and assertion language used in this paper, thus setting the stage for our model. In Section 4.3, we provide the syntax of the version of idealized algol we use in this paper. In particular, we introduce our separation-logic type system, which includes

an extended subtyping relation. In Section 4.4, we present the main contribution of the paper, a model which allows a sound interpretation, which we also show to be coherent and in harmony with the standard semantics. In the last sections we give pointers to related and future work, and conclude.

## 4.2 Storage Model and Assertion Language

We use the usual storage model of separation logic with one minor modification: we make explicit the shape of stack storage. Let  $\text{lds} = \{i, j, \dots\}$  be a countably infinite set of variables, and let  $\Delta$  range over finite subsets of  $\text{lds}$ . We use the following semantic domains:

$$\begin{aligned} \eta &\in \llbracket \Delta \rrbracket \stackrel{\text{def}}{=} \Delta \rightarrow \text{Int}, \\ h &\in \text{Heap} \stackrel{\text{def}}{=} \text{Nat} \rightarrow_{\text{FIN}} \text{Int}, \\ (\eta, h) &\in \text{State}(\Delta) \stackrel{\text{def}}{=} \llbracket \Delta \rrbracket \times \text{Heap}. \end{aligned}$$

The set  $\Delta$  models the set of variables in scope, and an element  $\eta$  in  $\llbracket \Delta \rrbracket$  specifies the values of those stack variables. We sometimes call  $\eta$  an *environment* instead of a *stack*, in order to emphasize that all variables are immutable. An element  $h$  in *Heap* denotes a heap; the domain of  $h$  specifies the set of allocated cells, and the actual action of  $h$  determines the contents of those allocated cells. We recall the disjointness predicate  $h\#h'$  and the (partial) heap combination operator  $h \cdot h'$  from separation logic. The predicate  $h\#h'$  means that  $\text{dom}(h) \cap \text{dom}(h') = \emptyset$ ; and,  $h \cdot h'$  is defined only for such disjoint heaps  $h$  and  $h'$ , and in that case, it denotes the combined heap  $h \cup h'$ .

Properties of states are expressed using the assertion language of classical separation logic [115]:<sup>1</sup>

$$\begin{aligned} E &::= i \mid 0 \mid 1 \mid E + E, \\ P &::= E = E \mid E \mapsto E \mid \emptyset \mid P * P \mid \text{true} \mid P \wedge P \mid \neg P \mid \forall i. P. \end{aligned}$$

The assertion  $E \mapsto E'$  means that the current heap has only one cell  $E$  and, moreover, that the content of the cell is  $E'$ . When we do not care about the contents, we write  $E \mapsto -$ ; formally, this is an abbreviation of  $\neg(\forall i. \neg E \mapsto i)$  for some  $i$  not occurring in  $E$ . The next two assertions,  $\emptyset$  and  $P * Q$ , are the most interesting features of this assertion language. The empty predicate  $\emptyset$  means that the current heap is empty, and the separating conjunction  $P * Q$  means that the current heap can be partitioned into two parts, one satisfying  $P$  and another satisfying  $Q$ .

As in the storage model, we make explicit which set of free variables we are considering an expression or an assertion under. Thus, letting  $\text{fiv}$  be a function that takes an expression or an assertion and returns the set of free variables, we often write assertions as  $\Delta \vdash P$  to indicate that  $\text{fiv}(P) \subseteq \Delta$ , and that  $P$  is currently being considered for environments of the shape  $\Delta$ . Likewise, we often write  $\Delta \vdash E$  for expressions.

The interpretations of an expression  $\Delta \vdash E$  and an assertion  $\Delta \vdash P$  are of the forms

$$\llbracket \Delta \vdash E \rrbracket : \llbracket \Delta \rrbracket \rightarrow \text{Int}, \quad \llbracket \Delta \vdash P \rrbracket : \llbracket \Delta \rrbracket \rightarrow \mathcal{P}(\text{Heap}).$$

<sup>1</sup>The assertion language of separation logic also contains the separating implication  $\ast$ . Since that connective does not raise any new issues in connection with the present work, we omit it here.

The interpretation of expressions is standard, just like that of assertions. We include part of the definition of the interpretation of assertions here.

$$\begin{aligned} \llbracket \Delta \vdash E \mapsto E' \rrbracket_\eta &= \{ \llbracket \Delta \vdash E \rrbracket_\eta \rightarrow \llbracket \Delta \vdash E' \rrbracket_\eta \} \\ \llbracket \Delta \vdash \emptyset \rrbracket_\eta &= \{ \} \\ \llbracket \Delta \vdash P * P' \rrbracket_\eta &= \\ &\{ h \cdot h' \mid h \# h' \wedge h \in \llbracket \Delta \vdash P \rrbracket_\eta \wedge h' \in \llbracket \Delta \vdash P' \rrbracket_\eta \} \end{aligned}$$

### 4.3 Programming Language

The programming language is Reynolds’s idealized algol [109] adapted for “separation-logic typing.” It is a call-by-name typed lambda calculus, extended with heap operations, dependent functions, and Hoare-triple types. As explained in the introduction, we only consider immutable variables.

The types of the language are defined as follows. We write  $\Delta \vdash \theta : \text{Type}$  for a type  $\theta$  in context  $\Delta$ . The set of types is defined by the following inference rules (in which  $P$  and  $Q$  range over assertions):

$$\begin{array}{c} \frac{\text{fiv}(P) \subseteq \Delta \quad \text{fiv}(Q) \subseteq \Delta}{\Delta \vdash \{P\} - \{Q\} : \text{Type}} \quad \frac{\Delta \vdash \theta : \text{Type} \quad \text{fiv}(P) \subseteq \Delta}{\Delta \vdash \theta \otimes P : \text{Type}} \\ \\ \frac{\Delta \cup \{i\} \vdash \theta : \text{Type} \quad i \notin \Delta}{\Delta \vdash \Pi_i \theta : \text{Type}} \quad \frac{\Delta \vdash \theta : \text{Type} \quad \Delta \vdash \theta' : \text{Type}}{\Delta \vdash \theta \rightarrow \theta' : \text{Type}} \end{array}$$

Note that the types are *dependent types*, in that they may depend on variables  $i$  (see the first rule above). One way to understand a type is to read it as a specification for terms, i.e., through the Curry-Howard correspondence. A Hoare-triple type  $\{P\} - \{Q\}$  is a direct import from separation logic; it denotes a set of commands  $c$  that satisfy the Hoare triple  $\{P\}c\{Q\}$ . An invariant extension  $\theta \otimes P$  is satisfied by a term  $M$  if and only if for one part of the heap, the behavior of  $M$  satisfies  $\theta$  and for the other part of the heap,  $M$  maintains the invariant  $P$ . For instance,  $\{P\} - \{Q\} \otimes P_0$  intuitively consists of commands that given an input state satisfying  $P * P_0$ , so that the input state may be split into a  $P$ -part and a  $P_0$ -part, change the  $P$ -part so that the result satisfies  $Q$ ; and for the  $P_0$ -part, the commands modify it freely, but maintain the invariant  $P_0$ .

The type  $\Pi_i \theta$  is a dependent product type, as in standard dependent type theory (under Curry-Howard it corresponds to the specification given by universally quantifying  $i$  in the specification corresponding to  $\theta$ ). Intuitively,  $\Pi_i \theta$  denotes functions from integers such that given an integer  $n$ , they return a value satisfying  $\theta[n/i]$ . For example, the type  $\Pi_i \{j \mapsto -\} - \{j \mapsto i!\}$  specifies a factorial function that computes the factorial of  $i$  and stores the result in the heap cell  $j$ .

The pre-terms of the language are given by the following grammar:

$$\begin{aligned} M ::= & x \mid \lambda x : \theta. M \mid MM \mid \lambda i. M \mid ME \\ & \mid \text{fix } M \mid \text{ifz } E M M \mid M; M \mid \text{let } i = \text{new in } M \\ & \mid \text{free}(E) \mid [E] := E \mid \text{let } i = [E] \text{ in } M, \end{aligned}$$

where  $E$  is an integer expression defined in Section 4.2. The language has the usual constructs for a higher-order imperative language with heap operations, but it has two distinct features. First, it treats the integer expressions as “second class”: the terms  $M$

never have the integer type, and all integer expressions inside a term are from the separate grammar for  $E$  defined in Section 4.2. Second, no “integer variables”  $i$  can be modified in this language; only heap cells can be modified. Note that the language has two forms of abstraction and application, one for general terms and the other for integer expressions. A consequence of this stratification is that all integer expressions terminate, because the grammar for  $E$  does not contain the recursion operator.

The language has four heap operations. Command  $\text{let } i = \text{new in } M$  allocates a heap cell, binds  $i$  to the address of the allocated cell, and executes the command  $M$ .<sup>2</sup> An allocated cell  $i$  can be disposed by  $\text{free}(i)$ . The remaining two commands access the content of a cell. The command  $[i] := E'$  changes the content of cell  $i$  by  $E'$ ; and  $\text{let } j = [i]$  in  $M$  reads the content of cell  $i$ , binds  $j$  to the read value, and executes  $M$ . Note that the allocation and lookup commands involve the “continuation”, and make the bound variable available in the continuation; such indirect-style commands are needed because all variables are immutable.

The typing rules of the language decide a judgment of the form  $\Gamma \vdash_{\Delta} M : \theta$ , where  $\Gamma$  is a list of type assignments to identifiers  $\Gamma = x_1 : \theta_1, \dots, x_n : \theta_n$ , and where the set  $\Delta$  contains all the free variables appearing in  $\Gamma, M, \theta$ .

The type system is shown in Figure 4.1. For notational simplicity we have omitted some obvious side-conditions of the form  $\Delta \vdash \theta : \text{Type}$  which ensure that, for a judgment  $\Gamma \vdash_{\Delta} M : \theta$ , the set  $\Delta$  always contains all the free variables appearing in  $\Gamma, M, \theta$ , and that the type assignment  $\Gamma$  is always well-formed. There are three classes of rules. The first class consists of the rules from the simply typed lambda calculus extended with dependent product types and recursion. The second class consists of the rules for the imperative constructs, all of which come from separation logic. The last class consists of the subsumption rule based on the subtyping relation  $\preceq_{\Delta}$ , which is the most interesting part of our type system. The proof rules for  $\preceq_{\Delta}$  define a preorder between types with free variables in  $\Delta$ , and include all the usual structural subtyping rules [97, chp. 15]. The rules specific to our system are: the encoding of Consequence in Hoare logic; the generalized frame rule that adds an invariant to all types; and the distribution rules for an added invariant assertion.

The generalized frame rule,  $\theta \preceq_{\Delta} \theta \otimes P_0$ , means that if a program satisfies  $\theta$  and an assertion  $P_0$  does not “mention” any cells described by  $\theta$ , then the program preserves  $P_0$ . Note that this rule indicates that the types in our system are *tight* [59, 115]: if a program satisfies  $\theta$ , it can only access heap cells “mentioned” in  $\theta$ . This is why an assertion  $P_0$  for “unmentioned” cells is preserved by the program. For instance, if a program  $M$  has a type of the form

$$\theta_1 \rightarrow \dots \rightarrow \theta_n \rightarrow \{P\} - \{Q\},$$

the tightness of the type says that all the cells that  $M$  can directly access must appear in the precondition  $P$ . Thus, if no cells in an assertion  $P_0$  appear in  $P$ , program  $M$  maintains  $P_0$ , as long as argument procedures maintain it. Such a fact can, indeed, be inferred by

---

<sup>2</sup>We consider single-cell allocation only in order to simplify the presentation; it is straightforward to adapt our results to a language with allocation of  $n$  consecutive cells.

$$\begin{array}{c}
\frac{}{\Gamma, x: \theta \vdash_{\Delta} x: \theta} \text{ (fiv}(\Gamma, \theta) \subseteq \Delta) \quad \frac{\Gamma, x: \theta \vdash_{\Delta} M: \theta'}{\Gamma \vdash_{\Delta} \lambda x: \theta. M: \theta \rightarrow \theta'} \\
\frac{\Gamma \vdash_{\Delta} M: \theta \rightarrow \theta \quad \Gamma \vdash_{\Delta} M': \theta'}{\Gamma \vdash_{\Delta} MM': \theta} \\
\frac{\Gamma \vdash_{\Delta \cup \{i\}} M: \theta'}{\Gamma \vdash_{\Delta} \lambda i. M: \Pi_i \theta'} \text{ (fiv}(\Gamma) \subseteq \Delta) \quad \frac{\Gamma \vdash_{\Delta} M: \Pi_i \theta}{\Gamma \vdash_{\Delta} ME: \theta[E/i]} \text{ (fiv}(E) \subseteq \Delta) \\
\frac{\Gamma \vdash_{\Delta} M: \theta \rightarrow \theta}{\Gamma \vdash_{\Delta} \text{fix } M: \theta} \\
\frac{\Gamma \vdash_{\Delta} M: \{P \wedge E=0\} - \{Q\} \quad \Gamma \vdash_{\Delta} M': \{P \wedge E \neq 0\} - \{Q\}}{\Gamma \vdash_{\Delta} \text{ifz } E M M': \{P\} - \{Q\}} \\
\frac{\Gamma \vdash_{\Delta} M: \{P\} - \{P'\} \quad \Gamma \vdash_{\Delta} M': \{P'\} - \{Q\}}{\Gamma \vdash_{\Delta} (M; M'): \{P\} - \{Q\}} \\
\frac{\Gamma \vdash_{\Delta \cup \{i\}} M: \{P * i \mapsto -\} - \{Q\}}{\Gamma \vdash_{\Delta} \text{let } i = \text{new in } M: \{P\} - \{Q\}} \text{ (} i \notin \text{fiv}(\Gamma, P, Q)) \\
\frac{\Gamma \vdash_{\Delta \cup \{i\}} M: \{P * E \mapsto i\} - \{Q\}}{\Gamma \vdash_{\Delta} \text{let } i = [E] \text{ in } M: \{P * E \mapsto -\} - \{Q\}} \text{ (} i \notin \text{fiv}(\Gamma, E, P, Q)) \\
\frac{}{\Gamma \vdash_{\Delta} \text{free}(E): \{E \mapsto -\} - \{\emptyset\}} \text{ (fiv}(\Gamma, E) \subseteq \Delta) \\
\frac{}{\Gamma \vdash_{\Delta} [E] := E': \{E \mapsto -\} - \{E \mapsto E'\}} \text{ (fiv}(\Gamma, E, E') \subseteq \Delta) \\
\frac{\Gamma \vdash_{\Delta} M: \theta \quad \theta \preceq_{\Delta} \theta'}{\Gamma \vdash_{\Delta} M: \theta'}
\end{array}$$

where  $\preceq_{\Delta}$  is the usual structural subtyping relation [97, chp. 15] for types over  $\Delta$ , extended with the following rules:

$$\begin{array}{l}
\{P'\} - \{Q'\} \preceq_{\Delta} \{P\} - \{Q\} \text{ (when for all } \eta \in [\Delta], [P]_{\eta} \subseteq [P']_{\eta} \text{ and } [Q']_{\eta} \subseteq [Q]_{\eta}) \\
\theta \preceq_{\Delta} \theta \otimes P \quad (\{P\} - \{Q\}) \otimes P_0 \preceq_{\Delta} \{P * P_0\} - \{Q * P_0\} \\
(\Pi_i \theta) \otimes P \preceq_{\Delta} \Pi_i(\theta \otimes P) \quad (\theta \otimes Q) \otimes P \preceq_{\Delta} \theta \otimes (Q * P) \\
(\theta \rightarrow \theta') \otimes P \preceq_{\Delta} (\theta \otimes P \rightarrow \theta' \otimes P)
\end{array}$$

Figure 4.1: Typing Rules

the generalized frame rule together with the distribution rules:

$$\begin{aligned}
& \theta_1 \rightarrow \dots \rightarrow \theta_n \rightarrow \{P\} - \{Q\} \\
\preceq_{\Delta} & \quad (\because \theta \preceq_{\Delta} \theta \otimes P_0) \\
& (\theta_1 \rightarrow \dots \rightarrow \theta_n \rightarrow \{P\} - \{Q\}) \otimes P_0 \\
\preceq_{\Delta} & \quad (\because (\theta \rightarrow \theta') \otimes P_0 \preceq_{\Delta} (\theta \otimes P_0 \rightarrow \theta' \otimes P_0)) \\
& (\theta_1 \otimes P_0 \rightarrow \dots \rightarrow \theta_n \otimes P_0 \rightarrow \{P\} - \{Q\} \otimes P_0) \\
\preceq_{\Delta} & \quad (\because \{P\} - \{Q\} \otimes P_0 \preceq_{\Delta} \{P * P_0\} - \{Q * P_0\}) \\
& (\theta_1 \otimes P_0 \rightarrow \dots \rightarrow \theta_n \otimes P_0 \rightarrow \{P * P_0\} - \{Q * P_0\}).
\end{aligned}$$

The generalized frame rule, the distribution rules, and the structural subtyping rule for function types all together give many interesting higher-order frame rules, including the second-order frame rule. The common mechanism for obtaining such a rule is: first, add an invariant assertion by the generalized frame rule, and then, propagate the added assertion all the way down to a base triple type by the distribution rules. The structural subtyping rule for the function type allows us to apply this construction for a sub type-expression in an appropriate covariant or contravariant way. For instance, we can derive a third-order frame rule as follows:

$$\begin{aligned}
& (\{P_1\} - \{Q_1\} \rightarrow \{P_2\} - \{Q_2\}) \rightarrow \{P_3\} - \{Q_3\} \\
\preceq_{\Delta} & \quad (\because \theta \preceq_{\Delta} \theta \otimes P) \\
& \left( (\{P_1\} - \{Q_1\} \rightarrow \{P_2\} - \{Q_2\}) \rightarrow \{P_3\} - \{Q_3\} \right) \otimes P \\
\preceq_{\Delta} & \quad (\because (\theta \rightarrow \theta') \otimes P \preceq_{\Delta} (\theta \otimes P \rightarrow \theta' \otimes P)) \\
& (\{P_1\} - \{Q_1\} \otimes P \rightarrow \{P_2\} - \{Q_2\} \otimes P) \rightarrow \{P_3\} - \{Q_3\} \otimes P \\
\preceq_{\Delta} & \quad (\because \text{structural subtyping}) \\
& (\{P_1\} - \{Q_1\} \otimes P \rightarrow \{P_2\} - \{Q_2\}) \rightarrow \{P_3\} - \{Q_3\} \otimes P \\
\preceq_{\Delta} & \quad (\because \{P_0\} - \{Q_0\} \otimes P \simeq_{\Delta} \{P_0 * P\} - \{Q_0 * P\}) \\
& (\{P_1 * P\} - \{Q_1 * P\} \rightarrow \{P_2\} - \{Q_2\}) \rightarrow \{P_3 * P\} - \{Q_3 * P\}.
\end{aligned}$$

## 4.4 Semantics

In this section we present our main contribution, the semantics that formalizes the underlying intuitions of the separation-logic type system. In particular, we formalize the following three intuitive properties of the type system:

1. The types in the separation-logic type system refine the conventional types. A separation-logic type specifies a stronger property of a term, and restricts clients of such terms by asking them to only depend upon what can be known from the type. For instance, the type  $\{1 \mapsto 3\} - \{1 \mapsto 0\}$  of a term  $M$  indicates not just that  $M$  is a command, but also that  $M$  stores 0 to cell 1 if cell 1 contains 3 initially. Moreover, this type forces clients to run  $M$  only when cell 1 contains 3.
2. The higher-order frame rules in the type system imply that all programs behave locally.
3. The type system, however, does not change the computational behavior of each program.

We formalize the first intuitive property by means of partial equivalence relations. Roughly, each type  $\theta$  in our semantics determines a partial equivalence relation (in short, per) over

the meaning of the “underlying type”  $\bar{\theta}$ . The domain of a per over a set  $A$  is a subset of  $A$ ; this indicates that  $\theta$  indeed specifies a stronger property than  $\bar{\theta}$ . The other part of a per, namely the equivalence relation part, explains that the type system restricts the clients, so that no type-checked clients can tell apart two equivalent programs. For instance,  $\{1 \mapsto 3\} - \{1 \mapsto 0\}$  determines a per over the set of all commands. The domain of this per consists of commands satisfying  $\{1 \mapsto 3\} - \{1 \mapsto 0\}$ , and the per equates two such commands if they behave identically when cell 1 contains 3 initially. The equivalence relation implies that type-checked clients run a command of  $\{1 \mapsto 3\} - \{1 \mapsto 0\}$  only when cell 1 contains 3.

We justify the other two intuitive properties by proving technical lemmas about our semantics. For number 2, we prove the soundness of all the subtyping rules, including the generalized frame rule and the distribution rules. For number 3, we prove that our semantics has been obtained by extracting and then quotienting semantic elements in the conventional semantics; yet, this extraction and quotienting does not reduce the computational information of semantic elements.

In this section, we first define categories  $\mathcal{C}$  and  $\mathcal{D}$ , corresponding to the extraction and quotienting, respectively. Next we give the interpretation of types and terms. Finally, we connect our semantics with the conventional semantics, and prove that our semantics is indeed obtained by extracting and quotienting from the conventional semantics.

#### 4.4.1 Categories $\mathcal{C}$ and $\mathcal{D}$

We construct  $\mathcal{C}$  and  $\mathcal{D}$  by modifying the category CPO of pointed cpos and continuous functions. For  $\mathcal{C}$ , we impose a parameterized per on each cpo, and extract only those morphisms in CPO that preserve such pers (at all instantiations). Each per formalizes that each type  $\theta$  corresponds to a specification over the underlying type  $\bar{\theta}$ , and the preservation of the pers ensures that all the morphisms in  $\mathcal{C}$  satisfy the corresponding specifications. The parameterization of each per guarantees that all morphisms in  $\mathcal{C}$  behave locally (in the sense of higher-order frame rules). The other category  $\mathcal{D}$  is a quotient of  $\mathcal{C}$ . Intuitively, the quotienting of  $\mathcal{C}$  reflects that our type system also restricts the clients of a term; thus, more terms cannot be distinguished observationally.

We define the “extracting” category  $\mathcal{C}$  first. Let  $Pred$  be the set of *predicates*, i.e., subsets of *Heap*. We recall the semantic version of separating connectives,  $emp$  and  $*$ , on  $Pred$ . For  $p, q \in Pred$ ,

$$\begin{aligned} h \in emp &\iff h = \lambda n.undef \\ h \in p * q &\iff \exists h_1 h_2. h_1 * h_2 = h \wedge h_1 \in p \wedge h_2 \in q. \end{aligned}$$

The category  $\mathcal{C}$  is defined as follows:

- objects:  $(A, R)$  where  $A$  is a pointed cpo, and  $R$  is a family of admissible pers<sup>3</sup> indexed by predicates such that

$$\forall p, q \in Pred. R(p) \subseteq R(p * q);$$

- morphisms:  $f: (A, R) \rightarrow (B, S)$  is a continuous function from  $A$  to  $B$  such that

$$\forall p. f[R(p) \rightarrow S(p)]f,$$

i.e.,  $f$  maps  $R(p)$  related elements to  $S(p)$  related elements.

---

<sup>3</sup>A per  $R_0$  on  $A$  is admissible iff  $(\perp, \perp) \in R_0$  and  $R_0$  is a sub-cpo of  $A \times A$ .

Intuitively, each object  $(A, R)$  denotes a specification parameterized by invariant extension. The first component  $A$  denotes the underlying set from which we select “correct” elements.  $R(\text{emp})$  denotes the initial specification of this object where no invariant is added by the frame rule. The domain  $|R(\text{emp})|$  of per  $R(\text{emp})$  indicates which elements satisfy the specification, and the equivalence relation on  $|R(\text{emp})|$  expresses how the specification is also used to limit the interaction of a client: the client can only do what the specification guarantees, so more elements become equivalent observationally. The per  $R(p)$  at another predicate  $p$  denotes an extended specification by the invariant  $p$ .

We illustrate the intuition of  $\mathcal{C}$  with a “Hoare-triple” object  $[p, q]$  for some  $p, q \in \text{Pred}$ . Let  $\text{comm}$  be the set of all functions  $c$  from  $\text{Heap}$  to  $\mathcal{P}(\text{Heap} \cup \{\text{wrong}\})$  that satisfy safety monotonicity and the frame property:

- *Safety Monotonicity*: for all  $h, h_0 \in \text{State}$ , if  $h \# h_0$  and  $\text{wrong} \notin c(h)$ , then  $\text{wrong} \notin c(h * h_0)$ ;
- *Frame Property*: for all  $h, h_0, h'_1 \in \text{State}$ , if  $h \# h_0$ ,  $\text{wrong} \notin c(h)$ , and  $h'_1 \in c(h * h_0)$ , then there exists  $h'$  such that  $h'_1 = h_0 * h'$  and  $h' \in c(h)$ .

Intuitively,  $\text{comm}$  contains all commands that satisfy the (first-order) frame rule [85], and it forms the underlying set for all Hoare-triple specifications. Indeed, it is the first component of the Hoare-triple object  $[p, q]$ , where the order on  $\text{comm}$  is given by:

$$c \sqsubseteq c' \iff \forall h. c(h) \subseteq c'(h).$$

The real meaning of  $[p, q]$  is given by the second component  $R$ . For each predicate  $p_0$ , the domain of  $R(p_0)$  consists of all “commands” in  $\text{comm}$  that satisfy  $\{p * p_0\} - \{q * p_0\}$ :

$$c \in |R(p_0)| \iff \forall h \in p * p_0. c(h) \subseteq q * p_0.$$

The equivalence relation  $R(p_0)$  relates  $c$  and  $c'$  in  $|R(p_0)|$  iff  $c$  and  $c'$  behave the same for the inputs in  $p * p_0 * \text{true}$ :

$$\begin{aligned} \text{true} &= \{h \mid h \in \text{Heap}\} \\ c[R(p_0)]c' &\iff \forall h \in p * p_0 * \text{true}. c(h) = c'(h). \end{aligned}$$

Intuitively, this equivalence relation means that the type system allows a client to execute  $c$  or  $c'$  in  $h$  only when  $h$  satisfies  $p * p_0 * p'$  for some  $p'$ ; the predicate  $p'$  reflects that the frame rule in the type system allows the “widening” of a specification by an invariant.

The category  $\mathcal{C}$  is cartesian closed, and it has all small products. The terminal object is  $(\{\perp\}, R)$  where  $R(p)$  is  $\{(\perp, \perp)\}$  for all  $p$ , and all the small products are given pointwise; for instance,  $(A, R) \times (B, S)$  is  $(A \times B, \{R(p) \times S(p)\}_p)$ . The exponential of  $(A, R)$  and  $(B, S)$  is subtle, and its per component involves the quantification over all predicates. Let  $R \Rightarrow S$  be the following family of pers on the continuous function space  $A \Rightarrow B$ :

$$f[(R \Rightarrow S)(p)]g \iff \forall q \in \text{Pred}. f[R(p * q) \rightarrow S(p * q)]g.$$

Here  $f$  and  $g$  range over the domain of  $(R \Rightarrow S)(p)$ , which contains only local functions: it consists of functions  $f$  in  $|R(p) \rightarrow S(p)|$  such that

$$\forall q. f[R(p * q) \rightarrow S(p * q)]f.$$

Note that this condition is precisely the semantic version of the frame rule. The exponential  $(A, R) \Rightarrow (B, S)$  is given by  $(A \Rightarrow B, R \Rightarrow S)$ .

**Lemma 4.1.**  $\mathcal{C}$  is cartesian closed, and has all small limits.

Another important feature of  $\mathcal{C}$  is that it validates higher-order frame rules. Let  $\mathcal{P}_r$  be the preorder  $(\text{Pred}, \sqsubseteq)$  with  $\sqsubseteq$  defined by predicate extension:

$$p \sqsubseteq r \iff \exists q. p * q = r.$$

Category  $\mathcal{C}$  has an “invariant-extension” functor  $\text{inv}$  from  $\mathcal{C} \times \mathcal{P}_r$  to  $\mathcal{C}$  defined by:

$$\text{inv}((A, R), p) = (A, R(p * -)) \text{ and } \text{inv}(f, p \sqsubseteq q) = f.$$

Functor  $\text{inv}$  corresponds to the type constructor  $\otimes$  in our language; given a “type”  $(A, R)$  and a predicate  $p$ ,  $\text{inv}$  extends  $(A, R)$  by adding an invariant  $p$ . For instance, when a triple object  $[p', q']$  is extended with  $p$ , it becomes  $[p' * p, q' * p]$ .

Functor  $\text{inv}$  validates the subtyping rules that express higher-order frame rules: the generalized frame rule  $\theta \preceq \theta \otimes P$  and the rules for distributing  $\otimes$  over each type constructor. We first show that the functoriality of  $\text{inv}$  gives the soundness of the generalized frame rule. Note that for all predicates  $p$ ,  $\text{emp} \sqsubseteq p$ , and that  $\text{inv}(-, \text{emp})$  is the identity functor on  $\mathcal{C}$ . Thus, for each  $(A, R)$ , the functoriality of  $\text{inv}$  gives a morphism from  $(A, R)$  to  $\text{inv}((A, R), p)$ . This morphism gives the soundness of the subtyping rule  $\theta \preceq \theta \otimes P$ .

The soundness of the other distribution rules follows from the fact that for all  $p$ ,  $\text{inv}(-, p)$  preserves most of the structure of  $\mathcal{C}$ . For instance,  $\text{inv}(-, p)$  preserves the exponential of  $\mathcal{C}$ , because for all objects  $(A, R)$  and  $(B, S)$  and all predicates  $q$ , we have that

$$\begin{aligned} & f[(R(p * -) \Rightarrow S(p * -))(q)]g \\ \iff & \forall q'. f[R(p * (q * q')) \rightarrow S(p * (q * q'))]g \\ \iff & \forall q'. f[R((p * q) * q') \rightarrow S((p * q) * q')]g \\ \iff & f[(R \Rightarrow S)(p * q)]g. \end{aligned}$$

**Lemma 4.2.** For each predicate  $p$ ,  $\text{inv}(-, p)$  preserves the cartesian closed structure and all the small products of  $\mathcal{C}$  on the nose.

**Lemma 4.3.** For all predicates  $p$  and  $q$ ,  $\text{inv}(-, p) \circ \text{inv}(-, q) = \text{inv}(-, p * q)$ .

For now, the final remark on  $\mathcal{C}$  is that the triple-object generator  $[-, -]$  can be made into a functor, whose morphism action validates the subtyping rule for Consequence. Let  $\mathcal{P}$  be the set of predicates ordered by the subset inclusion  $\subseteq$ . Generator  $[-, -]$  can be extended to a functor  $\text{tri}$  from  $\mathcal{P}^{\text{op}} \times \mathcal{P}$  to  $\mathcal{C}$ :

$$\text{tri}(p, q) = [p, q] \text{ and } \text{tri}(p \subseteq p', q' \subseteq q)(c) = c.$$

Note that  $\text{tri}$  is contravariant in the first argument and covariant on the second argument. This mixed variance reflects that the pre-condition of a triple can be strengthened, and the post-condition can be weakened; thus, it validates the subtyping rule for Consequence. We also note that the subtyping rule that moves an invariant assertion into the pre- and post-conditions is sound.

**Lemma 4.4.** For each predicate  $p$ , let  $\multimap p: \mathcal{P} \rightarrow \mathcal{P}$  be a functor that maps a predicate  $q$  to  $q * p$ . Then,

$$\text{inv}(-, p) \circ \text{tri} = \text{tri}(\multimap p, \multimap p).$$

The category  $\mathcal{D}$  is obtained from  $\mathcal{C}$  by equating morphisms according to an equivalence relation  $\sim$ . Morphisms  $f$  and  $g$  in  $\mathcal{C}[(A, R), (B, S)]$  are related by  $\sim$  iff

$$\forall p \in \text{Pred}. f[R(p) \rightarrow S(p)]g.$$

Relation  $\sim$  is an equivalence relation; it is reflexive, because each morphism in  $\mathcal{C}[(A, R), (B, S)]$  should map  $R(p)$ -related elements to  $S(p)$ -related elements, for all  $p$ ; and it is symmetric and transitive because, for all  $p$ ,  $R(p)$  and  $S(p)$  are symmetric and transitive. The interesting property of  $\sim$  is that it is preserved by all the structure of  $\mathcal{C}$ :

**Lemma 4.5 (Preservation).** *The relation  $\sim$  is preserved by the following operators in  $\mathcal{C}$ :*

- the composition of morphisms;
- the currying of morphisms;
- the pairing into all the small products; and
- the functor  $\text{inv}(-, p \sqsubseteq q)$  on  $\mathcal{C}$ , for all predicates  $p, q$  such that  $p \sqsubseteq q$ .

This lemma ensures that taking a quotient of morphisms in  $\mathcal{C}$  gives a well-defined category, which we call  $\mathcal{D}$ . Category  $\mathcal{D}$  inherits all the interesting structure of  $\mathcal{C}$  by Lemma 4.5; it is cartesian closed, has all small products, and has a functor  $\text{inv}' : \mathcal{D} \times \mathcal{P}_r \rightarrow \mathcal{D}$  that preserves the CCC structure and the small products of  $\mathcal{D}$ . Let  $E$  be the “quotienting” functor from  $\mathcal{C}$  to  $\mathcal{D}$ , and  $\text{tri}' : \mathcal{P}^{\text{op}} \times \mathcal{P} \rightarrow \mathcal{D}$  the composition of  $E$  with  $\text{tri}$ . We summarize the main property of  $\mathcal{D}$  in the following two lemmas:

**Lemma 4.6.** *The category  $\mathcal{D}$  is a CCC with all small products, and has two functors  $\text{inv}' : \mathcal{D} \times \mathcal{P}_r \rightarrow \mathcal{D}$  and  $\text{tri}' : \mathcal{P}^{\text{op}} \times \mathcal{P} \rightarrow \mathcal{D}$  such that*

1.  $\text{inv}'(-, p)$  preserves all the CCC structure and the small products of  $\mathcal{D}$ ;
2.  $\text{inv}'(-, p) \circ \text{inv}'(-, q) = \text{inv}'(-, p * q)$ ; and
3.  $\text{inv}'(-, p) \circ \text{tri}' = \text{tri}'(- * p, - * p)$ .

**Lemma 4.7.** *The functor  $E$  from  $\mathcal{C}$  to  $\mathcal{D}$  is full, preserves the CCC structure as well as small products, and makes the following diagrams commute:*

$$\begin{array}{ccc} \mathcal{C} \times \mathcal{P}_r & \xrightarrow{\text{inv}} & \mathcal{C} \\ E \times \text{Id} \downarrow & & \downarrow E \\ \mathcal{D} \times \mathcal{P}_r & \xrightarrow{\text{inv}'} & \mathcal{D} \end{array} \qquad \begin{array}{ccc} \mathcal{P}^{\text{op}} \times \mathcal{P} & \xrightarrow{\text{tri}} & \mathcal{C} \\ \text{Id} \downarrow & & \downarrow E \\ \mathcal{P}^{\text{op}} \times \mathcal{P} & \xrightarrow{\text{tri}'} & \mathcal{D} \end{array}$$

#### 4.4.2 Interpretation of the Language

We interpret the language in the family fibration  $\text{Fam}(\mathcal{D}) \rightarrow \mathbf{Set}$ . Each base set in the fibration models all the possible environments for a fixed shape of the stack (i.e., a fixed set of integer variables  $\Delta$ ). For instance, the object  $\{(A, R)_\eta\}_{\eta \in \llbracket \Delta \rrbracket}$  assumes that all the available integer variables are in  $\Delta$ , and it specifies a type dependent on the values of such variables, given by  $\eta$ . The types and terms of our language are interpreted using the categorical structure of this fibration.

The interpretation is explicit about the set of variables under which we consider types, type assignments, and terms. Write  $\Delta \vdash \Gamma$  to mean that  $\Delta \vdash \Gamma(x) : \text{Type}$ , for all  $x$  in the domain of  $\Gamma$ .

The semantics of  $\Delta \vdash \theta (: \text{Type})$  and  $\Delta \vdash \Gamma$  is given by a family of objects in  $\mathcal{D}$  indexed by the environments in  $\llbracket \Delta \rrbracket$ . The precise definition of  $\llbracket \theta \rrbracket$  and  $\llbracket \Gamma \rrbracket$  is given as follows: for  $\eta$  in  $\llbracket \Delta \rrbracket$ ,

$$\begin{aligned} \llbracket \Delta \vdash \{P\} - \{Q\} \rrbracket_\eta &= \text{tri}'(\llbracket \Delta \vdash P \rrbracket_\eta, \llbracket \Delta \vdash Q \rrbracket_\eta) \\ \llbracket \Delta \vdash \theta \otimes P \rrbracket_\eta &= \text{inv}'(\llbracket \Delta \vdash \theta \rrbracket_\eta, \llbracket \Delta \vdash P \rrbracket_\eta) \\ \llbracket \Delta \vdash \theta \rightarrow \theta' \rrbracket_\eta &= \llbracket \Delta \vdash \theta \rrbracket_\eta \Rightarrow \llbracket \Delta \vdash \theta' \rrbracket_\eta \\ \llbracket \Delta \vdash \Pi_i \theta \rrbracket_\eta &= \prod_{n \in \text{Int}} \llbracket \Delta \cup \{i\} \vdash \theta \rrbracket_{\eta[i \rightarrow n]} \\ \llbracket \Delta \vdash \Gamma \rrbracket_\eta &= \prod_{x \in \text{dom}(\Gamma)} \llbracket \Delta \vdash \Gamma(x) \rrbracket_\eta \end{aligned}$$

Note that  $\text{tri}'$  is used to interpret the triple type  $\{P\} - \{Q\}$ , and  $\text{inv}'$  to interpret the invariant extension  $\theta \otimes P$ .

We interpret each subtype relation  $\theta \preceq_\Delta \theta'$  as a family of morphisms in  $\mathcal{D}$  of the following shape:

$$\{f_\eta : \llbracket \Delta \vdash \theta \rrbracket_\eta \rightarrow \llbracket \Delta \vdash \theta' \rrbracket_\eta\}_{\eta \in \llbracket \Delta \rrbracket}.$$

The semantics is given by induction on the derivation. The subtyping rule for Consequence is interpreted using the morphism action of functor  $\text{tri}'$ :

$$\begin{aligned} \llbracket \{P'\} - \{Q'\} \preceq_\Delta \{P\} - \{Q\} \rrbracket_\eta \\ = \text{tri}'(\llbracket P \rrbracket_\eta \subseteq \llbracket P' \rrbracket_\eta, \llbracket Q' \rrbracket_\eta \subseteq \llbracket Q \rrbracket_\eta) \end{aligned}$$

The other important subtyping rules are the ones for the higher-order frame rules. The interpretation of these rules uses the property of  $\text{inv}'$ . The generalized frame rule is interpreted by the morphism action of  $\text{inv}'$ :

$$\llbracket \theta \preceq_\Delta \theta \otimes P \rrbracket_\eta = \text{inv}'(\llbracket \theta \rrbracket_\eta, \text{emp} \subseteq \llbracket P \rrbracket_\eta)$$

and the rules for distributing an invariant is interpreted by the identity; this interpretation “typechecks,” because  $\text{inv}'(-, p)$  preserves the exponentials and the small products on the nose, and because of (items 2 and 3 of) Lemma 4.6.

Finally, we define the semantics of each typing judgment  $\Gamma \vdash_\Delta M : \theta$  by an indexed family of morphisms in  $\mathcal{D}$  of the form:

$$\{f_\eta : \llbracket \Delta \vdash \Gamma \rrbracket_\eta \rightarrow \llbracket \Delta \vdash \theta \rrbracket_\eta\}_{\eta \in \llbracket \Delta \rrbracket}.$$

The semantics is given by induction on the derivation of the judgment, and it is shown in Figure 4.2. The interpretation of terms is given using the categorical structure of  $\mathcal{D}$  in a standard way. The only specific parts are the interpretation of basic imperative operations, where we use five basic semantic constants

*seq, new, read, free, and write,*

which are also defined in the figure.

For this interpretation of terms, the question of well-definedness arises, because of the introduction and elimination of dependent function type  $\Pi_i \theta$ . The semantic definition of  $\lambda i.M$  assumes that if  $\Gamma$  does not contain the variable  $i$ , it is interpreted as the same object in  $\mathcal{D}$  no matter how we change or even drop the value of  $i$  in the index. The definition of  $\llbracket ME \rrbracket$  assumes that the reindexing precisely models the substitution. The following lemmas show that these two assumptions indeed hold.

$$\begin{aligned}
\llbracket \Gamma, x : \theta \vdash_\Delta x : \theta \rrbracket_\eta &= \pi_x \\
\llbracket \Gamma \vdash_\Delta \lambda x : \theta. M : \theta \rightarrow \theta' \rrbracket_\eta &= \text{curry}(\llbracket \Gamma, x : \theta \vdash_\Delta M : \theta' \rrbracket_\eta \circ \text{iso}(\llbracket \Gamma \rrbracket_\eta \times \llbracket \theta \rrbracket_\eta, \llbracket \Gamma, x : \theta \rrbracket_\eta)) \\
\llbracket \Gamma \vdash_\Delta M M' : \theta \rrbracket_\eta &= \text{eval} \circ \langle \llbracket \Gamma \vdash_\Delta M : \theta' \rightarrow \theta \rrbracket_\eta, \llbracket \Gamma \vdash_\Delta M' : \theta' \rrbracket_\eta \rangle \\
\llbracket \Gamma \vdash_\Delta \lambda i. M : \Pi_i \theta \rrbracket_\eta &= \langle \llbracket \Gamma \vdash_{\Delta \cup \{i\}} M : \theta \rrbracket_{\eta[i \mapsto n]} \rangle_{n \in \text{Int}} \\
\llbracket \Gamma \vdash_\Delta M E : \theta[E/i] \rrbracket_\eta &= \pi_{[E]_\eta} \circ \llbracket \Gamma \vdash_\Delta M : \Pi_i \theta \rrbracket_\eta \\
\llbracket \Gamma \vdash_\Delta M : \theta' \rrbracket_\eta &= \llbracket \theta \preceq_\Delta \theta' \rrbracket_\eta \circ \llbracket \Gamma \vdash_\Delta M : \theta \rrbracket_\eta \\
\llbracket \Gamma \vdash_\Delta \text{fix } M : \theta \rrbracket_\eta &= \text{lfix} \circ \llbracket \Gamma \vdash_\Delta M : \theta \rightarrow \theta \rrbracket_\eta \\
\llbracket \Gamma \vdash_\Delta \text{ifz } E M M' : \{P\} - \{Q\} \rrbracket_\eta &= \text{if} (\llbracket \Delta \vdash E \rrbracket_\eta = 0) \text{ then } \llbracket \Gamma \vdash_\Delta M : \{P \wedge E=0\} - \{Q\} \rrbracket_\eta \\
&\quad \text{else } \llbracket \Gamma \vdash_\Delta M' : \{P \wedge E \neq 0\} - \{Q\} \rrbracket_\eta \\
\llbracket \Gamma \vdash_\Delta M; M' : \{P\} - \{Q\} \rrbracket_\eta &= \text{seq} \circ \langle \llbracket \Gamma \vdash_\Delta M : \{P\} - \{P'\} \rrbracket_\eta, \llbracket \Gamma \vdash_\Delta M' : \{P'\} - \{Q\} \rrbracket_\eta \rangle \\
\llbracket \Gamma \vdash_\Delta \text{free}(E) : \{E \mapsto -\} - \{\emptyset\} \rrbracket_\eta &= \text{free}(\llbracket \Delta \vdash E \rrbracket_\eta) \circ !_{\llbracket \Delta \vdash \Gamma \rrbracket_\eta} \\
\llbracket \Gamma \vdash_\Delta \text{let } i = \text{new} \text{ in } M : \{P\} - \{Q\} \rrbracket_\eta &= \text{new} \circ \langle \llbracket \Gamma \vdash_{\Delta \cup \{i\}} M : \{P * i \mapsto -\} - \{Q\} \rrbracket_{\eta[i \mapsto n]} \rangle_{n \in \text{Int}} \\
\llbracket \Gamma \vdash_\Delta \text{let } i = [E] \text{ in } M : \{P * E \mapsto -\} - \{Q\} \rrbracket_\eta &= \text{read}(\llbracket \Delta \vdash E \rrbracket_\eta) \circ \langle \llbracket \Gamma \vdash_{\Delta \cup \{i\}} M : \{P * E \mapsto i\} - \{Q\} \rrbracket_{\eta[i \mapsto n]} \rangle_{n \in \text{Int}} \\
\llbracket \Gamma \vdash_\Delta [E] := E' : \{E \mapsto -\} - \{E \mapsto E'\} \rrbracket_\eta &= \text{write}(\llbracket \Delta \vdash E \rrbracket_\eta, \llbracket \Delta \vdash E' \rrbracket_\eta) \circ !_{\llbracket \Delta \vdash \Gamma \rrbracket_\eta}
\end{aligned}$$

where  $\text{seq}$ ,  $\text{new}$ ,  $\text{read}(n)$ ,  $\text{free}(n)$ , and  $\text{write}(n, n')$  are morphisms in  $\mathcal{D}$  (equivalence classes) defined as follows:

$$\begin{aligned}
\text{seq}_{p,q,q'} &: \text{tri}(p, q) \times \text{tri}(q, q') \rightarrow \text{tri}(p, q') \\
\text{seq} &= [\lambda(c, c'). \lambda h. \{wrong \mid wrong \in c(h)\} \cup \cup \{c'(h') \mid h' \in c(h)\}] \\
\text{new}_{p,q} &: (\prod_{n \in \text{Int}} \text{tri}(p * n \mapsto -, q)) \rightarrow \text{tri}(p, q) \\
\text{new} &= [\lambda c. \lambda h. \cup \{c(n)(h[n \mapsto m]) \mid m, n \in \text{Int} \wedge n \notin \text{dom}(h)\}] \\
\text{read}(n)_{p,q} &: (\prod_{m \in \text{Int}} \text{tri}(p * n \mapsto m, q)) \rightarrow \text{tri}(p * n \mapsto -, q) \\
\text{read}(n) &= [\lambda c. \lambda h. \text{if } n \in \text{dom}(h) \text{ then } c(h(n))(h) \text{ else } \{wrong\}] \\
\text{free}(n) &: \mathbf{1} \rightarrow \text{tri}(\llbracket n \mapsto - \rrbracket, \llbracket \text{emp} \rrbracket) \\
\text{free}(n) &= [\lambda x. \lambda h. \text{if } n \in \text{dom}(h) \text{ then } \{h[n \rightarrow \text{undef}]\} \text{ else } \{wrong\}] \\
\text{write}(n, n') &: \mathbf{1} \rightarrow \text{tri}(\llbracket n \mapsto - \rrbracket, \llbracket n \mapsto n' \rrbracket) \\
\text{write}(n, n') &= [\lambda x. \lambda h. \text{if } n \in \text{dom}(h) \text{ then } \{h[n \rightarrow n']\} \text{ else } \{wrong\}]
\end{aligned}$$

Figure 4.2: Interpretation of Terms

**Lemma 4.8.** *If  $\text{fiv}(\Gamma) \subseteq \Delta$ , then*

$$\forall \eta \in \llbracket \Delta \rrbracket. \forall n \in \text{Int}. \llbracket \Delta \vdash \Gamma \rrbracket_\eta = \llbracket \Delta \cup \{i\} \vdash \Gamma \rrbracket_{\eta[i \mapsto n]}.$$

**Lemma 4.9.** *If  $\text{fiv}(\theta) \subseteq \Delta \cup \{i\}$  and  $\text{fiv}(E) \subseteq \Delta$ , then*

$$\forall \eta \in \llbracket \Delta \rrbracket. \llbracket \Delta \vdash \theta[E/i] \rrbracket_\eta = \llbracket \Delta \cup \{i\} \vdash \theta \rrbracket_{\eta[i \mapsto \llbracket E \rrbracket_\eta]}.$$

### 4.4.3 Adequacy

Our semantics of terms needs further justification in two ways. First, the interpretation of a typing judgment needs to be shown coherent. The interpretation is defined over a proof derivation of the judgment, so two different derivations of the same judgment might have different denotations. This is troublesome for us especially, because our goal is to give a semantics of a programming language with a separation-logic type system, instead of a semantics of a proof in separation logic. Second, the connection with the standard semantics needs to be provided. Our semantics uses subsumption which never arises in the standard interpretation. Thus, our interpretation could be substantially different from the standard interpretation. In this section, we provide justification for both of these two issues.

We consider two other interpretations of our language. The first interpretation, called the standard interpretation, ignores all assertions in the types. In the standard interpretation,  $\{P\} - \{Q\}$  means the same thing no matter what  $P$  and  $Q$  are, and for all  $P$ ,  $\theta \otimes P$  and  $\theta$  have identical interpretations. Let  $\text{tri}''$  be the constant functor from  $\mathcal{P}^{\text{op}} \times \mathcal{P}$  to  $\text{CPO}$  such that  $\text{tri}''(p, q) = \text{comm}$ , and let  $\text{inv}''$  be a functor given by the first projection from  $\text{CPO} \times \mathcal{P}_r$  to  $\text{CPO}$ . Then, the standard interpretation is the interpretation of the previous section, where we use  $\text{CPO}$ ,  $\text{tri}''$  and  $\text{inv}''$  instead of  $\mathcal{D}$ ,  $\text{tri}'$  and  $\text{inv}'$ , and we take an equivalence class representative in the definition of constants; for instance, the constant  $\text{read}(n)$  now means

$$\lambda c. \lambda h. \text{if } n \in \text{dom}(h) \text{ then } c(h(n))(h) \text{ else } \{\text{wrong}\}.$$

The second interpretation uses the category  $\mathcal{C}$ . It is obtained by simply replacing  $\text{tri}'$  and  $\text{inv}'$  in the previous section by  $\text{tri}$  and  $\text{inv}$ , and eliminating the embedding  $[-]$  to the equivalence class in the definition of constants.

The three interpretations are very closely related. Note that from the category  $\mathcal{C}$  to  $\text{CPO}$ , there is a forgetful functor  $F$  that maps an object  $(A, R)$  to  $A$ , and a morphism  $f$  to  $f$ . This forgetful functor preserves all the categorical structure of  $\mathcal{C}$  that we use to interpret the types of our language:

**Lemma 4.10.**  *$F$  is a faithful functor that preserves the CCC structure and the small products of  $\mathcal{C}$ , and makes the following diagrams commute.*

$$\begin{array}{ccc} \mathcal{C} \times \mathcal{P}_r & \xrightarrow{\text{inv}} & \mathcal{C} \\ F \times \text{Id} \downarrow & & \downarrow F \\ \text{CPO} \times \mathcal{P}_r & \xrightarrow{\text{inv}''} & \text{CPO} \end{array} \quad \begin{array}{ccc} \mathcal{P}^{\text{op}} \times \mathcal{P} & \xrightarrow{\text{tri}} & \mathcal{C} \\ \text{Id} \downarrow & & \downarrow F \\ \mathcal{P}^{\text{op}} \times \mathcal{P} & \xrightarrow{\text{tri}''} & \text{CPO} \end{array}$$

Note that Lemmas 4.5 and 4.10 imply that the interpretation of the types in  $\mathcal{D}$  and  $\text{CPO}$  factors through the interpretation in  $\mathcal{C}$ . The following lemma shows that the interpretation of the terms has a similar property.

**Lemma 4.11.** *Both the forgetful functor  $F$  from  $\mathcal{C}$  to  $\text{CPO}$  and the quotienting functor  $E$  from  $\mathcal{C}$  to  $\mathcal{D}$  preserve the interpretation of terms. That is, for all typing judgments  $\Gamma \vdash_{\Delta} M : \theta$  and all  $\eta \in \llbracket \Delta \rrbracket$ ,*

$$\begin{aligned} F(\llbracket \Gamma \vdash_{\Delta} M : \theta \rrbracket_{\eta}^{\mathcal{C}}) &= \llbracket \Gamma \vdash_{\Delta} M : \theta \rrbracket_{\eta}^{\text{CPO}} \\ E(\llbracket \Gamma \vdash_{\Delta} M : \theta \rrbracket_{\eta}^{\mathcal{C}}) &= \llbracket \Gamma \vdash_{\Delta} M : \theta \rrbracket_{\eta}^{\mathcal{D}}. \end{aligned}$$

Since  $E$  is full and  $F$  is faithful, intuitively, Lemma 4.11 says that our semantics is obtained by first selecting some elements, and then quotienting those selected elements.

**Corollary 4.12.** *Our semantics is coherent: the semantics of a typing judgment does not depend on derivations.*

*Proof:* Let  $\mathcal{P}_1$  and  $\mathcal{P}_2$  be two derivations of a judgment  $\Gamma \vdash_{\Delta} M : \theta$ . We note that the standard semantics is coherent; only the subsumption rule is not syntax-directed, but in the standard semantics, this rule does not contribute to the interpretation, because all the subtyping rules denote the identity function. Thus, for all environments  $\eta$ , we have

$$\llbracket \mathcal{P}_1 \rrbracket_{\eta}^{\text{CPO}} = \llbracket \mathcal{P}_2 \rrbracket_{\eta}^{\text{CPO}}.$$

Then, by Lemma 4.11 and the faithfulness of  $F$ ,

$$\begin{aligned} \llbracket \mathcal{P}_1 \rrbracket_{\eta}^{\text{CPO}} = \llbracket \mathcal{P}_2 \rrbracket_{\eta}^{\text{CPO}} &\implies F(\llbracket \mathcal{P}_1 \rrbracket_{\eta}^{\mathcal{C}}) = F(\llbracket \mathcal{P}_2 \rrbracket_{\eta}^{\mathcal{C}}) \\ &\implies \llbracket \mathcal{P}_1 \rrbracket_{\eta}^{\mathcal{C}} = \llbracket \mathcal{P}_2 \rrbracket_{\eta}^{\mathcal{C}} \\ &\implies E(\llbracket \mathcal{P}_1 \rrbracket_{\eta}^{\mathcal{C}}) = E(\llbracket \mathcal{P}_2 \rrbracket_{\eta}^{\mathcal{C}}) \\ &\implies \llbracket \mathcal{P}_1 \rrbracket_{\eta}^{\mathcal{D}} = \llbracket \mathcal{P}_2 \rrbracket_{\eta}^{\mathcal{D}} \end{aligned}$$

□

## 4.5 Related Work

The (first order) frame rule was discovered in the early days of separation logic [59], and it was a main reason for the success of that logic. For example, it was vital in the proofs of garbage collection algorithms in [134] and [23]. Recently, the second-order frame rule, which allows reasoning about simple first-order modules, was discovered [88]. This naturally encouraged the question of whether there are more general frame rules that apply to higher types.

Unlike in [88], our soundness result of higher-order frame rules does not require that invariant assertions be *precise*. This preciseness requirement is needed because higher-order frame rules interact badly with the conjunction rule [88]. In our system, such bad interaction does not exist, because the system does not have the conjunction rule.

Other type systems which track state changes have been proposed in the work on typed assembly languages [77, 4, 124]. Their main focus is to obtain sound rules for proving the safety of programs. Thus, they mostly use easy-to-define conventional operational semantics, and prove the soundness of the proof system syntactically (i.e., by subject reduction and progress lemmas), or “logically” [124]: each type is interpreted as a subset of a single universe of “meanings,” and a typing judgment is interpreted as a specification for the behavior of programs, like a Hoare triple in separation logic. Our separation-logic type system is more refined in that it allows the full power of separation logic in the types and, moreover, we also treat higher-order procedures.

The semantics of idealized algol has been studied intensively [89, 109, 86, 106]. Normally, the semantics is parameterized by the shape of the memory. The indexing in the fibration in our semantics follows this tradition, and it models the shape of the stack. However, the other indexing of our semantics, the indexing by invariant predicates over heaps, has not been used in the literature before.

The construction of the category  $\mathcal{D}$  is an instance of the Kripke quotient by Mitchell and Moggi [72]. The families of pers in  $\mathcal{D}$  form a Kripke logical relation on CPO indexed by the preorder category  $\mathcal{P}_r$ ; our condition on each family ensures that the requirement of Kripke monotonicity holds. This Kripke logical relation produces  $\mathcal{D}$  by Mitchell and Moggi’s construction.

The idea of proving coherence by relating two languages comes from Reynolds [114]. Reynolds proved the coherence of the semantics of typed lambda calculus with subtyping, by connecting it with the semantics of untyped lambda calculus. We use the general direction of Reynolds’s proof, but the details of our proof are quite different from Reynolds’s, because we consider very different languages.

## 4.6 Conclusion and Future Directions

We have presented a type system for idealized algol extended with heaps that includes separation-logic specifications as types and, moreover, defined the coherent semantics of idealized algol typed with this system.

One shortcoming of our type system is that the higher-order frame rules in the system allow only static modularity [91]. With the higher-order frame rules alone, we cannot capture all the the information hiding aspect of dynamically allocated data structures as needed for modeling abstract data types. However, it is well-known that abstract data types can be modeled using existential types and we are currently considering to enrich the assertion language with predicate variables, as in the recently introduced higher-order version of separation logic [20], and to extend the types with dependent product and sums over *predicates*.

Another shortcoming of the type system is that it cannot have the disjunction rule in separation logic. The disjunction rule has two judgments in the premise, and requires that both judgments are about the same program. Such a requirement about the same program cannot be expressed in our type system. We plan to overcome this problem by extending the type system with intersection types [111].

Yet another future direction is to define a parametric model. Uday Reddy pointed out that separation-logic types should validate stronger reasoning principles for data abstraction than ordinary types, because they let us control what clients can access more precisely. Formalizing his intuition is the goal of the parametricity semantics. We currently plan to use category  $\mathcal{C}'$  which replaces each *predicate-indexed* family of pers in  $\mathcal{C}$  by a *relation-indexed* family of *saturated relations*: an object in  $\mathcal{C}'$  is a cpo paired with a family  $T$  of *binary* relations such that (1)  $T$  is indexed by a “typed” relation  $r: p \leftrightarrow q$  on heaps (i.e.,  $r \subseteq p \times q$ ); (2) for each predicate  $p$ ,  $T$  at the diagonal relation  $\Delta_p$  is a per; (3) for all  $r: p \leftrightarrow q$ ,  $T(r)$  is a saturated relation between pers  $T(\Delta_p)$  and  $T(\Delta_q)$ ; (4)  $T(r) \subseteq T(r * r')$ . The morphisms in  $\mathcal{C}'$  are continuous functions that preserve the families of relations. This category has all the categorical structure of  $\mathcal{C}$  that we used in the semantics of this paper. However, it is difficult to interpret the triple types such that the interpretation of the memory allocator `new` lives in the category. Overcoming this problem will be the focus of our research in this direction.

Finally, we would like to extend the relational separation logic [135] to higher-order, following the style of system R [1], and we want to explore the Curry-Howard correspondence of our type system with specification logic [110].

### **Acknowledgements**

We have benefitted greatly from discussions with Uday Reddy and Peter O'Hearn. Yang was supported by grant No. R08-2003-000-10370-0 from the Basic Research Program of the Korea Science & Engineering Foundation. Yang, Birkedal and Torp-Smith were supported by Danish Technical Research Council Grant 56-00-0309.



## Chapter 5

# Refinement and Separation Contexts

### Abstract

A separation context is a client program which does not dereference internals of a module with which it interacts. We use certain “precise” relations to unambiguously describe the storage of a module and prove that separation contexts preserve such relations. We also show that a *simulation* theorem holds for separation contexts, while this is not the case for arbitrary client programs.

### Preface

This chapter is a reprint of the conference paper [70], co-authored with Ivana Mijajlović and Peter O’Hearn, both from Queen Mary, University of London.

## 5.1 Introduction

Pointers wreak havoc with data abstractions [54, 55, 10, 107]. To see why, suppose that a data abstraction uses a linked list in its internal representation; for example, an implementation of resource manager will use a free list. If a client program dereferences or otherwise accesses a pointer into this representation, then it will be sensitive to changes to the internal representation of the module. In theoretical terms, this havoc is manifest in the failure of classical “abstraction, logical relation, simulation” theorems for data abstraction. For example, the client program will behave differently if, say, the first rather than the second field in a cons cell is used to link together elements of a free list.

Data refinement is a method where one starts with an abstract specification of a data type and derives its concrete representation. Hoare introduced a method of refinement for imperative programs [52, 49]. His treatment of refinement assumes a static-scope based separation between the abstract data type and variables of the client. Pointers break those assumptions, as described above.

Previous approaches to abstraction in the presence of pointers [54, 10, 107, 36, 30] typically work by restricting what can point across certain boundaries. These solutions are limited and complex, and have difficulty coping with situations where pointers transfer between program components or where pointers across boundaries do exist without being dereferenced at the wrong time.

Separation logic [115], on the other hand, enables us to check code of a client for safety, even if there are pointers into the internals of a module [88]. It just ensures that pointers not be dereferenced at the wrong time, without permission.

This paper takes a first step towards bringing the ideas from separation logic into refinement. We present a model, but not yet a logic, which ensures separation between a client and a module, throughout the process of refinement of the module. Our conditions for abstraction, based on a notion of *separation context*, are considerably simpler than ones developed by Banerjee et al [10] and Reddy et al [107], and can easily handle examples with dangling pointers and examples of dynamic ownership transfer. We illustrate this with the nastiest problem we know of – toy versions of `malloc` and `free` [64].

The paper is organized as follows: we give some basic ideas and motivation in Section 5.2. In Section 5.3, we give relevant definitions regarding the programming language and relations on states. This enables us to define *unary separation contexts* in Section 5.4, and to prove properties about them. A separation context is a client program that does not dereference pointers into module internals. The idea that a module owns a part of the heap is described by a *precise* relation, which is a special kind of relation that unambiguously identifies a specific portion of the heap. We show that separation contexts respect these unary relations, where arbitrary contexts do not. Finally, in Section 5.5, we prove a *simulation theorem* which is a cousin of a classic logical relations or abstraction theorem, and which fails when a context is not a separation context. We also give a condition which ensures that a separation context for an abstract module is automatically a separation context for all its refinements.

## 5.2 Basic Ideas

We discuss two simple examples in which we consider two different pieces of client code. In both programs we assume that the client code interacts with the memory manager module through two provided operations, `new()` and `dispose()`, for allocating and disposing memory, respectively. Suppose the module keeps locations available for allocating to a client, in a singly linked list.

To begin with we regard the program state as being separated into two parts, one of which belongs to a client, and the other which belongs to the module. The module's part always contains the free list. The statement `new(x);` takes a location from the free list puts it into `x`; at this point we regard the boundary between the client and module states as shifting: the ownership of the cell has *transferred* from the module to the client, so that the separation between client and module states is dynamic rather than determined once-and-for-all before a program runs. Similarly, when a client disposes a location we regard the ownership of that location as transferring from the client to the module. As mentioned, the concept of “ownership” here is simple: at any program point we regard the state as being separated into two parts, and ownership is just membership in one or the other component of the separated states.

Now, some programs respect this concept of separation while others do not. Consider the following client code.

```
new(x); do something with x; dispose(x); dispose(x)
```

This simple program behaves very badly – it disposes the same location twice. This is possible because after disposing the location pointed to by `x` the first time, `x` still holds the value of the location. Depending on the implementation of `dispose`, this code could

destroy the structure of the free list, and might eventually cause a program crash. This program contradicts our assumption of separation: the second  $\text{dispose}(x)$  statement accesses a cell which the client does not own, since it was previously transferred to the module.

In fact, *any* attempt to use the location after first  $\text{dispose}$  will contradict separation, say if we replace the second  $\text{dispose}$  by a statement  $[x] := 42$  that mutates  $x$ 's location. And both cases contradict abstraction. For instance, if the manager uses the  $[x]$  field as a pointer to the next node in the free list, then  $[x] := 42$  will corrupt the free list, but if the manager uses a different representation of the free list, corruption might not occur: depends whether or not it is representation-dependent.

In contrast, the following code obeys separation: the client code reads and writes to its own part, and disposes only a location which belongs to it.

$$\text{new}(x); [x] := 15; y := [x]; \text{dispose}(x)$$

The issue here is not exclusive to low-level programming languages. In a garbage collected language, thread and connection pools are sometimes used to avoid the overhead of creating and destroying threads and database connections (such as when in a web server). Then, a thread or connection id should not be used after it has been returned to a pool, until it has been doled out again.

In the formal development to follow, a “separation context” will be a piece of client code together with a precondition which ensures respect for separation.

### 5.3 Preliminary Definitions

In this section, we give relevant definitions regarding the storage model and relations in it, as well as a programming language and its semantics.

#### Storage Model

We describe our models in an abstract way, which will allow various realizations of “heaps”. We assume a countably infinite set  $\text{Var}$  of variables given. Let  $S : \text{Var} \rightarrow \text{Val}$  be the set of *stacks* (that is, finite, partial maps from variables to values), and let  $H$  be a set of *heaps*, where we just assume that we have a set with a partial commutative monoid structure  $(H, *, e)$ . In effect, our development is on the level of the abstract model theory of BI [104], rather than the single model used in separation logic [59, 115]. We assume that  $*$  is injective in the sense that for each  $h$ , the partial function  $h * - : H \rightarrow H$  is injective. The set of *states* is the set of stack-heap pairs.

The subheap order  $\sqsubseteq$  is induced by  $*$  in the standard way

$$h_1 \sqsubseteq h_2 \iff \exists h_3. h_1 * h_3 = h_2.$$

Two heaps  $h_1$  and  $h_2$  are *disjoint*, denoted  $h_1 \# h_2$ , if  $h_1 * h_2$  is defined.

We will often take  $H$  to be a set of finite partial functions

$$H = \text{Ptr} \rightarrow_{\text{fin}} \text{Val}, \text{ where } \text{Ptr} = \{0, 1, 2, \dots\} \quad \text{Val} = \{\dots, -1, 0, 1, \dots\}.$$

The combination  $h * h'$  of two such heaps is defined only when they have disjoint domains, in which case it is the union of the graphs of the two functions. We will not restrict ourselves to this (RAM) model, but will assume it in examples unless stated differently.

## Separation logic

Separation logic is an extension of Hoare logic, where *heaps* have been added to the storage model. The usual assertion language of Hoare logic is extended with assertions that express properties about heaps

$$A, B ::= \text{emp} \mid e_1 \mapsto e_2 \mid A * B \mid \top \mid \forall_* p \in m. A \mid \dots$$

The first asserts that the heap is empty, the second says that the current heap has exactly one pointer in its domain, and the third is the *separating conjunction* and means that the current heap can be split into two disjoint parts for which  $A$  and  $B$  hold, respectively. The fourth is true for any state, and the last assertion form is an iterated separating conjunction over a finite set. The semantics of assertions is given by a judgement  $s, h \models A$  which asserts that the assertion  $A$  holds in the state  $(s, h)$ . More about separation logic can be found in [115].

## Unary relations

Certain special properties are used to identify the heap portion owned by a module [88].

**Definition 5.1.** A relation  $M \subseteq S \times H$  is *precise* if for any state  $s, h$  there is at most one subheap  $h_0 \sqsubseteq h$ , such that  $(s, h_0) \in M$ .

We illustrate precise unary relations with an example in the RAM model. Let  $\alpha$  be a sequence of integers. The predicate  $\text{list}(\alpha, x)$  is defined inductively on the sequence  $\alpha$  by

$$\text{list}(\varepsilon, x) \stackrel{\text{def}}{=} x = \text{nil} \wedge \text{emp}, \quad \text{list}(a \cdot \alpha, x) \stackrel{\text{def}}{=} x = a \wedge \exists y. x \mapsto y * \text{list}(\alpha, y)$$

where  $\varepsilon$  represents the empty sequence and  $\cdot$  conses an element  $a$  onto the front of a sequence  $\alpha$ . This predicate says that  $x$  points to a non-circular singly-linked list whose addresses are the sequence  $\alpha$  (this is called a “Bornat list” in [115]). For any given  $(s, h)$ , there can be at most one subheap of  $h$  which satisfies  $\text{list}(\alpha, x)$ , consisting of the cells in  $\alpha$ . Generally, a precise relation gives you a way to “pick out the relevant cells”.

We define the *separating conjunction of unary relations*  $M, M' \subseteq S \times H$  by

$$M * M' = \{(s, h) \mid \exists h_0, h_1. h_0 \# h_1 \wedge h = h_0 * h_1 \wedge (s, h_0) \in M \wedge (s, h_1) \in M'\}.$$

Taking into account that  $*$  is injective, a precise relation  $M$  induces a unique splitting of a state  $(s, h)$ . We write  $(s, h_M)$  for the substate of  $(s, h)$  uniquely described by  $M$ , if it exists. Otherwise,  $(s, h_M) = e$ , the unit.

## The Model

Our model will use a simple language with two kinds of atomic operations: the client operations and the module operations. The denotation of client commands will be given by functions  $f : (S \times H) \rightarrow (S \times H) \uplus \{\text{wrong}\}$ , and the denotation of module operations will be given by binary relations  $t \subseteq (S \times H) \times (S \times H) \uplus \{\text{wrong}\}$ . The special state *wrong* results when a program illegally accesses storage beyond the current heap. We presume there is a fixed set of module variables  $\text{Var}_M$ , which are never changed by the client:

$$\forall x \in \text{Var}_M. \begin{aligned} f(s, h) = \text{wrong} &\Leftrightarrow \forall v. f(s \setminus \{x \mapsto v\}, h) = \text{wrong} \text{ and} \\ f(s, h) = (s', h') &\Leftrightarrow \forall v. f(s \setminus \{x \mapsto v\}, h) = (s' \setminus \{x \mapsto v\}, h'). \end{aligned}$$

For a unary relation on states  $M$ , we write  $M_{wrong}$  to denote  $M \cup \{wrong\}$ . We will write  $(s, h)[t](s', h')$  to denote that the states  $(s, h)$  and  $(s', h')$  are in the binary relation  $t$ .

The relation  $M \subseteq S \times H$  is said to be *preserved* by a function  $f$  (respectively relation  $t$ ) on states, if for all  $(s, h), (s', h')$ , such that state  $(s, h)$  is in  $M$  and  $f(s, h) = (s', h')$  (respectively  $(s, h)[t](s', h')$ ), imply  $(s', h') \in M_{wrong}$ .

The reader will have recognized an asymmetry in our model: client primitive operations are required to be deterministic, while in module operations nondeterminism is allowed. One effect of this is that, when frame conditions are imposed later, the client operations will not be able to do any allocation; allocation will have to be viewed as a module operation. Technically, the determinism restriction is needed for our simple simulation theorem.

### Local Functions and Relations

We consider functions and relations on states that access resources in a local way. More formally, we say that a function  $f : (S \times H) \rightarrow (S \times H) \uplus \{wrong\}$  (relation  $t \subseteq (S \times H) \times (S \times H) \uplus \{wrong\}$ ) is *local* [88] if it satisfies the properties:

- **Safety Monotonicity:** For all states  $(s, h)$  and heaps  $h_1$  such that  $h \# h_1$ , if  $f(s, h) \neq wrong$  (respectively  $\neg(s, h)[t]wrong$ ), then  $f(s, h * h_1) \neq wrong$  (respectively  $\neg(s, h * h_1)[t]wrong$ ).
- **Frame Property:** For all states  $(s, h)$  and heaps  $h_1$  with  $h \# h_1$ , if  $f(s, h) \neq wrong$  (respectively  $\neg(s, h)[t]wrong$ ) and  $f(s, h * h_1) = (s', h')$ , (respectively  $(s, h * h_1)[t](s', h')$ ) then there is a subheap  $h'_0 \sqsubseteq h'$  such that  $h'_0 \# h_1$ ,  $h'_0 * h_1 = h'$  and  $f(s, h) = (s', h'_0)$  (respectively  $(s, h)[t](s', h'_0)$ ).

The properties are the ones needed for soundness of the Frame Rule of separation logic; see [136]. We will only consider local functions and relations.

### Programming Language

The programming language is an extension of the simple while-language with a finite set of atomic client operations  $f_j$  ( $j \in J$ ) and a finite set of module operations  $oper_i$ ,  $i \in I$ . The syntax of the *user language* is

$$\begin{aligned} c_{user} &::= f_j, j \in J \mid oper_i, i \in I \mid c_1; c_2 \mid \mathbf{if} \ e \ \mathbf{then} \ c \ \mathbf{else} \ c \mid \mathbf{while} \ e \ \mathbf{do} \ c, \\ e &::= \mathit{int} \mid \mathit{var} \mid e + e \mid e \times e \mid e - e, \ \mathit{int} \in \mathbb{Z}, \ \mathit{var} \in \mathit{Var}, \\ \mathbb{Z} &= \{\dots, -1, 0, 1, \dots\}, \ \mathit{Var} = \{x, y, \dots\}, \ I, J - \text{finite indexing sets.} \end{aligned}$$

The expressions used in the language do not access heap storage. Commands such as  $x := e$ ,  $[e_1] := e_2$ ,  $x := [e]$ , etc. are examples of atomic operations.

The semantics of the language is parameterized by a precise relation  $M$  and a collection  $(oper_i)_{i \in I}$  of binary relations that preserve  $M * T$ . It defines a big-step transition relation  $\rightsquigarrow \subseteq (c_{user} \times (S \times H)) \times ((S \times H) \uplus \{wrong, av\})$  on configurations, where  $av$  denotes a state in which client code illegally accesses the heap storage owned by the module, and will be referred to as “access violation”. The operational semantics of the language is given in Table 5.1. State  $(s, h_M) \in M$  denotes the substate of  $(s, h)$  uniquely determined by relation  $M$  in the second rule. What is left over,  $(s, h_U)$ , is the client’s state.  $K$  denotes an element of  $(S \times H) \uplus \{av, wrong\}$ .

Table 5.1: Operational semantics

$$\begin{array}{c}
\hline
\frac{f_j(s, h) = (s', h')}{f_j, s, h \rightsquigarrow s', h'} \quad \frac{(s, h) = (s, h_M) * (s, h_U) \quad f_j(s, h) \neq \text{wrong} \quad f_j(s, h_U) = \text{wrong}}{f_j, s, h \rightsquigarrow \text{av}} \quad \frac{f_j(s, h) = \text{wrong}}{f_j, s, h \rightsquigarrow \text{wrong}} \\
\frac{(s, h)[\text{oper}_i](s', h')}{\text{oper}_i, s, h \rightsquigarrow s', h'} \quad \frac{(s, h)[\text{oper}_i]\text{wrong}}{\text{oper}_i, s, h \rightsquigarrow \text{wrong}} \quad \frac{c_1, s, h \rightsquigarrow s', h' \quad c_2, s', h' \rightsquigarrow K}{c_1; c_2, s, h \rightsquigarrow K} \\
\frac{c_1, s, h \rightsquigarrow \text{wrong}}{c_1; c_2, s, h \rightsquigarrow \text{wrong}} \quad \frac{\llbracket e \rrbracket s = 0}{\text{while } e \text{ do } c, s, h \rightsquigarrow s, h} \quad \frac{\llbracket e \rrbracket s \neq 0 \quad c; \text{while } e \text{ do } c, s, h \rightsquigarrow K}{\text{while } e \text{ do } c, s, h \rightsquigarrow K} \\
\frac{c_1, s, h \rightsquigarrow \text{av}}{c_1; c_2, s, h \rightsquigarrow \text{av}} \quad \frac{\llbracket e \rrbracket s \neq 0 \quad c_1, s, h \rightsquigarrow K}{\text{if } e \text{ then } c_1 \text{ else } c_2, s, h \rightsquigarrow K} \quad \frac{\llbracket e \rrbracket s = 0 \quad c_2, s, h \rightsquigarrow K}{\text{if } e \text{ then } c_1 \text{ else } c_2, s, h \rightsquigarrow K} \\
\hline
\end{array}$$

## 5.4 Unary Separation Contexts

An essential point in the semantics in Table 5.1 is the way that module state is subtracted when client operations  $f_j$  are performed. If a client operation does not go wrong in a global state, but goes wrong when the module state is subtracted, we judge that this was due to an attempt to access the module’s state; in the semantics this is rendered as an access violation, and a separation context is then a program (with a precondition) that does not lead to access violation.

**Definition 5.2.** Let  $M \subseteq S \times H$  be a precise unary relation, let  $P$  be a unary predicate on states, and for  $i \in I$  let  $\text{oper}_i \subseteq (S \times H) \times (S \times H) \uplus \{\text{wrong}\}$  preserve relation  $M * T$ . A program  $c$  is a *unary separation context* for  $M, P$  and  $(\text{oper}_i)_{i \in I}$  if for all executions and all  $(s, h) \in M * P$   $c, s, h \not\rightsquigarrow \text{av}$ .

The idea is that  $M$  describes the heap storage owned by the module, and a separation context will never access that storage. Separation contexts preserve the resource invariant of a module because they change storage owned by the module only through the provided operations.

**Theorem 5.3.** Let  $M \subseteq S \times H$  be a precise relation, let  $P$  be a unary predicate on states, and for  $(i \in I)$  let  $\text{oper}_i \subseteq S \times H \times (S \times H) \uplus \{\text{wrong}\}$  preserve  $M * T$ , and let  $c$  be a separation context for  $M, P$  and  $(\text{oper}_i)_{i \in I}$ . Then for all such  $P$  and all states  $(s, h)$  and  $(s', h')$ , if  $(s, h) \in M * P$ , and  $c, s, h \rightsquigarrow s', h'$ , then  $(s', h') \in (M * T)_{\text{wrong}}$ .

### Separation Context Examples

We now revisit the ideas discussed in Section 5.2 in our more formal setting. In order to specify the operations of the memory manager module, we make use of the “greatest relation” for the specification  $\{P\}\text{oper}\{Q\}[X]$ , which is the largest local relation satisfying a triple  $\{P\} - \{Q\}$  and changing only the variables in the set  $X$ . It is similar to the “generic commands” introduced by Schwarz [118] and the “specification statements” studied in the refinement literature, but adapted to work with locality conditions in [88].

The predicate  $\exists \alpha. \text{list}(\alpha, ls)$  describes the free list, and we choose it as the  $M$  component in the definition of a separation context. The operations  $\text{new}(x)$  and  $\text{dispose}(x)$  are the greatest relations satisfying the following specifications.

$$\begin{aligned} \text{new}_C(x) : & \quad \{\text{list}(a \cdot \alpha, ls)\} - \{\text{list}(\alpha, ls) * x \mapsto a\}[x, ls] \\ & \quad \{\text{list}(\varepsilon, ls)\} - \{\text{list}(\varepsilon, ls) * x \mapsto -\}[x, ls] \\ \text{dispose}_C(x) : & \quad \{\text{list}(\alpha, ls) * x \mapsto a\} - \{\text{list}(a \cdot \alpha, ls)\}[ls] \end{aligned}$$

For future reference, we will call this the *concrete* interpretation of the memory manager module. With these definitions we can judge whether a program (together with a precondition) is a separation context.

Consider the following three programs

$$\begin{array}{lll} \text{Program}_1 : & \text{Program}_2 : & \text{Program}_3 : \\ \text{new}(x); & \text{dispose}(x); & [81] := 42 \\ [x] := 47; & [x] := 47; & \\ \text{dispose}(x); & & \end{array}$$

We indicate whether a program, together with a precondition, is a separation context in the following table.

Context	Separation context?
$\{\text{emp}\} \text{Program}_1$	✓
$\{x \mapsto -\} \text{Program}_2$	✓
$\{\text{emp}\} \text{Program}_2$	×
$\{81 \mapsto -\} \text{Program}_3$	✓
$\{\text{emp}\} \text{Program}_3$	×

Most of the entries are easy to explain, and correspond to our informal discussion from earlier. The last one, though, requires some care. For, how do we know that  $[81] := 42$  interferes with the free list? The answer is that we do not. It might or might not be the case that location 81 is in the free list, at any given point in time. But, the notion of separation context is fail-safe: if there is *any* possibility that 81 is in the free list, on any run, then the program is judged not to be a separation context. And we can easily construct an example state where 81 is indeed in the free list. On the other hand in the second-last entry the precondition  $81 \mapsto -$  ensures that 81 cannot be in the free list. This is because of the use of  $*$  to separate the module and client states.

## 5.5 Refinement and Separation

In this section we first introduce precise binary relations and the separating conjunction of binary relations. We give a definition of refinement and prove a binary relation-preservation theorem.

Let  $R \subseteq (S_0 \times H_0) \times (S_1 \times H_1)$  be a binary relation. We say that  $R$  is *precise*, if each of its two projections on the corresponding set of states is precise. Formally, for any state  $(s_i, h_i) \in (S_i \times H_i)$  there is at most one  $h'_i \sqsubseteq h_i$  such that there exists a state  $(s_{1-i}, h_{1-i}) \in (S_{1-i} \times H_{1-i})$  such that  $(s_i, h'_i)[R](s_{1-i}, h_{1-i})$ , for  $i=0,1$ .

We illustrate precise binary relations with an example. Suppose we have two different implementations of a memory manager module. In the first implementation we assume that  $f$  is a set variable, which keeps track of all owned locations. In the second implementation, we let this information be kept in a list. We use the list predicate

$\text{list}(\alpha, ls)$ , defined in Section 5.3. Now, a precise binary relation

$$R = \left\{ ((s, h), (s', h')) \mid \begin{array}{l} (s, h \models \forall_* p \in f. p \mapsto -) \wedge (s', h' \models \text{list}(\alpha, ls)) \wedge \\ \text{set}(\alpha) = s(f) \end{array} \right\},$$

where  $\text{set}(\alpha)$  is defined as the set of pointers in the sequence  $\alpha$ , relates these two implementations. Relation  $R$  relates pairs of states, such that one state can be described as a set of different pointers, while the other is determined by the list of exactly the pointers that appear in the mentioned set.

For two binary relations  $R, R' \subseteq (S_1 \times H_1) \times (S_2 \times H_2)$  on states, we define their separating conjunction [107] as

$$R * R' = \left\{ ((s_1, h_1), (s_2, h_2)) \mid \begin{array}{l} \exists h'_1, h''_1, h'_2, h''_2. h_1 = h'_1 * h''_1 \wedge h_2 = h'_2 * h''_2 \wedge \\ (s_1, h'_1)[R](s_2, h'_2) \wedge (s_1, h''_1)[R'](s_2, h''_2) \end{array} \right\}$$

Similarly to the unary case, for a binary relation on states  $R$  we will write  $R_{\text{wrong}}$  to denote  $R \cup \{(\text{wrong}, \text{wrong})\}$ .

### 5.5.1 Refinement and Separation Contexts

In this section, we formally express what it means for one module to be a *refinement* (or an implementation) of another. For simplicity, we assume that there is only one operation of the module, i.e., that the index set  $I$  from the syntax of the user language is singleton. In previous work on refinement [49], our definition of refinement is called a downward simulation.<sup>1</sup>

In the following, we will take  $H_1$  and  $H_2$  to be two (in general different, but possibly equal) heap models, assuming that  $(H_1, *_1, e_1)$  and  $(H_2, *_2, e_2)$  have partial commutative monoid structure.

**Definition 5.4.** Let  $Z \subseteq (S_1 \times H_1) \times (S_2 \times H_2)$  be a binary relation. We define  $\text{oper}^2 \subseteq (S_2 \times H_2) \times (S_2 \times H_2) \uplus \{\text{wrong}\}$  to be a *refinement* of  $\text{oper}^1 \subseteq (S_1 \times H_1) \times (S_1 \times H_1) \uplus \{\text{wrong}\}$  with respect to  $Z$ , if

- for all states  $(s_1, h_1), (s_2, h_2), (s'_2, h'_2)$ , such that  $(s_1, h_1)[Z](s_2, h_2)$  and  $(s_2, h_2)[\text{oper}^2](s'_2, h'_2)$  there exists a state  $(s'_1, h'_1)$ , such that  $(s_1, h_1)[\text{oper}^1](s'_1, h'_1)$ , and  $(s'_1, h'_1)[Z](s'_2, h'_2)$ , and
- for all states  $(s_1, h_1), (s_2, h_2)$ , such that  $(s_1, h_1)[Z](s_2, h_2)$  if  $(s_2, h_2)[\text{oper}^2]\text{wrong}$ , then  $(s_1, h_1)[\text{oper}^1]\text{wrong}$ .

In order to prove the relation preservation theorem, we instantiate the refinement relation to a separating conjunction of binary relations,  $R, Q \subseteq (S_1 \times H_1) \times (S_2 \times H_2)$ . We assume that the following properties hold:

- $R$  is precise
- $Q$  is such that for any two states  $(s_1, h_1), (s_2, h_2)$  related by  $Q$  and a guard (condition of **if** and **while** statements)  $b$ ,  $s_1(b) \Leftrightarrow s_2(b)$
- $\text{oper}^2 \subseteq (S_2 \times H_2) \times (S_2 \times H_2) \uplus \{\text{wrong}\}$  is a refinement of  $\text{oper}^1 \subseteq (S_1 \times H_1) \times (S_1 \times H_1) \uplus \{\text{wrong}\}$  with respect to  $R * Q$

<sup>1</sup>It can sometimes be confusing to read the early literature on refinement, but we stick to the jargon of [49, 53, 79]

- We denote a pair  $(f_j^1, f_j^2)$  by  $\mathbf{f}_j$ . Pair  $\mathbf{f}_j$  is such that it maps  $Q$ -related states to  $Q_{wrong}$ -related states.

The role of  $R$  is to relate abstract and concrete subheaps which belong to the module, while  $Q$  relates the clients' parts of the heaps.

**Simulation Theorem (Informally):** Suppose we have two instantiations of a client program, which use calls to concrete and abstract module operations respectively, related by a refinement relation. Then, provided both of these two instantiations are separation contexts with respect to the corresponding modules, the effect of the concrete computation can be tracked by the abstract.

Stating this more formally requires some notation. For a program  $c$ , let  $c_i \subseteq (S_i \times H_i) \times (S_i \times H_i) \uplus \{wrong\}$  be a relation denoted by  $c$  in the operational semantics defined by  $R_i$  and  $oper^i$ ,  $i = 1, 2$ , where  $R_i$  is the projection of  $R$  onto  $(S_i \times H_i)$ . Also, let  $Q_P$  denote  $Q \cap (P \times Q(P))$ , where  $Q$  is a binary relation on states,  $P$  is a unary relation on states, and  $Q(P)$  is their composition.

**Theorem 5.5 (Simulation Theorem).** *Let  $R, Q, oper^i, c, c_i$  for  $i = 1, 2$ , be as above, and let  $P \subseteq Q_1$  be a unary relation on states. Let  $c_1$  be a separation context for  $R_1, P$  and  $oper^1$ , and let  $c_2$  be a separation context for  $R_2, Q(P)$  and  $oper^2$ . Then for all such  $P$  and all  $(s_1, h_1), (s_2, h_2), (s'_2, h'_2)$ , if  $(s_1, h_1) [R * Q_P] (s_2, h_2)$  and  $(s_2, h_2) [c_2] (s'_2, h'_2)$  then there exists a state  $(s'_1, h'_1)$  such that  $(s_1, h_1) [c_1] (s'_1, h'_1)$  and  $(s'_1, h'_1) [R * Q] (s'_2, h'_2)$ .*

The crucial assumption is that  $c_1$  and  $c_2$  are separation contexts for the given modules and preconditions; without this condition, the theorem fails.

One shortcoming is that we have to check whether both  $c_1$  and  $c_2$  are separation contexts to apply Theorem 5.5. From the point of view of program development it would be better if we knew that when we had a separation context for an abstract module then it would automatically remain a separation context for all its refinements. Then the check could be done once and for all. In order to realize this aim, an extra concept is needed: safety. A safe separation context is a client which does not touch any storage not in its possession.

**Definition 5.6 (Safe Separation Context).** Let  $c$  be a separation context for the precise relation  $M$ , precondition  $P$  and family of operations  $(oper^i)_{i \in I}$ . Program  $c$  is a *safe separation context* for  $M, P, (oper^i)_{i \in I}$  if for all executions and all states  $(s, h) \in M * P, c, s, h \not\rightarrow wrong$ .

**Theorem 5.7.** *Let  $R, Q, oper^i, c, c_i$  for  $i = 1, 2$  be as in Theorem 5.5, and let  $P \subseteq Q_1$  be a unary relation on states. If  $c_1$  is a safe separation context for  $R_1, P$  and  $oper^1$ , then  $c_2$  is a safe separation context for  $R_2, Q(P)$  and  $oper^2$ .*

### Safe Separation Context Example

To see the role of the concept of safety, consider an *abstract* version of the memory manager procedures, the “magical malloc module”. It is magical in that the module does not own any locations at all, producing them as if out of thin air (In implementation terms, the thin air is like a call to a system routine such as `sbrk`). Therefore, the resource invariant of the module,  $M$  in our formal setup, is the predicate `emp`, which is clearly precise.

Now, we define the abstract operations  $\text{new}_A(x)$  and  $\text{dispose}_A(x)$  as the greatest relations satisfying the following specifications.

$$\text{new}_A(x) : \{\text{emp}\} - \{x \mapsto -\}[x], \quad \text{dispose}_A(x) : \{x \mapsto -\} - \{\text{emp}\}[]$$

This is the meaning of allocation and disposal that is usually presumed in separation logic. Because the manager owns no storage whatsoever, there is no way for a client to trample on it. As a result, *every* client program is a separation context for this abstract module.

But, not every context is safe. Consider the context

$$\{\text{emp}\} [81] := 42$$

from the Separation Context Examples in Section 5.4. It immediately goes wrong, and so is not safe. Recall also that in the more concrete semantics, from the same section, this is not even an ordinary separation context.

This shows the import of Theorem 5.7. If we know that our context is safe in the abstract setting, then this ensures that module internals will not be tampered with in refinements. Put another way, module tampering in a concrete implementation can show up as going wrong in the abstract, and the concept of safe separation context protects against this.

### Refinement Examples

Here, we illustrate refinement relations between different interpretations of the memory manager module with two examples.

To define the refinement relations we borrow some notation from relational separation logic [135]. Let  $S_1 \times H_1$  and  $S_2 \times H_2$  be two state spaces. Let  $P \subseteq S_1 \times H_1$ ,  $Q \subseteq S_2 \times H_2$  and  $R \subseteq (S_1 \times H_1) \times (S_2 \times H_2)$  be predicates. We let

$$\left( \begin{array}{c} P \\ Q \end{array} \right) \wedge R \quad \text{denote} \quad \{(s_1, h_1), (s_2, h_2) \mid (s_1, h_1 \models P \wedge s_2, h_2 \models Q) \wedge R\}.$$

The first example involves refinement between the *abstract* and the *concrete* interpretations of the memory manager module. We have already specified both interpretations, the abstract – in the Safe Separation Context Example above, and the concrete – in the ordinary Separation Context Example from Section 5.4.

The refinement relation  $Z_{AC}$  between these two interpretations is a separating conjunction of binary relations  $R_{AC}$  and  $Q_{AC}$ . These are given by

$$R_{AI} = \left( \begin{array}{c} \text{emp} \\ \exists \alpha. \text{list}(\alpha, ls) \end{array} \right) \quad Q_{AI} = \mathbf{Id}.$$

Relation  $R_{AI}$  relates modules' states of the two interpretations and is basically the relation between their resource invariants. Relation  $Q_{AC}$  relates the clients' states and is the identity relation.

In the second example, we introduce an *intermediate* version of the memory manager module. We do this for two reasons. First, this illustrates the use of two different heap models, as allowed in our formal setting. Second, considering refinement between the intermediate and the concrete interpretations requires a subtler refinement relation.

On the intermediate level, the intention is to keep locations owned by the module in a set, without committing to the representation of the set. If this set becomes empty, we call a “system routine” (like `sbrk`) to get a new location.

For this interpretation, we assume the following heap model. Let  $Loc$  be an infinite set of locations. A heap will be an element of the Cartesian product  $\mathcal{P}_{fin}(Loc) \times H_1$ , where  $(H_1, *_1, e_1)$  is the partial commutative monoid of the RAM model. We say that a pair  $(N, h)$  from this product is *well-defined* if  $N \cap \text{dom}(h) = \emptyset$ . The intermediate heap model  $H$  consists of these well-defined elements. Two intermediate heaps  $(N_1, h_1)$  and  $(N_2, h_2)$  are disjoint,  $(N_1, h_1) \#_1 (N_2, h_2)$ , whenever  $N_1 \cap N_2 = \emptyset$  and  $N_1 \cap \text{dom}(h_2) = \emptyset$  and  $N_2 \cap \text{dom}(h_1) = \emptyset$  and  $\text{dom}(h_1) \cap \text{dom}(h_2) = \emptyset$ . We define  $*$  between two heaps by

$$(N_1, h_1) * (N_2, h_2) = \begin{cases} (N_1 \cup N_2, h_1 *_1 h_2) & \text{if } (N_1, h_1) \#_1 (N_2, h_2) \text{ and} \\ & (N_1, h_1), (N_2, h_2) \text{ well defined;} \\ \text{undefined} & \text{otherwise} \end{cases}$$

We say that  $s, (N, h) \models \text{act}(p)$  if and only if  $p \in N$ . The resource invariant can be described with  $\forall_* p \in f. \text{act}(p)$ , where  $f$  is a set variable. We now define operations  $\text{new}_I(x)$  and  $\text{dispose}_I(x)$  as the greatest relations satisfying the specifications

$$\begin{aligned} \text{new}_I(x) : & \quad \{ \forall_* p \in f. \text{act}(p) \wedge f = Y \neq \emptyset \} - \{ (\forall_* p \in f. \text{act}(p) \wedge f = Y \setminus \{x\}) * \\ & \quad x \mapsto - \} [x, f] \\ & \quad \{ \forall_* p \in f. \text{act}(p) \wedge f = \emptyset \} - \{ (\forall_* p \in f. \text{act}(p) \wedge f = \emptyset) * x \mapsto - \} [x] \\ \text{dispose}_I(x) : & \quad \{ (\forall_* p \in f. \text{act}(p) \wedge f = Y) * x \mapsto - \} - \{ \forall_* p \in f. \text{act}(p) \wedge \\ & \quad f = Y \cup \{x\} \} [f] \end{aligned}$$

The variable  $Y$  is used to keep track of the initial contents of  $f$ , similarly to how  $\alpha$  was used in the concrete interpretation. Note that it is not altered because it is not in the modifies set, a set of actual locations owned by the module. We intend that  $\text{new}_I(x)$  is the greatest relation satisfying both stated specifications.

Now, the refinement relation  $Z_{IC}$  between intermediate and concrete relations is a separating conjunction of binary relations  $R_{IC}$  and  $Q_{IC}$  given by

$$R_{IC} = \left( \begin{array}{c} f \\ \text{list}(\alpha, ls) \end{array} \right) \wedge \text{set}(\alpha) \text{val}(f) \quad Q_{IC} = \mathbf{Id},$$

where  $\text{val}(f)$  is the value of set variable  $f$ . It can be verified that the operations preserve these relations as required in the definition of refinement.

In these two examples we have not exercised the possibility of using a non-identity relation to relate the abstract and concrete client states. A good such example compares two implementations of a buffer, one of which copies two values where the other passes a single pointer to the two values.

### Directions for future work

There are several directions for further work. First, we have, for simplicity, considered the interaction between a client and a single module; in the future we plan on investigating independence between modules. Second, it would be worthwhile to consider multiple-instance classes (e.g. [10]); here we have, in effect, a single single-instance class. It would also be important to remove the restriction of determinism, imposed to the client operations. Finally, we would like to use the model to make the connection back

to logic. Perhaps a relational version of the hypothetical frame rule, or the modular procedure rule, from [88] can be formulated, borrowing from Yang's relational separation logic [135].

### **Acknowledgements**

We would like to thank Hongseok Yang, Josh Berdine, Richard Bornat and Cristiano Calcagno for invaluable discussions and anonymous referees for their careful comments. Torp-Smith's research was partially supported by Danish Natural Science Research Council Grant 51-00-0315 and Danish Technical Research Council Grant 56-00-0309. Mija-jlović and O'Hearn were supported by the EPSRC.

## Chapter 6

# Towards a Parametric Model for Separation-logic Typing

### Abstract

In a recent LICS paper [25], we present a categorical model for a  $\lambda$ -calculus with imperative constructs for heap manipulations equipped with a separation logic type system. Here, we take the first steps toward a *parametric* model for separation logic typing.

### Preface

This chapter is a previously unpublished manuscript, co-authored with Lars Birkedal from IT University of Copenhagen and Hongseok Yang from ERC-ACI, Seoul National University.

## 6.1 Introduction

In the paper [25] (which we henceforth refer to as “the LICS paper”), we provide a categorical model for a version of idealized algol adapted for separation logic typing. That work brings ideas from separation logic into higher-order programming languages, and the separation-logic type system includes many interesting frame rules.

Since the programming language in the LICS paper is a typed  $\lambda$ -calculus, it is reasonable to work towards a *parametric* model, as mentioned in the future work section of the LICS paper. One purpose of this is to give reasoning principles for data abstraction in the presence of pointers since separation-logic types let us control access to data structures better, as it was pointed out by Uday Reddy. Stated informally, the work in the LICS paper lets us give separation-logic types to terms in a typed imperative  $\lambda$ -calculus. Recently, there has been some work on *relational reasoning* for imperative programs [135, 13], and this is the direction, we set out to pursue in this work. In contrast to the just mentioned works, which both consider a simple first-order programming language, we consider a higher-order programming language here. The goal, both in the present work and in the works [135, 13] just mentioned, is to give reasoning principles for proving *relationships between programs*, rather than properties of a single program, as it is done in the LICS paper and most of the hitherto published literature on separation logic. Motivations for such a goal are many; for example, good reasoning principles

for deriving relationships between programs can be used in connection with optimizing transformations in compilers, and for data abstraction.

In Yang's paper on *relational separation logic* [135], the meaning of the central notion of a *quadruple*

$$\{R\} \begin{array}{c} C_1 \\ C_2 \end{array} \{S\} \quad (6.1)$$

is that two states that are related by the relation  $R$  are taken to states that are related by the relation  $S$  by the commands  $C_1, C_2$ , respectively (if they terminate). With the work of the LICS paper in mind, it is a reasonable goal to, stated very informally, extend these quadruples to a higher-order programming language by essentially giving a *relational interpretation of types* from the LICS paper, and *quadruple types* to pairs of programs, and to show a property resembling the fundamental theorem of logical relations [101]: if a program  $M$  has a type  $\theta$  in separation-logic typing, the pair  $(M, M)$  preserves the identity relation corresponding to the relational interpretation of  $\theta$ . In order to prove such properties, we have to exhibit a model which captures the notions mentioned above.

In this paper, we present the first step towards such a model. Much of the work follows the same lines as the LICS paper, and since this work is supposed to extend the work in the LICS paper, it is recommended that readers familiarize themselves with that.

Before we proceed with the technical development, we briefly summarize the intuition and motivation behind it in a synopsis.

## 6.2 Synopsis of the Categorical Model

In the LICS paper, types as are modeled objects in the category  $\mathcal{C}$ , which has as objects pairs  $(o, P)$ , where  $o$  is a pcpo, and  $P$  is a family of admissible pers on  $o$ , indexed by predicates (*i.e.*, subsets of heaps), such that  $P(p) \subseteq P(p * q)$  for all predicates  $p, q$ . The morphisms  $f : (o, P) \rightarrow (o', P')$  are continuous functions  $f : o \rightarrow o'$  with  $f[P(p) \rightarrow P'(p)]f$  for all predicates  $p$ . An alternative description of  $\mathcal{C}$  can be obtained by considering a full subcategory of the category of functors between the preorder  $\mathcal{E}_v$  induced by  $*$  on the set of heaps and the category  $\mathcal{D}_v$  whose objects are pairs of a cpo and an admissible per on it and whose morphisms are continuous maps preserving the pers. It is easy to see that  $\mathcal{C}$  is equivalent to the full subcategory of this functor category which consists of functors that are constant on the cpo part.

The development towards a parametric model can be described in steps. First, we use a different indexing. For the sake of generality, we simply use a commutative monoid  $(M, \cdot)$  instead of the specific monoid  $\mathcal{E}_v$  from before. In the second step, our model is extended to give a *relational interpretation of types*. This follows the lines of earlier work [116, 86, 40] and uses *reflexive graphs* [86, 41] to give a relational interpretation. Hence we extend  $\mathcal{E}_v$  and  $\mathcal{D}_v$  to reflexive graphs. This is straightforward for  $\mathcal{E}_v$ , and work in the papers [8, 1, 67] indicates that the right way to extend  $\mathcal{D}_v$  to a reflexive graph is to define the category  $\mathcal{D}_e$  to have as objects *saturated relations*  $r : p \leftrightarrow q$  between pers  $p$  and  $q$ . These can, informally, be seen as "relations on pers". Hence, we have the reflexive graphs as in the diagram

$$\begin{array}{ccc} \mathcal{E}_e & \xrightarrow{F_e} & \mathcal{D}_e \\ \delta_0 \left( \begin{array}{c} \uparrow \\ Id \\ \downarrow \end{array} \right) \delta_1 & & \delta'_0 \left( \begin{array}{c} \uparrow \\ Id' \\ \downarrow \end{array} \right) \delta'_1 \\ \mathcal{E}_v & \xrightarrow{F_v} & \mathcal{D}_v \end{array} ,$$

and the relational interpretation of types then takes place in the category of functor pairs between these reflexive graphs that are constant on the cpo-part. As it turns out, these functor pairs are determined by the functor between  $\mathcal{E}_e$  and  $\mathcal{D}_e$ . The interpretation from the LICS paper corresponds to the  $F_v$  components of such functor pairs. We define a category  $\mathcal{P}_v$  which is a description of this functor category in the style of the LICS paper, and show that this category has much of the same structure as the category  $\mathcal{C}$  from the LICS paper. Objects of  $\mathcal{P}_v$  are thus families of saturated relations, indexed by elements in the monoid  $\mathcal{E}_e$  with certain properties.

Following the lines of the work mentioned above, relations between types are modelled by the so-called “ $W$ -construction”. A relation between types is interpreted as a relation between the basic interpretations of the types (not the relational interpretations of the types), as in the following diagram.

$$\begin{array}{ccc}
 \text{Rel}_1 & & \text{Rel}_2 \\
 \curvearrowright & \mathcal{R} & \curvearrowleft \\
 & \searrow & \swarrow \\
 & \text{Type}_1 & \text{Type}_2 \\
 & \swarrow & \searrow \\
 & & \curvearrowright
 \end{array}
 \tag{6.2}$$

Here, the two “wings” are interpretations of types, including relational interpretations, and the middle component  $\mathcal{R}$  is a relation between the “basic” interpretations of the types. There is a category of functors between such “ $W$ -objects” which is used to interpret relations between types. We define a category in the style of the LICS paper which is an alternative description of this functor category. Here, objects consist of two objects from  $\mathcal{P}_v$  and a family of relations between them, satisfying certain conditions.

The last step in the construction is to go from the abstract reflexive  $\mathcal{E}$  graph (which is a reflexive graph of monoids) back to a more concrete one. In this more concrete setting, more structure, both in  $\mathcal{P}_v$  and  $\mathcal{P}_e$  exists, much like the structure of  $\mathcal{C}$  in the LICS paper. Moreover,  $\mathcal{P}_v$  and  $\mathcal{P}_e$  themselves form a reflexive graph, and the reflexive graph functors preserve much of the structure mentioned above. This, among other things, ensures that a lemma corresponding to the fundamental theorem of logical relations holds.

We emphasize here that this paper is primarily a *semantic* study — the calculi for types, the programming language, *etc.*, are deferred to after the semantics, and will not be given in all details. In what follows, more details of the development of our model are presented.

### 6.3 Developing a Parametric Model

The development of the model proceeds in several steps.

1. Give an alternative definition of a slightly generalized version of the category  $\mathcal{C}$  from the LICS paper as a subcategory of a functor category  $\mathcal{E}_v \rightarrow \mathcal{D}_v$ .
2. Extend  $\mathcal{E}_v$  and  $\mathcal{D}_v$  to reflexive graphs.
3. Define a suitable subcategory the category  $\text{RGFun}(\mathcal{E}, \mathcal{D})$  of reflexive graph functors and give a definition of this category in the style of the LICS paper.
4. Show properties of  $\mathcal{P}_v$ : Cartesian closure, invariant extension, quotienting.
5. Define the category of  $W$ -objects and in particular the objects  $\mathcal{E}_W$  and  $\mathcal{D}_W$ .

6. Define a suitable subcategory of the category  $\mathbf{WFun}(\mathcal{E}_W, \mathcal{D}_W)$  of functor quadruples between  $\mathcal{E}_W$  and  $\mathcal{D}_W$  and give the alternative definition  $\mathcal{P}_e$  in the style of the LICS paper.
7. Show properties of  $\mathcal{P}_e$ , essentially the same as for  $\mathcal{P}_v$ .
8. Define the reflexive graph  $\mathcal{P} = \mathcal{P}_v \rightleftarrows \mathcal{P}_e$  and show interplay between structure on vertices  $\mathcal{P}_v$  and edges  $\mathcal{P}_e$ .
9. Define more concrete  $\mathcal{P}_v$  and  $\mathcal{P}_e$  and show operations in the more concrete model. Show how these operations interact with the structure on the reflexive graph  $\mathcal{P}$ . In particular, show a version of the fundamental theorem of logical relations.

As can be seen from these steps, equivalent descriptions of the same categories are used, namely descriptions as subcategories of functor categories and descriptions in the style of the LICS paper. The reason for this “double development” is that the first kind of description is guided by earlier work on relational reasoning and reflexive graphs, whereas categories in the style of the LICS paper are easier to understand and work with; the double description illustrate that our definitions are not “taken out of thin air”.

### 6.3.1 Alternative description of the LICS model

Here is an alternative description of the category  $\mathcal{C}$  from the LICS paper. First, consider the set  $\mathcal{P}(\text{Heaps})$  of predicates, with the operation  $*$ . It induces a preordered monoid  $(\mathcal{P}(\text{Heaps}), *)$ , where the order is given by  $p \sqsubseteq q$  iff  $q = p * p'$  for some  $p'$ . Let  $\mathcal{E}_v$  be the category induced by this preordered monoid. Also, let  $\mathcal{D}_v$  be the category with objects: pairs  $(o, P)$ , where  $o$  is a pcpo and  $P$  is an admissible per on  $o$ , and morphisms: continuous functions that respect the pers. It is easy to see that  $\mathcal{C}$  is equivalent to the full subcategory of the functor category  $\mathcal{E}_v \Rightarrow \mathcal{D}_v$  to which  $F$  belongs iff  $F(p) = (o, P)$  and  $F(p') = (o', P')$  imply  $o = o'$ , i.e.,  $F$  is “constant on the first component”.

The intuition behind the components in this exposition of the model is as follows. The objects in  $\mathcal{D}_v$  are pers on cpos, which we think of as programs. These pers ensure that “clients cannot distinguish between implementations of a module with a given interface specification” (to use the terminology of the paper [88]). The indexing over predicates builds in the higher-order frame rules; the intuition is that each per corresponds to a predicate that has been added to a type with invariant extension.

### 6.3.2 The Reflexive Graphs $\mathcal{E}$ and $\mathcal{D}$

As mentioned above, we extend the categories  $\mathcal{E}_v$  and  $\mathcal{D}_v$  to reflexive graphs, but with a slightly more general indexing category  $\mathcal{E}_v$ . Let  $\mathcal{E}_v$  and  $\mathcal{E}_e$  be total commutative monoids  $(M_v, \cdot)$ ,  $(M_e, \cdot)$ , both preordered by  $p \sqsubseteq q$  iff  $q = p \cdot p'$  for some  $p'$ . Furthermore, assume we have projections  $\delta_0, \delta_1 : \mathcal{E}_e \rightarrow \mathcal{E}_v$ , and a “diagonal”  $I : \mathcal{E}_v \rightarrow \mathcal{E}_e$ , satisfying  $\delta_0 I = \delta_1 I = \text{Id}_{\mathcal{E}_v}$  and which respect compositions, i.e., for all objects  $p, q \in \mathcal{E}_v$ , and  $r, r' \in \mathcal{E}_e$ ,

$$I(p \cdot q) = I(p) \cdot I(q) \quad \delta_i(r \cdot r') = \delta_i(r) \cdot \delta_i(r'),$$

i.e., we have a reflexive graph of monoids. We often write  $r : p \leftrightarrow q$  for an object  $r \in \mathcal{E}_e$  with  $\delta_0(r) = p$  and  $\delta_1(r) = q$ , and  $\Delta_p$  instead of  $I(p)$  for objects  $p \in \mathcal{E}_v$ .

The reflexive graph  $\mathcal{D}$  is defined as follows. The vertex part  $\mathcal{D}_v$  has

**Objects** pairs  $(o, P)$ , where  $o$  is a pcpo and  $P$  is an admissible per on  $o$ .

**Morphisms**  $f : (o, P) \rightarrow (o', P')$  is a continuous function  $f : o \rightarrow o'$  with the property that  $a[P]a' \Rightarrow f(a)[P]f(a')$  for all  $a, a'$  in  $o$ .

The edge part  $\mathcal{D}_e$  of this reflexive graph is the category with

**Objects** tuples  $(o, P_0, P_1, R)$ , where  $o$  is a pcpo,  $P_i$  are admissible pers on  $o$ , and  $R$  is an admissible and *saturated* (see below) relation (with respect to the  $P_i$ s).

**Morphisms**  $f : (o, P_0, P_1, R) \rightarrow (o', P'_0, P'_1, R')$  are continuous functions  $f : o \rightarrow o'$  with  $f[P_i \rightarrow P'_i]f$  and  $f[R \rightarrow R']f$ .

In general, if  $P, P'$  are pers on a set  $M$ , the relation  $R \subseteq M \times M$  is *saturated* with respect to  $P, P'$  if for all  $a, a', a''$ ,

$$a[R]a' \text{ implies } \begin{cases} - a[P]a \text{ and } a'[P']a', \\ - (a[P]a'' \Rightarrow a''[R]a'), \text{ and} \\ - (a'[P']a'' \Rightarrow a[R]a''). \end{cases}$$

Note that this makes  $R$  a per on  $M$ . This condition can be written more succinctly as  $R = P; R; P'$ , where relation composition is in “diagrammatic” order.

The identity functor  $I : \mathcal{D}_v \rightarrow \mathcal{D}_e$  is defined as  $I(o, P) = (o, P, P, P)$ , and maps morphisms to themselves (it is easy to see that a per is saturated with respect to itself). Projections just pick the appropriate per  $P_i$  on  $o$ . It is easy to see that this indeed defines a reflexive graph  $\mathcal{D} = (\mathcal{D}_v, \mathcal{D}_e)$ .

### 6.3.3 A Functor Category and an Alternative Description

Recall that, given two reflexive graphs  $\mathcal{G}, \mathcal{G}'$ , there is a category  $\text{RGFun}(\mathcal{G}, \mathcal{G}')$  of *reflexive graph functors* between  $\mathcal{G}$  and  $\mathcal{G}'$ . Objects are pairs  $F = (F_v, F_e)$  that make the diagram

$$\begin{array}{ccc} \mathcal{G}_e & \xrightarrow{F_e} & \mathcal{G}'_e \\ \left( \begin{array}{c} \uparrow \\ \downarrow \end{array} \right) & & \left( \begin{array}{c} \uparrow \\ \downarrow \end{array} \right) \\ \mathcal{G}_v & \xrightarrow{F_v} & \mathcal{G}'_v \end{array}$$

commute, and morphisms are pairs of natural transformations making an obvious diagram commute. Here we consider a full subcategory of  $\text{RGFun}(\mathcal{E}, \mathcal{D})$ . The objects are those functors that constant on the cpo part. Here is a description of this category in the style of the LICS paper. First, note that in our case, where the source reflexive graph has a monoidal structure, the vertex part of the functor pairs are redundant; this is captured by the following lemma, which is easy to show.

**Lemma 6.1.** *If  $(F_v, F_e), (G_v, G_e) \in \text{RGFun}(\mathcal{E}, \mathcal{D})$  and  $G_e = F_e$ , then  $G_v = F_v$ .*

Note that this does *not* imply that all functors  $F_e : \mathcal{E}_v \rightarrow \mathcal{D}_v$  give rise to a reflexive graph functor. The conditions for a functor  $F_e : \mathcal{E}_e \rightarrow \mathcal{D}_e$  to be part of a functor pair  $(F_v, F_e) \in \text{RGFun}(\mathcal{E}, \mathcal{D})$  are

1. for all  $p \in \mathcal{E}_v$ ,  $F_e(I_{\mathcal{E}}(p)) = I_{\mathcal{D}}(x)$  for some  $x \in \mathcal{D}_v$ .
2. for all  $r : p \leftrightarrow q \in \mathcal{E}_e$ ,  $F_e(r) : F_e(I_{\mathcal{D}}(p)) \leftrightarrow F_e(I_{\mathcal{D}}(q))$ .

Note that in the first requirement,  $x$  is unambiguous, and this makes it possible to define  $F_v$  on objects:  $F_v(p) = x$  such that  $F_e(I_{\mathcal{E}}(p)) = I_{\mathcal{D}}(x)$ . This description of the functors in question is used to give the alternative definition. Hence,  $\mathcal{P}_v$  is the category with

**Objects** pairs  $(o, R)$ , where  $o$  is a pcpo, and  $R$  is a family of triples  $R(r : p \leftrightarrow q) = (R(r), P_0(p), P_1(q))$ <sup>1</sup>, indexed by objects  $r \in \mathcal{E}_v$ , and such that  $P_0(p), P_1(q)$  are admissible pers on  $o$ , and  $R(r)$  is an admissible per on  $o$ , saturated with respect to  $P_0(p)$  and  $P_1(q)$ . This family must satisfy

- $R(I_{\mathcal{E}}(p))$  is the identity saturated relation with respect to  $P_0(p)$ , i.e., the three components of  $R(I_{\mathcal{E}}(p))$  are the same.
- $R(r : p \leftrightarrow q)$  is a saturated relation between  $R(I_{\mathcal{E}}(p))$  and  $R(I_{\mathcal{E}}(q))$ .
- for all  $r, r', R(r) \subseteq R(r * r')$ .

**Morphisms**  $f : (o, R) \rightarrow (o', R')$  is a continuous map  $f : o \rightarrow o'$  with the property that for all  $r : p \leftrightarrow q$ ,

$$a[R(r)]a' \text{ implies } f(a)[R'(r)]f(a').$$

It is easy to see that  $\mathcal{P}_v$  is equivalent to the subcategory of  $\text{RGFun}(\mathcal{E}, \mathcal{D})$  described above. The category  $\mathcal{P}_v$  has much of the same structure as the category  $\mathcal{C}$  from the LICS paper, as we shall see.

### 6.3.4 Properties of $\mathcal{P}_v$

A lot of the properties of the category  $\mathcal{C}$  also hold in  $\mathcal{P}_v$ . In particular,  $\mathcal{P}_v$  is cartesian closed and we can define an invariant extension and a quotienting with the same properties as in the LICS paper.

**Exponentials in  $\mathcal{P}_v$**  The category  $\mathcal{P}_v$  defined above is a cartesian closed category. Indeed, the terminal object and products are defined as expected, so we focus on exponentials. Given objects  $(o, R), (o', R')$ , define  $(o, R) \Rightarrow (o', R')$  as  $(o \Rightarrow o', R \Rightarrow R')$ , where  $o \Rightarrow o'$  is the usual exponential of pointed cpos. The family of saturated relations on this pcpo has components

$$(R \Rightarrow R')(r : p \leftrightarrow q) = ((R \Rightarrow R')(r), (P_0 \Rightarrow P_0)(p), (P_1 \Rightarrow P_1)(q)),$$

which is called  $(S, S_0, S_1)$  for brevity. For reference, assume that the two given objects have  $(R(r), P_0(p), P_1(q))$  and  $(R'(r), P'_0(p), P'_1(q))$ , respectively, as their components.

First, we focus on the definition of the per  $S_0$  on  $o \Rightarrow o'$ . This partial equivalence relation is defined by

$$f[S_0]f' \text{ iff for all } r' : p' \leftrightarrow q'. f[R(\Delta_p \cdot r') \rightarrow R'(\Delta_p \cdot r)]f'$$

It is easy to see that this is a per, and that it is admissible, given that the two given objects have these properties.

---

<sup>1</sup>We use the letter  $R$  for the whole family as well as for the first member of each of its components. Although messy, this should not cause confusion.

Next, we define the relation  $S \subseteq (o \Rightarrow o') \times (o \Rightarrow o')$  by

$$f[S]f' \text{ iff } \begin{cases} - f[S_0]f, f'[S_1]f', \text{ and} \\ - \forall r' : p' \leftrightarrow q'. f[R(r \cdot r') \rightarrow R'(r \cdot r')]f'. \end{cases}$$

Again, it is easy to see that this is admissible and saturated. It is also not hard to see that this indeed defines an object in the category  $\mathcal{P}_v$ . To see that it is the exponential, it suffices to show that the evaluation map  $\varepsilon : o \times (o \Rightarrow o') \rightarrow o'$  from the category of pcpos is indeed a morphism in  $\mathcal{P}_v$ . Hence, we show that for all  $r : p \leftrightarrow q$ ,

$$\varepsilon[R(r) \times (R \Rightarrow R')(r) \rightarrow R'(r)]\varepsilon.$$

If  $(a, f)[R(r) \times (R \Rightarrow R')(r)](a', f')$ , it is clear that

$$\varepsilon(f, a) = f(a)[R'(r)]f'(a') = \varepsilon(f', a'),$$

as desired. This means that

**Lemma 6.2.** *The category  $\mathcal{P}_v$  is a cartesian closed category.*

**Invariant Extension in  $\mathcal{P}_v$**  Just as in the LICS paper, there is an “invariant extension functor” which corresponds to the type constructor  $\otimes$ . Define the functor  $\text{inv}_v : \mathcal{P}_v \times \mathcal{E}_v \rightarrow \mathcal{P}_v$  by

$$\text{inv}_v((o, R), p_0) = (o, R(\Delta_{p_0} \cdot -)) \quad \text{and} \quad \text{inv}_v(f, p \sqsubseteq p \cdot q) = f.$$

Just as in the LICS paper,

**Lemma 6.3.** *For each  $p \in \mathcal{E}_v$ , the functor  $\text{inv}_v(-, p) : \mathcal{P}_v \rightarrow \mathcal{P}_v$  preserves the cartesian closed structure of  $\mathcal{P}_v$  on the nose.*

This is not hard to show. Furthermore, by definition,

**Lemma 6.4.** *Let  $p, q$  be elements in  $\mathcal{E}_v$ . Then,*

$$\text{inv}_v(-, q) \circ \text{inv}_v(-, p) = \text{inv}_v(-, p \cdot q).$$

**Quotienting in  $\mathcal{P}_v$**  The quotienting from the LICS paper works in this setting as well. The equivalence relation is defined as follows: for two morphisms  $f, g : (o, R) \rightarrow (o', R')$ ,

$$f \sim g \quad \text{iff} \quad f[R(r) \rightarrow R'(r)]g \text{ for all } r : p \leftrightarrow q.$$

This relation is reflexive due to the requirements for morphisms, and it is symmetric and transitive because both  $R$  and  $R'$  are. The preservation in the LICS paper also holds here.

**Lemma 6.5.** *The relation  $\sim$  is preserved by the following operators in  $\mathcal{P}_v$ :*

- *compositions of morphisms;*
- *currying of morphisms;*
- *pairing into small products; and*
- *The functor  $\text{inv}_v(-, p \sqsubseteq q)$  on  $\mathcal{P}_v$  for all predicates with  $p \sqsubseteq q$ .*

*Proof.* Composition and pairing are straightforward to verify, just like  $\text{inv}_v(-, p \sqsubseteq q)$ , since its action on morphisms is the identity, so we focus on currying. Let

$$f \sim g : (o, R) \times (o', R') \rightarrow (o'', R''),$$

and let  $\tilde{f}, \tilde{g} : (o, R) \rightarrow ((o', R') \Rightarrow (o'', R''))$  be the corresponding curried morphisms. We check that  $\tilde{f} \sim \tilde{g}$ . Let  $a[R(r)]b$ ; it is then our task to show

$$\tilde{f}(a)[(R \Rightarrow R')(r)]\tilde{g}(b).$$

The three conditions required to verify this are all checked with a diagram like that below, where  $h$  is replaced by  $f$  or  $g$ , where appropriate.

$$\begin{array}{ccc} a' & \xrightarrow{\tilde{h}} & a'' = (\tilde{h}(a))(a') = h(a, a') \\ R'(-) \uparrow & & \downarrow R''(-) \\ b' & \xrightarrow{\tilde{h}} & b'' = (\tilde{h}(a))(a') = h(a, a') \end{array}$$

This shows the desired result.  $\square$

### 6.3.5 $W$ -objects

As briefly described in Section 6.2, relationships between types are interpreted as relations between the underlying interpretations; this is spelled out here.

First, a  $W$ -object consists of five categories and eight functors as in the diagram

$$\begin{array}{ccccc} \mathcal{C}_0 & & \mathcal{C}_4 & & \mathcal{C}_2 \\ \delta_1 \curvearrowright & & \delta_0 & & \delta_0 \curvearrowleft \\ \text{Id} \searrow & & & & \nearrow \text{Id} \\ \delta_0 \curvearrowright & & & & \delta_1 \curvearrowleft \\ \mathcal{C}_1 & & \mathcal{C}_3 & & \end{array} \quad (6.3)$$

such that the “wings” on the left and right are reflexive graphs. An example can be obtained by taking the two wings to be the reflexive graph  $\mathcal{E}$  from before, the middle category to be  $\mathcal{E}_e$ , and the two middle functors to be the same projections as in the reflexive graph  $\mathcal{E}$ . Call this  $W$ -object  $\mathcal{E}_W$ . Another example of a  $W$ -object used subsequently, requires yet another category. Define  $\mathcal{D}'_e$  to be the category with

**Objects** Tuples  $(o, P, o', P', R)$ , where

- $(o, P), (o', P')$  are objects in  $\mathcal{D}_v$ , and
- $R \subseteq (o \times o')$  is a relation that is saturated with respect to  $P, P'$ .

**Morphisms**  $(o_0, P_0, o'_0, P'_0, R_0) \rightarrow (o_1, P_1, o'_1, P'_1, R_1)$  are pairs

$$(f : (o_0, P_0) \rightarrow (o_1, P_1), f' : (o'_1, P'_1) \rightarrow (o'_0, P'_0))$$

of maps in  $\mathcal{D}_v$  with the property that

$$a[R_0]a' \text{ implies } f(a)[R_1(r)]f'(a').$$

The  $W$ -object  $\mathcal{D}_W$  has the reflexive graph  $\mathcal{D}$  in its “wings”, the category  $\mathcal{D}'_e$  in the middle, and the obvious projections as the two middle functors. There is a category  $\text{WFun}(\mathcal{E}_W, \mathcal{D}_W)$  with  $W$ -objects and quadruples of functors making the obvious diagram commute. For such functor quadruples, note that the “middle” functor only determines the relation part; that is, if

$$F(r : p \leftrightarrow q) = (o, P, o', P', R),$$

the cpos and relations  $o, o', P, P'$  are determined by the other functors' actions on  $r$ .

**Remark 6.6.** As a side-remark, let us consider the category  $\text{RGFun}(\mathcal{E}, \mathcal{D})$  and the category  $\text{WFun}(\mathcal{E}_w, \mathcal{D}_W)$  of functor quadruples. There are projections

$$\delta_i : \text{WFun}(\mathcal{E}_w, \mathcal{D}_W) \rightarrow \text{RGFun}(\mathcal{E}, \mathcal{D}), i = 0, 1,$$

that take a functor quadruple to the functor pairs between the “wings”, and a functor

$$I : \text{RGFun}(\mathcal{E}, \mathcal{D}) \rightarrow \text{WFun}(\mathcal{E}_W, \mathcal{D}_W),$$

which takes a functor pair to the quadruple of functors where both of the “wings” are the pair we started with, and the middle functor is the edge part of the functor pair. It is easy to see that this defines a reflexive graph. In fact, this is the exponential of the objects  $\mathcal{E}$  and  $\mathcal{D}$  in the category of reflexive graphs and reflexive graph functors. One can show that both  $\text{RGFun}(\mathcal{E}, \mathcal{D})$  and  $\text{WFun}(\mathcal{E}_W, \mathcal{D}_W)$  are cartesian closed and that  $\delta_0, \delta_1$ , and  $I$  preserve the ccc-structure.

### 6.3.6 The Category $\mathcal{P}_e$

We consider a subcategory of the category  $\text{WFun}(\mathcal{E}_W, \mathcal{D}_W)$  with functor-quadruples between  $\mathcal{E}_W$  and  $\mathcal{D}_W$  as objects, and quadruples of natural transformations between these functors as morphisms. Following the lines of the construction of the category  $\mathcal{P}_v$ , this subcategory consists of functors where the two functors between  $\mathcal{E}_v$  and  $\mathcal{D}_v$  are constant (but not necessarily the same) on the pcpo parts.

Again, we present an alternative description of this functor category, in the style of the LICS paper. Intuitively, an object in this category consists of two objects from  $\mathcal{P}_v$  and a (saturated) relation between them. More formally, define the category  $\mathcal{P}_e$  which has

**Objects** tuples  $(o, R, o', R', \mathcal{R})$ , where

1.  $(o, R)$  and  $(o', R')$  are objects in  $\mathcal{P}_v$ .
2.  $\mathcal{R}$  is a family, indexed by objects of the category  $\mathcal{E}_e$ , of saturated relations, such that  $\mathcal{R}(r : p \leftrightarrow q)$  is a saturated relation between the pers  $R(I_{\mathcal{E}}(p))$  and  $R'(I_{\mathcal{E}}(q))$ .
3.  $\mathcal{R}(r) \subseteq \mathcal{R}(r \cdot r')$  for all  $r, r'$  in  $\mathcal{E}_e$ .

**Morphisms**  $(o_0, R_0, o'_0, R'_0, \mathcal{R}_0) \rightarrow (o_1, R_1, o'_1, R'_1, \mathcal{R}_1)$  are pairs

$$(f : (o_0, R_0) \rightarrow (o_1, R_1), f' : (o'_0, R'_0) \rightarrow (o'_1, R'_1))$$

of morphisms in  $\mathcal{P}_v$  such that for all  $r : p \leftrightarrow q$  in  $\mathcal{E}_e$ ,

$$f[\mathcal{R}_0(r) \rightarrow \mathcal{R}_1(r)]f'.$$

The family  $\mathcal{R}$  of saturated relations in this definition plays the role of the saturated relation between two objects of  $\mathcal{P}_v$  that was mentioned before the definition.

### 6.3.7 Properties of $\mathcal{P}_e$

Many of the properties of  $\mathcal{P}_v$  are also enjoyed by  $\mathcal{P}_e$ , as we show here.

#### Exponentials in $\mathcal{P}_e$

**Lemma 6.7.** *The category  $\mathcal{P}_e$  is a cartesian closed category.*

*Proof.* Again, the focus is on exponentials. Given two objects

$$(o_0, R_0, o'_0, R'_0, \mathcal{R}_0) \quad \text{and} \quad (o_1, R_1, o'_1, R'_1, \mathcal{R}_1)$$

in  $\mathcal{P}_e$ , define the exponential  $(o_0, R_0, o'_0, R'_0, \mathcal{R}_0) \Rightarrow (o_1, R_1, o'_1, R'_1, \mathcal{R}_1)$  by

$$(o_0 \Rightarrow o_1, R_0 \Rightarrow R_1, o'_0 \Rightarrow o'_1, R'_0 \Rightarrow R'_1, \mathcal{R}_0 \Rightarrow \mathcal{R}_1),$$

where

- $(o_0 \Rightarrow o_1, R_0 \Rightarrow R_1)$  and  $(o'_0 \Rightarrow o'_1, R'_0 \Rightarrow R'_1)$  are the exponentials in  $\mathcal{P}_v$  of  $(o_0, R_0)$  and  $(o_1, R_1)$ , and  $(o'_0, R'_0)$  and  $(o'_1, R'_1)$ , respectively, and
- $(\mathcal{R}_0 \Rightarrow \mathcal{R}_1)(r : p \leftrightarrow q) \subseteq (o_0 \Rightarrow o_1) \times (o'_0 \Rightarrow o'_1)$  is defined by

$$f[(\mathcal{R}_0 \Rightarrow \mathcal{R}_1)(r)]f' \text{ iff } \begin{cases} - f[(R_0 \Rightarrow R_1)(\Delta_p)]f', \\ - f'[(R'_0 \Rightarrow R'_1)(\Delta_q)]f' \text{ and} \\ - f[\mathcal{R}_0(r \cdot r') \rightarrow \mathcal{R}_1(r \cdot r')]f' \text{ for all } r' : p' \leftrightarrow q' \in \mathcal{E}_e. \end{cases}$$

This definition of an exponential indeed defines an object in  $\mathcal{P}_e$ . To see that the relation  $(\mathcal{R}_0 \Rightarrow \mathcal{R}_1)(r : p \leftrightarrow q)$  is a saturated relation with respect to  $(R_0 \Rightarrow R_1)(\Delta_p)$  and  $(R'_0 \Rightarrow R'_1)(\Delta_q)$ , suppose  $f[(\mathcal{R}_0 \Rightarrow \mathcal{R}_1)(r)]f'$  and  $f'[(R'_0 \Rightarrow R'_1)(\Delta_q)]f''$ . We then have to show that  $f[(\mathcal{R}_0 \Rightarrow \mathcal{R}_1)(r)]f''$ . Clearly,  $f[(R_0 \Rightarrow R_1)(\Delta_p)]f'$ , and  $f'[(R'_0 \Rightarrow R'_1)(\Delta_q)]f''$ , since this is a symmetric relation. So, let  $r' : p' \leftrightarrow q'$  be an object in  $\mathcal{E}_e$ , and suppose  $a[\mathcal{R}_0(r \cdot r')]a'$ . Since  $\mathcal{R}_0(r \cdot r')$  is saturated with respect to  $R'_0(\Delta_{q \cdot q'})$ ,  $a'[\mathcal{R}_0(\Delta_{q \cdot q'})]a'$ . This implies  $f(a)[\mathcal{R}_1(r \cdot r')]f'(a')$ , and also that  $f'(a')[\mathcal{R}_1(\Delta_{q \cdot q'})]f''(a')$ , which in turn entails  $f(a)[\mathcal{R}_1(r \cdot r')]f''(a')$ , since  $\mathcal{R}_1$  is saturated. It is clear that  $\mathcal{R}$  is monotone, and that the pair

$$(\varepsilon, \varepsilon') : (((o_0 \Rightarrow o'_0) \times o_0) \times ((o_1 \Rightarrow o'_1) \times o_1) \rightarrow (o'_0 \times o'_1))$$

is a morphism. □

**Invariant Extension in  $\mathcal{P}_e$**  The category  $\mathcal{P}_e$  has an invariant extension functor. Whereas the invariant extension in  $\mathcal{P}_v$  extended objects with an object in  $\mathcal{E}_v$ , invariant extension in  $\mathcal{P}_e$  extends objects with an object in  $\mathcal{E}_e$  (which can be seen as a relation between two objects in  $\mathcal{E}_v$ ). More formally, define the functor  $\text{inv}_e : \mathcal{P}_e \times \mathcal{E}_e \rightarrow \mathcal{P}_e$  by

$$\begin{aligned} \text{inv}_e((o, R, o', R', \mathcal{R}), r : p \leftrightarrow q) &= (o, R(\Delta_p \cdot -), o', R'(\Delta_q \cdot -), \mathcal{R}(r \cdot -)) \\ \text{inv}_e((f, f'), r \sqsubseteq r') &= (f, f') \end{aligned}$$

Notice how  $\text{inv}_e$  has the same action as  $\text{inv}_v$  on the two components  $(o, R)$  and  $(o', R')$  that are objects in  $\mathcal{P}_v$ . Again, invariant extension commutes with composition and exponentials, this is captured by the two following lemmas.

**Lemma 6.8.** *Let  $r$  and  $r'$  be elements of  $\mathcal{E}_e$ . Then,*

$$\text{inv}_v(-, r) \circ \text{inv}_v(-, r') = \text{inv}_v(-, r \cdot r')$$

**Lemma 6.9.** *For an object  $r : p \leftrightarrow q$  in  $\mathcal{E}_e$  the functor  $\text{inv}_e(r, -)$  preserves the cartesian closed structure of  $\mathcal{P}_e$  on the nose.*

### 6.3.8 The Reflexive Graph $\mathcal{P}$

Informally, every object in  $\mathcal{P}_e$  contains two objects from  $\mathcal{P}_v$ , and is a relation between them. This intuition is made explicit here. First, note that there are functors  $\delta_i : \mathcal{P}_e \rightarrow \mathcal{P}_v$  defined by

$$\delta_i(o_0, R_0, o_1, R_1, \mathcal{R}) = (o_i, R_i) \quad \text{for } i = 0, 1,$$

and a functor  $I : \mathcal{P}_v \rightarrow \mathcal{P}_e$ , defined by

$$I(o, R) = (o, R, o, R, R).$$

These functors clearly satisfy  $\delta_i \circ I = \text{Id}$  for  $i = 0, 1$ , so there is a reflexive graph [86, 40]  $\mathcal{P}$  with  $\mathcal{P}_v$  as the vertex graph and  $\mathcal{P}_e$  as the edge graph.

**Lemma 6.10.** *The projections and the diagonal functor in the reflexive graph  $\mathcal{P}$  preserve the cartesian closed structure of  $\mathcal{P}_v$  and  $\mathcal{P}_e$ .*

*Proof.* Again, we focus on the exponential part. We show that given two objects

$$(o_0, R_0, o'_0, R'_0, \mathcal{R}_0) \quad \text{and} \quad (o_1, R_1, o'_1, R'_1, \mathcal{R}_1)$$

in  $\mathcal{P}_e$  and two objects  $(o, R), (o', R')$  in  $\mathcal{P}_v$ ,

$$\begin{aligned} \delta_i((o_0, R_0, o'_0, R'_0, \mathcal{R}_0) \Rightarrow (o_1, R_1, o'_1, R'_1, \mathcal{R}_1)) &= \\ \delta_i((o_0, R_0, o'_0, R'_0, \mathcal{R}_0)) \Rightarrow \delta_i((o_1, R_1, o'_1, R'_1, \mathcal{R}_1)) &\text{ and} \\ I((o, R) \Rightarrow (o', R')) = I(o, R) \Rightarrow I(o', R'). \end{aligned}$$

This is an easy unwinding of definitions. □

Note the connection between Lemma 6.10 and Remark 6.6. The two results are *not* the same, since  $\mathcal{P}_v$  corresponds to only a subset of all reflexive graph functors (those that are constant on the cpo part).

There is an interplay between the functors  $\text{inv}$  and  $I$ , captured by

**Lemma 6.11.** *Given an object  $p \in \mathcal{E}_v$ ,*

$$I(\text{inv}_v(-, p)) = \text{inv}_e(I(-), \Delta_p) : \mathcal{P}_v \rightarrow \mathcal{P}_e.$$

### 6.3.9 The Heap Model

To move closer to the model in the LICS paper, we require extra properties of the “world reflexive graph”  $\mathcal{E}$ , used so far. The new reflexive graph has pers on heaps as vertices and saturated relations between pers as edges. Formally, recall first the  $*$  operation on relations from the paper [107], defined by

$$h[p * p']h' \text{ iff } \exists h_0, h_1, h'_0, h'_1. h = h_0 * h_1 \wedge h' = h'_0 * h'_1 \wedge h_0[p]h'_0 \wedge h_1[p']h'_1. \quad (6.4)$$

As for predicates, there is a notion of preciseness for relations. Recall that a subset  $M \subseteq \text{Heaps}$  is called *precise*, if for all heaps  $h$ , there is at most one subheap  $h' \subseteq h$  with  $h \in M$ . A per  $p \subseteq \text{Heaps} \times \text{Heaps}$  is called *precise*, if both projections of  $M$  onto  $\text{Heaps}$  are precise subsets. We restrict to precise pers in order for the  $*$  operation to be well-defined; it is not enough that  $p, p'$  are pers in order for the relation defined in (6.4) to be transitive. Possibly, there is a smaller class of pers that makes  $*$  well-defined as an operation on pers, but for now we stick to precise pers. When we write “per” in the rest of the paper, we implicitly mean “precise per”. Let  $\text{Per}(\text{Heaps})$  be the set of pers on heaps; then it is easy to see that  $(\text{Per}(\text{Heaps}), *)$  is a commutative monoid, with preorders induced both by  $*$  and by subset inclusion. Furthermore, define the monoid  $\mathcal{E}'_e$  as follows: elements are of the form  $r : p \leftrightarrow q$ , where  $p$  and  $q$  are elements of  $\mathcal{E}_v$  and  $r$  is a relation on heaps that is saturated with respect to  $p$  and  $q$ . It is easy to see (using preciseness) that if  $r$  and  $r'$  are saturated with respect to  $p, q$  and  $p', q'$ , respectively, then  $r * r'$  is saturated with respect to  $p * p'$  and  $q * q'$ . Since any per is saturated with respect to itself, there are obvious functors  $I : \mathcal{E}'_v \rightarrow \mathcal{E}_e, \delta_0, \delta_1 : \mathcal{E}_e \rightarrow \mathcal{E}_v$  that make this a reflexive graph  $\mathcal{E}'$ , which is an instance of the reflexive graph  $\mathcal{E}$  from the beginning of this section. Thus, all the analysis from before is still valid if  $\mathcal{E}$  is replaced by  $\mathcal{E}'$ . In the rest of this paper, we use the “version” of  $\mathcal{P}$  that arises from using  $\mathcal{E}'$  instead of  $\mathcal{E}$ . We shall sometimes view  $\mathcal{E}_v$  as a preordered category (ordered by inclusion) as mentioned above; when doing so, it is named  $\mathcal{P}_r$ . Note that  $p * -$  respects the subset ordering; in other words, it is an endofunctor on  $\mathcal{P}_r$ .

Note that, in contrast to the LICS paper, where the “indexing category”  $\mathcal{E}_v$  consisted of predicates on heaps, we index over pers here. The reason is that the indexing of the LICS paper causes problems with parametricity, as we briefly explain here. Consider the specification for **cons**

$$\{\text{emp}\} x := \mathbf{cons}() \{x \mapsto -\} \quad (6.5)$$

If we lift this specification to a quadruple in the expected way, we get

$$\{\Delta_{\text{emp}}\} \begin{array}{l} x := \mathbf{cons}() \\ x := \mathbf{cons}() \end{array} \{\Delta_{x \mapsto -}\},$$

where  $h$  and  $h'$  are related by  $\Delta_p$  iff  $h = h' \in P$ . This quadruple does not satisfy the binary version of the frame rule. Indeed, consider the relation  $R$  which relates the singleton heap  $[2 \mapsto 3]$  and the empty heap only. The binary version of the frame rule then gives:

$$\{\Delta_{\text{emp}} * R\} \begin{array}{l} x := \mathbf{cons}() \\ x := \mathbf{cons}() \end{array} \{\Delta_{x \mapsto -} * R\},$$

and this does *not* hold, since one of the computations can set  $x$  to 2, whereas the second cannot. Our interpretation of the specification (6.5) should reflect the nondeterministic nature of **cons**; all the possible values chosen by **cons** should in some sense be “equal”. In other words, we need to allow  $\alpha$ -renaming, in the sense of the article [33], in our interpretation of assertions, and therefore we use pers rather than predicates on heaps as the indexing category  $\mathcal{E}_v$ .

### Triple Objects

Let  $p_0, q_0$  be precise pers on heaps. We define an object  $[p_0, q_0]$  in  $\mathcal{P}_v$  which is to be the interpretation of the type  $\{p_0\} - \{q_0\}$ . The underlying pcpo comm is the usual set of

commands, that is, functions  $c$  from  $H$  to  $\mathcal{P}(H \cup \{wrong\})$  that satisfy safety monotonicity and the frame property [25]. The order is given by pointwise inclusion, and it is easy to see that this constitutes a pcpo. Next, define the required family of relations on this pcpo as follows. Let  $r : p \leftrightarrow q$  be a saturated relation with respect to the pers  $p, q$ . For brevity, set  $[p_0, q_0](r) = (S, S_0, S_1)$ . The per  $S_0$  is defined as follows. For  $c, c'$  in comm,

$$\begin{aligned} c[S_0]c' \text{ iff for all saturated relations } r' : p' \leftrightarrow q' \text{ between some pers} \\ p' \text{ and } q' \text{ on heaps, and all heaps } h, h', \\ h[p_0 * p * r']h' \text{ implies } wrong \notin c(h) \cup c'(h'), \text{ and} \\ - \forall h_0 \in c(h). \exists h'_0 \in c'(h'). h_0[q_0 * p * r']h'_0 \\ - \forall h_1 \in c'(h'). \exists h'_1 \in c(h). h'_1[q_0 * p * r']h_1, \end{aligned}$$

The per  $S_1$  is defined analogously, and the saturated relation  $S$  is defined by

$$\begin{aligned} c[S]c' \text{ iff } c[S_0]c, c'[S_1]c', \text{ and} \\ \text{for all saturated relations } r' : p' \leftrightarrow q' \text{ between some pers} \\ p' \text{ and } q' \text{ on heaps, and all heaps } h, h', \\ h[p_0 * r * r']h' \text{ implies } wrong \notin c(h) \cup c'(h'), \text{ and} \\ - \forall h_0 \in c(h). \exists h'_0 \in c'(h'). h_0[q_0 * r * r']h'_0 \\ - \forall h_1 \in c'(h'). \exists h'_1 \in c(h). h'_1[q_0 * r * r']h_1. \end{aligned}$$

Note that the two clauses  $c[S_0]c \wedge c'[S_1]c'$  are *not* redundant.

To see that this is a saturated relation, suppose  $c[S]c'$  and  $c[S_0]c''$ , so that  $c''[S_0]c''$ . Let  $r' : p' \leftrightarrow q'$  be a saturated relation, and suppose  $h[p_0 * r * r']h'$ , which implies  $h[p_0 * p * p']h$ . Now, let  $h'' \in c''(h)$ ; then there exists  $h'_0$  with  $h_0[q_0 * p * p']h'_0$ , and since  $c[S]c'$ , there is a  $h''_0$  with  $h''_0[q_0 * r * r']h''_0$ . Since this is a saturated relation,

$$h_0[q_0 * r * r']h''_0,$$

as desired. The two conditions needed to verify that  $[p_0, q_0]$  is indeed an object of  $\mathcal{P}_v$ , namely preservation of identities and monotonicity, are easy to check.

As in the LICS paper, the triple object generator can be extended to a functor  $\text{tri}$ . Define the functor  $\text{tri} : \mathcal{P}_r^{\text{op}} \times \mathcal{P}_r \rightarrow \mathcal{P}_v$  by

$$\text{tri}(p, q) = [p, q] \quad \text{and} \quad \text{tri}(p \subseteq p', q' \subseteq q) = id.$$

To see that this is indeed a functor, we verify that the identity is a morphism. Suppose  $q_0 \subseteq q'_0$ , i.e.,  $h[q_0]h' \Rightarrow h[q'_0]h'$  for all  $h, h'$ ; we then need to show  $id : [p_0, q_0] \rightarrow [p_0, q'_0]$ . For all saturated relations  $r : p \leftrightarrow q, r' : p' \leftrightarrow q'$ , we therefore check that if  $h[p_0 * r * r']h'$ , then  $wrong \notin c(h) \cup c'(h')$  and for all  $h_0 \in c(h)$ , there is  $h'_0 \in c'(h')$  s.t.  $h_0[q_0 * r * r']h'_0$ . This is easy.

This functor's interplay with invariant extension is captured by the following lemma.

**Lemma 6.12.** *If  $p$  is a per on heaps,*

$$\text{inv}_v(-, p) \circ \text{tri} = \text{tri}(- * p, - * p).$$

*Proof.* For a per  $p_1$  on heaps, and all saturated relations  $r : p \leftrightarrow q$ , we prove

$$[p_0 * p_1, q_0 * p_1](r) = ((p_1 * -)([p_0, q_0]))(r) (= [p_0, q_0](\Delta_p * r)).$$

Set  $(S, S_0, S_1) = [p_0, q_0](\Delta_p * r)$  and  $(T, T_0, T_1) = [p_0 * p_1, q_0 * p_1](r)$ . We then need to show, for example, that for all commands  $c, c'$ ,

$$\begin{aligned} c[S_0]c' &\text{ iff } c[T_0]c', \text{ and} \\ c[S]c' &\text{ iff } c[T]c'. \end{aligned}$$

This, too, is a simple matter of unwinding definitions.  $\square$

In addition, define an object  $[p_0, q_0]$  in  $\mathcal{P}_v$  corresponding to the total correctness specification  $[p_0] - [q_0]$ . This object has  $\text{comm}$  as its cpo-part, and for a saturated relation  $r : p \leftrightarrow q$ , define  $(T, T_0, T_1) = [p_0, q_0](r)$  as follows. For  $c, c' \in \text{comm}$ ,

$$\begin{aligned} c[T_0]c' &\text{ iff for all saturated relations } r' : p' \leftrightarrow q' \text{ between some pers} \\ & p' \text{ and } q' \text{ on heaps, and all heaps } h, h', \\ & h[p_0 * p * r']h' \text{ implies } c(h) \neq \emptyset, c'(h') \neq \emptyset, \text{ wrong} \notin c(h) \cup c'(h'), \text{ and} \\ & - \forall h_0 \in c(h). \exists h'_0 \in c'(h'). h_0[q_0 * p * r']h'_0 \\ & - \forall h_1 \in c'(h'). \exists h'_1 \in c(h). h'_1[q_0 * p * r']h_1, \end{aligned}$$

and

$$\begin{aligned} c[T]c' &\text{ iff } c[S_0]c, c'[S_1]c', \text{ and} \\ & \text{for all saturated relations } r' : p' \leftrightarrow q' \text{ between some pers} \\ & p' \text{ and } q' \text{ on heaps, and all heaps } h, h', \\ & h[p_0 * r * r']h' \text{ implies } c(h) \neq \emptyset, c'(h') \neq \emptyset, \text{ wrong} \notin c(h) \cup c'(h'), \text{ and} \\ & - \forall h_0 \in c(h). \exists h'_0 \in c'(h'). h_0[q_0 * r * r']h'_0 \\ & - \forall h_1 \in c'(h'). \exists h'_1 \in c(h). h'_1[q_0 * r * r']h_1. \end{aligned}$$

This generator extends to a functor  $\text{ttri}$  in the same way as  $\text{tri}$ . Note that the identity is a morphism from  $[p_0, q_0]$  to  $[p_0, q_0]$ , corresponding to the fact that total correctness implies partial correctness.

### Some Morphisms

The interpretation of terms in the LICS paper is standard, except for the imperative constructions. The same constructions exist in our model.

The definitions are the same as in the LICS paper.

$$\begin{aligned} \text{seq}_{p,q,q'} & : (\text{ttri}(p, q) \times \text{ttri}(q, q')) \rightarrow \text{ttri}(p, q') \\ \text{seq}_{p,q,q'} & \equiv \lambda(c, c'). \{ \text{wrong} \mid \text{wrong} \in c(h) \} \cup (\bigcup \{ c'(h') \mid h' \in c(h) \}) \\ \text{read}_{n,p_0,q_0} & : \prod_{m \in \text{Int}} ((\text{ttri}(p_0 * n \mapsto m, q_0)) \rightarrow \text{ttri}(p_0 * m \mapsto -, q_0)) \\ \text{read}_{n,p_0,q_0} & \equiv \lambda c. \lambda h. \text{if } n \in \text{dom}(h) \text{ then } c_{h(n)}(h) \text{ else } \text{wrong} \\ \text{free}(n) & : 1 \rightarrow ((\text{ttri}(n \mapsto -, \text{emp})) \\ \text{free}(n) & \equiv \lambda \cdot . \lambda h. \text{if } n \in \text{dom}(h) \text{ then } \{ h[n \rightarrow \text{undef}] \} \text{ else } \{ \text{wrong} \} \\ \text{write}(n, n') & : 1 \rightarrow (\text{ttri}(n \mapsto -, n \mapsto n')) \\ \text{write}(n, n') & \equiv \lambda \cdot \lambda h. \text{if } n \in \text{dom}(h) \text{ then } \{ h[n \mapsto n'] \} \text{ else } \{ \text{wrong} \} \end{aligned}$$

The challenge, of course, is to show that these are indeed morphisms in  $\mathcal{P}_v$ . This is fairly straightforward.

### Quadruples

As mentioned in Section 6.1, the meaning of a quadruple (6.1) is that all pairs of states related by  $p$  are taken to pairs of states by  $c_0, c_1$  that are related by  $p'$ . There are more conditions, but we pursue the idea of pairs of programs that take related arguments to related results, in the spirit of logical relations [101, 71]. A relation in the setting of relational separation logic [135] can be seen as a relation between two predicates (sets of heaps). Since we use pers on heaps rather than predicates, we use saturated relations between pers in our version of quadruples rather than relations.

More concretely, given two objects  $r_0 : p_0 \leftrightarrow q_0$  and  $r_1 : p_1 \leftrightarrow q_1$  in  $\mathcal{E}_e$ , construct the quadruple object  $[r_0, r_1]$ , given by

$$[r_0, r_1] = (\text{comm}, R_0, \text{comm}, R_1, \mathcal{R}),$$

where

- $(\text{comm}, R_0)$  and  $(\text{comm}, R_1)$  are the triple objects  $[p_0, p_1]$  and  $[q_0, q_1]$ , respectively.
- $\mathcal{R}$  is the family (indexed by saturated relations on heaps) of relations on  $\text{comm}$  given by (for a saturated relation  $r$  between pers  $p, q$  on heaps)

$$\begin{aligned} c[\mathcal{R}(r : p \leftrightarrow q)]c' \text{ iff } & c[R_0(\Delta_p)]c, c'[R_1(\Delta_q)]c', \text{ and} \\ & \text{for all saturated relations } r' : p' \leftrightarrow q' \text{ between pers } p \text{ and } q, \\ & h[r * r' * r_0]h' \text{ implies } \text{wrong} \notin c(h) \cup c'(h') \text{ and} \\ & - \forall h_0 \in c(h). \exists h'_0 \in c'(h'). h_0[r * r' * r_1]h'_0 \\ & - \forall h_0 \in c'(h'). \exists h'_0 \in c(h). h'_0[r * r' * r_1]h_0. \end{aligned}$$

First, we check that this indeed defines an object in  $\mathcal{P}_e$ ; we have already done most of the work, and functoriality of  $\mathcal{R}$  follows from the quantification over all saturated relations. The only task left is to check that  $\mathcal{R}(r : p \leftrightarrow q)$  is saturated with respect to the pers  $R_0(\Delta_p)$  and  $R_1(\Delta_q)$ . Assume  $c[\mathcal{R}(r)]c'$  and  $c'[R(\Delta_q)]c''$ ; then we should show  $c[\mathcal{R}(r)]c''$ . To this end, let  $r' : p' \leftrightarrow q'$  be a saturated relation between the pers  $p', q'$ . First, note that if  $h[r_0 * r * r']h'$ , then  $h'[p_0 * q * r']h'$ . The following diagram shows  $c[\mathcal{R}(r)]c''$ .

$$\begin{array}{ccc} h & \xrightarrow{c} & h_0 \\ \uparrow r_0 * r * r' & & \uparrow r_1 * r * r' \\ h' & \xrightarrow{c'} & h'_0 \\ \uparrow p_0 * q * r' & & \uparrow q_0 * q * r' \\ h' & \xrightarrow{c''} & h''_0 \end{array}$$

Just as for the triple objects in  $\mathcal{P}_v$ , the quadruple object generator  $[-, -]$  extends to a functor. Let  $\mathcal{R}_r$  be the category of saturated relations between pers on heaps, ordered by inclusion. Then, define the functor  $\text{quad} : \mathcal{R}_r^{\text{OP}} \times \mathcal{R}_r \rightarrow \mathcal{P}_e$  in the same manner as the tri functor.

$$\text{quad}(r, r') = [r, r'] \quad \text{and} \quad \text{quad}(r_0 \subseteq r'_0, r_1 \subseteq r'_1) = \text{id}.$$

As for triple objects, there is an interplay between quadruples and invarian extension.

**Lemma 6.13.** *For a saturated relation  $r : p \leftrightarrow q$ ,*

$$\text{inv}_e(-, r) \circ \text{quad} = \text{quad}(r * -, r * -)$$

Note the interplay between the structure of the reflexive graph  $\mathcal{P}$  and the tri and functors, captured by

**Lemma 6.14.** *Let  $p_0, q_0$  be pers on heaps. Then,*

$$I(\text{tri}(p_0, q_0)) = \text{quad}(\Delta_{p_0}, \Delta_{q_0})$$

*Proof.* Let  $r : p \leftrightarrow q$ . Then,

$$c[(I[p_0, q_0])(r)]c' \quad \text{iff} \quad c[[p_0, q_0](r)]c'$$

and  $c[[\Delta_{p_0}, \Delta_{q_0}](r)]c'$  if and only if

- $c[[p_0, q_0](\Delta_p)]c$ ;
- $c'[[p_0, q_0](\Delta_q)]c'$ ; and
- some simulation condition.

The simulation condition mentioned is exactly the definition of  $c[[p_0, q_0](r)]c'$ . The other conditions hold due to identity extension of the object  $[p_0, q_0]$ .  $\square$

As a corollary, note the following property, which resembles the fundamental theorem of logical relations.

**Theorem 6.15.** *Suppose  $c$  is a map from the terminal object to  $[p_0, q_0]$  in  $\mathcal{P}_v$ . Then, the pair  $(c, c)$  is a map from the terminal object to  $[\Delta_{p_0}, \Delta_{q_0}]$  in  $\mathcal{P}_e$ .*

*Proof.* Easy, since  $I$  sends  $c$  to  $(c, c)$ ,  $[p_0, q_0]$  to  $[\Delta_{p_0}, \Delta_{q_0}]$  (according to Lemma 6.13) and preserves the terminal object.  $\square$

This gives us a way to go from triples to quadruples, inasmuch as this implies soundness of the rule

$$\frac{M : \{p_0\} - \{q_0\}}{(M, M) : \{\Delta_{p_0}\} = \{\Delta_{q_0}\}}.$$

Total correctness types are included here (they were not present in the LICS paper, although it would have been straightforward to add them), because we want an embedding rule like the one in Yang's relational separation logic [135] to hold. In our terminology, this rule reads

$$\frac{\vdash_{\Delta} M : [p_0] - [q_0] \quad \vdash_{\Delta} M' : [p'_0] - [q'_0]}{\vdash_{\Delta} (M, M') : \left\{ \begin{pmatrix} p_0 \\ p'_0 \end{pmatrix} \right\} = \left\{ \begin{pmatrix} q_0 \\ q'_0 \end{pmatrix} \right\}}. \quad (6.6)$$

First, note that total correctness is required, *i.e.*, both of the programs  $c_0, c_1$  are required to terminate; we have the same requirement here. We show the semantic version of this rule. The setup is different, since we deal with pers and not predicates. Before the proof, though, define the relation  $\begin{pmatrix} p_0 \\ p_1 \end{pmatrix}$ , given two pers  $p_0, p_1$ , by

$$h \left[ \begin{pmatrix} p_0 \\ p_1 \end{pmatrix} \right] h' \quad \text{iff} \quad h[p_0]h \text{ and } h'[p_1]h'.$$

It is straightforward to show that this is a saturated relation with respect to  $p_0$  and  $p_1$ , so we proceed to the proof of soundness of (6.6).

Suppose  $M : 1 \rightarrow \lfloor p_0, q_0 \rfloor$  and  $M' : 1 \rightarrow \lfloor p_0, q_0 \rfloor$  are morphisms in  $\mathcal{P}_v$ ; this means, for example, that  $M[\lfloor p_0, q_0 \rfloor(r)]M$  for all  $r : p \leftrightarrow q$ . We need to show that  $(M, M')$  is a morphism  $1 \rightarrow \text{quad}(\binom{p_0}{p'_0}, \binom{q_0}{q'_0})$  in  $\mathcal{P}_e$ , i.e., that for all  $r : p \leftrightarrow q$ ,

$$M[\text{quad}(\binom{p_0}{p'_0}, \binom{q_0}{q'_0})(r)]M'.$$

Let  $P$  and  $Q$  be the pers  $\binom{p_0}{p'_0}$  and  $\binom{q_0}{q'_0}$ , respectively, and let  $r : p \leftrightarrow q$  and  $r' : p' \leftrightarrow q'$  be saturated pers. Suppose  $h = j_0 * j_1 * j_2 [P * r * r'] j'_0 * j'_1 * j'_2 = h'$  with  $j_0 [P] j'_0$ ,  $j_1 [r] j'_1$ , and  $j_2 [r'] j'_2$ . Then  $h[p_0 * r * r'] h$ .  $M$  terminates, and the frame property implies that if  $h_0 \in M(h_0)$ , then  $h_0 = l_0 * j_1 * j_2$  for some  $l_0$  in  $M(j_0)$ . Since  $M$  is related to itself, this  $l_0$  is related to itself in  $q_0 : l_0 [q_0] l_0$ , which entails  $h_0 [q_0 * r * r'] h_0$ . The same argument can be carried out for  $M'(h')$ , resulting in a heap  $h'_0$  in  $M'(h')$  with  $h'_0 [q_1 * r * r'] h'_0$ . This means that  $h_0 [Q * r * r'] h'_0$ , as desired.

## 6.4 Storage Model and Assertions

In this section, we present a calculus in the style of the LICS paper which is meant to capture the properties of our model.

We give syntax and semantics for expressions and assertions; much of this is the same as in the LICS paper, but the semantics of assertions is interpreted as a map from stacks to pers on heaps rather than predicates on heaps. We interchangeably use the phrases “identifier” and “variables” since they are immutable. Let  $i, j, \dots$  range over these, and define the semantic domains:

$$\begin{aligned} \eta &\in \llbracket \Delta \rrbracket \stackrel{\text{def}}{=} \Delta \rightarrow \text{Int} \\ h &\in \text{Heap} \stackrel{\text{def}}{=} \text{Nat} \rightarrow_{\text{fin}} \text{Int} \\ (\eta, h) &\in \text{State}(\Delta) \stackrel{\text{def}}{=} \llbracket \Delta \rrbracket \times \text{Heap} \end{aligned}$$

The expression and assertion languages are the same as in classical separation logic [115].

$$\begin{aligned} E &::= i \mid 0 \mid 1 \mid E + E \\ P &::= E = E \mid E \mapsto E \mid \text{emp} \mid P * P \mid \mathbf{true} \mid P \wedge P \mid \forall i. P \mid \exists i. P \end{aligned}$$

Write  $\Delta \vdash E$  and  $\Delta \vdash P$  to indicate that the free variables of an expression or assertion are included in  $\Delta$ . The interpretation of an expression  $\Delta \vdash E$  is of the form

$$\llbracket \Delta \vdash E \rrbracket : \llbracket \Delta \rrbracket \rightarrow \text{Int},$$

and it is completely standard.

Assertions  $\Delta \vdash P$  are interpreted as precise pers on the set of heaps, i.e., the interpretation of an assertion  $\Delta \vdash P$  is of the form  $\llbracket \Delta \rrbracket \rightarrow \text{Per}(\text{Heap})$ , and it is given by

$$\begin{aligned} \llbracket E = E' \rrbracket \eta &= \begin{cases} \emptyset & \text{if } \llbracket E \rrbracket \eta \neq \llbracket E' \rrbracket \eta \\ \text{Id} & \text{if } \llbracket E \rrbracket \eta = \llbracket E' \rrbracket \eta \end{cases} \\ \llbracket E \mapsto E' \rrbracket \eta &= p, \text{ where } h[p]h' \text{ iff } h = h' = \llbracket \llbracket E \rrbracket \eta \rightarrow \llbracket E' \rrbracket \eta \rrbracket \\ \llbracket \text{emp} \rrbracket \eta &= p, \text{ where } h[p]h' \text{ iff } h = h' = [] \\ \llbracket P * P' \rrbracket \eta &= p * p' \text{ where } p = \llbracket P \rrbracket \eta \text{ and } p' = \llbracket P' \rrbracket \eta \\ \llbracket \mathbf{true} \rrbracket \eta &= \text{Id} \\ \llbracket P \wedge P' \rrbracket \eta &= p \text{ where } h[p]h' \text{ iff } h[\llbracket P \rrbracket \eta]h' \text{ and } h[\llbracket P' \rrbracket \eta]h' \\ \llbracket \forall i. P \rrbracket \eta &= p \text{ where } h[p]h' \text{ iff } h[\llbracket P \rrbracket \eta_{[i \mapsto v]}]h' \text{ for all } v \in \text{Int} \\ \llbracket \exists i. P \rrbracket \eta &= p \text{ where } h[p]h' \text{ iff } h[\llbracket P \rrbracket \eta_{[i \mapsto v]}]h' \text{ for some } v \in \text{Int} \end{aligned}$$

where we use  $Id$  for the diagonal per defined by  $h[Id]h'$  iff  $h = h'$ . This interpretation is not very rich, inasmuch as all interpretations are “diagonals”. This elaborated upon in Section 6.6.

## 6.5 Programming Languages

We present a programming language for “simple programs” which is essentially the language from the LICS paper, and a simple programming language for “program pairs”; the latter are used as preterms in a calculus, ideas from separation-logic typing are combined with those of relational separation logic [135].

### 6.5.1 Programming Language for Triple-typing

We start with the language for simple programs. It is the language from the LICS paper, except that heap allocation is left out. Types of this language are defined as follows. Write  $\Delta \vdash \theta : \text{Type}$  for a type  $\theta$  in context  $\Delta$ ; this is defined as follows.

$$\frac{\text{fiv}(P) \subseteq \Delta \quad \text{fiv}(Q) \subseteq \Delta}{\Delta \vdash_{\Delta} [P] - [Q] : \text{Type}} \quad \frac{\text{fiv}(P) \subseteq \Delta \quad \text{fiv}(Q) \subseteq \Delta}{\Delta \vdash \{P\} - \{Q\} : \text{Type}}$$

$$\frac{\Delta \vdash \theta : \text{Type} \quad \text{fiv}(P) \subseteq \Delta}{\Delta \vdash \theta \otimes P : \text{Type}}$$

$$\frac{\Delta \cup \{i\} \vdash \theta : \text{Type} \quad i \notin \Delta}{\Delta \vdash \Pi_i \theta : \text{Type}} \quad \frac{\Delta \vdash \theta : \text{Type} \quad \Delta \vdash \theta' : \text{Type}}{\Delta \vdash \theta \rightarrow \theta' : \text{Type}}$$

This is the same as in the LICS paper, except that an extra base type  $[P] - [Q]$  for total correctness is included, as discussed at the end of Section 6.3.9.

For these types, there is a subtyping relation  $\preceq_{\Delta}$ , which is essentially the same as in the LICS paper. It is the usual structural subtyping relation [71] (*i.e.*, it is reflexive, transitive, and on function types, and it is contravariant in argument types and covariant in result types), extended with the following rules.

$$\begin{aligned} [P] - [Q] &\preceq_{\Delta} \{P\} - \{Q\} \\ \{P'\} - \{Q'\} &\preceq_{\Delta} \{P\} - \{Q\} \quad (\text{when for all } \eta, \mathbf{f}P_{\eta} \subseteq \mathbf{f}P'_{\eta} \text{ and } \mathbf{f}Q'_{\eta} \subseteq \mathbf{f}Q_{\eta}) \\ \theta &\preceq_{\Delta} \theta \otimes P & (\{P\} - \{Q\}) \otimes P_0 &\simeq_{\Delta} \{P * P_0\} - \{Q * P_0\} \\ (\theta \otimes Q) \otimes P &\simeq_{\Delta} \theta \otimes (Q * P) & (\theta \rightarrow \theta') \otimes P &\simeq_{\Delta} (\theta \otimes P \rightarrow \theta' \otimes P) \\ (\Pi_i \theta) \otimes P &\simeq_{\Delta} \Pi_i \theta \otimes P \end{aligned}$$

The only difference compared to the LICS paper is that total correctness is a subtype of partial correctness.

Pre-terms of the language are given by two grammars. The first of these is for *simple triple preterms*. Intuitively, these are the “simple commands” from the programming language of standard separation logic, which are known to terminate. The simple triple pre-terms are given by

$$\begin{aligned} M^s ::= & \text{ifz } E \ M^s \ M^s \\ & | \ M^s ; M^s \\ & | \ \text{free}(E) \\ & | \ [E] := E \\ & | \ \text{let } i = [E] \ \text{in } M^s \\ & | \ \text{while } E \ M^s. \end{aligned} \tag{6.7}$$

The simple triple pre-terms are a subset of the pre-terms of the programming language from the LICS paper. The latter has pre-terms given by

$$\begin{array}{l}
 M ::= M^s \\
 \quad | x \\
 \quad | \lambda x:\theta.M \\
 \quad | MM \\
 \quad | \lambda i.M \\
 \quad | ME \\
 \quad | \text{fix } M
 \end{array} \tag{6.8}$$

This is the same as the preterms of the language in the LICS paper, except that new is omitted.

The typing rules of the language are defined by a judgment  $\Gamma \vdash_{\Delta} M : \theta$ . We first give the typing rules for the simple triple preterms  $M^s$ . They are

$$\frac{\Gamma \vdash_{\Delta \cup \{i\}} M^s : \langle P * E \mapsto i \rangle - \langle Q \rangle}{\Gamma \vdash_{\Delta} \text{let } i = [E] \text{ in } M^s : \langle P * E \mapsto - \rangle - \langle Q \rangle} \quad i \notin \text{fiv}(\Gamma, E, P, Q)$$

$$\frac{}{\text{free}(E) : [E \mapsto -] - [\text{emp}]} \quad \text{fiv}(\Gamma, E) \subseteq \Delta$$

$$\frac{}{\Gamma \vdash_{\Delta} [E] := E' : [E \mapsto -] - [E \mapsto E']} \quad \text{fiv}(\Gamma, E, E') \subseteq \Delta$$

$$\frac{\Gamma \vdash_{\Delta} M^s : \langle P \rangle - \langle P' \rangle \quad \Gamma \vdash_{\Delta} M'^s : \langle P' \rangle - \langle Q \rangle}{\Gamma \vdash_{\Delta} M^s; M'^s : \langle P \rangle - \langle Q \rangle}$$

$$\frac{\Gamma \vdash_{\Delta} M^s : \langle P \wedge E = 0 \rangle - \langle Q \rangle \quad \Gamma \vdash_{\Delta} M'^s : \langle P \wedge E \neq 0 \rangle - \langle Q \rangle}{\Gamma \vdash_{\Delta} \text{ifz } E M^s M'^s : \langle P \rangle - \langle Q \rangle}$$

$$\frac{\Gamma \vdash_{\Delta} M^s : \{(E \neq 0) \wedge P\} - \{P\}}{\Gamma \vdash_{\Delta} \text{while } E M^s : \{P\} - \{P \wedge E = 0\}}$$

In these rules, we use angled brackets  $\langle - \rangle$  to indicate that a rule is valid for both partial and total correctness types. For example, the first rule above is shorthand for two rules. In one, the term  $M^s$  is required to have the type  $\{P * E \mapsto i\} - \{Q\}$ , whereas in the other,  $M^s$  is required to have type  $[P * E \mapsto i] - [Q]$ .

The typing rules for the rest of the terms, *i.e.*, the terms defined in the grammar (6.8), are defined in the same way as in the LICS paper, so we refrain from giving it here.

### 6.5.2 Programming Language for Quadruple-typing

We turn to program pairs and quadruple types. Recall that, stated informally, the meaning of a quadruple

$$\{p\} \begin{array}{c} c_0 \\ c_1 \end{array} \{p'\}, \tag{6.9}$$

where  $p$  and  $p'$  are *relations* on states and  $c_0, c_1$  are programs, is that all pairs of states related by  $p$  are taken to pairs of states by  $c_0, c_1$  that are related by  $p'$ . This idea is pursued by assigning *quadruple types* to terms which, intuitively, denote pairs of programs.

First, assume a grammar for relations on heaps, in the style of relational separation logic [135]. We refrain from fixing this grammar here; there is a brief discussion of this in Section 6.6.3. Some relation forms, that need to be present in the language, however, are

- “Pure relations”, *i.e.*, relations whose truth do not depend on the heap.
- $\text{Emp}$ , which is the singleton relation that relates the empty heap to itself.
- $R_0 * R_2$ , the separating conjunction for relations, which was introduced in the article [107].
- $I_P$ , where  $P$  is taken from the grammar of assertions introduced earlier.
- $\binom{P}{P'}$ , the *assertion pair* from relational separation logic.

We abuse terminology and call the types in the quadruple language *kinds* instead of types, to avoid confusion. As for types, there is a calculus  $\Delta \vdash \tau : \text{Kind}$ , which means that  $\tau$  is a kind in the context  $\Delta$ . This judgment is defined by

$$\frac{\text{fiv}(R) \subseteq \Delta \quad \text{fiv}(S) \subseteq \Delta \quad \Delta \vdash \tau : \text{Kind} \quad \text{fiv}(R) \subseteq \Delta}{\Delta \vdash \{R\} \sqsubseteq \{S\} : \text{Kind}} \quad \frac{\Delta \vdash \tau : \text{Kind} \quad \text{fiv}(R) \subseteq \Delta}{\Delta \vdash \tau \odot R : \text{Kind}}$$

$$\frac{\Delta \vdash \tau : \text{Kind} \quad \Delta \vdash \tau : \text{Kind}}{\Delta \vdash \tau \rightarrow \tau : \text{Kind}} .$$

Notice that quadruple types are base types, and that invariant extension (by the  $\odot$  connective) is with relations rather than assertions. Following the lines of the subtyping relation  $\preceq_{\Delta}$ , there is a “subkinding relation”  $\ll$  for kinds. It extends usual subtyping with

$$\begin{aligned} \tau &\ll \tau \odot R & (\{R\} \sqsubseteq \{S\}) \odot R' &\equiv \{R * R'\} \sqsubseteq \{S * R'\} \\ (\tau \rightarrow \tau') \odot R &\equiv (\tau \odot R) \rightarrow (\tau' \odot R) \\ I(\{P\} \sqsubseteq \{Q\}) &\equiv \{I_P\} \sqsubseteq \{I_Q\} & I(\theta \rightarrow \theta') &\equiv I(\theta) \rightarrow I(\theta') \\ I(\theta \otimes P) &\equiv I(\theta) \odot I_P. \end{aligned}$$

The pre-terms of the quadruple programming language is a simply typed  $\lambda$ -calculus, with “program pairs” as basic constants. The grammar for these pre-terms is

$$N ::= x \mid \lambda x:\tau. N \mid NN \mid (M, M), \tag{6.10}$$

where  $M$  ranges over programs from the grammar (6.8). The typing rules of this language are defined by a judgment  $\Xi \vdash_{\Delta} N : \tau$ , where  $\Xi$  is a list of kind assignments and  $\Delta$  contains all the free variables in  $\Xi, N, \tau$ . As mentioned, this is a simply typed  $\lambda$ -calculus, extended with a typing rule for pairs of simple programs. The judgment is defined by standard rules for simply typed  $\lambda$ -calculus, and rules for pairs of terms from

the grammar (6.8). The  $\lambda$ -calculus rules are

$$\frac{}{\Xi, x : \tau \vdash_{\Delta} : \tau}$$

$$\frac{\Xi, x : \tau \vdash_{\Delta} N : \tau'}{\Xi \vdash_{\Delta} \lambda x : \tau. N : \tau \rightarrow \tau}$$

$$\frac{\Xi \vdash_{\Delta} N : \tau' \rightarrow \tau \quad \Xi \vdash_{\Delta} N' : \tau'}{\Xi \vdash_{\Delta} NN' : \tau} .$$

$$\frac{\Xi \vdash_{\Delta} N : \tau \quad \tau \ll \tau'}{\Xi \vdash_{\Delta} N : \tau'} ,$$

and the rules for pairs of terms are

$$\frac{\vdash_{\Delta} M : \theta}{\vdash_{\Delta} (M, M) : I(\theta)}$$

$$\frac{\vdash_{\Delta} M^s : [P] - [Q] \quad \vdash_{\Delta} M'^s : [P'] - [Q']}{\vdash_{\Delta} (M^s, M'^s) : \left\{ \binom{P}{P'} \right\} = \left\{ \binom{Q}{Q'} \right\}}$$

$$\frac{R \Rightarrow (E_0 \stackrel{0}{=} E_1) \quad \vdash_{\Delta} (M_0^s, M_1^s) : \{R \wedge (E \neq 0)\} = \{R\}}{\vdash_{\Delta} (\text{while } E_0 M_0^s, \text{while } E_1 M_1^s) : \{R\} = \{R \wedge (E = 0)\}}$$

$$\frac{R \Rightarrow (E_0 \stackrel{0}{=} E_1) \quad \begin{array}{l} \vdash_{\Delta} (M_0^s, M_1^s) : \{R \wedge (E = 0)\} = \{S\} \\ \vdash_{\Delta} (M_0^s, M_1^s) : \{R \wedge (E \neq 0)\} = \{S\} \end{array}}{\vdash_{\Delta} (\text{ifz } E_0 M_0^s M_0^s, \text{ifz } E_1 M_1^s M_1^s) : \{R\} = \{S\}}$$

$$\frac{\vdash_{\Delta} (M_0^s, M_1^s) : \{R\} = \{R'\} \quad \vdash_{\Delta} (M_0^s, M_1^s) : \{R'\} = \{S\}}{\vdash_{\Delta} (M_0^s; M_0^s, M_1^s; M_1^s) : \{R\} = \{S\}} ,$$

where we use the shorthand notation  $E_0 \stackrel{0}{=} E_1$  for  $E_0 = 0 \iff E_1 = 0$ . The problem with this calculus is, of course, its lack of base cases. As Yang mentions [135], programs have to “have the same structure” in order for us to apply this calculus. At this point, we do not have any convincing examples, although we believe it possible to fit the Schorr-Waite example from the relational separation logic paper [135] into this framework. The rules presented here (which can be seen as an internal language for the model) are, in other words, far from complete.

### Interpretation of Quadruple Terms

The calculus for quadruple terms is, like that for triple terms, a simply typed  $\lambda$ -calculus, and the category  $\mathcal{P}_e$  is cartesian closed, so part of the interpretation is completely standard. Moreover, the constructions corresponding to pairing both two terms of ground type (the embedding rule) and for pairing a term of arbitrary type with itself (our semantical version of the fundamental theorem of logical relations) can be carried out; this gives a way to interpret these terms. The interpretation of the terms formed by the rest of the typing rules here is omitted here.

## 6.6 Conclusion and Problems

We have presented the an extension of the categorical model for separation-logic typing presented in the LICS paper. We also demonstrated that this is a promising direction towards relational reasoning for higher-order programs, inasmuch as there are morphisms corresponding to many of the rules for quadruples in relational separation logic [135], and there is a morphism corresponding to the fundamental theorem of logical relations. However, there are some problems, as illustrated in the remainder of this paper.

Recall from Section 6.3.9 that we changed from indexing by predicates to indexing by pers on heaps, in order to deal with the non-determinism assumed from allocation of new heap cells. We show here that this is not enough to solve the problem. At this point, we do not have a detailed solution to the problem, but we briefly outline possible ways to approach the it.

Consider again **cons** and a per-specification for it.

$$\{\text{emp}\} x := \mathbf{cons}() \{R\} \quad (6.11)$$

The problem is to give a candidate for  $R$ . As explained in Section 6.3.9,  $R$  should allow  $\alpha$ -renaming, in order to keep soundness of the binary version of the frame rule. The problem is that  $\alpha$ -renaming induces a partial equivalence relation on *states* (including stacks), not just heaps: two *states*  $(\eta, h)$  and  $(\eta', h')$  should be related by  $R$  iff there are integers  $n, n'$  such that  $\eta(x) = n, h = [n \rightarrow -]$  and  $\eta'(x) = n', h' = [n' \rightarrow -]$ , *i.e.*, the relation  $R$  relates states with different stacks.

So far, our work has followed the lines of the LICS paper quite tightly, but the difficulty just mentioned might force us to deviate substantially from the LICS paper. In the LICS paper, we use the family fibration  $Fam(\mathcal{D}) \rightarrow \mathbf{Set}$  to interpret terms. Informally this means that each store shape gives an interpretation of expressions, assertions, types, and terms, *i.e.*, the interpretation of, e.g., an assertion is a map from stacks to predicates on heaps. To follow those lines, we would take the interpretation of an assertion to be a map from stacks to pers on heaps. Now, consider the relation  $R$  in the specification (6.11) above. As outlined above, this per *depends on the value of  $x$  in the stack*, *i.e.*, it is not possible to capture the required properties of  $R$  from the knowledge of only one stack. Hence it is not possible to define the meaning of  $R$  as a map from stacks to pers on heaps, and this implies that the *Fam*-construction from the LICS paper will not work without complications in our setting. This, in turn, means that we have to fundamentally change our semantics in order to at the same time make the model parametric and allow (nondeterministic) allocation of new heap cells in the programming language.

As mentioned, we do not have a detailed solution at this point, but we outline two ideas that could lead to a resolution are. Future work will reveal whether one of them, a combination of them, or some other ideas will lead the way.

### 6.6.1 Pers on States

We used the *Fam*-construction to model local variables in the LICS paper; this is quite standard, since separation-logic typing can be seen as a dependent type system. It might be possible to keep this view and still interpret the language along standard lines.

As mentioned, the development of our model in Sections 6.3.1 - 6.3.8 is quite general and does not depend on the heap model or whether predicates on heaps, pers on heaps, or some other indexing category is used, as long as this has a monoidal structure. Further, all the properties in Section 6.3.9 hold when going from pers on heaps to pers on

*states*, since the extra properties we require from the storage model also hold for states. Although we have not checked it, we *believe* that both the categories  $\mathcal{P}_v$  and  $\mathcal{P}_e$  are *locally cartesian closed* [61], *i.e.*, each slice category in both of the categories are cartesian closed, and the functors that make  $\mathcal{P} = \mathcal{P}_v \rightleftarrows \mathcal{P}_e$  a reflexive graph preserve all the locally cartesian closed structure. Given a locally cartesian closed category  $\mathcal{C}$ , it is standard [119, 61] to interpret dependent type theories in the co-domain fibration  $\mathcal{C}^\rightarrow \rightarrow \mathcal{C}$ . Then, the categorical product corresponds to the dependent product of dependent type theory (and coproducts correspond to dependent sums). As mentioned, this might well be possible, but it changes the intuition behind the model considerably. For instance, this interpretation will cause the interpretation of the contexts  $\Delta$  to live in the same categories as the interpretation of the types, and that means that they might potentially express properties of heaps, whereas in the LICS model, contexts were only used to keep track of variables. Possibly, the contexts can be interpreted in a class of “pure objects” of the category, but on the other hand, we might gain new insights from having contexts whose interpretation can depend on the heap.

### 6.6.2 Non-determinism of Heap Allocation

The next approach to solving the problem outlined in this section is potentially more fundamental, inasmuch as it challenges the way we think about heap allocation. The reason to change the indexing category from predicates on heaps to pers on heaps comes from the fact that we assume that **cons** is *non-deterministic*. Somewhat controversially, we claim that this might be a pseudo-problem, since assuming non-determinism of allocation is not the right level of abstraction. It might be more correct to view memory allocation as a *module* instead of an operation of the programming language. To do so, we should really use a specification of a memory allocator (e.g., as in the papers [23, 33]) and treat **cons**/**new** simply as a procedure rather than a atomic operation.

### 6.6.3 Other Problems

The main problem of the work in this paper is, as mentioned, to give a good interpretation of assertions as pers on heaps/states. This induces two other minor problems, which we have glossed over so far. They are mentioned briefly here, but an in-depth analysis of them does not make sense as long as the basic interpretation of assertions, types, *etc.*, is not fixed.

First, we did not fix a syntax for relations in Section 6.5.2. To our knowledge, no grammars for pers on states exist, and it is a challenge in itself to come up with a “good” grammar for such pers, in the sense that it is simple and at the same time captures all relevant pers. Although this is certainly an interesting problem, we choose not to approach it since the focus of this paper is on semantics rather than syntax.

Another issue is that the rules in Section 6.5.2 for going from a simple program (with a triple type) to a program pair (with a quadruple type) all require the contexts to be empty. When the interpretation of contexts is fixed, contexts possibly need not be empty, so that we can use contexts of the form “ $I(\Delta)$ ” in the conclusions of the rules in Section 6.5.2. However, as long as we do not have a fixed interpretation of contexts (and hence no way of going from  $\Delta$ ’s to  $\Xi$ ’s), we stick to the stricter rules with empty contexts.



# Bibliography

- [1] ABADI, M., CARDELLI, L., AND CURIEN, P.-L. Formal parametric polymorphism. *Theoretical Computer Science* 121, 1 & 2 (1993), 9–58.
- [2] ADITYA, S., AND CARO, A. Compiler-directed type reconstruction for polymorphic languages. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA '93)* (Copenhagen, Denmark, 1993), ACM Press, pp. 74–82.
- [3] ADITYA, S., FLOOD, C. H., AND HICKS, J. E. Garbage collection for strongly-typed languages using run-time type reconstruction. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming (LFP '94)* (Orlando, Florida, United States, 1994), ACM Press, pp. 12–23.
- [4] AHMED, A., JIA, L., AND WALKER, D. Reasoning about hierarchical storage. In *Proc. of the Eighteenth Annual IEEE Symposium on Logic in Computer Science (LICS'03)* (Ottawa, Canada, June 2003), IEEE Press.
- [5] AHMED, A., AND WALKER, D. The logical approach to stack typing. In *Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003)* (New Orleans, LA, USA, January 2003), pp. 74–85.
- [6] APPEL, A. W. Foundational proof carrying code. In *Proc. of the Sixteenth IEEE Symposium on Logic in Computer Science (LICS'01)* (Boston, MA, USA, June 2001), IEEE Press.
- [7] APPEL, A. W., AND GONÇALVES, M. J. R. Hash-consing garbage collection. Tech. Rep. CS-TR-412-93, Princeton University, February 1993.
- [8] BAINBRIDGE, E. S., FREYD, P. J., SCEDROV, A., AND SCOTT, P. J. Functorial polymorphism. *Theoretical Computer Science* 70, 1 (1990), 35–64.
- [9] BANERJEE, A., AND NAUMANN, D. A. State based ownership, reentrance, and encapsulation. In *Proc. of the Nineteenth European Conference on Object-Oriented Programming (ECOOP'05)*.
- [10] BANERJEE, A., AND NAUMANN, D. A. Representation independence, confinement and access control [extended abstract]. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'02)* (Portland, OR, USA, 2002), ACM Press, pp. 166–177.
- [11] BANERJEE, A., AND NAUMANN, D. A. Ownership confinement ensures representation independence for object-oriented programs. *Journal of the ACM* 52, 6 (November 2005). Supercedes the conference paper [10].

- [12] BEN-ARI, M. Algorithms for on-the-fly garbage collection. *ACM Transactions of Principles on Programming Languages and Systems (TOPLAS)* 6, 3 (1984), 333–344.
- [13] BENTON, N. Simple relational correctness proofs for static analyses and program transformations. In *Proceedings of the 31st Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL'04)* (Venice, Italy, January 2004), ACM Press, pp. 14–25.
- [14] BENTON, N., AND LEPCHEY, B. Relational reasoning in a nominal semantics for storage. In *Proceedings of the Seventh International Conference on Typed Lambda Calculi and Applications (TLCA '05)* (Nara, Japan, April 2005), P. Urzyczyn, Ed., vol. 3461 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 86–101.
- [15] BERDINE, J., CALCAGNO, C., AND O'HEARN, P. W. A decidable fragment of separation logic. In *Proceedings of the Twenty-Fourth Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'04)* (2004), vol. 3328 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 97–109.
- [16] BERDINE, J., CALCAGNO, C., AND O'HEARN, P. W. Symbolic execution with separation logic. In *Proceedings of the Third Asian Symposium on Programming Languages and Systems (APLAS'05)* (Tsukuba, Japan, November 2005), Springer Verlag, pp. 52–68.
- [17] BERGER, M., HONDA, K., AND YOSHIDA, N. A logical analysis of aliasing in imperative higher-order functions. Tech. rep., Queen Mary, University of London, 2005. Long version of [18]. Available at <http://www.dcs.qmul.ac.uk/~kohei/logics/>.
- [18] BERGER, M., HONDA, K., AND YOSHIDA, N. A logical analysis of aliasing in imperative higher-order functions. In *Proc. of The 10th ACM SIGPLAN International Conference on Functional Programming (ICFP'05)* (Tallinn, Estonia, September 2005), ACM Press. Accepted for publication.
- [19] BIERING, B. On the logic of bunched implications and its relation to separation logic. Master's thesis, University of Copenhagen, 2004.
- [20] BIERING, B., BIRKEDAL, L., AND TORP-SMITH, N. BI-hyperdoctrines and higher order separation logic. In *Proc. of ESOP 2005: The European Symposium on Programming* (Edinburgh, Scotland, April 2005), pp. 233–247.
- [21] BIERING, B., BIRKEDAL, L., AND TORP-SMITH, N. Bi hyperdoctrines, higher-order separation logic, and abstraction. Tech. Rep. ITU-TR-2005-69, IT University of Copenhagen, Copenhagen, Denmark, July 2005. Extended version of [20].
- [22] BIRKEDAL, L., MØGELBERG, R. E., AND PETERSEN, R. L. Parametric domain-theoretic models of polymorphic intuitionistic / linear lambda calculus. In *Proceedings of Mathematical Foundations of Programming Semantics XXI (MFPS'05)* (Birmingham, United Kingdom, May 2005), Elsevier.
- [23] BIRKEDAL, L., TORP-SMITH, N., AND REYNOLDS, J. Local reasoning about a copying garbage collector. In *Proc. of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'04)* (Venice, Italy, January 2004), pp. 220–231.

- [24] BIRKEDAL, L., TORP-SMITH, N., AND REYNOLDS, J. C. Local reasoning about a copying garbage collector. *ACM Transactions of Principles on Programming Languages and Systems (TOPLAS)* (2005). Extended version of [23]. Accepted modulo revisions.
- [25] BIRKEDAL, L., TORP-SMITH, N., AND YANG, H. Semantics of separation-logic typing and higher-order frame rules. In *Proceedings of the Twentieth Annual IEEE Symposium on Logic in Computer Science (LICS 2005)* (Chicago, IL, USA, June 2005), IEEE Press, pp. 260–269.
- [26] BORNAT, R. Proving pointer programs in Hoare logic. In *Proceedings of the 5th International Conference on Mathematics of Program Construction* (Ponte De Lima, Portugal, July 2000), vol. 1837 of *Lecture Notes In Computer Science*, Springer Verlag, pp. 102–126.
- [27] BORNAT, R. Correctness of copydag via local reasoning. Private Communication, March 2003.
- [28] BORNAT, R., CALCAGNO, C., AND O’HEARN, P. Local reasoning, separation and aliasing. In *Proceedings of SPACE 2004* (Venice, Italy, January 2004).
- [29] BORNAT, R., CALCAGNO, C., O’HEARN, P., AND PARKINSON, M. Permission accounting in separation logic. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL’05)* (Long Beach, CA, USA, January 2005), ACM.
- [30] BOYAPATI, C., LISKOV, B., AND SHRIRA, L. Ownership types for object encapsulation. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL’03)* (New Orleans, LA, USA, January 2003), pp. 213–223.
- [31] BURDY, L. B vs Coq to prove a garbage collector. In *Proc. of the 14th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2001)* (Edinburgh, Scotland, September 2001), vol. 2152 of *LNCS*, Springer Verlag.
- [32] CALCAGNO, C., ISHTIAQ, S., AND O’HEARN, P. W. Semantic analysis of pointer aliasing, allocation and disposal in Hoare logic. In *Proc. of the Second International Conference on Principles and Practice of Declarative Programming (PPDP’00)* (Montreal, Canada, September 2000).
- [33] CALCAGNO, C., O’HEARN, P. W., AND BORNAT, R. Program logic and equivalence in the presence of garbage collection. *Theoretical Computer Science* 298, 3 (2003), 557–587.
- [34] CALCAGNO, C., YANG, H., AND O’HEARN, P. W. Computability and complexity results for a spatial assertion language for data structures. In *Proceedings of the Twenty-First Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS’01)* (2001), vol. 2245 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 108–119.
- [35] CHENEY, C. J. A nonrecursive list compacting algorithm. *Communications of the ACM* 13, 11 (November 1970), 677–678.

- [36] CLARKE, D. G., NOBLE, J., AND POTTER, J. M. Simple ownership types for object containment. In *Proc. of the Fifteenth European Conference on Object-Oriented Programming (ECOOP'01)* (Budapest, Hungary, June 2001), Springer-Verlag, pp. 53–76.
- [37] COUPET-GRIMAL, S. C. nouvel. *Journal of Logic and Computation* 13, 6 (December 2003), 815–833.
- [38] CRARY, K., WALKER, D., AND MORRISETT, G. Typed memory management in a calculus of capabilities. In *Proc. of the 26th ACM SIGPLAN-SIGACT on Principles of Programming Languages (POPL'99)* (San Antonio, TX, USA, January 1999), pp. 262–275.
- [39] DIJKSTRA, E. W., LAMPORT, L., MARTIN, A. J., SCHOLTEN, G. S., AND STEFFENS, E. M. F. On-the-fly garbage collection: an exercise in cooperation.
- [40] DUNPHY, B., AND REDDY, U. S. Parametric limits. In *Proc. of the 19th Annual IEEE Symposium on Logic in Computer Science (LICS'04)* (Turku, Finland, July 2004), IEEE Press.
- [41] DUNPHY, B. P. *Parametricity as a Notion of Uniformity in Reflexive Graphs*. PhD thesis, University of Illinois, Urbana-Champaign, 2002.
- [42] FLUET, M., AND WANG, D. Implementation and performance evaluation of a safe runtime system in Cyclone. In *Informal Proceedings of the Second Workshop on Semantics, Program Analysis and Computing Environments for Memory Management (SPACE'04)* (Venice, Italy, January 2004), ACM/SIGPLAN.
- [43] GOTO, E. Monocopy and associative algorithms in extended lisp. Tech. Rep. TR 74-03, University of Tokyo, 1974.
- [44] GRIES, D., AND LEVIN, G. Assignment and procedure call proof rules. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 2, 4 (October 1980), 564–579.
- [45] GUTTMANN, J., RAMSDELL, J., AND WAND, M. VLISP: A verified implementation of scheme. *Lisp and Symbolic Computation* 8, 1–2 (1994), 5–32.
- [46] GUTTMANN, J. D., MONK, L. G., RAMSDELL, J. D., FARMER, W. M., AND SWARUP, V. A guide to VLISP, a verified programming language implementation. Tech. Rep. M92B091, The MITRE Corporation, 1992.
- [47] HALLENBERG, N., ELSMAN, M., AND TOFTE, M. Combining region inference and garbage collection. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'02)* (Berlin, June 2002), ACM Press.
- [48] HAVELUND, K. Mechanical verification of a garbage collector. In *Proc. of the Fourth International Workshop Formal Methods for Parallel Programming : Theory and Applications (FMPPTA'99)* (San Juan, Puerto Rico, April 1999).
- [49] HE, J., HOARE, C. A. R., AND SANDERS, J. W. Data refinement refined (resume). In *Proceedings of the First European Symposium on Programming (ESOP'86)*

- (Saarbrücken, Federal Republic of Germany, March 1986), B. Robinet and R. Wilhelm, Eds., vol. 213 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 187–196.
- [50] HOARE, C. A. R. An axiomatic approach to computer programming. *Communications of the ACM* 12, 583 (1969), 576–580.
- [51] HOARE, C. A. R. Procedures and parameters: An axiomatic approach. In *Symp. on Semantics of Algorithmic Languages* (1971), E. Engler, Ed., Springer-Verlag, Berlin, pp. 102–116.
- [52] HOARE, C. A. R. Proof of correctness of data representations. *Acta Informatica* 1 (1972), 271–281.
- [53] HOARE, C. A. R., HE, J., AND SANDERS, J. W. Prespecification in data refinement. *Information Processing Letters* 25, 2 (May 1987), 71–76.
- [54] HOGG, J. Islands: Aliasing protection in object-oriented languages. In *Proceedings of the Sixth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'91)* (Phoenix, AZ, USA, October 1991), A. Paepcke, Ed., vol. 26 of *SIGPLAN Notices*, pp. 271–285.
- [55] HOGG, J., LEA, D., WILLS, A., DECHAMPEAUX, D., AND HOLT, R. The geneva convention on the treatment of object aliasing. *SIGPLAN OOPS Messenger* 3, 2 (1992), 11–16.
- [56] HONDA, K., YOSHIDA, N., AND BERGER, M. An observationally complete program logic for imperative higher-order functions. Tech. rep., Queen Mary, University of London, 2005. Extended version of [57]. Available at <http://www.dcs.qmul.ac.uk/~kohei/logics/>.
- [57] HONDA, K., YOSHIDA, N., AND BERGER, M. An observationally complete program logic for imperative higher-order functions. In *Proc. of Twentieth Annual IEEE Symposium on Logic in Computer Science (LICS 2005)* (Chicago, IL, USA, June 2005), IEEE Press, pp. 270–279.
- [58] HUWIG, H., AND POIGNE, A. A note on inconsistencies caused by fixpoints in a cartesian closed category. *Theoretical Computer Science* 73, 1 (June 1990), 101–112.
- [59] ISHTIAQ, S., AND O'HEARN, P. W. BI as an assertion language for mutable data structures. In *Proceedings of the 28th Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL'01)* (London, 2001).
- [60] JACKSON, P. B. Verifying a garbage collection algorithm. In *Proc. of the 11th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'98)* (Canberra, Australia, September 1998), vol. 1479 of *LNCS*, Springer Verlag, pp. 225–244.
- [61] JACOBS, B. *Categorical Logic and Type Theory*, vol. 141 of *Studies in Logic and the Foundations of Mathematics*. North-Holland Publishing Co., Amsterdam, 1999.
- [62] JIA, L., AND WALKER, D. ILC: A foundation for automated reasoning about pointer programs, March 2005. Draft. Available at <http://www.cs.princeton.edu/~ljia/>.

- [63] JIM, T., MORRISETT, G., GROSSMAN, D., HICKS, M., CHENEY, J., AND WANG, Y. Cyclone: A safe dialect of C. In *Proc. of the USENIX Annual Technical Conference* (Monterey, CA, USA, June 2002), pp. 275–288.
- [64] KERNIGHAN, B. W., AND RITCHIE, D. M. *The C Programming Language*, second ed. Prentice-Hall, 1988.
- [65] LAMBEK, J., AND SCOTT, P. J. *Introduction to Higher Order Categorical Logic*, vol. 7 of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press, New York, NY, USA, 1986.
- [66] LAWVERE, F. Adjointness in foundations. *Dialectica* 23, 3/4 (1969), 281–296.
- [67] MA, Q., AND REYNOLDS, J. C. Types, abstraction, and parametric polymorphism, part 2. In *Proceedings of The International Conference on Mathematical Foundations of Programming Semantics (MFPS'91)* (Pittsburgh, PA, USA, March 1991), S. D. Brookes, M. Main, A. Melton, M. Mislove, and D. A. Schmidt, Eds., vol. 598 of *LNCS*, Springer-Verlag, pp. 1–40.
- [68] MEHTA, F., AND NIPKOW, T. Proving pointer programs in higher-order logic. In *Automated Deduction – CADE-19* (2003).
- [69] MEYER, A. R., AND SIEBER, K. Towards fully abstract semantics for local variables. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'88)* (San Diego, CA, USA, January 1998), ACM Press, pp. 191–203.
- [70] MIJAJLOVIĆ, I., TORP-SMITH, N., AND O'HEARN, P. W. Refinement and separation contexts. In *Proc. of the 24th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'04)* (Chennai, India, December 2004), vol. 3328 of *LNCS*, pp. 421–433.
- [71] MITCHELL, J. C. *Foundations of programming languages*. MIT Press, Cambridge, MA, USA, 1996.
- [72] MITCHELL, J. C., AND MOGGI, E. Kripke-style models for typed lambda calculus. *Annals of Pure and Applied Logic* 51 (1991), 99–124.
- [73] MITCHELL, J. C., AND PLOTKIN, G. D. Abstract types have existential type. In *Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages (POPL'85)* (New Orleans, LA, USA, 1985), pp. 37–51.
- [74] MØGELBERG, R. E. *Categorical and Domain Theoretic Models of Parametric Polymorphism*. PhD thesis, IT University of Copenhagen, Copenhagen, Denmark, 2005.
- [75] MONNIER, S., AND SHAO, Z. Typed regions. Tech. Rep. YALEU/DCS/TR-1242, Dept. of Computer Science, Yale University, New Haven, CT, 2002.
- [76] MORRISETT, G., FELLEISEN, M., AND HARPER, R. Abstract models of memory management. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture (FPCA '95)* (La Jolla, California, United States, 1995), ACM Press, pp. 66–77.

- [77] MORRISETT, G., WALKER, D., CRARY, K., AND GLEW, N. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems* 21, 3 (1999), 527–568.
- [78] NAUMANN, D. A. Predicate transformer semantics of a higher order imperative language with record subtyping. *Science of Computer Programming* 41, 1 (2001), 1–51.
- [79] NAUMANN, D. A. Soundness of data refinement for a higher order imperative language. *Theoretical Computer Science* 278 (2002), 271–301.
- [80] NECULA, G. C. Proof-carrying code. In *Proc. of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)* (Paris, France, January 1997), pp. 106–119.
- [81] NECULA, G. C., AND LEE, P. Safe kernel extensions without run-time checking. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI'96)* (Berkeley, CA, USA, October 1996), pp. 229–243.
- [82] NIPKOW, T., PAULSON, L. C., AND WENZEL, M. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, vol. 2283 of LNCS. Springer, 2002.
- [83] O'HEARN, P., AND PYM, D. J. The logic of bunched implications. *Bulletin of Symbolic Logic* 5, 2 (June 1999).
- [84] O'HEARN, P. W. Resources, concurrency and local reasoning. In *Proceedings of the 15th International Conference on Concurrency Theory (CONCUR'04)* (London, England, September 2004), vol. 3170 of LNCS, pp. 49–67.
- [85] O'HEARN, P. W., REYNOLDS, J. C., AND YANG, H. Local reasoning about programs that alter data structures. In *Proceedings of the Annual Conference of the European Association for Computer Science Logic (CSL 2001)* (Berlin, Germany, September 2001).
- [86] O'HEARN, P. W., AND TENNENT, R. D. Parametricity and local variables. In *Algol-like Languages, Volume 2*, P. W. O'Hearn and R. D. Tennent, Eds. Birkhauser, 1997, ch. 16, pp. 109–163.
- [87] O'HEARN, P. W., YANG, H., AND REYNOLDS, J. C. Separation and information hiding (work in progress). Extended version of [88], 2003.
- [88] O'HEARN, P. W., YANG, H., AND REYNOLDS, J. C. Separation and information hiding. In *Proceedings of the 31st ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL'04)* (Venice, Italy, 2004), pp. 268–280.
- [89] OLES, F. J. *A Category-Theoretic Approach to the Semantics of Programming Languages*. PhD thesis, Syracuse University, Syracuse, NY, USA, 1982.
- [90] OWICKI, S., AND GRIES, D. An axiomatic proof technique for parallel programs. *Acta Informatica* 6, 4 (1976), 319–340.
- [91] PARKINSON, M., AND BIERMAN, G. Separation logic and abstraction. In *Proceedings of the 32nd Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL'05)* (Long Beach, CA, USA, January 2005), pp. 247–258.

- [92] PARKINSON, M. J. *Local Reasoning for Java*. PhD thesis, University of Cambridge, October 2005.
- [93] PARNAS, D. L. Information distribution aspects of design methodology. In *Information Processing Conference 1971 (1972)*, pp. 339–344.
- [94] PARNAS, D. L. On the criteria to be used in decomposing systems into modules. *Communications of the ACM* 15, 12 (1972), 1053–1058.
- [95] PARNAS, D. L. The secret history of information hiding. In *Software Pioneers: Contributions to Software Engineering*, M. Broy and E. Denert, Eds. Springer-Verlag, New York, NY, USA, 2002, pp. 399–409.
- [96] PETERSEN, L., HARPER, R., CRARY, K., AND PFENNING, F. A type theory for memory allocation and data layout. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'03)* (New Orleans, LA, USA, January 2003), pp. 172–184.
- [97] PIERCE, B. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.
- [98] PITTS, A. M. Categorical logic. In *Handbook of Logic in Computer Science, Volume 5: Algebraic and Logical Structures*, S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, Eds. Clarendon Press, Oxford, 2001, ch. 2.
- [99] PITTS, A. M. Tripos theory in retrospect. *Mathematical Structures in Computer Science* 12, 3 (2002), 265–279.
- [100] PIXLEY, C. An incremental garbage collection algorithm for multimutator systems. *Distributed Computing* 3, 1 (1988), 41–50.
- [101] PLOTKIN, G. D. Lambda-definability in the full type hierarchy. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, J. Hindley and J. Seldin, Eds. Academic Press, 1980, pp. 365–373.
- [102] PYM, D. *The Semantics and Proof Theory of the Logic of Bunched Implications*, vol. 26 of *Applied Logics Series*. Kluwer, 2002.
- [103] PYM, D. J. Errata and remarks for the semantics and proof theory of the logic of bunched implications. Addendum to [102]. Available at <http://www.cs.bath.ac.uk/~pym/>.
- [104] PYM, D. J., O'HEARN, P. W., AND YANG, H. Possible worlds and resources: the semantics of BI. *Theoretical Computer Science* 315, 1 (2004), 257–305.
- [105] REDDY, U. Objects and classes in Algol-like languages. In *Proc. of The Fifth International Workshop on Foundations of Object-Oriented Languages (FOOL'98)* (San Diego, CA, USA, January 1998).
- [106] REDDY, U., AND YANG, H. Correctness of data representations involving heap data structures. *Science of Computer Programming* 50, 1 (March 2004), 129–160. Extended version of [107].

- [107] REDDY, U. S., AND YANG, H. Correctness of data representations involving heap data structures. In *Proc. of the 12th European Symposium on Programming, ESOP 2003* (2003), P. Degano, Ed., Springer Verlag, pp. 223–237.
- [108] REYNOLDS, J. C. *The Craft of Programming*. Prentice-Hall International Series in Computer Science. Prentice-Hall, London, 1981.
- [109] REYNOLDS, J. C. The essence of algol. In *Algorithmic Languages: Proceedings of the International Symposium on Algorithmic Languages* (Amsterdam, the Netherlands, October 1981), J. W. de Bakker and J. C. van Vliet, Eds., North-Holland Publishing Co., pp. 345–372.
- [110] REYNOLDS, J. C. Idealized algol and its specification logic. In *Tools and Notions for Program Construction*, D. Nel, Ed. Cambridge University Press, Cambridge, UK, 1982, pp. 121–161.
- [111] REYNOLDS, J. C. Design of the programming language Forsythe. Report CMU-CS-96-146, Carnegie Mellon University, Pittsburgh, Pennsylvania, June 1996.
- [112] REYNOLDS, J. C. *Theories of Programming Languages*. Cambridge University Press, 1998.
- [113] REYNOLDS, J. C. Intuitionistic reasoning about shared mutable data structures. In *Millennial Perspectives in Computer Science*, J. Davies, B. Roscoe, and J. Woodcock, Eds. Palgrave, Houndsmill, Hampshire, 2000, pp. 303–321.
- [114] REYNOLDS, J. C. The meaning of types — from intrinsic to extrinsic semantics. Research Series RS-00-32, BRICS, DAIMI, Department of Computer Science, University of Aarhus, December 2000. <http://www.brics.dk/RS/00/32/>.
- [115] REYNOLDS, J. C. Separation logic: A logic for shared mutable data structures. In *Proc. of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS'02)* (Copenhagen, Denmark, July 2002), IEEE Press, pp. 55–74.
- [116] ROBINSON, E. P., AND ROSOLINI, G. Reflexive graphs and parametric polymorphism. In *Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science (LICS 1994)* (Paris, France, July 1994), IEEE Press, pp. 364–371.
- [117] RUSSINOFF, D. M. A mechanically verified incremental garbage collector. *Formal Aspects of Computing* 6 (1994), 359–390.
- [118] SCHWARZ, J. Generic commands - A tool for partial correctness formalisms. *Computer Journal* 20, 2 (1977), 151–155.
- [119] SEELY, R. A. G. Locally cartesian closed categories and type theories. *Math. Proc. Cambridge Phil. Soc.* 95 (1984), 33–48.
- [120] SILBERSCHATZ, A., AND GALVIN, P. *Operating Systems Concepts*, Fifth ed. World Student Series. Addison-Wesley, Reading, MA, USA, 1998.
- [121] SMITH, F., WALKER, D., AND MORRISETT, G. Alias types. In *Proceedings of the 9th European Symposium on Programming Languages and Systems (ESOP'00)* (Berlin, Germany, March 2000), pp. 366–381.

- [122] STOYLE, G., HICKS, M., BIERMAN, G., SEWELL, P., AND NEAMTIU, I. Mutatis mutandis: Safe and predictable dynamic software updating. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'05)* (Long Beach, CA, USA, 2005), ACM Press, pp. 183–194.
- [123] SWARUP, V., FARMER, W. M., GUTTMANN, J. D., MONK, L. G., AND RAMSDELL, J. D. The VLISP byte-code interpreter. Tech. Rep. M 92B096, The MITRE Corporation, 1992.
- [124] TAN, G., APPEL, A. W., SWADI, K. N., AND WU, D. Construction of a semantic model for a typed assembly language. In *Proc. of the Fifth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI '04)* (Venice, Italy, January 2004).
- [125] TOFTE, M., AND BIRKEDAL, L. A region inference algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 20, 4 (July 1998), 734–767.
- [126] TOFTE, M., BIRKEDAL, L., ELSMAN, M., AND HALLENBERG, N. A retrospective on region-based memory management. *Higher-Order Symbolic Computation* 17, 3 (September 2004), 245–265.
- [127] TOFTE, M., AND TALPIN, J.-P. Implementation of the call-by-value lambda-calculus using a stack of regions. In *Proc. of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'94)* (Portland, OR, USA, 1994), pp. 188–201.
- [128] VAN OOSTEN, J. Basic category theory. Tech. Rep. LS-95-1, BRICS, January 1995.
- [129] WADLER, P. L. Analysis of an algorithm for real time garbage collection. *Communications of the ACM* 19, 9 (September 1976), 491–500.
- [130] WANG, D., AND APPEL, A. W. Type preserving garbage collectors. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'01)* (London, United Kingdom, 2001), pp. 166–178.
- [131] WEBER, T. Towards mechanized program verification with separation logic. In *Annual Conference of the European Association for Computer Science Logic (CSL'04)* (Karpacz, Poland, September 2004).
- [132] WINSKEL, G. *The Formal Semantics of Programming Languages: an introduction*. MIT Press, Cambridge, MA, USA, 1993.
- [133] WIRTH, N. Program development by stepwise refinement. *Communications of the ACM* 14, 4 (April 1971), 221–227.
- [134] YANG, H. *Local Reasoning for Stateful Programs*. PhD thesis, University of Illinois, Urbana-Champaign, 2001.
- [135] YANG, H. Relational separation logic. *Theoretical Computer Science* (2004). Submitted.
- [136] YANG, H., AND O'HEARN, P. A semantic basis for local reasoning. Springer-Verlag, pp. 402–416. Proceedings of the 5th International Conference on Foundations of Software Science and Computation Structures (FOSSACS'02).