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

Resource Reasoning and Labelled Separation Logic

Mohammad Raza

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

Abstract

This thesis develops resource reasoning with separation logic in the areas of modular program specification, program optimization, and concurrency verification for heap-manipulating programs.

In the first part, we investigate the resources that are required for modular and complete program specifications. Since the safety footprints of a program (the resources required for safe execution) do not always yield complete specifications, we first characterize the notion of the *relevance footprint*. We show that the relevance footprints are the only elements essential for a complete specification, and also identify the conditions for sufficiency. We then introduce a novel semantic model of heaps which establishes the correspondence between safety and relevance footprints, and we identify a general condition that guarantees this correspondence in arbitrary resource models.

In the second part, we present *labelled separation logic* for introducing optimizations such as automatic parallelization in heap manipulating programs. In order to detect dependences between distant statements in a program, we annotate spatial conjuncts in separation logic formulae with the labels of accessing commands, and propagate these labels through program proofs. We also identify the notion of 'allocation dependences' which, in addition to standard stack and heap dependences, are needed to ensure the safety of optimizations.

In the final part, we address the analysis of resource ownership transfers in concurrent programs, and present an algorithm for automating concurrent separation logic proofs. This is based on a form of labelled separation logic in which ownership constraints are tracked through a proof and ownership is inferred from heap accesses at arbitrary program points. Unlike previous methods, the algorithm presented here does not require user annotations about ownership distribution, and we demonstrate how it can verify programs that could not be handled by previous methods.

Declaration of Originality

This thesis is my own work and all related work is appropriately referenced.

Mohammad Raza

Acknowledgements

I would like to thank God for everything, including logic. Thank you to my parents, my wife and my family for all of their infinite love and support.

I would like to thank my supervisor, Philippa Gardner, for her invaluable support and friendship, and my second supervisor, Cristiano Calcagno, for his encouragement and advice. I am also very grateful to Peter O'Hearn and Byron Cook, for taking interest in my work and making this journey so much more interesting.

I would also like to thank the following people for very helpful and enjoyable discussions about this and other work: Josh Berdine, Lars Birkedal, Thomas Dinsdale-Young, Dino Distefano, Mike Dodds, Alexey Gotsman, Chris Hankin, Laurie Hendren, Suresh Jagannathan, Stephen Magill, Matthew Parkinson, Simon Peyton-Jones, Jiri Simsa, Satnam Singh, Gareth Smith, Don Syme, Viktor Vafeiadis, Mark Wheelhouse, Adam Wright and Hongseok Yang. Finally, I would like to thank the Engineering and Physical Sciences Research Council and the Overseas Research Studentship scheme for awarding me the funding to undertake this research.

for my parents and fatima and sana

Contents

List of Figures

Chapter 1

Introduction

This thesis is about formal, logic-based approaches for ensuring the quality of computer programs, in terms of both reliability and performance. In general, software today is unreliable and susceptible to errors and viruses. Standard testing techniques cannot provide guarantees, as testing can only verify a limited set of possible executions. Performance is also a factor of concern as heavier demands are made on computer systems. Becoming increasingly important is the exploitation of software parallelism, as manufacturers turn to developing multicore architectures which provide increased concurrency instead of increased clock speed. There is therefore a growing need for robust and powerful methods for the verification and optimization of computer programs. Formal approaches based on the rigour of mathematical logic hold the promise of delivering such methods.

A central concern in ensuring the quality of computer systems is the correct and efficient management of the resources that are available to the system. The focus here is especially on heap manipulation in imperative programs with dynamic memory allocation and pointer data structures. Such programs have been out of the reach of tractable analyses based on formal logics, but a significant advance has recently been made with *separation logic*, which is a program logic based on the idea of reasoning spatially about resource. In analogy with temporal logic, in which statements can describe *when* a property holds, statements in separation logic can describe *where* a property holds. This approach has led to elegant proofs of program correctness, as well as modular and scalable automated verification methods. However, deeper properties of how resources are manipulated in programs have remained out of the reach of existing methods based on separation logic:

- what are the resources needed to describe the behaviour of a program, so that we can provide modular and complete program specifications?
- how can we detect the resources accessed in different parts of a program, so that the program may be executed more efficiently?
- how can we detect how resources are shared between different processes, so that we can verify concurrent programs?

In this thesis we develop resource reasoning with separation logic to address these questions. We first give a background on Hoare logic and separation logic, and then describe the motivation and contributions of the thesis in more detail.

1.1 Background

Hoare Logic A rigorous approach to program verification is provided by Hoare logic [19, 27], in which formal proofs of program correctness are constructed using mathematical logic. Such proofs guarantee the properties for all possible runs rather than a few test cases. In Hoare logic, program properties are specified as 'Hoare triples' of the form $\{P\}\ C\ \{Q\}$, where C is a command and P and Q are logical formulae that describe the pre- and post-conditions of the command. A Hoare triple may have a partial or total correctness interpretation. The partial correctness interpretation of the triple $\{P\}\ C\ \{Q\}$ is that if P is true before the execution of C, then Q is true after the execution if C terminates. The total correctness interpretation in addition guarantees the termination of C. In his original paper, Hoare provided an axiomatic treatment for a simple imperative programming language based on partial correctness triples of this form. He demonstrated how formal proofs of properties can be expressed as triples, and formally proved by logical inference using axioms and inference rules such as the following:

Axiom of Assignment: $\{P[E/x]\}\ x := E\ \{P\}$

Rule of Consequence: $$\frac{P' \Rightarrow P \quad \{P\}\ C\ \{Q\} \quad Q \Rightarrow Q'}{\{P'\}\ C\ \{Q'\}}$$

Rule of Composition: $$\frac{\{P\}\ C1\ \{Q\} \quad \{Q\}\ C2\ \{R\}}{\{P\}\ C1; C2\ \{R\}}$$

The Hoare logic approach to verification has been extensively studied and extended for a variety of programming constructs. However, the approach is rarely used in practice, mainly because the inference methods do not scale well to realistic programs. The situation is worse in the presence of pointers and dynamic data structures, since assertions in first-order logic describe the global state of memory and so intricate aliasing relationships between pointers must be expressed to be able to prove program properties. This leads to an uncontrollable complexity of specifications and proofs, and a breakdown in modularity of reasoning.

Separation logic Separation logic was introduced by O'Hearn, Reynolds and Yang [51, 41, 32] to address the difficulties encountered by Hoare logic in reasoning about pointer programs. The central idea here is to reason spatially about resources (such as the program's heap) by introducing a new connective to first-order logic, which is the spatial separating conjunction $*$. This connective allows properties of separate resources to be specified in isolation from one another. Thus, when reasoning about heap-manipulating programs for example, a formula $P*Q$ asserts that the memory heap can be separated into two disjoint parts, one of which satisfies P and the other satisfies Q. The other spatial connective added by separation logic is the adjoint of the separating conjunction, which is the separating implication $-\!\!*$. A formula $P -\!\!* Q$ asserts that if the current heap is added to any heap satisfying P, then the resulting heap will satisfy Q. Such assertions allow one to easily and concisely express properties of heap layout and aliasing, which are the cause of the difficulty of reasoning in classical logic. For example, we may have the formula $\texttt{list}(x) * \texttt{list}(y)$, which asserts that the heap consists of two disjoint linked lists at x and y. In contrast, an assertion $\texttt{list}(x) \wedge \texttt{list}(y)$ in classical logic only describes the presence of two lists in the heap which may or may not share any number of heap cells.

Furthermore, the spatial assertion language of separation logic also facilitates *local reasoning* about programs, in which specifications and proofs can focus on the resources that are relevant to a program, instead of having to describe the global state of the system. Local reasoning is natural because of the local way in which programs behave: there are certain resources that a program accesses and are required for safe execution, which is known as the program's *safety footprint*, and any other resource is unaffected by the program.

In line with this local behaviour of programs, the interpretation of a Hoare triple $\{P\}\ C\ \{Q\}$ in separation logic extends the standard interpretation with the additional constraint that the pre-condition P should include the safety footprint of the command C, so that there are no memory access faults when executing the command from this pre-condition. Moreover, one can now give *small specifications* for programs, in which the pre-condition only describes the safety footprint. For example, a small specification for the heap deallocation command is given as

$$\{l \mapsto \_\}\ \texttt{dispose}(l)\ \{\texttt{emp}\}$$

The pre-condition describes the safety footprint, which in this case is the single heap cell at location l that the command deallocates (the contents of the heap cell are unspecified as they are not accessed in the command's execution). Deallocating the cell results in a post-condition which is the empty heap. To infer the behaviour of commands on arbitrary larger states, one can use the main inference rule of local reasoning known as the *frame rule*:

$$\frac{\{P\}\ C\ \{Q\}}{\{R * P\}\ C\ \{R * Q\}}$$

The frame rule encodes the local behaviour of programs: if the command C executes safely on pre-condition P, then it does not access any other resources, and hence the part of the heap described by formula R will remain unchanged after the execution (note that there may be additional side conditions associated with the rule depending on the specific form of the logic being used). For example, given the small specification for dispose, we may construct the following proof using the frame rule and the rule for composition:

$$\cfrac{\cfrac{\{l \mapsto \_\}\ \texttt{dispose}(l)\ \{\texttt{emp}\}}{\{l' \mapsto \_ \ * \ l \mapsto \_\}\ \texttt{dispose}(l)\ \{l' \mapsto \_\ \}}\ \text{Frame} \qquad \{l' \mapsto \_\}\ \texttt{dispose}(l')\ \{\texttt{emp}\}}{\{l' \mapsto \_ \ * \ l \mapsto \_\}\ \texttt{dispose}(l);\texttt{dispose}(l')\ \{\texttt{emp}\}}\ \text{Composition}$$

Such local reasoning with separation logic has led to elegant and modular proofs of some difficult programs [54, 41, 47].

Automated analysis with separation logic Apart from providing simple specifications and by-hand proofs, separation logic has also made significant advances in automated analysis of heap-manipulating programs. The foundation for this was laid by Berdine, Calcagno and O'Hearn in [3, 2], who developed a form of symbolic execution based on the proof rules of separation logic. The formulation of this symbolic execution method is based on the spatial nature of separation logic reasoning. For example, consider the axiom for the heap mutation command

$$\{P * x \mapsto [f : y]\}\ \ x \to f := z\ \ \{P * x \mapsto [f : z]\}$$

where the command accesses the heap cell at location x and updates the value of field f to z. The precondition, which asserts the presence of the cell at x (in which the field f initially has value y), is updated *in-place*, in a way that mirrors the imperative update of the actual heap that occurs during program execution. This in-place update can be done because the separating conjunction avoids the need for checking the possibility of aliasing in the global heap, because it separates the cell of interest from the rest of the heap.

To formulate separation logic in terms of this kind of symbolic execution, two restrictions are imposed on the kind of formulae that can be used. First, formulae are restricted to a format of the form $\Pi \wedge \Sigma$, where Π is a *pure formula* (describing properties of variables, independent of the heap) and Σ is a *spatial formula* describing the heap, which is a *-combination of heap predicates. The other restriction is that formulae do not describe the detailed contents of data structures, but only describe the shapes of heap structures (in the sense of shape analysis). For example, apart from the basic points-to predicate $x \mapsto \_$, we may have a shape predicate $\texttt{ls}(x, y)$ which describes a heap that is a linked list segment from x to y, or the predicate $\texttt{tree}(x)$ which describes a binary tree at x. These kind of formulae are commonly known as *symbolic heaps* in the literature, as their restricted format closely resembles the structure of concrete heaps.

For this restricted symbolic heap fragment, Berdine et al. developed a symbolic execution framework which, given user annotations for pre- and post-conditions and loop invariants, is able to automatically verify safety properties of heap manipulating programs, ensuring the absence of

```
new(x);               ║ with r when f = 1
with r when f = 0     ║ {
{                     ║   y := c;
  c := x;             ║   f := 0;
  f := 1;             ║ }
}                     ║ dispose(y);
```

Figure 1.1: Cell-transferring buffer

memory errors such as dereferences of dangling pointers and memory leaks. This basic foundation has been developed in many directions since, especially in the area of shape analysis. Firstly, abstraction (or widening) operators were added to symbolic execution to infer invariants automatically via a fixed-point calculation, in the usual manner of abstract interpretation [17, 36]. Advances in various other directions, especially on the issues of scalability, compositionality and data structures, have led to the first automatic proofs of pointer safety in entire industrial programs, with verification of Microsoft and Linux device drivers with up to 10,000 lines of code [55, 8, 1].

Concurrent Separation Logic The locality and modularity of reasoning provided by separation logic has also brought about important advances in the verification of concurrent programs. Concurrent program analysis is a difficult problem because of the need to consider possible interleavings between concurrent processes, which becomes even more complicated in the presence of aliasing and dynamic data structures in the heap. Concurrent Separation Logic (CSL), introduced by O'Hearn in [42], made a breakthrough in modular Hoare-style reasoning about concurrent heap-manipulating programs.

The basic idea behind CSL is the use of formulae called *resource invariants* to describe the heap that is shared between concurrent processes, and to hide away the interference between processes with the use of resource invariants in program proofs. We consider concurrrent programs in which synchronization is implemented using the *conditional critical region (CCR)* construct `with` r `when` B `do` C, where r is a resource name, B is a boolean condition, and C is a command. A thread executing this command waits until the boolean condition B is satisfied and no other CCR for r is executing, and then it executes the body C. Two CCRs for the same resource name cannot be executed simultaneously, which ensures that shared resources are accessed in mutual exclusion. For example, consider the program shown in figure 1.1. This program uses a single heap cell buffer shared between two threads, access to which is protected by resource name r. The buffer heap cell is pointed to by variable c and the flag variable f indicates whether the buffer is full or empty (the initial pre-condition of the program is that $f = 0$ and $c = \texttt{nil}$). The left thread allocates a new cell and places it in the buffer, and the right thread removes the cell from the buffer and disposes it.

CSL provides reasoning about such programs by associating each resource name r with a formula $I(r)$ called the resource invariant, which describes the state of the shared resources protected by r at all times. For the buffer example, the resource invariant specifies that it is always the case that either the buffer is empty and $f = 0$, or there is a single heap cell pointed to by c and $f = 1$, which is given by the formula

$$I(r) \overset{def}{=} \{(\texttt{emp} \wedge f = 0) \vee (c \mapsto \_ \wedge f = 1)\}$$

The idea is that the resource invariant defines a contract between threads about how the shared state must be maintained: inside critical regions, a thread will gain ownership of the shared state and expect it to satisfy the resource invariant. It can then work on the shared state in mutual exclusion, possibly invalidating the resource invariant, but then it must re-establish the invariant at the end of the critical region, before giving up ownership of the shared state. This is modelled by the inference rule for critical regions:

$$\frac{\{(P * I(r)) \wedge B\}\, C\, \{Q * I(r)\}}{\{P\}\, \texttt{with}\ r\ \texttt{when B}\ C\, \{Q\}} \ \text{no other process modifies variables free in } P \text{ or } Q$$

The body of the CCR has access to the shared state and the thread's local state P, and the invariant is re-established after execution along with some other post-condition Q. Outside the CCR, the resource invariant is 'hidden away', and reasoning proceeds without knowledge of the shared state. For example, in the left thread in the buffer program, after the $\texttt{new}(x)$ command creates $x \mapsto \_$, the CCR rule is used as follows

$$\frac{\dfrac{\{(x \mapsto \_ \wedge f = 0\}\quad c := x;\ f := 1;\quad \{x \mapsto \_ \wedge c = x \wedge f = 1\}}{\{(x \mapsto \_ * I(r)) \wedge f = 0\}\quad c := x;\ f := 1;\quad \{\texttt{emp} * I(r)\}}\ \text{rule of consequence}}{\{x \mapsto \_\}\ \ \texttt{with}\ r\ \texttt{when}\ \texttt{f} = \texttt{0}\ \{c := x;\ f := 1;\}\ \ \{\texttt{emp}\}}\ \text{CCR rule}$$

Hence, ownership of the cell allocated in the thread moves into the buffer after the CCR execution, and the thread is left with the empty post-condition $\texttt{emp}$. We can similarly derive the following specification for the CCR in the second thread

$$\{\texttt{emp}\}\ \ \texttt{with}\ r\ \texttt{when}\ \texttt{f} = \texttt{1}\,\{y := c;\ f := 0;\}\ \ \{y \mapsto \_\}$$

where ownership of the cell moves from the buffer into the thread, with post-condition $y \mapsto \_$, which the thread then safely disposes with the dispose command. By hiding away the resource invariant describing the shared state, the proof of each thread can proceed independently of other threads, giving the illusion of non-interference. This is modelled by the rule for parallel composition, which simply combines the local states of each of the threads:

$$\frac{\{P_1\}\,C_1\,\{Q_1\} \cdots \{P_n\}\,C_n\,\{Q_n\}}{\{P_1 * \cdots * P_n\}\,C_1 \parallel \cdots \parallel C_n\,\{Q_1 * \cdots * Q_n\}}\ \text{no variable free in } P_i \text{ or } Q_i \text{ is changed in } C_j \text{ when } j \neq i$$

In the buffer example, using the specifications derived for the CCRs (in which the resource invariant is hidden away), we get the specifications $\{\texttt{emp}\}\,T_i\,\{\texttt{emp}\}$ for each thread T_i. Using the parallel composition rule we get the overall specification $\{\texttt{emp}\}\,T_1 \parallel T_2\,\{\texttt{emp}\}$ for the parallel composition. Such a proof in CSL shows that there are no memory errors or data races in the program. The soundness of concurrent separation logic was shown by Brookes in [6].

1.2 Motivation and Contributions

Although resource reasoning with separation logic has made a significant advance in program verification, there have remained important questions about the nature of resource manipulation in relation to modular program specification, optimization and concurrency analysis. In this section we describe these issues and the contributions made in this thesis to address them.

1.2.1 Resources required for modular specifications

Consider again the small specification for the dispose command that we described above:

$$\{l \mapsto \_\}\ \texttt{dispose}(l)\ \{\texttt{emp}\}$$

where the pre-condition is the safety footprint of the command, that is, the minimum resource required for safe execution. This is a complete specification for the command since we can use the frame rule to infer the behaviour of the command on all larger states. This suggests that the safety footprint should be enough to specify the complete behaviour of any program. This, however, is not true in general, as shown by the following program:

$$AD ::= \quad \texttt{new}(x); \texttt{dispose}(x)$$

This *allocate-deallocate* program allocates a new cell with address stored in the stack variable x, and then deallocates the cell. The smallest heap on which the program is safe is the empty heap `emp`. The specification of the program on this pre-condition is:

$$\{\texttt{emp}\} \quad AD \quad \{\texttt{emp}\} \tag{1.1}$$

We can extend our reasoning to larger heaps by applying the frame rule. For example, extending to a one-cell heap with address l gives

$$\{l \mapsto \_\} \quad AD \quad \{l \mapsto \_\} \tag{1.2}$$

However, the specification 1.1 is not a complete specification of the AD program. For example, the following triple is also valid:

$$\{l \mapsto \_\} \quad AD \quad \{l \to \_ \wedge x \neq l\} \tag{1.3}$$

This is because if l is already allocated, then the new address stored in x cannot be l. But this triple is not derivable from 1.1. However, the triples 1.1 and 1.3 together do provide a complete specification from which all valid triples can be derived.

This example shows that the safety footprint of a program is not always sufficient to describe the complete behaviour of a program. Firstly, this opens the question of how the *relevance footprint* of a program should be defined, that is, the resources that are required for a complete specification of a program. We need to know if such a notion exists and what the formal definition and properties of relevance footprints would be, in order to determine how to construct complete specifications for arbitrary programs. Secondly, it seems unnatural that the safety footprint is not sufficient to describe all the properties of a program, since the local behaviour of programs indicates that they act independently of any resources that they do not access. In the AD program, for example, it is not clear why we should need to refer to the single-cell heaps in the pre-condition in order to describe the program's complete behaviour, even though these cells are not accessed by the program. The need to mention these additional resources seems to bring some redundancy to specifications and makes them more cumbersome and less modular. It poses the question of whether it is possible to formulate a more natural framework of local reasoning in which complete specifications can be obtained from the safety footprint.

Contributions In Chapter 2, we investigate relevance footprints in the generic framework of local reasoning introduced in [11], where programs are modelled as local functions that act on monoids representing abstract models of resource. We introduce the formal definition of the relevance footprint of a local function, and show how this definition provides the correct footprints for a complete specification of the AD example discussed earlier. We then prove the central result characterizing relevance footprints that, for any local function, the relevance footprints are the only elements which are *essential* for a complete specification of the function.

In section 2.4 we investigate the question of *sufficiency*: the conditions under which a smallest complete specification for a local function can be constructed using only the relevance footprints. For well-founded resource models such as the standard heap model (where there are no infinite descending chains of smaller resources), we show that the relevance footprints are always sufficient. In the non-well-founded case, we find that sufficiency depends on the presence of negativity in the resource model, which is when non-empty elements of resource can be combined to produce the empty state. For models without negativity, such as heaps with infinitely divisible fractional permissions [4], we show that either the relevance footprints are sufficient for a complete specification or a smallest complete specification does not exist. For models with negativity, such as the integers under addition, we show that it is possible to construct smallest complete specifications using non-essential elements.

In the final section, we apply the theory of relevance footprints to investigate the issue of regaining the correspondence between safety footprints and relevance footprints. We present an alternative model of heaps, and prove that in this new model the relevance footprint of *every* program, including AD, corresponds to the safety footprint. Furthermore, we identify a general condition on the primitive commands of a programming language under which this correspondence holds in arbitrary resource models.

```
          listInit($x$) {
               local $x_1$;
               if ($x \neq$ nil) {
$l_1$ :              $x_1 := x \rightarrow next$;
$l_2$ :              $x \rightarrow f_1 :=$ nil;
$l_3$ :              $x \rightarrow f_2 :=$ nil;
$l_4$ :              listInit($x_1$);
               }
          }
```

Figure 1.2: Linked list traversal program

The results in this chapter were first published in [49]. The journal version containing proofs and final section on the regaining of safety footprints appeared in [50].

1.2.2 Resource dependence detection for optimization

Optimization techniques are generally based on a detection of how resources are accessed in different parts of the program, as this information can be used to effect optimizations such as parallelizing statements that access separate resources, or reordering statements to improve temporal locality of reference. Such techniques have been extensively studied and successfully applied for programs with simple data types and arrays, but there has been very limited progress for programs that manipulate pointers and dynamic data structures. The difficulty is that dynamically-allocated heap locations are not named by program variables or array indexes, and it is therefore difficult to detect when two distant statements in a program may access the same heap locations. Even the number of heap locations may not be statically determined when programs operate on dynamically-allocated pointer data structures, such as linked lists of arbitrary size. For example, figure 1.2 shows a recursive procedure that traverses a nil-terminated linked list of unknown length. While the list is non-empty, the procedure first sets x_1 to the next element in the list (pointed at by the *next* field), and sets fields f_1 and f_2 in the current cell to `nil`, and then makes the recursive call on the tail of the list.

In this case the statements l_2 and l_3 access the heap location which is the head of the linked list and the recursive call at l_4 accesses all locations in the tail of the list, and hence it is possible to execute the recursive call in parallel with statements l_2 and l_3. Although separation logic has provided tractable and scalable verification techniques for such programs, these analyses cannot be used for dependence detection such as in the case of this program. The reason for this is that separation logic assertions only describe the spatial separation and memory layout at a single program point, and so a region of memory may be described by different formulae at different program points. To be able to detect dependences between distant statements in the program, we need to be able to relate the heap states at arbitrary points throughout the execution of the program, such as detecting that statements l_4 and l_2 access separate heap locations in the list traversal program.

Contributions In chapter 3 we extend separation logic to address the optimization of heap-manipulating programs, by detecting dependences between program statements. The main conceptual step is to express memory separation properties throughout a program's lifetime by annotating separation-logic formulae with *labels* of accessing commands. Symbolic execution based on separation logic is extended so that the memory region accessed by a command is marked with the label of the command. This label tracking is directly determined for primitive commands, but for composite commands such as procedure calls or while loops, the label tracking is determined by an adaptation of a *frame inference* method from [3], which allows us to infer the part of a formula that is not accessed by a command. We discuss examples of the kind of program optimizations we can perform and present experimental results on performance improvement in the area of hardware synthesis from heap-manipulating programs.

Although our focus is on the use of separation logic to determine heap dependences between commands, the ultimate aim is to show that any optimization based on the inferred dependences will produce the same output states as the original program. Program optimizations have often been proposed based on the *assumption* that if two commands access separate heap and variables in all possible executions, then they can be parallelized to give an equivalent program. We describe here how this assumption actually does not hold in the presence of dynamic memory allocation and deallocation, and optimizations based on the assumption can produce results that are significantly different from the original program. We discuss example programs to illustrate the problem, and introduce the notion of *allocation dependences*, in addition to heap and stack dependences, in order to guarantee the safety of optimizations. We formally demonstrate the soundness of our optimizations using an action trace semantics of programs.

A preliminary version of the work in this chapter was presented in [48, 14].

1.2.3 Resource sharing in concurrent programs

The detection of how resources are shared between parallel processes is essential for the successful analysis of concurrent programs. This is especially difficult for so-called *daring* concurrent programs, where resources may be accessed by concurrent processes outside of critical regions, and ownership of shared resources is dynamically transferred during execution. For example, consider the cell-transferring buffer program from Figure 1.1, in which the single heap cell in the buffer is shared between two threads. The program illustrates the essence of dynamic ownership transfer: there is no static separation of the heap into the heap that is owned by the left thread, the heap owned by the right thread, and the shared resource which is the buffer cell. Instead, there is a single heap cell whose ownership moves between the threads and the shared state in the course of the execution: the cell is allocated in the left thread, its ownership is then transferred into the shared buffer, and it then comes out of the buffer and into the right thread, which finally disposes it.

Also notice that this notion of ownership transfer is not a construct of the programming language, or something that is explicitly defined in the program. It is instead an implicit property of the concurrent program brought about by how threads agree to safely share resources without causing memory errors or data races. Indeed, only changing the way in which the threads access the buffer cell may bring about a different pattern of ownership transfer. For example, the program in figure 1.3 is obtained by moving the disposal of the heap cell from the right thread to the left thread. In this case, even though the critical regions are exactly as before, the ownership of the cell remains in the left thread, since it is now the one that disposes it. Ownership never gets transferred to the right thread, and this is fine since the right thread never makes a heap access. This program can therefore be seen as an *address-transferring buffer* program, since the right thread is only able to read the value of the address of the buffer cell and cannot access the cell itself. As an example of an unsafe ownership policy, if both threads were to dispose the cell, then this would result in a data race and a double dispose.

Inferring the ownership policy of a program is therefore essential for successful analysis of such programs and to ensure the absence of memory errors or data races. Concurrent separation logic provides a good framework for such analysis, where the ownership policy of a program can be explicitly specified in the resource invariants. For example, the programs in figures 1.1 and 1.3

```
new(x);                ||
with r when f = 0      ||  with r when f = 1
{                      ||  {
   c := x;             ||     y := c;
   f := 1;             ||     f := 0;
}                      ||  }
dispose(x);            ||
```

Figure 1.3: Address-transferring buffer

can each be verified in CSL with a different choice of resource invariant describing the appropriate ownership policy. The cell-transferring program has resource invarant $\{(f = 0 \wedge \texttt{emp}) \vee (f = 1 \wedge c \mapsto \texttt{nil})\}$, which specifies that the cell is owned by the buffer when the flag is set. The address-transferring program has invariant $I(r) \overset{def}{=} \{(f = 0 \wedge \texttt{emp}) \vee (f = 1 \wedge \texttt{emp})\}$, which specifies that the buffer never obtains ownership of the cell. Given these invariants, correctness of each program can be verified with the inference rules of CSL. Thus the problem of ownership inference can be posed as the ability to infer the resource invariants for a concurrent program, so that a program proof in concurrent separation logic can be automated. Thus far, methods proposed for automating CSL [23, 9] fail on simple programs due to problems with ownership inference.

Contributions In chapter 4 we present a new method for inferring resource invariants for the automation of concurrent separation logic proofs. This method addresses the ownership inference problem using a resource labelling technique that is similar in nature to the one used for the dependence analysis in chapter 3. It extends the fixpoint method of [23] with a form of labelling in which *ownership constraints* are propagated through a program proof, which allows ownership of shared resources to be inferred from heap accesses made at possibly arbitrary program points.

We demonstrate how the algorithm verifies programs which both of the previous approaches [23, 9] could not handle. Also unlike the previous methods, our algorithm does not require user annotations about ownership distribution in the pre-condition of the concurrent program, as it is able to infer this automatically. We also present a generic technique for programs with while loops, which is parametric in any resource-invariant inference method for loop-free programs.

Chapter 2

Footprints and Complete Specifications

In this chapter, we investigate the resources that are required to construct complete specifications for programs. For generality, the discussion is based on the abstract separation logic framework of [11], where programs are modelled as local functions that act on abstract models of resource represented by partial commutative monoids. We start by giving a background on abstract separation logic and then formulate the notion of complete specifications for programs in section 2.2. In section 2.3, the formal definition of the relevance footprint is introduced, based on the definition of locality of functions, and we prove the central result that the relevance footprints are the only essential elements required for a complete specification. In section 2.4, we give results about the sufficiency of relevance footprints, which depend on properties of the resource models.

In section 2.5, we explore how a correspondence between safety and relevance footprints may be regained. A new heap model is presented in which the relevance footprint of *every* program is the safety footprint. We also identify a general condition on the primitive commands of a programming language under which this correspondence holds in arbitrary resource models.

2.1 Background

We begin with a description of the abstract separation logic framework introduced in [11]. Separation logic reasoning has been applied to several memory models, including heaps based on pointer arithmetic [41], heaps with permissions [4], and the combination of heaps with variable stacks which views variables as resource [5, 46]. In each case, the basic soundness and completeness results for local Hoare reasoning are similar. For this reason, Calcagno, O'Hearn and Yang [11] characterised the underlying principles of local reasoning, introducing the notion of local functions that act on abstract resource models called separation algebras. This generalises the specific examples of local imperative commands and memory models. They introduce abstract separation logic for local reasoning in this abstract setting, and give general soundness and completeness results. However, a formal understanding of relevance footprints is missing in this abstract theory, and we will provide such a formulation in this chapter of the thesis.

Separation Algebras and Local Functions Separation algebras provide a model of resource which generalises over the specific heap models used in various applications of separation logic. Informally, a separation algebra models resource as a set of elements that can be 'glued' together to create larger elements. The 'glueing' operator satisfies properties in accordance with this resource intuition, such as commutativity and associativity, as well as the cancellation property which requires that 'ungluing' a certain portion from a resource element gives us a unique element.

Definition 2.1 (Separation Algebra) *A* **separation algebra** *is a cancellative, partial commutative monoid* $(\Sigma, \bullet, u)$*, where* Σ *is a set and* $\bullet$ *is a partial binary operator with unit* u*. The operator satisfies the familiar axioms of associativity, commutativity and unit, using a partial equality on* Σ *where either both sides are defined and equal, or both are undefined. It also satisfies the cancellative property stating that, for each* $\sigma \in \Sigma$*, the partial function* $\sigma \bullet (\cdot) : \Sigma \rightharpoonup \Sigma$ *is injective.*

We shall sometimes overload notation, using Σ to denote the separation algebra $(\Sigma, \bullet, u)$. Examples of separation algebras include multisets with union and unit $\emptyset$, the natural numbers with addition and unit 0, heaps as finite partial functions from locations to values ([11] and example 1), heaps with permissions [11, 4], and the combination of heaps and variable stacks enabling us to model programs with variables as local functions ([11], [46] and example 1). These examples all have an intuition of resource, with $\sigma_1 \bullet \sigma_2$ intuitively giving more resource than just σ_1 and σ_2 for $\sigma_1, \sigma_2 \neq u$. However, notice that the general notion of a separation algebra also permits examples which may not have this resource intuition, such as $\{a, u\}$ with $a \bullet a = u$. Since our aim is to investigate general properties of local reasoning, our inclination is to impose minimal restrictions on what counts as resource and to work with this simple definition of a separation algebra.

Definition 2.2 (Separateness and substate) *Given a separation algebra $(\Sigma, \bullet, u)$, the* **separateness** *(#) relation between two states $\sigma_0, \sigma_1 \in \Sigma$ is given by $\sigma_0 \# \sigma_1$ if and only if $\sigma_0 \bullet \sigma_1$ is defined. The* **substate** *($\preceq$) relation is given by $\sigma_0 \preceq \sigma_1$ if and only if $\exists \sigma_2. \sigma_1 = \sigma_0 \bullet \sigma_2$. We write $\sigma_0 \prec \sigma_1$ when $\sigma_0 \preceq \sigma_1$ and $\sigma_0 \neq \sigma_1$.*

Lemma 2.1 (Subtraction) *For $\sigma_1, \sigma_2 \in \Sigma$, if $\sigma_1 \preceq \sigma_2$ then there exists a unique element denoted $\sigma_2 - \sigma_1 \in \Sigma$, such that $(\sigma_2 - \sigma_1) \bullet \sigma_1 = \sigma_2$.*

Proof: *Existence follows by definition of $\preceq$. For uniqueness, assume there exist $\sigma', \sigma'' \in \Sigma$ such that $\sigma' \bullet \sigma_1 = \sigma_2$ and $\sigma'' \bullet \sigma_1 = \sigma_2$. Then we have $\sigma' \bullet \sigma_1 = \sigma'' \bullet \sigma_1$, and thus by the cancellation property we have $\sigma' = \sigma''$.* ■

We consider functions on separation algebras that generalise imperative programs operating on heaps. Such programs can behave non-deterministically, and can also *fault*. To model non-determinism, we consider functions from a separation algebra Σ to its powerset $\mathcal{P}(\Sigma)$. To model faulting, we add a special top element $\top$ to the powerset. We therefore consider total functions of the form $f : \Sigma \rightarrow \mathcal{P}(\Sigma)^\top$. On any element of Σ, the function can either map to a set of elements, which models *safe* execution with non-deterministic outcomes, or to $\top$, which models a faulting execution. Mapping to the empty set represents divergence (non-termination).

Definition 2.3 *The standard subset relation on the powerset is extended to $\mathcal{P}(\Sigma)^\top$ by defining $p \sqsubseteq \top$ for all $p \in \mathcal{P}(\Sigma)^\top$. The binary operator $*$ on $\mathcal{P}(\Sigma)^\top$ is given by*

$$\begin{aligned} p * q &= \{\sigma_0 \bullet \sigma_1 \mid \sigma_0 \# \sigma_1 \wedge \sigma_0 \in p \wedge \sigma_1 \in q\} \quad \textit{if } p, q \in \mathcal{P}(\Sigma) \\ &= \top \quad otherwise \end{aligned}$$

$\mathcal{P}(\Sigma)^\top$ is a total commutative monoid under $*$ with unit $\{u\}$.

Definition 2.4 (Function ordering) *For functions $f, g : \Sigma \rightarrow \mathcal{P}(\Sigma)^\top$, $f \sqsubseteq g$ if and only if $f(\sigma) \sqsubseteq g(\sigma)$ for all $\sigma \in \Sigma$.*

We shall only consider functions that act *locally* with respect to resource. For imperative commands on the heap model, the locality conditions were first characterised in [56], where they were used to prove soundness of local reasoning for the specific heap model. The conditions identified were

- *Safety monotonicity*: if the command is safe on some heap, then it is safe on any larger heap.
- *Frame property*: if the command is safe on some heap then, in any outcome of applying the command on a larger heap, the additional heap portion will remain unchanged by the command.

In [11], these two properties were amalgamated to provide the following definition of a local function acting on a separation algebra.

Definition 2.5 (Local Function) *A* **local function on** Σ *is a total function* $f : \Sigma \rightarrow \mathcal{P}(\Sigma)^{\top}$ *which satisfies the* **locality condition***:*

$$\sigma \# \sigma' \ \textit{ implies } \ f(\sigma' \bullet \sigma) \sqsubseteq \{\sigma'\} * f(\sigma)$$

We let $LocFunc$ *be the set of local functions on* Σ*.*

Intuitively, we think of a command to be local if, whenever the command executes safely on any resource element, then the command will not 'touch' any additional resource that may be added to the initial state. Safety monotonicity follows from the above definition because, if f is safe on σ (that is, $f(\sigma) \sqsubset \top$), then it is safe on any larger state, since $f(\sigma' \bullet \sigma) \sqsubseteq \{\sigma'\} * f(\sigma) \sqsubset \top$.

The frame property follows by the fact that when the additional state σ' is added to σ, any output state is in the set $\{\sigma'\} * f(\sigma)$, and so the additional state σ' is unchanged. However, we note that the $\sqsubseteq$ ordering allows for reduced non-determinism on larger states. This, for example, is the case for the AD command from the introduction which allocates a cell, assigns its address to stack variable x, and then deallocates the cell. On the empty heap, its result would allow all possible values for variable x. However, on the larger heap where cell 1 is already allocated, its result would allow all values for x except 1, and we therefore have a more deterministic outcome on this larger state.

Lemma 2.2 *Locality is preserved under sequential composition, non-deterministic choice and Kleene-star, which are defined as*

$$(f;g)(\sigma) = \begin{cases} \top & \text{if } f(\sigma) = \top \\ \bigsqcup\{g(\sigma') \mid \sigma' \in f(\sigma)\} & \text{otherwise} \end{cases}$$

$$(f+g)(\sigma) = f(\sigma) \sqcup g(\sigma)$$

$$f^*(\sigma) = \bigsqcup_n f^n(\sigma)$$

Example 1 (Separation algebras and local functions)

1. **Plain heap model**. *A simple example is the separation algebra of heaps* $(H, \bullet, u_H)$*, where* $H = L \rightharpoonup_{\mathit{fin}} Val$ *are finite partial functions from a set of locations* L *to a set of values* Val *with* $L \subseteq Val$*, the partial operator* $\bullet$ *is the union of partial functions with disjoint domains, and the unit* u_H *is the function with the empty domain. For* $h \in H$*, let* $dom(h)$ *be the domain of* h*. We write* $l \mapsto v$ *for the partial function with domain* $\{l\}$ *that maps* l *to* v*. For* $h_1, h_2 \in H$*, if* $h_2 \preceq h_1$ *then* $h_1 - h_2 = h_1\,|_{dom(h_1)-dom(h_2)}$*. An example of a local function is the* $dispose[l]$ *command that deletes the cell at location* l*:*

$$dispose[l](h) = \begin{cases} \{h - (l \mapsto v)\} & h \succeq (l \mapsto v) \\ \top & \text{otherwise} \end{cases}$$

The function is local: if $h \not\succeq (l \mapsto v)$ *then* $dispose[l](h) = \top$*, and* $dispose[l](h' \bullet h) \sqsubseteq \top$*. Otherwise,* $dispose[l](h' \bullet h) = \{(h' \bullet h) - (l \mapsto v)\} \sqsubseteq \{h'\} * \{h - (l \mapsto v)\} = \{h'\} * dispose[l](h)$*.*

2. **Heap and stack**. *There are two approaches to modelling the stack in the literature. One is to treat the stack as a total function from variables to values, and only combine two heap and stack pairs if the stacks are the same. The other approach, which we use here, is to allow the variable stack to be split, treating it as part of the resource. We can incorporate the variable stack into the heap model by using the set* $H = L \cup Var \rightharpoonup_{\mathit{fin}} Val$*, where* L *and* Val *are as before and* Var *is the set of stack variables* $\{x, y, z, ...\}$*. The* $\bullet$ *operator combines heap and stack portions with disjoint domains, and is undefined otherwise. The unit* u_H *is the function with the empty domain which represents the empty heap and empty stack.*

Although this approach is limited to disjoint reference to stack variables, this constraint can be lifted by enriching the separation algebra with permissions *[4]. However, this added complexity using permissions can be avoided for the discussion here. For a state $h \in H$, we let $loc(h)$ and $var(h)$ denote the set of heap locations and stack variables in the domain of h respectively. In this model we can define the allocation and deallocation commands as*

$$new[x](h) = \begin{cases} \{h' \bullet x \mapsto l \bullet l \mapsto w \mid w \in Val, l \in L \backslash loc(h')\} & h = h' \bullet x \mapsto v \\ \top & \textit{otherwise} \end{cases}$$

$$dispose[x](h) = \begin{cases} \{h' \bullet x \mapsto l\} & h = h' \bullet x \mapsto l \bullet l \mapsto v \\ \top & \textit{otherwise} \end{cases}$$

Commands for heap mutation and lookup can be defined as

$$mutate[x, v](h) = \begin{cases} \{h' \bullet x \mapsto l \bullet l \mapsto v\} & h = h' \bullet x \mapsto l \bullet l \mapsto w \\ \top & \textit{otherwise} \end{cases}$$

$$lookup[x, y](h) = \begin{cases} \{h' \bullet x \mapsto l \bullet l \mapsto v \bullet y \mapsto v\} & h = h' \bullet x \mapsto l \bullet l \mapsto v \bullet y \mapsto w \\ \top & \textit{otherwise} \end{cases}$$

The AD *command described in the introduction, which is the composition $new[x]; dispose[x]$, corresponds to the following local function*

$$AD(h) = \begin{cases} \{h' \bullet x \mapsto l \mid l \in L \backslash loc(h')\} & h = h' \bullet x \mapsto v \\ \top & \textit{otherwise} \end{cases}$$

Note that in all cases, any stack variables that the command refers to should be in the stack in order for the command to execute safely, otherwise the command will be acting non-locally.

3. **Integers**. *The integers form a separation algebra under addition with identity 0. In this case we have that any 'adding' function $f(x) = \{x + c\}$ that adds a constant c is local, while a function that multiplies by a constant c, $f(x) = \{cx\}$, is non-local in general. However, the integers under multiplication also form a separation algebra with identity 1, and in this case every multiplying function is local but not every adding function. This illustrates the point*

that the notion of locality of commands depends on the notion of separation of resource that is being used.

Predicates, Specifications and Local Hoare Reasoning We now present the local reasoning framework for local functions on separation algebras. This is an adaptation of Abstract Separation Logic [11], with some minor changes in formulation for the purposes of this thesis. Predicates over separation algebras are treated simply as subsets of the separation algebra.

Definition 2.6 *A* **predicate** p *over* Σ *is an element of the powerset* $\mathcal{P}(\Sigma)$*.*

Note that the top element $\top$ is not a predicate and that the $*$ operator, although defined on $\mathcal{P}(\Sigma)^\top \times \mathcal{P}(\Sigma)^\top \to \mathcal{P}(\Sigma)^\top$, acts as a binary connective on predicates. We have the distributive law for union that, for any $X \subseteq \mathcal{P}(\Sigma)$,

$$(\bigsqcup X) * p = \bigsqcup\{x * p \mid x \in X\}$$

The same is not true for intersection in general, but does hold for $precise$ predicates. A predicate is precise if, for any state, there is at most a single substate that satisfies the predicate.

Definition 2.7 (Precise predicate) *A predicate* $p \in \mathcal{P}(\Sigma)$ *is* **precise** *if and only if, for every* $\sigma \in \Sigma$*, there exists at most one* $\sigma_p \in p$ *such that* $\sigma_p \preceq \sigma$*.*

With precise predicates, there is at most a unique way to break any state in order to get a substate that satisfies the predicate. Any singleton predicate $\{\sigma\}$ is precise. Another example of a precise predicate is $\{l \mapsto v \mid v \in Val\}$ for some l, while $\{l \mapsto v \mid l \in L\}$ for some v is not precise.

Lemma 2.3 (Precision characterization) *A predicate* p *is precise if and only if, for all* $X \subseteq \mathcal{P}(\Sigma)$*,* $(\bigsqcap X) * p = \bigsqcap\{x * p \mid x \in X\}$

Proof: *We first show the left to right direction. Assume* p *is precise. We have to show that, for all* $X \subseteq \mathcal{P}(\Sigma)$*,* $(\bigsqcap X) * p = \bigsqcap\{x * p \mid x \in X\}$*. Assume* $\sigma \in (\bigsqcap X) * p$*. Then there exist* σ_1, σ_2 *such that* $\sigma = \sigma_1 \bullet \sigma_2$ *and* $\sigma_1 \in \bigsqcap X$ *and* $\sigma_2 \in p$*. Thus, for all* $x \in X$*,* $\sigma \in x * p$*, and hence* $\sigma \in \bigsqcap\{x * p \mid x \in X\}$*. Now assume* $\sigma \in \bigsqcap\{x * p \mid x \in X\}$*. Then* $\sigma \in x * p$ *for all* $x \in X$*.*

Hence there exists $\sigma_1 \preceq \sigma$ such that $\sigma_1 \in p$. Since p is precise, σ_1 is unique. Let $\sigma_2 = \sigma - \sigma_1$. Thus, we have $\sigma_2 \in x$ for all $x \in X$, and so $\sigma_2 \in \bigsqcap X$. Hence, we have $\sigma \in (\bigsqcap X) * p$.

For the other direction, we assume that p is not precise and show that there exists an X such that $(\bigsqcap X) * p \neq \bigsqcap\{x * p \mid x \in X\}$. Since p is not precise, there exists $\sigma \in \Sigma$ such that, for two distinct $\sigma_1, \sigma_2 \in p$, we have $\sigma_1 \preceq \sigma$ and $\sigma_2 \preceq \sigma$. Let $\sigma'_1 = \sigma - \sigma_1$ and $\sigma'_2 = \sigma - \sigma_2$. Now let $X = \{\{\sigma'_1\}, \{\sigma'_2\}\}$. Since $\sigma \in \{\sigma'_1\} * p$ and $\sigma \in \{\sigma'_2\} * p$, we have $\sigma \in \bigsqcap\{x * p \mid x \in X\}$. However, because of the cancellation property, we also have that $\sigma'_1 \neq \sigma'_2$, and so $(\bigsqcap X) * p = \emptyset * p = \emptyset$. Hence, $\sigma \notin (\bigsqcap X) * p$, and we therefore have $(\bigsqcap X) * p \neq \bigsqcap\{x * p \mid x \in X\}$. ■

Our Hoare reasoning framework is formulated with tuples of pre- and post- conditions, rather than the usual Hoare triples that include the function as in [11]. In our case the standard triple shall be expressed as a function f *satisfying* a tuple (p, q), written $f \models (p, q)$. The reason for this is that we shall be examining the properties that a pre- and post- condition tuple may have with respect to a given function, such as whether a given tuple is complete for a given function. This approach is very similar to the notion of the *specification statement* (a Hoare triple with a 'hole') introduced by Morgan in [39], which is used in refinement calculi, and was also used to prove completeness of a local reasoning system in [56].

Definition 2.8 (Specification) *Let Σ be a separation algebra. A* **statement** *on Σ is a tuple (p, q), where $p, q \in \mathcal{P}(\Sigma)$ are predicates. A* **specification** *ϕ on Σ is a set of statements. We let $\Phi_\Sigma = \mathcal{P}(\mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma))$ denote the set of all specifications on Σ. We shall exclude the subscript when it is clear from the context. The* **domain** *of a specification is defined as $D(\phi) = \bigsqcup\{p \mid (p, q) \in \phi\}$.* **Domain equivalence** *is defined as $\phi \cong_D \psi$ if and only if $D(\phi) = D(\psi)$.*

The domain is the union of the pre-conditions of all the statements in the specification. It is one possible measure of *size*: how much of Σ the specification is referring to.

Definition 2.9 (Satisfaction) *A local function f satisfies a statement (p, q), written $f \models (p, q)$, if and only if, for all $\sigma \in p$, $f(\sigma) \sqsubseteq q$. It satisfies a specification $\phi \in \Phi$, written $f \models \phi$, if and only if $f \models (p, q)$ for all $(p, q) \in \phi$.*

Definition 2.10 (Semantic consequence) *Let $p, q, r, s \in \mathcal{P}(\Sigma)$ and $\phi, \psi \in \Phi$. Each judgement*

$(p,q) \models (r,s), \phi \models (p,q)$, $(p,q) \models \phi$, and $\phi \models \psi$ holds if and only if all local functions that satisfy the left-hand side also satisfy the right-hand side.

Proposition 2.4 (Order Characterization) *$f \sqsubseteq g$ if and only if, for all $p, q \in \mathcal{P}(\Sigma)$, $g \models (p,q)$ implies $f \models (p,q)$.*

For every specification ϕ, there exists a *best local action* satisfying ϕ, which is a local function such that all statements satisfied by this function are also satisfied by any local function satisfying ϕ. For example, in the heap and stack separation algebra of example 1.2, consider the specification

$$\phi_{new} = \{(\{x \mapsto v\}, \{x \mapsto l \bullet l \mapsto w \mid l \in L, w \in Val\}) \mid v \in Val\}$$

There are many local functions that satisfy this specification. Trivially, the local function that always diverges satisfies it. Another example is the local function that assigns the value w of the newly allocated cell to be 0, rather than any non-deterministically chosen value. However, the best local action for this specification is the $new[x]$ function described in example 1.2, as it can be checked that for any local function f satisfying ϕ_{new}, we have $f \sqsubseteq new[x]$. The notion of the best local action shall be used when addressing questions about completeness of specifications. It is adapted from [11], except that we generalise to the best local action of a specification rather than a single pre- and post-condition pair.

Definition 2.11 (Best local function) *For a specification $\phi \in \Phi$, the best local action of ϕ, written $bla[\phi]$, is the function of type $\Sigma \to \mathcal{P}(\Sigma)^\top$ defined by*

$$bla[\phi](\sigma) = \bigsqcap \{\{\sigma'\} * q \mid \sigma = \sigma' \bullet \sigma'', \sigma'' \in p, (p,q) \in \phi\}$$

As an example, it can be checked that the best local action $bla[\phi_{new}]$ of the specification ϕ_{new} given above is indeed the function $new[x]$ described in example 1.2. The following lemma presents the important properties which characterise the best local action.

Lemma 2.5 *Let $\phi \in \Phi$. The following hold:*

- *$bla[\phi]$ is local*

- $bla[\phi] \models \phi$
- *if f is local and $f \models \phi$ then $f \sqsubseteq bla[\phi]$*

Proof: *To show that $bla[\phi]$ is local, consider σ_1, σ_2 such that $\sigma_1 \# \sigma_2$. We then calculate*

$$\begin{array}{rl} & bla[\phi](\sigma_1 \bullet \sigma_2) \\ = & \bigsqcap\{\{\sigma'\} * q \mid \sigma_1 \bullet \sigma_2 = \sigma' \bullet \sigma'', \sigma'' \in p, (p,q) \in \phi\} \\ \sqsubseteq & \bigsqcap\{\{\sigma_1 \bullet \sigma'''\} * q \mid \sigma_2 = \sigma''' \bullet \sigma'', \sigma'' \in p, (p,q) \in \phi\} \\ = & \bigsqcap\{\{\sigma_1\} * \{\sigma'''\} * q \mid \sigma_2 = \sigma''' \bullet \sigma'', \sigma'' \in p, (p,q) \in \phi\} \\ = & \{\sigma_1\} * \bigsqcap\{\{\sigma'''\} * q \mid \sigma_2 = \sigma''' \bullet \sigma'', \sigma'' \in p, (p,q) \in \phi\} \\ = & \{\sigma_1\} * bla[\phi](\sigma_2) \end{array}$$

In the second-last step we used the property that $\{\sigma_1\}$ is precise (lemma 2.3).

To show that $bla[\phi]$ satisfies ϕ, consider any $(p,q) \in \phi$ and $\sigma \in p$. Then $bla[\phi](\sigma) \sqsubseteq \{u\} * q = q$.

For the last point, suppose f is local and $f \models \phi$. Then, for any σ such that $\sigma = \sigma_1 \bullet \sigma_2$ and $\sigma_2 \in p$ and $(p,q) \in \phi$,

$$\begin{array}{rcl} f(\sigma) & = & f(\sigma_1 \bullet \sigma_2) \\ & \sqsubseteq & \{\sigma_1\} * f(\sigma_2) \\ & \sqsubseteq & \{\sigma_1\} * q \end{array}$$

Thus $f(\sigma) \sqsubseteq bla[\phi](\sigma)$.

In the case that there do not exist σ_1, σ_2 such that $\sigma = \sigma_1 \bullet \sigma_2$ and $\sigma_2 \in D(\phi)$, then

$$\begin{array}{rcl} bla[\phi](\sigma) & = & \bigsqcap \emptyset \\ & = & \top \end{array}$$

So in this case also $f(\sigma) \sqsubseteq bla[\phi](\sigma)$. ■

Lemma 2.6 *For $\phi \in \Phi$ and $p, q \in \mathcal{P}(\Sigma)$, $bla[\phi] \models (p,q) \Leftrightarrow \phi \models (p,q)$.*

$$\frac{(p,q)}{(p*r, q*r)}\ \textit{Frame} \qquad \frac{p' \sqsubseteq p \quad (p,q) \quad q \sqsubseteq q'}{(p',q')}\ \textit{Consequence} \qquad \frac{(p_i,q_i),\ \text{all } i \in I}{(\bigsqcup_{i\in I} p_i, \bigsqcup_{i\in I} q_i)}\ \textit{Union} \qquad \frac{(p_i,q_i),\ \text{all } i \in I, I \neq \emptyset}{(\bigsqcap_{i\in I} p_i, \bigsqcap_{i\in I} q_i)}\ \textit{Intersection}$$

Figure 2.1: Inference rules for local Hoare reasoning

Proof:

$$\begin{aligned}
& bla[\phi] \models (p,q) \\
\Leftrightarrow\ & \textit{for all local functions } f,\ f \models \phi \Rightarrow f \models (p,q) \quad \textit{(by lemma 2.5)} \\
\Leftrightarrow\ & \phi \models (p,q) \quad \textit{(by definition 2.10).}
\end{aligned}$$

■

The inference rules of the proof system are given in figure 2.1. Consequence, union and intersection are adaptations of standard rules of Hoare logic. The frame rule is what permits local reasoning, as it codifies the fact that, since all functions are local, any assertion about a separate part of resource will continue to hold for that part after the application of the function. We omit the standard rules for basic constructs such as sequential composition, non-deterministic choice, and Kleene-star which can be found in [11].

Definition 2.12 (Proof-theoretic consequence) *For predicates p, q, r, s and specifications ϕ, ψ, each of the judgements $(p,q) \vdash (r,s), \phi \vdash (p,q)$, $(p,q) \vdash \phi$, and $\phi \vdash \psi$ holds if and only if the right-hand side is derivable from the left-hand side by the rules in figure 2.1.*

The proof system of figure 2.1 is sound and complete with respect to the satisfaction relation.

Theorem 2.7 (Soundness and Completeness) $\phi \vdash (p,q) \Leftrightarrow \phi \models (p,q)$

Proof: Soundness can be checked by checking each of the proof rules in figure 2.1. The frame rule is sound by the locality condition, and the others are easy to check.

For completeness, assume we are given $\phi \models (p,q)$. By lemma 2.6, we have $bla[\phi] \models (p,q)$. So for all $\sigma \in p$, $bla[\phi](\sigma) \sqsubseteq q$, which implies

$$\bigsqcup_{\sigma \in p} bla[\phi](\sigma) \sqsubseteq q \quad (*)$$

Now we have the following derivation:

$$\begin{array}{c}
\phi \\ \hline
(r, s) \quad \text{for all } (r,s) \in \phi \\ \hline
(\{\sigma'\}, s) \quad \text{for all } \sigma' \in r, (r,s) \in \phi \\ \hline
(\{\sigma - \sigma'\} * \{\sigma'\}, \{\sigma - \sigma'\} * s) \quad \text{for all } \sigma' \in r, (r,s) \in \phi, \sigma' \preceq \sigma, \sigma \in p \\ \hline
\Big(\bigsqcap_{\substack{\sigma' \preceq \sigma \\ \sigma' \in r \\ (r,s) \in \phi}} \{\sigma - \sigma'\} * \{\sigma'\}, \bigsqcap_{\substack{\sigma' \preceq \sigma \\ \sigma' \in r \\ (r,s) \in \phi}} \{\sigma - \sigma'\} * s \Big) \quad \text{for all } \sigma \in p \\ \hline
(\{\sigma\}, bla[\phi](\sigma)) \quad \text{for all } \sigma \in p \\ \hline
(\bigsqcup_{\sigma \in p} \{\sigma\}, \bigsqcup_{\sigma \in p} bla[\phi](\sigma)) \\ \hline
(p, q)
\end{array}$$

The last step in the proof is by $(*)$ and the rule of consequence. Note that the intersection rule can be safely applied because the argument of the intersection is necessarily non-empty (if it were empty then $bla[\phi](\sigma) = \top$, which contradicts $bla[\phi](\sigma) \sqsubseteq q$). ■

2.2 Properties of Specifications

We discuss certain properties of specifications as a prerequisite for our main discussion on relevance footprints. We define the notion of a *complete* specification for a local function, which is a specification from which follows every property that holds for the function. However, a function may have many complete specifications, so we introduce a canonical form for specifications. We show that there exists a unique canonical complete specification for every domain on which a local function can be completely specified. As discussed in the introduction, an important notion of local reasoning is the *small specification* which completely describes the behaviour of a local function by mentioning only the footprint. Thus, as a prerequisite to investigating their existence, we formalise small specifications as complete specifications with the smallest possible domain. Similarly, we define *big* specifications as complete specifications with the biggest domain.

Definition 2.13 (Complete Specification) *A specification $\phi \in \Phi$ is a* **complete specification** *for*

f, written $complete(\phi, f)$, if and only if, for all $p, q \in \mathcal{P}(\Sigma)$, $f \models (p, q) \Leftrightarrow \phi \models (p, q)$. Let $\Phi_{comp(f)}$ be the set of all complete specifications of f.

ϕ is complete for f whenever the tuples that hold for f are *exactly* the tuples that follow from ϕ. This also means that any two complete specifications ϕ and ψ for a local function are semantically equivalent (that is, $\phi \dashv\models \psi$). The following proposition illustrates how the notions of best local action and complete specification are closely related.

Proposition 2.8 *For all $\phi \in \Phi$ and local functions f, $complete(\phi, f) \Leftrightarrow f = bla[\phi]$.*

Proof: *Assume f = $bla[\phi]$. Then, by lemma 2.6, we have that ϕ is a complete specification for f.*

For the converse, assume $complete(\phi, f)$. We shall show that for any $\sigma \in \Sigma$, $f(\sigma) = bla[\phi](\sigma)$.

case 1: *$f(\sigma) = \top$. If $bla[\phi](\sigma) \neq \top$, then $bla[\phi] \models (\{\sigma\}, bla[\phi](\sigma))$. This means that $\phi \models (\{\sigma\}, bla[\phi](\sigma))$ by lemma 2.6, and so $f \models (\{\sigma\}, bla[\phi](\sigma))$, but this is a contradiction. Therefore, $bla[\phi](\sigma) = \top$.*

case 2: *$bla[\phi](\sigma) = \top$. If $f(\sigma) \neq \top$, then $f \models (\{\sigma\}, f(\sigma))$. This means that $\phi \models (\{\sigma\}, f(\sigma))$, and so $bla[\phi] \models (\{\sigma\}, f(\sigma))$, but this is a contradiction. Therefore, $f(\sigma) = \top$.*

case 3: *$bla[\phi](\sigma) \neq \top$* **and** *$f(\sigma) \neq \top$. We have*

$$\begin{array}{ll} & f \models (\{\sigma\}, f(\sigma)) \\ \Rightarrow & bla[\phi] \models (\{\sigma\}, f(\sigma)) \\ \Rightarrow & bla[\phi](\sigma) \sqsubseteq f(\sigma) \end{array}$$

$$\begin{array}{ll} & bla[\phi] \models (\{\sigma\}, bla[\phi](\sigma)) \\ \Rightarrow & f \models (\{\sigma\}, bla[\phi](\sigma)) \\ \Rightarrow & f(\sigma) \sqsubseteq bla[\phi](\sigma) \end{array}$$

Therefore $f(\sigma) = bla[\phi](\sigma)$ ■

Any specification is therefore only complete for a unique local function, which is its best local action. However, a local function may have lots of complete specifications. For example, if ϕ is a complete specification for f and $(p, q) \in \phi$, then $\phi \cup \{(p, q')\}$ is also complete for f if $q \subseteq q'$. For this reason it will be useful to have a canonical form for specifications.

Definition 2.14 (Canonicalisation) *The* **canonicalisation** *of a specification ϕ is defined as $\phi_{can} = \{(\{\sigma\}, bla[\phi](\sigma)) \mid \sigma \in D(\phi)\}$. A specification is in* **canonical** *form if it is equal to its canonicalisation. Let $\Phi_{can(f)}$ denote the set of all canonical complete specifications of f.*

Notice that a given local function does not necessarily have a *unique* canonical complete specification, as there may exist canonical specifications with different domains. For example, both $\{(\{u\}, \{u\})\}$ and $\{(\{u\}, \{u\}), (\{\sigma\}, \{\sigma\})\}$, for some $\sigma \in \Sigma$, are canonical complete specifications for the identity function.

Proposition 2.9 *For any specification ϕ, we have $\phi \dashv\models \phi_{can}$.*

Proof: *We first show $\phi \models \phi_{can}$. For any $(p, q) \in \phi_{can}$, we have that (p, q) is of the form $(\{\sigma\}, bla[\phi](\sigma))$ for some $\sigma \in D(\phi)$. So we have $bla[\phi] \models (p, q)$, and so $\phi \models (p, q)$ by lemma 2.6.*

We now show $\phi_{can} \models \phi$. For any $(p, q) \in \phi$, we have $bla[\phi] \models (p, q)$ by lemma 2.6. So for all $\sigma \in p$, $bla[\phi](\sigma) \sqsubseteq q$, which implies

$$\bigsqcup_{\sigma \in p} bla[\phi](\sigma) \sqsubseteq q \quad (*)$$

Now we have the following derivation:

$$\cfrac{\cfrac{\cfrac{\phi_{can}}{(\{\sigma\}, bla[\phi](\sigma)) \quad \text{\small\emph{for all} } \sigma \in p}}{(\bigsqcup_{\sigma \in p}\{\sigma\}, \bigsqcup_{\sigma \in p} bla[\phi](\sigma))}}{(p, q)}$$

The last step is by $(*)$ and consequence. So we have $\phi_{can} \vdash \phi$, and by soundness $\phi_{can} \models \phi$. ■

Thus, the canonicalisation of a specification is logically equivalent to the specification. The following corollary shows that all complete specifications that have the same domain have a unique canonical form, and specifications of different domains have different canonical forms.

Corollary 2.10 *$\Phi_{can(f)}$ is isomorphic to the quotient set $\Phi_{comp(f)}/\cong_D$, under the isomorphism that maps $[\phi]_{\cong_D}$ to ϕ_{can}, for every $\phi \in \Phi_{comp(f)}$.*

Proof: *By proposition 2.8, all complete specifications for f have the same best local action, which is f itself. So by the definition of canonicalisation, it can be seen that complete specifications with different domains have different canonicalisations, and complete specifications with the same domain have the same canonicalisation. This shows that the mapping is well-defined and injective. Every canonical complete specification ϕ is also complete, and $[\phi]_{\cong_D}$ maps to $\phi_{can} = \phi$, so the mapping is surjective.* ■

Definition 2.15 (Small and Big specifications) *ϕ is a* **small specification** *for f if and only if $\phi \in \Phi_{comp(f)}$ and there is no $\psi \in \Phi_{comp(f)}$ such that $D(\psi) \sqsubset D(\phi)$. A* **big specification** *is defined similarly.*

Small and *big* specifications are thus the specifications with the smallest and biggest domains respectively. The question is if/when small and big specifications exist. The following result shows that a canonical big specification exists for every local function.

Proposition 2.11 (Big Specification) *For any local function f, the canonical big specification for f is given by $\phi_{big(f)} = \{(\{\sigma\}, f(\sigma)) \mid f(\sigma) \sqsubset \top\}$.*

Proof: *$f \models \phi_{big(f)}$ is trivial to check. To show $complete(\phi_{big(f)}, f)$, assume $f \models (p, q)$ for some $p, q \in \mathcal{P}(\Sigma)$. Note that, for any $\sigma \in p$, $f(\sigma) \sqsubseteq q$ and so $\bigsqcup_{\sigma \in p} f(\sigma) \sqsubseteq q$. We then have the derivation*

$$\cfrac{\cfrac{\cfrac{\phi_{big(f)}}{(\{\sigma\}, f(\sigma))} \;\; \textit{for all } f(\sigma) \sqsubset \top}{(\bigsqcup_{\sigma \in p} \{\sigma\}, \bigsqcup_{\sigma \in p} f(\sigma))}}{(p, q)}$$

By soundness we get $\phi_{big(f)} \models (p, q)$. $\phi_{big(f)}$ has the biggest domain because f would fault on any element not included in $\phi_{big(f)}$. ■

The notion of a small specification has until now been used in an informal sense in local reasoning papers [41, 4, 10] as specifications that completely specify the behaviour of an update command by only describing the command's behaviour on the part of the resource that it affects. Although these papers present examples of such specifications for specific commands, the notion has so far not received a formal treatment in the general case. The question of the existence of small specifications is strongly related to the concept of footprints, since finding a small specification is about finding a complete specification with the smallest possible domain, and therefore enquiring about which elements of Σ are essential and sufficient for a complete specification. This requires a formal characterisation of the footprint notion, which we shall now present.

2.3 Relevance Footprints

In the introduction we discussed how the *AD* program demonstrates that the safety footprint of a program (the smallest states for safe execution) do not always yield a complete specification of the program. To obtain a complete specification, we needed larger states in the pre-condition of the specification, which may be thought of as the *relevance footprint* of the program. This raises the question of how the relevance footprint of an arbitrary local function can be formally defined. We address this question by first analysing the notion of locality. Recall that the definition of locality (definition 2.5) states that the action on a certain state σ_1 imposes a *limit* on the action on a bigger state $\sigma_2 \bullet \sigma_1$. This limit is the set $\{\sigma_2\} * f(\sigma_1)$, since we have $f(\sigma_2 \bullet \sigma_1) \sqsubseteq \{\sigma_2\} * f(\sigma_1)$.

Another way of viewing this definition is that for any state σ, the action of the function on that state has to be within the limit imposed by *every* substate σ' of σ, that is, $f(\sigma) \sqsubseteq \{\sigma - \sigma'\} * f(\sigma')$. In the case where $\sigma' = \sigma$, this condition is trivially satisfied for any function (local or non-local). The distinguishing characteristic of local functions is that this condition is also satisfied by every strict substate of σ, and thus we have

$$f(\sigma) \sqsubseteq \prod_{\sigma' \prec \sigma} \{\sigma - \sigma'\} * f(\sigma')$$

We define this overall constraint imposed on σ by all of its strict substates as the *local limit* of f on σ, and show that the locality definition is equivalent to satisfying the local limit constraint.

Definition 2.16 (Local limit) *For a local function f on Σ and $\sigma \in \Sigma$, the* **local limit** *of f on σ is defined as*

$$L_f(\sigma) = \prod_{\sigma' \prec \sigma} \{\sigma - \sigma'\} * f(\sigma')$$

Proposition 2.12 *f is local $\quad\Leftrightarrow\quad f(\sigma) \sqsubseteq L_f(\sigma) \quad$ for all $\sigma \in \Sigma$*

Proof: *Assume f is local. So for any σ, for every $\sigma' \prec \sigma$, $f(\sigma) \sqsubseteq \{\sigma - \sigma'\} * f(\sigma')$. $f(\sigma)$ is therefore smaller than the intersection of all these sets, which is $L_f(\sigma)$.*

For the converse, assume the rhs and that $\sigma_1 \bullet \sigma_2$ is defined. If $\sigma_1 = u$ then $f(\sigma_1 \bullet \sigma_2) \sqsubseteq \{\sigma_1\} * f(\sigma_2)$ and we are done. Otherwise, $\sigma_2 \prec \sigma_1 \bullet \sigma_2$ and we have $f(\sigma_1 \bullet \sigma_2) \sqsubseteq L_f(\sigma_1 \bullet \sigma_2) \sqsubseteq \{\sigma_1\} * f(\sigma_2)$. ■

Thus for any local function f acting on a certain state σ, the local limit determines a upper bound on the possible outcomes on σ, based on the outcomes on all smaller states. If this upper bound does correspond exactly to the set of all possible outcomes on σ, then σ is 'large enough' that all the smaller states determine the behaviour of f on σ. In this case we do not think of σ as being relevant for a description of f, since smaller states are sufficient to determine the action of f on σ. With this observation, we define footprints as those states on which the action of f cannot be determined by the smaller states alone, that is, the set of outcomes is a *strict* subset of the local limit.

Definition 2.17 (Relevance Footprint) *For a local function f and $\sigma \in \Sigma$, σ is a relevance footprint of f, written $F_f(\sigma)$, if and only if $f(\sigma) \sqsubset L_f(\sigma)$. We denote the set of relevance footprints of f by $F(f)$.*

Note that an element σ is therefore not a footprint if and only if the action of f on σ is at the local limit, that is $f(\sigma) = L_f(\sigma)$.

Lemma 2.13 *For any local function f, the smallest safe states of f are always relevance footprints of f.*

Proof: *Let σ be a smallest safe state for f. Then for any $\sigma' \prec \sigma$, $f(\sigma') = \top$. Therefore $L_f(\sigma) = \top$ and so $f(\sigma) \sqsubset L_f(\sigma)$.* ■

Example 2 (Dispose) *The footprints of the $dispose[l]$ command in the plain heap model (example 1.1) are the cells at location l. We check this by considering the following cases*

1. *The empty heap, u_H, is not a footprint since $L_{dispose[l]}(u_H) = \top = dispose[l](u_H)$*

2. *Every cell $l \mapsto v$ for some v is a footprint*

$$L_{dispose[l]}(l \mapsto v) = \{l \mapsto v\} * dispose[l](u_H) = \{l \mapsto v\} * \top = \top$$

$$dispose[l](l \mapsto v) = \{u_H\} \sqsubset L_{dispose[l]}(l \mapsto v)$$

3. *Every state σ such that $\sigma \succ (l \mapsto v)$ for some v is not a footprint*

$$L_{dispose[l]}(\sigma) \sqsubseteq \{\sigma - (l \mapsto v)\} * dispose[l](l \mapsto v) = \{\sigma - (l \mapsto v)\} = dispose[l](\sigma)$$

   By proposition 2.12, we have $L_{dispose[l]}(\sigma) = dispose[l](\sigma)$. The intuition is that σ does not characterise any 'new' behaviour of the function: its action on σ is just a consequence of its action on the cells at location l and the locality property of the function.

4. *Every state σ such that $\sigma \not\succ (l \mapsto v)$ for some v is not a footprint*

$$L_{dispose[l]}(\sigma) \sqsubseteq \{\sigma\} * dispose[l](u_H) = \{\sigma\} * \top = \top = dispose[l](\sigma)$$

   Again by proposition 2.12, $L_{dispose[l]}(\sigma) = dispose[l](\sigma)$.

However, as discussed in the introduction, the smallest safe states are not always the only relevant states. We now demonstrate how definition 2.17 delivers the correct footprint for the *AD* program, which is the empty heap and all the single-cell heaps, as discussed in the introduction.

Example 3 (*AD* command) *The* AD *(Allocate-Deallocate) command was defined on the heap and stack model in example 1.2. We have the following cases for σ.*

1. *$\sigma \not\succeq x \mapsto v_1$ for some v_1 is not a footprint, since $L_{AD}(\sigma) = \top = AD(\sigma)$.*

2. *$\sigma = x \mapsto v_1$ for some v_1 is a footprint since $L_{AD}(\sigma) = \top$ (by case (1)) and $AD(\sigma) = \{x \mapsto w \mid w \in L\} \sqsubset L_{AD}(\sigma)$.*

3. *$\sigma = l \mapsto v_1 \bullet x \mapsto v_2$ for some l, v_1, v_2 is a footprint.*

$$\begin{aligned} L_{AD}(\sigma) &= \{l \mapsto v_1\} * AD(x \mapsto v_2) \\ &\quad \textit{(AD faults on all other elements strictly smaller than } \sigma\textit{)} \\ &= \{l \mapsto v_1\} * \{x \mapsto w \mid w \in L\} \\ &= \{l \mapsto v_1 \bullet x \mapsto w \mid w \in L\} \end{aligned}$$

$$AD(\sigma) \quad = \{l \mapsto v_1 \bullet x \mapsto w \mid w \in L, w \neq l\} \sqsubset L_{AD}(\sigma)$$

4. *$\sigma = h \bullet x \mapsto v_1$ for some v_1, and where $|loc(h)| > 1$, is not a footprint.*

$$L_{AD}(\sigma) \sqsubseteq \bigsqcap_{h \succ l \mapsto v} \{h - (l \mapsto v)\} * AD(l \mapsto v \bullet x \mapsto v_1)$$

$$= \{h \bullet x \mapsto w \mid w \notin loc(h)\} = AD(\sigma)$$

By proposition 2.12, we get $L_{AD}(\sigma) = AD(\sigma)$.

Definition 2.17 therefore works correctly for these example programs. We now give the formal result to show that this definition of relevance footprints is the correct one for arbitrary local functions.

Theorem 2.14 (Essentiality) *The relevance footprints of a local function are the only essential domain elements for any complete specification of that function, that is,*

$$F_f(\sigma) \quad \Leftrightarrow \quad \forall \phi \in \Phi_{comp(f)}.\, \sigma \in D(\phi)$$

Proof: Assume some fixed f and σ. We establish the following equivalent statement :

$$\neg F_f(\sigma) \quad \Leftrightarrow \quad \exists \phi \in \Phi_{comp(f)}.\, \sigma \notin D(\phi)$$

We first show the right to left implication. So assume ϕ is a complete specification of f such that $\sigma \notin D(\phi)$. Since $complete(\phi, f)$, by proposition 2.8, we have $f = bla[\phi]$. So

$$f(\sigma) = \bigsqcap_{\sigma_1 \preceq \sigma, \sigma_1 \in p, (p,q) \in \phi} \{\sigma - \sigma_1\} * q$$

Now for any set $\{\sigma - \sigma_1\} * q$ in the above intersection, we have that $\sigma_1 \in p$, and $(p, q) \in \phi$ for some p. Since $\sigma_1 \in p$, we have $f(\sigma_1) \sqsubseteq q$, and therefore $\{\sigma - \sigma_1\} * f(\sigma_1) \sqsubseteq \{\sigma - \sigma_1\} * q$. Also, $\sigma_1 \neq \sigma$, because otherwise we would have $\sigma \in p$, which would contradict the assumption that $\sigma \notin D(\phi)$. So $\sigma_1 \prec \sigma$ and we have

$$L_f(\sigma) \sqsubseteq \{\sigma - \sigma_1\} * f(\sigma_1) \sqsubseteq \{\sigma - \sigma_1\} * q$$

So the local limit is smaller than each set $\{\sigma - \sigma_1\} * q$ in the intersection, and therefore it is smaller than the intersection itself: $L_f(\sigma) \sqsubseteq f(\sigma)$. We know from proposition 2.12 that $f(\sigma) \sqsubseteq L_f(\sigma)$, so we get $f(\sigma) = L_f(\sigma)$ and therefore $\neg F_f(\sigma)$.

We now show the left to right implication. Assume that σ is not a footprint of f. We shall use the big specification, $\phi_{big(f)}$, to construct a complete specification of f which does not contain σ in its domain. If $f(\sigma) = \top$ then the big specification itself is such a specification, and we are done. Otherwise assume $f(\sigma) \sqsubset \top$. Let $\phi = \phi_{big(f)} / \{(\{\sigma\}, f(\sigma))\}$. It can be seen that $\sigma \notin D(\phi)$. Now we need to show that ϕ is complete for f. For this it is sufficient to show $\phi \dashv\vdash \phi_{big(f)}$ because we know that $\phi_{big(f)}$ is complete for f. The right to left direction, $\phi \dashv \phi_{big(f)}$, is trivial.

For $\phi \vdash \phi_{big(f)}$, we just need to show $\phi \vdash (\{\sigma\}, f(\sigma))$. We have the following derivation:

$$\dfrac{\dfrac{\dfrac{\phi}{(\{\sigma'\}, f(\sigma')) \quad \text{for all } \sigma' \prec \sigma,\, f(\sigma') \sqsubset \top}}{(\{\sigma - \sigma'\} * \{\sigma'\}, \{\sigma - \sigma'\} * f(\sigma')) \quad \text{for all } \sigma' \prec \sigma,\, f(\sigma') \sqsubset \top}}{\dfrac{(\{\sigma\}, \displaystyle\bigsqcap_{\sigma' \prec \sigma, f(\sigma') \sqsubset \top} \{\sigma - \sigma'\} * f(\sigma'))}{(\{\sigma\}, L_f(\sigma))}}$$

The intersection rule can be safely applied as there is at least one $\sigma' \prec \sigma$ such that $f(\sigma') \sqsubset \top$. This is because $f(\sigma) \sqsubset \top$, so if there were no such σ' then σ would be a footprint, which is a contradiction. Note that the last step uses the fact that

$$\bigsqcap_{\sigma' \prec \sigma, f(\sigma') \sqsubset \top} \{\sigma - \sigma'\} * f(\sigma') = \bigsqcap_{\sigma' \prec \sigma} \{\sigma - \sigma'\} * f(\sigma') = L_f(\sigma)$$

because adding the top element to an intersection does not change its value. Since σ is not a footprint, $f(\sigma) = L_f(\sigma)$, and so $\phi \vdash (\{\sigma\}, f(\sigma))$. ■

2.4 Sufficiency and Small Specifications

In the last section it was shown that the relevance footprints are the only elements that are *essential* for a complete specification of a local function. We now investigate when a set of elements is *sufficient* for a complete specification of a local function, in the sense that a complete specification of the function can be constructed from only these elements. To study this, we first identify the notion of the *basis* of a local function.

Bases The local limit of a function f on a state σ was defined in the previous section as the constraint imposed on f by all the strict substates of σ. To address the question of which states are *sufficient* to determine the behaviour of f, we first generalise the local limit definition to consider the constraint imposed by only the substates taken from a given set.

Definition 2.18 (Local limit imposed by a set) *For a subset A of a separation algebra Σ, the* **local limit** *imposed by A on the action of f on σ is defined by*

$$L_{A,f}(\sigma) = \bigsqcap_{\sigma' \preceq \sigma, \sigma' \in A} \{\sigma - \sigma'\} * f(\sigma')$$

When the local limit imposed by a set A is enough to completely determine f, we call A a *basis* for f.

Definition 2.19 (Basis) *$A \sqsubseteq \Sigma$ is a* **basis** *for f, written $basis(A, f)$, if and only if $L_{A,f} = f$.*

This means that, when given the action of f on elements in A alone, we can determine the action of f on any element in Σ by just using the locality property of f. Every local function has at least one basis, namely the trivial basis Σ itself. We next show the correspondence between the bases and complete specifications of a local function.

Lemma 2.15 *Let $\phi_{A,f} = \{(\{\sigma\}, f(\sigma)) \mid \sigma \in A, f(\sigma) \sqsubset \top\}$. Then we have $basis(A, f) \Leftrightarrow complete(\phi_{A,f}, f)$.*

Proof: *We have $L_{A,f} = bla[\phi_{A,f}]$ by definition. The result follows by proposition 2.8 and the definition of basis.* ■

For every canonical complete specification $\phi \in \Phi_{can(f)}$, we have $\phi = \phi_{D(\phi),f}$. By the previous lemma it follows that $D(\phi)$ forms a basis for f. The lemma therefore shows that every basis determines a complete canonical specification, and vice versa. This correspondence also carries over to all complete specifications for f by the fact that every domain-equivalent class of complete specifications for f is represented by the canonical complete specification with that domain (corollary 2.10). By the essentiality of relevance footprints (theorem 2.14), it follows that the relevance footprints are present in every basis of a local function.

Lemma 2.16 *The relevance footprints of f are included in every basis of f.*

Proof: *Every basis A of f determines a complete specification for f the domain of which is a subset of A. By the essentiality theorem (2.14), the domain includes the footprints.* ■

The question of sufficiency is about how small the basis can get. Given a local function, we wish to know if it has a smallest basis. Since every basis must contain the footprints, it follows that if the relevance footprints alone form a basis then they are sufficient to construct a smallest complete specification. We find that for well-founded resource models, this is indeed the case.

Theorem 2.17 (Sufficiency I) *If a separation algebra Σ is well-founded under the $\preceq$ relation, then the relevance footprints of any local function form a basis for it: that is, $f = L_{F(f),f}$.*

Proof: Assume that Σ is well-founded under $\preceq$. We shall show by induction that $f(\sigma) = L_{F(f),f}(\sigma)$ for all $\sigma \in \Sigma$. The induction hypothesis is that, for all $\sigma' \prec \sigma$, $f(\sigma') = L_{F(f),f}(\sigma')$

case 1: Assume σ is a footprint of f. We have $f(\sigma) = \{u\} * f(\sigma)$, and so, by definition 2.18, we have $L_{F(f),f}(\sigma) \sqsubseteq f(\sigma)$. We have by locality that $f(\sigma) \sqsubseteq L_{F(f),f}(\sigma)$, and so $f(\sigma) = L_{F(f),f}(\sigma)$.

case 2: Assume σ is not a footprint of f. We have

$$\begin{aligned}
& f(\sigma) = L_f(\sigma) \quad \textit{(because } \sigma \textit{ is not a footprint of f)} \\
& = \bigsqcap_{\sigma' \prec \sigma} \{\sigma - \sigma'\} * f(\sigma') \\
& = \bigsqcap_{\sigma' \prec \sigma} \big(\{\sigma - \sigma'\} * \bigsqcap_{\sigma'' \preceq \sigma', F_f(\sigma'')} \{\sigma' - \sigma''\} * f(\sigma'')\big) \quad \textit{(by the induction hypothesis)} \\
& = \bigsqcap_{\sigma' \prec \sigma, \sigma'' \preceq \sigma', F_f(\sigma'')} \{\sigma - \sigma'\} * \{\sigma' - \sigma''\} * f(\sigma'') \quad \textit{(by the precision of } \{\sigma - \sigma'\}\textit{)} \\
& = \bigsqcap_{\sigma'' \prec \sigma, F_f(\sigma'')} \{\sigma - \sigma''\} * f(\sigma'') \\
& = \bigsqcap_{\sigma'' \preceq \sigma, F_f(\sigma'')} \{\sigma - \sigma''\} * f(\sigma'') \quad \textit{(because } \sigma \textit{ is not a footprint of f)} \\
& = L_{F(f),f}(\sigma)
\end{aligned}$$

■

In section 2.2, the notions of big and small specifications were introduced (definition 2.15), and the existence of a big specification was shown (proposition 2.11). We are now in a position to show the existence of the small specification for well-founded resource. If Σ is well-founded, then every local function has a small specification whose domain is the relevance footprints.

Corollary 2.18 (Small specification) *For well-founded separation algebras, every local function has a small specification given by $\phi_{F(f),f}$.*

Proof: *$\phi_{F(f),f}$ is complete by theorem 2.17 and lemma 2.15. It has the smallest domain by the essentiality theorem.* ■

Thus, for well-founded resource, the relevance footprints are always essential and sufficient, and specifications need not consider any other elements. Note that in practice, small specifications may not always be in canonical form, even though they always have the same domain as the canonical form. For example, the heap dispose command can have the specification $\{(\{l \mapsto v \mid v \in Val\}, \{u_H\})\}$ rather than the canonical one given by $\{(\{l \mapsto v\}, \{u_H\}) \mid v \in Val\}$.

Non-well-founded Resource In most practical situations the resource model is usually well-founded. A notable exception is the fractional permissions model [4] in which the resource includes ‘permissions to access’, which can be indefinitely divided. If a separation algebra is non-well-founded under the $\preceq$ relation, then there is some infinite descending chain of elements $\sigma_1 \succ \sigma_2 \succ \sigma_3$.... From a resource-oriented point of view, there are two distinct ways in which this could happen. One way is when it is possible to remove non-empty pieces of resource from a state indefinitely, as in the separation algebra of non-negative real numbers under addition. In this case any infinite descending chain does not have more than one occurrence of any element. Another way is when an infinite chain may exist because of repeated occurrences of some elements. This happens when there is *negativity* present in the resource: it may be possible to add two non-unit elements together to produce the unit element. An example is the separation algebra of integers under addition, where $1+(-1) = 0$, so adding -1 to 1 is like adding positive and negative resource to get nothing. Since $1 = 0 + 1$, we have that $1 \succ 0 \succ 1$... forms an infinite chain.

Definition 2.20 (Negativity) *A separation algebra Σ has* **negativity** *if and only if there exists a non-unit element $\sigma \in \Sigma$ that has an inverse; that is, $\sigma \neq u$ and $\sigma \bullet \sigma' = u$ for some $\sigma' \in \Sigma$. We say that Σ is* **non-negative** *if no such element exists.*

All separation algebras with negativity are non-well-founded, because for elements σ and σ' such that $\sigma \bullet \sigma' = u$, the set $\{\sigma, u\}$ forms an infinite descending chain as there is no least element. For the general non-negative case, we find that either the footprints form a basis, or there is no smallest basis.

Theorem 2.19 (Sufficiceny II) *If Σ is non-negative then, for any local f, either the relevance footprints form a smallest basis or there is no smallest basis for f.*

Proof: Let A be a basis for f (we know there is at least one, which is the trivial basis Σ itself). If A is the set of footprints then we are done. So assume A contains some non-footprint μ. We shall show that there exists a smaller basis for f, which is $A/\{\mu\}$. So it suffices to show $f(\sigma) = L_{A/\{\mu\},f}(\sigma)$ for all $\sigma \in \Sigma$. We have

$$f(\sigma) = L_{A,f}(\sigma) = \bigsqcap_{\sigma' \preceq \sigma, \sigma' \in A} \{\sigma - \sigma'\} * f(\sigma')$$

case 1: $\mu \not\preceq \sigma$. We have $f(\sigma) = \bigsqcap_{\sigma' \preceq \sigma, \sigma' \in A/\{\mu\}} \{\sigma - \sigma'\} * f(\sigma') = L_{A/\{\mu\},f}(\sigma)$

case 2: $\mu \preceq \sigma$. In this case

$$f(\sigma) = \Big(\bigsqcap_{\sigma' \preceq \sigma, \sigma' \in A/\{\mu\}} \{\sigma - \sigma'\} * f(\sigma')\Big) \quad \sqcap \quad (\{\sigma - \mu\} * f(\mu)) \quad (1)$$

To complete the proof, we need to show that $f(\sigma)$ is equal to the left-hand side of the above intersection:

$$f(\sigma) = \bigsqcap_{\sigma' \preceq \sigma, \sigma' \in A/\{\mu\}} \{\sigma - \sigma'\} * f(\sigma')$$

To show this, we just need to show that the right-hand side of the intersection in (1) contains the left-hand side:

$$\bigsqcap_{\sigma'' \preceq \sigma, \sigma'' \in A/\{\mu\}} \{\sigma - \sigma''\} * f(\sigma'') \sqsubseteq \{\sigma - \mu\} * f(\mu)$$

This is shown as follows:

$$\begin{aligned}
& \{\sigma - \mu\} * f(\mu) \\
& = \{\sigma - \mu\} * L_f(\mu) \quad \textit{(because } \mu \textit{ is not a footprint of f)} \\
& = \{\sigma - \mu\} * \bigsqcap_{\sigma' \prec \mu} \{\mu - \sigma'\} * f(\sigma') \\
& = \{\sigma - \mu\} * \bigsqcap_{\sigma' \prec \mu} \Big(\{\mu - \sigma'\} * \bigsqcap_{\sigma'' \preceq \sigma', \sigma'' \in A/\{\mu\}} \{\sigma' - \sigma''\} * f(\sigma'')\Big) \\
& \quad \textit{(case 1 applies because } \Sigma \textit{ is non-negative, so } \sigma' \prec \mu \Rightarrow \mu \npreceq \sigma'\textit{)} \\
& = \bigsqcap_{\sigma' \prec \mu} \bigsqcap_{\sigma'' \preceq \sigma', \sigma'' \in A/\{\mu\}} \{\sigma - \mu\} * \{\mu - \sigma'\} * \{\sigma' - \sigma''\} * f(\sigma'') \quad \textit{(by precision)} \\
& = \bigsqcap_{\sigma' \prec \mu} \bigsqcap_{\sigma'' \preceq \sigma', \sigma'' \in A/\{\mu\}} \{\sigma - \sigma''\} * f(\sigma'') \\
& = \bigsqcap_{\sigma'' \prec \mu, \sigma'' \in A/\{\mu\}} \{\sigma - \sigma''\} * f(\sigma'') \\
& \sqsupseteq \bigsqcap_{\sigma'' \preceq \sigma, \sigma'' \in A/\{\mu\}} \{\sigma - \sigma''\} * f(\sigma'')
\end{aligned}$$

■

Corollary 2.20 (Small Specification) *If Σ is non-negative, then every local function either has a small specification given by $\phi_{F(f),f}$ or there is no smallest complete specification for the function.*

Example 4 (Permissions) *The fractional permissions model [4] is non-well-founded and non-negative. It can be represented by the separation algebra $HPerm = L \rightharpoonup_{fin} Val \times P$ where L*

and Val are as in example 1, and P is the interval (0, 1] of rational numbers. Elements of P represent 'permissions' to access a heap cell. A permission of 1 for a cell means both read and write access, while any permission less than 1 is read-only access. The operator $\bullet$ joins disjoint heaps and adds the permissions together for any cells that are present in both heaps only if the resulting permission for each heap cell does not exceed 1; the operation is undefined otherwise. In this case, the write function that updates the value at a location requires a permission of at least 1 and faults on any smaller permission. It therefore has a small specification with pre-condition being the cell with permission 1. The read function, however, can execute safely on any positive permission, no matter how small. Thus, this function can be completely specified with a specification that has a pre-condition given by the cell with permission z, for all $0 < z \leq 1$. However, this is not a smallest *specification, as a smaller one can be given by further restricting $0 < z \leq 0.5$. We can therefore always find a smaller specification by reducing the value of z but keeping it positive.*

For resource with negativity, we find that it is possible to have small specifications that include non-essential elements. These elements are non-essential in the sense that there exist other complete specifications that do not include these elements.

Example 5 (Integers) *An example of a model with negativity is the separation algebra of integers $(\mathbb{Z}, +, 0)$. In this case there can be local functions which can have small specifications that contain non-footprints. Let $f : \mathbb{Z} \rightarrow \mathcal{P}(\mathbb{Z})^\top$ be defined as $f(n) = \{n + c\}$ for some constant c, as in example 1. f is local, but it has no footprints. This is because for any n, $f(n) = 1 + f(n-1)$, and so n is not a footprint of f. However, f does have small specifications, for example, $\{(\{0\}, \{c\})\}$, $\{(\{5\}, \{5 + c\})\}$, or indeed $\{(\{n\}, \{n + c\})\}$ for any $n \in \mathbb{Z}$. So although every element is non-essential, some element is required to give a complete specification.*

2.5 Regaining Safety Footprints

In the introduction we discussed how the notion of footprints as the smallest safe states - the *safety footprint*- is inadequate for giving complete specifications, as illustrated by the *AD* example. For this reason, so far we have investigated the general notion of relevance footprint of local functions. Equipped with this general theory, we now investigate how, with different resource modelling choices, we may refine the reasoning framework to obtain a correspondence between safety footprints and relevance footprints. We start by presenting an alternative model of heaps, based on an investigation of why the AD phenomenon occurs in the standard model. We then demonstrate that the footprints of the AD command in this new model do correspond to the safety footprints. In the final section we identify, for arbitrary separation algebras, a condition on local functions which guarantees the equivalence of the safety and relevance footprint. It is shown that if this condition is met by all the primitive commands of a programming language then the correspondence is achieved for every program in the language, and this is indeed the case in our new heap model.

An alternative model We begin by taking a closer look at why the *AD* anomaly occurs in the standard heap and stack model described in example 1.2. Consider an application of the allocation command in this model:

$$new[x](42 \mapsto v \bullet x \mapsto w) = \{42 \mapsto v \bullet x \mapsto l \bullet l \mapsto r \mid l \in L \backslash \{42\}, r \in Val\}$$

The intuition of locality is that the initial state $42 \mapsto v \bullet x \mapsto w$ is only describing a local region of the heap and the stack, rather than the whole global state. In this case it says that the address 42 is initially allocated, and the definition of the allocation command is that the resulting state will have a new cell, the address of which can be anything other than 42. However, we notice that the initial state is in fact not just describing only its local region of the heap. It does state that 42 is allocated, but it also implicitly states a very global property: that *all other addresses are not allocated*. This is why the allocation command can choose to allocate any location that is not 42. Thus in this model, every local state implicitly contains some global allocation information which is used by the allocation command. In contrast, a command such as mutate does not require this global ‘knowledge’ of the allocation status of any other cell that it is not affecting. Now the key point is that this global information about which cells are free in the heap *changes* as more resource is added to the initial state. This can lead to program behaviour being sensitive to the addition of more resource to the initial state, a sensitivity that can be observed in the case of the *AD* program.

Based on this observation, we consider an alternative model of the heap. As before, a state $l \mapsto v$

will represent a local allocated region of the heap at address l with value v. However, unlike before, this state will say nothing about the allocation status of any locations other than l. This information about the allocation status of other locations will be represented explicitly in a *free* set, which will contain every location that is not allocated in the *global heap*. The model can be interpreted from an ownership point of view, where the free set is to be thought of as a unique, atomic piece of resource, ownership of which needs to be obtained by a command if it wants to do allocation or deallocation. An analogy is with the permissions model: a command that wants to read or write to a cell needs ownership of the appropriate permission on that cell. In the same way, in our new model, a command that wants to do allocation or deallocation needs to have ownership of the free set: the 'permission' to see which cells are free in the global heap so that it can choose one of them to allocate, or update the free set with the address that it deallocates. On the other hand, commands that only read or write to cells shall not require ownership of the free set.

Example 6 (Heap model with free set) *Formally, we work with a separation algebra $(H, \bullet, u_H)$. Let L, Var and Val denote sets of locations, variables and values, as before. States $h \in H$ are given by the grammar:*

$$h ::= u_H \mid l \mapsto v \mid x \mapsto v \mid F \mid h \bullet h$$

where $l \in L$, $v \in Val$, $x \in Var$ and $F \in \mathcal{P}(L)$. The operator $\bullet$ is undefined for states with overlapping locations or variables. Let $loc(h)$ and $var(h)$ be the set of allocated locations and variables in state h respectively. The set F carries the information of which locations are free. Thus we allow at most one free set in a state, and the free set must be disjoint from all locations in the state. So $h \bullet F$ is only defined when $loc(h) \cap F = \emptyset$ and $h \neq h' \bullet F'$ for any h' and F'. We assume $\bullet$ is associative and commutative with unit u_H.

In this model, the allocation command requires ownership of the free set for safe execution, since it chooses the location to allocate from this set. It removes the chosen address from the free set as it allocates the cell. It is defined as

$$new[x](h) = \begin{cases} \{h' \bullet x \mapsto l \bullet l \mapsto w \bullet F \backslash \{l\} \mid w \in Val, l \in F\} & h = h' \bullet x \mapsto v \bullet F \\ \top & \textit{otherwise} \end{cases}$$

Note that the output states $h' \bullet x \mapsto l \bullet l \mapsto w \bullet F\backslash\{l\}$ are defined, since we have $l \notin F\backslash\{l\}$ and the input state $h' \bullet x \mapsto v \bullet F$ implies that $loc(h')$ is disjoint from $F\backslash\{l\}$. The deallocation command *also requires the free set, as it updates the set with the address of the cell that it deletes:*

$$dispose[x](h) = \begin{cases} \{h' \bullet x \mapsto l \bullet F \cup \{l\}\} & h = h' \bullet x \mapsto l \bullet l \mapsto v \bullet F \\ \top & \textit{otherwise} \end{cases}$$

Again, the output states are defined, since the input state implies that $loc(h') \cup \{l\}$ is disjoint from F, and so $loc(h')$ is disjoint from $F \cup \{l\}$. Notice that, in this model, only the allocation and deallocation commands require ownership of the free set, since commands such as mutation and lookup are completely independent of the allocation status of other cells, and they are defined exactly as in example 1.2:

$$mutate[x, v](h) = \begin{cases} \{h' \bullet x \mapsto l \bullet l \mapsto v\} & h = h' \bullet x \mapsto l \bullet l \mapsto w \\ \top & \textit{otherwise} \end{cases}$$

$$lookup[x, y](h) = \begin{cases} \{h' \bullet x \mapsto l \bullet l \mapsto v \bullet y \mapsto v\} & h = h' \bullet x \mapsto l \bullet l \mapsto v \bullet y \mapsto w \\ \top & \textit{otherwise} \end{cases}$$

Lemma 2.21 *The functions $new[x]$, $dispose[x]$, $mutate[x, v]$ and $lookup[x, y]$ are all local in the separation algebra $(H, \bullet, u_H)$ from example 6.*

Proof: *Let $f = new[x]$ and assume $h' \# h$. We want to show $f(h' \bullet h) \sqsubseteq \{h'\} * f(h)$. Assume $h = h'' \bullet x \mapsto v \bullet F$ for some h'', x, l, v and F, because otherwise $f(h) = \top$ and we are done. So we have*

$$\begin{aligned} f(h' \bullet h) &= \{h' \bullet h'' \bullet x \mapsto l \bullet l \mapsto w \bullet F \backslash \{l\} \mid w \in Val, l \in F\} \\ &= \{h'\} * \{h'' \bullet x \mapsto l \bullet l \mapsto w \bullet F \backslash \{l\} \mid w \in Val, l \in F\} \\ &= \{h'\} * f(h) \end{aligned}$$

The other functions can be checked in a similar way. ■

As an aside, a very interesting property of the free set model is that it provides a more robust treatment of dynamic memory allocation than the standard model. In the standard model, in order to keep commands local, one is forced to work with a non-deterministic allocation function which assumes an infinite amount of memory, which is obviously not the case in the real world. We find that in the free set model, we can very naturally model deterministic allocation or allocation in bounded memory as local functions.

Example 7 (Deterministic and bounded memory allocation) *In the free set model of example 6, we can define a deterministic allocation function as*

$$new_d[x](h) = \begin{cases} \{h' \bullet x \mapsto l \bullet l \mapsto w \bullet F \backslash \{l\}\} & h = h' \bullet x \mapsto v \bullet F \ \wedge \ l = N(F) \\ \top & \textit{otherwise} \end{cases}$$

where $N(F) : \mathcal{P}(L) \to L$ is a function that chooses a specific location from F to be allocated next. We can define (non-deterministic) allocation in bounded memory by allowing the set of locations L to be finite, and defining

$$new_b[x](h) = \begin{cases} \{h' \bullet x \mapsto l \bullet l \mapsto w \bullet F \backslash \{l\} \mid w \in Val, l \in F\} & h = h' \bullet x \mapsto v \bullet F \ \wedge \ F \neq \emptyset \\ \top & \textit{otherwise} \end{cases}$$

We can similarly define deterministic allocation in bounded memory by choosing a specific location using $N(F)$ rather than all possible locations in F. Each of these functions can be checked to be local in the free set model.

Safety and relevance footprint correspondence for *AD* We consider the relevance footprint of the *AD* command in the new model. The sequential composition $new[x]; dispose[x]$ gives the function

$$AD(h) = \begin{cases} \{h' \bullet x \mapsto l \bullet F \mid l \in F\} & h = h' \bullet x \mapsto v \bullet F \\ \top & \text{otherwise} \end{cases}$$

The smallest safe states are given by the set $\{x \mapsto v \bullet F \mid v \in Val, F \in \mathcal{P}(L)\}$. By lemma 2.13, these smallest safe states are footprints. However, unlike before, in this model these are the *only* relevance footprints of the AD command. To see this, consider a larger state $h \bullet x \mapsto v \bullet F$ for non-empty h. We have

$$\begin{aligned} AD(h \bullet x \mapsto v \bullet F) &= \{h \bullet x \mapsto l \bullet F \mid l \in F\} \\ &= \{h\} * \{x \mapsto l \bullet F \mid l \in F\} \\ &= \{h\} * AD(x \mapsto v \bullet F) \end{aligned}$$

Since the local limit $L_{AD}(h \bullet x \mapsto v \bullet F) \sqsubseteq \{h\} * AD(x \mapsto v \bullet F)$ by definition, we have by proposition 2.12 that $L_{AD}(h \bullet x \mapsto v \bullet F) = AD(h \bullet x \mapsto v \bullet F)$, and so $h \bullet x \mapsto v \bullet F$ is not a footprint of AD.

Thus the relevance footprints of AD in this model do not include any non-empty heaps. By corollary 2.18, in this model the AD command has a smallest complete specification in which the pre-condition is just the empty heap:

$$\{(\{x \mapsto v \bullet F\}, \{x \mapsto l \bullet F\}) \mid v \in Val, F \in \mathcal{P}(L), l \in F\}$$

Intuitively, it says that if initially the heap is empty, the variable x is present in the stack, and we know which cells are free in the global heap, then after the execution, the heap will still be empty, exactly the same cells will still be free, and x will point to one of those free cells. This completely describes the behaviour of the command for all larger states using the frame rule. For example, we get the complete specification on the larger state in which 42 is allocated:

$$\{(\{42 \mapsto w\} * \{x \mapsto v \bullet F\}, \{42 \mapsto w\} * \{x \mapsto l \bullet F\}) \mid v, w \in Val, F \in \mathcal{P}(L), l \in F\}$$

In the pre-condition, the presence of location 42 in the heap means that 42 is not in the free set F (by definition of $*$). Therefore, in the post-condition, x cannot point to 42. Notice that in order to check that we have 'regained' safety footprints, we only needed to check that the footprint definition (definition 2.17) corresponds to the smallest safe states. The desired properties such as essentiality, sufficiency, and small specifications then follow by the results established in previous sections.

Correspondence for arbitrary programs Now that we have regained the safety footprints for AD in the new model, we want to know if this is generally the case for any program. We consider

$$[\![c]\!] \in LocFunc \qquad [\![\texttt{skip}]\!](\sigma) = \{\sigma\}$$

$$[\![C_1; C_2]\!] = [\![C_1]\!]; [\![C_2]\!] \qquad [\![C_1 + C_2]\!] = [\![C_1]\!] \sqcup [\![C_2]\!] \qquad [\![C^\star]\!] = \bigsqcup_n [\![C^n]\!]$$

Figure 2.2: Denotational semantics for the imperative programming language

the imperative programming language given in [11]:

$$C \quad ::= \quad c \mid \texttt{skip} \mid C;C \mid C + C \mid C^\star$$

where c ranges over an arbitrary collection of primitive commands, $+$ is nondeterministic choice, ; is sequential composition, and $(\cdot)^\star$ is Kleene-star (iterated ;). As discussed in [11], conditionals and while loops can be encoded using $+$ and $(\cdot)^\star$ and assume statements. The denotational semantics of commands is given in Figure 2.2.

Taking the primitive commands to be $new[x]$, $dispose[x]$, $mutate[x, v]$, and $lookup[x, y]$, our original aim was to show that, for every command C, the footprints of $[\![C]\!]$ in the new model are the smallest safe states. However, in attempting to do this, we identified a general condition on primitive commands under which the result holds for arbitrary separation algebras.

Let f be a local function on a separation algebra Σ. If, for $A \in \mathcal{P}(\Sigma)$, we define $f(A) = \bigsqcup_{\sigma \in A} f(\sigma)$, then the locality condition (definition 2.5) can be restated as

$$\forall \sigma', \sigma \in \Sigma.\ f(\{\sigma'\} * \{\sigma\}) \sqsubseteq \{\sigma'\} * f(\{\sigma\})$$

The $\sqsubseteq$ ordering in this definition allows local functions to be more deterministic on larger states. This sensitivity of determinism to larger states is apparent in the AD command in the standard model from example 1.2. On the empty heap, the command produces an empty heap, and reassigns variable x to *any* value, while on the singleton cell 1, it disallows the possibility that $x = 1$ afterwards. In the new model, the AD command does not have this sensitivity of determinism in the output states. In this case, the presence or absence of the cell 1 does not affect the outcomes of the AD command, since the command can only assign x to a value chosen from the free set, which does not change no matter what additional cells may be framed in. With this observation, we consider the general class of local functions in which this sensitivity of determinism is not present.

Definition 2.21 (Determinism Constancy) *Let f be a local function and $safe(f)$ the set of states on which f does not fault. The function f has the determinism constancy property if and only if, for every $\sigma \in safe(f)$,*

$$\forall \sigma' \in \Sigma.\ f(\{\sigma'\} * \{\sigma\}) = \{\sigma'\} * f(\{\sigma\})$$

Notice that the determinism constancy property by itself implies that the function is local, and it can therefore be thought of as a form of 'strong locality'. Firstly, we find that local functions that have determinism constancy always have footprints given by the smallest safe states.

Lemma 2.22 *If a local function f has determinism constancy then its footprints are the smallest safe states.*

Proof: *Let $min(f)$ be the smallest safe states of f. These are footprints by lemma 2.13. For any larger state $\sigma' \bullet \sigma$ where $\sigma \in min(f)$, $\sigma' \in \Sigma$ and σ is non-empty, we have*

$$f(\sigma' \bullet \sigma) = f(\{\sigma'\} * \{\sigma\}) = \{\sigma'\} * f(\sigma)$$

Since $L_f(\sigma' \bullet \sigma) \sqsubseteq \{\sigma'\} * f(\sigma)$, by proposition 2.12 we have that $L_f(\sigma' \bullet \sigma) = f(\sigma' \bullet \sigma)$, and so $\sigma' \bullet \sigma$ is not a footprint of f. ■

We now demonstrate that the determinism constancy property is preserved by all the constructs of our programming language. This implies that if all the primitive commands of the programming language have determinism constancy, then the footprints of every program are the smallest safe states.

Theorem 2.23 *If all the primitive commands of the programming language have determinism constancy, then the footprint of every program is given by the smallest safe states.*

Proof: Assuming all primitive commands have determinism constancy, we shall show by induction that every composite command has determinism constancy and the result follows by lemma 2.22. So for commands C_1 and C_2, let $f = [\![C_1]\!]$ and $g = [\![C_2]\!]$ and assume f and g have determinism constancy. For sequential composition we have, for $\sigma \in \mathit{safe}(f;g)$ and $\sigma' \in \Sigma$,

$$\begin{array}{cl}
 & (f;g)(\{\sigma'\} * \{\sigma\}) \\
= & g(f(\{\sigma'\} * \{\sigma\})) \\
= & g(\{\sigma'\} * f(\{\sigma\})) \\
 & (f \text{ has determinism constancy and } \sigma \in \mathit{safe}(f) \text{ since } \sigma \in \mathit{safe}(f;g)) \\
= & g(\bigsqcup_{\sigma_1 \in f(\sigma)} \{\sigma'\} * \{\sigma_1\}) \\
= & \bigsqcup_{\sigma_1 \in f(\sigma)} g(\{\sigma'\} * \{\sigma_1\}) \\
= & \bigsqcup_{\sigma_1 \in f(\sigma)} \{\sigma'\} * g(\sigma_1) \\
 & (g \text{ has determinism constancy and } \sigma_1 \in \mathit{safe}(g) \text{ since } \sigma \in \mathit{safe}(f;g) \text{ and } \sigma_1 \in f(\sigma)) \\
= & \{\sigma'\} * \bigsqcup_{\sigma_1 \in f(\sigma)} g(\sigma_1) \quad \text{(distributivity)} \\
= & \{\sigma'\} * (f;g)(\sigma)
\end{array}$$

For non-deterministic choice, we have for $\sigma \in \mathit{safe}(f+g)$ and $\sigma' \in \Sigma$,

$$\begin{array}{cl}
 & (f+g)(\{\sigma'\} * \{\sigma\}) \\
= & f(\{\sigma'\} * \{\sigma\}) \sqcup g(\{\sigma'\} * \{\sigma\}) \\
= & \{\sigma'\} * f(\{\sigma\}) \sqcup \{\sigma'\} * g(\{\sigma\}) \\
 & (f \text{ and } g \text{ have determinism constancy and } \sigma \in \mathit{safe}(f) \text{ and } \sigma \in \mathit{safe}(g)) \\
= & \{\sigma'\} * (f(\{\sigma\}) \sqcup g(\{\sigma\})) \quad \text{(distributivity)} \\
= & \{\sigma'\} * (f+g)(\{\sigma\})
\end{array}$$

For Kleene-star, we have for $\sigma \in \mathit{safe}(f^\star)$ and $\sigma' \in \Sigma$,

$$\begin{aligned}
& (f^\star)(\{\sigma'\} * \{\sigma\}) \\
= \;& \bigsqcup_n f^n(\{\sigma'\} * \{\sigma\}) \\
= \;& \bigsqcup_n \{\sigma'\} * f^n(\{\sigma\}) \\
& \text{(determinism constancy preserved under sequential composition and } \sigma \in \mathit{safe}(f^n)\text{)} \\
= \;& \{\sigma'\} * \bigsqcup_n f^n(\{\sigma\}) \quad \text{(distributivity)} \\
= \;& \{\sigma'\} * (f^\star)(\{\sigma\})
\end{aligned}$$

■

Now that we have shown the general result, it remains to check that all the primitive commands in the new model of example 6 do have determinism constancy.

Proposition 2.24 *Let H_1 be the stack and heap model of example 1.2 and H_2 be the alternative model of example 6. The commands $\mathit{new}[x]$, $\mathit{mutate}[x, v]$ and $\mathit{lookup}[x, y]$ all have determinism constancy in both models. The $\mathit{dispose}[x]$ command has determinism constancy in H_2 but not in H_1.*

Proof: We give the proofs for the new and dispose commands in the two models, and the cases for mutate and lookup can be checked in a similar way. For $\mathit{dispose}[x]$ in H_1, the following counterexample shows that it does not have determinism constancy.

$$\begin{aligned}
& \mathit{dispose}[x](\{l \mapsto v\} * \{x \mapsto l \bullet l \mapsto w\}) \\
= \;& \mathit{dispose}[x](\emptyset) \\
= \;& \emptyset \\
\sqsubset \;& \{l \mapsto v \bullet x \mapsto l\} \\
= \;& \{l \mapsto v\} * \mathit{dispose}[x](x \mapsto l \bullet l \mapsto w)
\end{aligned}$$

For $\mathit{new}[x]$ in H_1, any safe state is of the form $h \bullet x \mapsto v$. For any $h' \in H_1$, we have

$$\{h'\} * \mathit{new}[x](h \bullet x \mapsto v) = \{h'\} * \{h \bullet x \mapsto l \bullet l \mapsto w \mid w \in Val, l \in L \backslash loc(h)\} \quad (\dagger)$$

If $h' \bullet h \bullet x \mapsto v$ is undefined then h' shares locations with $loc(h)$ or variables with $var(h) \cup \{x\}$. This means that the RHS in † is the empty set. We have $new[x](\{h'\} * \{h \bullet x \mapsto v\}) = new[x](\emptyset) = \emptyset = \{h'\} * new[x](h \bullet x \mapsto v)$. If $h' \bullet h \bullet x \mapsto v$ is defined, then

$$\begin{array}{rl} & new[x](\{h'\} * \{h \bullet x \mapsto v\}) \\ = & new[x](h' \bullet h \bullet x \mapsto v) \\ = & \{h' \bullet h \bullet x \mapsto l \bullet l \mapsto w \mid w \in Val, l \in L \backslash loc(h' \bullet h)\} \\ = & \{h'\} * \{h \bullet x \mapsto l \bullet l \mapsto w \mid w \in Val, l \in L \backslash loc(h' \bullet h)\} \\ = & \{h'\} * \{h \bullet x \mapsto l \bullet l \mapsto w \mid w \in Val, l \in L \backslash loc(h)\} \\ = & \{h'\} * new[x](h \bullet x \mapsto v) \end{array}$$

For $dispose[x]$ in H_2, any safe state is of the form $h \bullet x \mapsto l \bullet l \mapsto v \bullet F$. Let $h' \in H_2$. We have

$$\{h'\} * dispose[x](h \bullet x \mapsto l \bullet l \mapsto v \bullet F) = \{h'\} * \{h \bullet x \mapsto l \bullet F \cup \{l\}\} \quad (\dagger\dagger)$$

If $h' \bullet h \bullet x \mapsto l \bullet l \mapsto v \bullet F$ is undefined then either h' contains a free set or it contains locations in $loc(h) \cup \{l\}$ or variables in $var(h) \cup \{x\}$. If h' contains a free set or it contains locations in $loc(h)$ or variables in $var(h) \cup \{x\}$, then the RHS in †† is the empty set. If h' contains the location l then also the RHS in †† is the empty set since the free set $F \cup \{l\}$ also contains l. Thus in both cases the RHS in †† is the empty set, and we have $dispose[x](\{h'\} * \{h \bullet x \mapsto l \bullet l \mapsto v \bullet F\}) = \emptyset = \{h'\} * dispose[x](h \bullet x \mapsto l \bullet l \mapsto v \bullet F)$.

If $h' \bullet h \bullet x \mapsto l \bullet l \mapsto v \bullet F$ is defined then we have

$$\begin{array}{rl} & dispose[x](\{h'\} * \{h \bullet x \mapsto l \bullet l \mapsto v \bullet F\}) \\ = & dispose[x](h' \bullet h \bullet x \mapsto l \bullet l \mapsto v \bullet F) \\ = & \{h' \bullet h \bullet x \mapsto l \bullet F \cup \{l\}\} \\ = & \{h'\} * \{h \bullet x \mapsto l \bullet F \cup \{l\}\} \\ = & \{h'\} * dispose[x](h \bullet x \mapsto l \bullet l \mapsto v \bullet F) \end{array}$$

For $new[x]$ in H_2, any safe state is of the form $h \bullet x \mapsto v \bullet F$. Let $h' \in H_2$. We have

$$\{h'\} * new[x](h \bullet x \mapsto v \bullet F) = \{h'\} * \{h \bullet x \mapsto l \bullet l \mapsto w \bullet F \backslash \{l\} \mid w \in Val, l \in F\} \quad (\dagger\dagger\dagger)$$

If $h' \bullet h \bullet x \mapsto v \bullet F$ is undefined then either h' contains a free set or it contains locations in $loc(h)$ or variables in $var(h) \cup \{x\}$. In all these cases the RHS in †† † is the empty set, and so we have $new[x](\{h'\} * \{h \bullet x \mapsto v \bullet F\}) = \emptyset = \{h'\} * new[x](h \bullet x \mapsto v \bullet F)$.

If $h' \bullet h \bullet x \mapsto v \bullet F$ is defined then we have

$$\begin{aligned}
& new[x](\{h'\} * \{h \bullet x \mapsto v \bullet F\}) \\
= \; & new[x](h' \bullet h \bullet x \mapsto v \bullet F) \\
= \; & \{h' \bullet h \bullet x \mapsto l \bullet l \mapsto w \bullet F \backslash \{l\} \mid w \in Val, l \in F\} \\
= \; & \{h'\} * \{h \bullet x \mapsto l \bullet l \mapsto w \bullet F \backslash \{l\} \mid w \in Val, l \in F\} \\
= \; & \{h'\} * new[x](h \bullet x \mapsto v \bullet F)
\end{aligned}$$

■

Thus theorem 2.23 and proposition 2.24 tell us that using the alternative model of example 6, the footprint of every program is given by the smallest safe states, and hence we have regained safety footprints for all programs. In fact, the same is true for the original model of example 1.2 if we do not include the dispose command as a primitive command, since all the other primitive commands have determinism constancy. This, for example, would be the case when modelling a garbage collected language [43].

2.6 Conclusion

We have developed a general theory of relevance footprints in the abstract setting of local functions that act on separation algebras. Based on the definition of locality, we introduced the formal definition of the relevance footprint of a local function, demonstrated essentiality of relevance footprints and identified the conditions for sufficiency. The theory of relevance footprints was then used to characterize the conditions under which the safety footprint provides complete specifications. We introduced an alternative model of heaps in which the correspondence between safety and relevance footprints is achieved, and identified the general property of determinism constancy which guarantees the correspondence in arbitrary resource models.

Apart from the natural correspondence between safety and relevance footprints, the new heap model also provides a more robust treatment of memory allocation, in which deterministic memory allocation and allocation in a finite amount of memory may be modelled as local functions. This is not true of the standard heap model, where memory allocation must necessarily be non-deterministic and an unbounded amount of memory is assumed. It is a direction for future work to explore this stronger treatment of allocation in practical applications, such as the verification of memory usage in embedded systems.

Finally, we comment on some related work. The discussion here has been based on the static notion of footprints as *states* of the resource on which a program acts. A different notion of footprint has recently been described in [28], where footprints are viewed as *traces* of execution of a computation. O'Hearn has described how the AD problem is avoided in this more elaborate semantics, as the allocation of cells in an execution prevents the framing of those cells. Interestingly, however, the heap model from example 6 illustrates that it is not essential to move to this more elaborate setting and incorporate dynamic, execution-specific information into the notion of footprint in order to resolve the AD problem. Instead, with the explicit representation of free cells in states, one can remain in an extensional semantics and have a purely resource-based view of footprints.

Chapter 3

Dependence Analysis for Optimization

3.1 Introduction

In this chapter we investigate how the resource reasoning provided by separation logic can be used to determine dependences between program statements, which is the key to effecting optimizations such as automatic parallelization. Optimization techniques are generally based on a detection of the resources accessed by program statements. Such techniques have been extensively studied and successfully applied for programs with simple data types and arrays, but there has been limited progress for programs that manipulate pointers and dynamic data structures [25, 26, 29]. Separation logic has made significant advances in automated verification of such heap-manipulating programs [55, 8, 1, 24, 22], but these analyses cannot be used for program parallelization. This is because the $*$ connective can only express separation of memory at a single program point, and therefore cannot relate memory regions in different states in the execution of a program.

We introduce *labelled separation logic* to analyse memory separation properties throughout a program's lifetime, which involves annotating formulae with *labels* to keep track of memory regions that are accessed by commands. This label-tracking is implemented in a method of symbolic execution first presented in [3], which is the basis of automated program analysis with separation logic. The technique allows us to determine heap-access dependences between commands in the program, but we find that these dependences alone are not sufficient to safely optimize a program. The ultimate aim is to ensure that any optimization should produce the same output states as the original program. Program optimizations have often been based on the assumption that if two commands access separate heap and variables in all possible executions, then they can be parallelized or reordered to give an equivalent program [21, 31]. We describe here how this assumption actually does not hold in the presence of dynamic memory allocation and deallocation, and that optimizations based on the assumption can produce results that are different from the original program. We discuss example programs to illustrate the problem, and introduce the notion of *allocation dependences*, in addition to heap and stack dependences, in order to guarantee the safety of optimizations. We formally demonstrate the soundness of the optimizations based on these dependences using a trace semantics of programs.

Our approach is part of a wider field of using static analysis to detect dependences in programs that manipulate dynamic pointer data structures [26, 21, 30, 29, 38]. The departure point is the use of separation logic, which gives our approach the potential for scalablity [55] and compositionality [8]. Our method of labelling memory regions with labels of the accessing commands is similar to [29], but the underlying abstract domain there is based on the memory layout approximations of [33] rather than spatial logic formulae. A logic-based approach is also advocated in [30], where *aliasing axioms* and theorem proving are used to detect independence. However, this method has difficulty handling structural modifications to the data structure, which do not cause problems in our case. Our method also does not rely on *reachability* properties of data structures, as in [26]. Such approaches encounter difficulties with data structure ‘segments’, such as non-nil-terminated list segments, or when there is internal sharing within the data structure as in the case of doubly linked lists. Our approach does not suffer from these inherent limitations as it is based on detecting the cells that are actually accessed rather than all the ones that may possibly be accessed.

Shortly after the conference paper on which this work is based [48], a different approach to using separation logic for optimization was proposed in [31], based on the idea of proof rewriting. In this case rewriting rules for transforming a program proof into a proof for an optimized program are presented. This method encounters complications in comparing non-consecutive statements in programs, since formulae at distant program points may refer to different memory regions even if they are syntactically the same. In contrast, the label-tracking mechanism we introduce here provides a simple method for tracking memory regions through an execution in order to compare distant statements. Another difference is that the proof rewriting method directly implements program transformations and does not provide dependence information. In practice, it is better to supply dependence information to the compiler and leave it the choice of which optimizations to effect, since the practicality of different possible optimizations may depend on a number of circumstantial factors such as the cost of creating threads or the context of the program and input data. Having made these comparisons, we remark that both our approach and [31] are based on the insight that separation logic is a very useful tool not just for verification, but for optimization as well.

In this chapter we illustrate our method in the standard symbolic execution framework for separation logic, working with formulae describing linked lists and tree structures as in [3]. The proposed method is engineered so that it can be applied as a post-processing phase starting from the output of an existing shape analysis based on separation logic. In the next section we introduce labelled symbolic heaps, which are separation logic formulae extended with labels. We then describe the extended symbolic execution algorithm for determining dependences and discuss examples. We end with a proof of soundness of the method.

3.2 Labelled Symbolic Heaps

Automated analysis based on separation logic is usually implemented in a fragment of the logic known as *symbolic heaps*, which were first introduced in [3] and have since become the standard for such analyses [17, 22, 7, 1, 55, 8, 9]. We first give a brief description of symbolic heaps and then describe our extension to *labelled symbolic heaps*, which shall be used in the analysis for detecting dependences between program statements.

The concrete heap model is based on a set of fields `Fields`, a set of heap locations `Loc`, and a set `Val` of values that include `Loc` and the nil value $\texttt{nil} \in \texttt{Val}$. We assume a finite set `Var` of program variables and an infinite set $\texttt{Var}'$ of primed variables. Primed variables are logical variables that will not be used in programs, only in formulae where they will be implicitly existentially quantified. We then set

$$\texttt{Heaps} = \texttt{Loc} \rightharpoonup_{fin} (\texttt{Fields} \to \texttt{Val})$$

$$\texttt{Stacks} = (\texttt{Var} \cup \texttt{Var}') \to \texttt{Val}$$

The standard symbolic heap fragment is shown in figure 3.1. We let `SH` be the set of all symbolic heaps. The semantic interpretation is shown in figure 3.2, which uses a forcing relation $s, h \models A$

$x, y, ..$	$\in$	$\texttt{Var}$	program variables
$x', y', ..$	$\in$	$\texttt{Var}'$	primed variables
$f_1, f_2, ..$	$\in$	$\texttt{Fields}$	fields
E, F	$::=$	$\texttt{nil} \mid x \mid x'$	expressions
ρ	$::=$	$f_1 : E_1, ..., f_k : E_k$	record expressions
π	$::=$	$E = E \mid E \neq E$	simple pure formulae
Π	$::=$	$\texttt{true} \mid \pi \mid \Pi \wedge \Pi$	pure formulae
S	$::=$	$E \mapsto [\rho] \mid \texttt{ls}(E, F) \mid \texttt{tree}(E)$	simple spatial formulae
Ψ	$::=$	$\texttt{emp} \mid S \mid \Psi * \Psi$	spatial formulae
U	$::=$	$\Pi \wedge \Psi$	symbolic heaps

Figure 3.1: Standard Symbolic Heaps

where $s \in \texttt{Stacks}$, $h \in \texttt{Heaps}$, and A is any pure formula, spatial formula, or symbolic heap. Expressions are program or logical variables, or `nil`. Pure formulae are a conjunction of equalities or inequalities of expressions interpreted on the variable stack s, while spatial formulae specify properties of the heap h.

The simple spatial formulae include the points-to assertion $E \mapsto [\rho]$, which describes a single allocated heap cell with address E and contents described by record expression ρ, and we have inductively defined predicates for linked list segments and binary trees. These data structures use the fields $n, l, r \in \texttt{Fields}$, where n is the next field for list segments, and l and r are the left and right subtree fields for binary trees. We have the spatial formula `emp` which describes the empty heap in which there are no allocated cells. The spatial conjunction $\Psi_1 * \Psi_2$ of two spatial formulae uses the separating conjunction of separation logic. It holds for a heap h if the heap can be split into two disjoint parts h_1 and h_2 such that Ψ_1 holds in h_1 and Ψ_2 in h_2.

An overall symbolic heap $U = \Pi \wedge \Psi$ is the classical conjunction of the the pure formula Π and the spatial formula Ψ describing properties of the stack and heap respectively, with the interpretation of every primed variable as existentially quantified. The notation $s(x'_1 \mapsto v_1, \ldots, x'_n \mapsto v_n)$ in figure 3.2 represents the stack s in which the variables $x'_1, \ldots, x'_n$ have values $v_1, \ldots, v_n$.

In order to detect dependences between different parts of the program, our analysis will need to track regions of the heap during execution, and we do this with a notion of *labeling* on the simple spatial formulae in symbolic heaps. We assume a fixed set of labels `Lab`, and define a *labelled* symbolic heap as symbolic heap in which every simple spatial conjunct is assigned a set of labels from `Lab`. Formally, labelled symbolic heaps are given by the grammar:

$$[\![x]\!]s = s(x) \qquad [\![x']\!]s = s(x') \qquad [\![\mathtt{nil}]\!]s = \mathtt{nil}$$

$$\begin{array}{lll}
s,h \models E_1 = E_2 & \text{iff} & [\![E_1]\!]s = [\![E_2]\!]s \\
s,h \models E_1 \neq E_2 & \text{iff} & [\![E_1]\!]s \neq [\![E_2]\!]s \\
s,h \models \mathtt{true} & & \textit{always} \\
s,h \models \Pi_0 \wedge \Pi_1 & \text{iff} & s,h \models \Pi_0 \textit{ and } s,h \models \Pi_1 \\
s,h \models E_0 \mapsto [f_1 : E_1, ..., f_k : E_k] & \text{iff} & h = [[\![E_0]\!]s \rightarrow r] \text{ where } r(f_i) = [\![E_i]\!]s \textit{ for } i \in 1..k \\
s,h \models \mathtt{ls}(E,F) & \text{iff} & \textit{there is a linked list segment from } E \textit{ to } F \\
s,h \models \mathtt{tree}(E) & \text{iff} & \textit{there is a binary tree at } E \\
s,h \models \mathtt{emp} & \text{iff} & h = \emptyset \\
s,h \models \Psi_0 * \Psi_1 & \text{iff} & \exists h_0, h_1 .\, h = h_0 * h_1 \textit{ and } s,h_0 \models \Psi_0 \textit{ and } s,h_1 \models \Psi_1 \\
s,h \models \Pi \wedge \Psi & \text{iff} & \exists v_1, \ldots, v_n .\, s(x'_1 \mapsto v_1, \ldots, x'_n \mapsto v_n), h \models \Pi \\
 & & \textit{and } s(x'_1 \mapsto v_1, \ldots, x'_n \mapsto v_n), h \models \Psi \\
 & & \textit{where } x'_1, \ldots, x'_n \textit{ are all the primed variables in } \Pi \wedge \Psi
\end{array}$$

The list and tree formulae are formally defined as the least predicates satisfying the inductive definitions:

$$\begin{array}{lll}
\mathtt{ls}(E,F) & \Leftrightarrow & (E = F \wedge \mathtt{emp}) \vee (E \neq F \wedge \exists y. E \mapsto [n : y] * \mathtt{ls}(y,F)) \\
\mathtt{tree}(E) & \Leftrightarrow & (E = \mathtt{nil} \wedge \mathtt{emp}) \vee (\exists x,y. E \mapsto [l : x, r : y] * \mathtt{tree}(x) * \mathtt{tree}(y))
\end{array}$$

Figure 3.2: Interpretation of Symbolic Heaps

l	$\in$	$\mathtt{Lab}$	labels
L	$\in$	$P(\mathtt{Lab})$	label sets
Σ	$::=$	$\mathtt{emp} \mid \langle S \rangle_L \mid \Sigma * \Sigma$	labelled spatial formulae
H	$::=$	$\Pi \wedge \Sigma$	labelled symbolic heaps

We write $\mathtt{Lab}(H)$, $\mathtt{Var}(H)$ and $\mathtt{Var}'(H)$ for the set of all labels, program variables and primed variables in H respectively. The algorithm will use labels that are indices of program statements to mark the heap region that is accessed in the execution of a statement. For example, a formula

$$\langle \mathtt{ls}(x, \mathtt{nil}) \rangle_{\{19,42\}} * \langle \mathtt{ls}(y, \mathtt{nil}) \rangle_{\{3\}}$$

at some point in the symbolic execution expresses that statements 19 and 42 have accessed list x but not list y, while statement 3 accessed list y but not list x.

We let LSH be the set of all labelled symbolic heaps. Labelled symbolic heaps are given a formal interpretation by extending concrete heaps so that every location maps to a heap cell and a set of labels, that is,

$$\texttt{Heaps} = \texttt{Loc} \rightharpoonup_{fin} ((\texttt{Fields} \rightarrow \texttt{Val}) \times P(\texttt{Lab}))$$

The label set for each heap cell contains the labels of all the commands that have accessed that cell. This component will be updated with new labels as the program execution proceeds. Concrete labelled heaps can satisfy both unlabelled and labelled symbolic heaps. For unlabelled symbolic heaps, the interpretation ignores the labels in the concrete heap. For labelled symbolic heaps, the label sets in spatial formulae are an over-approximation of the label sets in the concrete heap. The formal definition is as follows.

Definition 3.1 (Satisfaction) *Assume we have a stack s and labelled concrete heap h. For an unlabelled symbolic heap U, we have $s, h \models U$ as defined in figure 3.2.*

$$\begin{array}{lll}
s,h \models \langle S\rangle_L & \textit{iff} & s,h \models S \textit{ and for all } \ell \in dom(h),\ \textit{if } h(\ell) = (r, L') \textit{ then } L' \subseteq L \\
s,h \models \Sigma_0 * \Sigma_1 & \textit{iff} & \exists h_0, h_1.\, h = h_0 * h_1 \textit{ and } s, h_0 \models \Sigma_0 \textit{ and } s, h_1 \models \Sigma_1 \\
s,h \models \Pi \wedge \Sigma & \textit{iff} & \exists v_1, \ldots, v_n.\ s(x'_1 \mapsto v_1, \ldots, x'_n \mapsto v_n), h \models \Pi \ \textit{ and} \\
 & & s(x'_1 \mapsto v_1, \ldots, x'_n \mapsto v_n), h \models \Sigma \\
 & & \textit{where } x'_1, \ldots, x'_n \textit{ are all the primed variables in } \Pi \wedge \Sigma
\end{array}$$

Notice that the symbolic heap only gives an over-approximation of the labels of the concrete heap. For example, assume we have s, h where h contains separate lists at x and y, the head of the list at x has label set $\{l\}$, and the label sets of the rest of the heap are all empty. Then we have

$$s,h \models \langle x \mapsto [n : x']\rangle_{\{l\}} * \langle \texttt{ls}(x', y)\rangle_{\emptyset} * \langle \texttt{ls}(y, \texttt{nil})\rangle_{\emptyset}$$

$$s,h \models \langle \texttt{ls}(x, y)\rangle_{\{l\}} * \langle \texttt{ls}(y, \texttt{nil})\rangle_{\{l\}}$$

$$s,h \not\models \langle \texttt{ls}(x, y)\rangle_{\emptyset} * \langle \texttt{ls}(y, \texttt{nil})\rangle_{\emptyset}$$

Definition 3.2 (Entailment) *An entailment $H \vdash H'$ between two labelled symbolic heaps holds if and only if every concrete state s, h that satisfies H also satisfies H'.*

We define a general formula as a set $P \in \mathcal{P}(\texttt{LSH})$ of labelled symbolic heaps, representing the disjunction of all the heaps in P.

3.3 Dependence Analysis

We now present the analysis for detecting dependences between statements in a program that is written in an imperative programming language with heap manipulating commands, while loops and procedures. For such programs, we first perform an inter-procedural shape analysis based on separation logic, such as the one described in [8], which gives us specifications for procedure calls and while loops. These specifications are pre- and post-conditions given in terms of standard (unlabelled) symbolic heaps as shown in figure 3.1. Formally, a specification obtained from the shape analysis is given as a *spec table*.

Definition 3.3 (Spec table) *A spec table $\mathcal{T} : \mathtt{SH} \rightharpoonup \mathcal{P}(\mathtt{SH})$ is a partial function from unlabelled symbolic heaps to sets of unlabelled symbolic heaps. A specification for a command given as a spec table $\mathcal{T}$ represents the set of Hoare triples for the command in which, for every $U \in dom(\mathcal{T})$, there is a triple with pre-condition U and post-condition $\bigvee_{U' \in \mathcal{T}(U)} U'$.*

Given these specifications from the shape analysis, our dependence analysis uses an intermediate language for commands in which every composite command (a procedure call, conditional or while loop) is represented by a *specified* command, $\mathtt{com}[\mathcal{T}]$, where $\mathcal{T}$ is the specification given for the composite command.

Definition 3.4 (Programming language) *The programming language is given by the following grammar:*

$$\begin{array}{rcll} b & ::= & E = E \mid E \neq E & \textit{boolean expressions} \\ c & ::= & x := E \mid x := E \to f \mid E_1 \to f := E_2 \mid \mathtt{new}(x) \mid \mathtt{dispose}(x) & \textit{atomic commands} \\ C & ::= & c \mid \mathtt{com}[\mathcal{T}] \mid C_1; C_2 & \textit{programs} \end{array}$$

Boolean expressions test equality or inequality between variable expressions (defined in figure 3.1). We have atomic commands c for manipulating the heap and program variables. The assignment command $x := E$ assigns to variable x the value of expression E. The heap lookup command $x := E \to f$ assigns to x the value of field f in the heap cell at address E. The heap mutation command $E_1 \to f := E_2$ sets the value of field f at heap cell E_1 to the value E_2. The allocation command $\mathtt{new}(x)$ allocates a new heap cell and sets x to its address, and $\mathtt{dispose}(x)$

deallocates the heap cell at address x. The formal semantics of these commands is given in section 3.6 when we demonstrate soundness. A program C is either an atomic command, a specified command or a sequential composition of commands. We also assume that we are given the sets $\mathtt{RV}(C)$ and $\mathtt{MV}(C)$ for the set of variables that are read and modified by command C respectively (these variables can be obtained from the syntax of commands).

The dependence analysis will be applied to a sequential block of commands of the form $l_1 : C_1; \ldots ; l_n : C_n$, where each component $l_i : C_i$ of the block is either an atomic command or a specified command, indexed by label $l_i \in \mathtt{Lab}$. The aim of the analysis is to detect the dependences between the component commands $C_1, \ldots, C_n$, so that any parallelization or optimization that respects these dependences produces the same results as the original program. Notice that although the bodies of composite commands such as procedures, conditionals and loops are abstracted away as they are replaced by specified commands, we can apply the analysis separately to different levels of nesting in the original program in order to optimize the bodies of composite commands.

For a sequential block C and a given pre-condition formula $P \in \mathcal{P}(\mathtt{LSH})$, the dependence analysis is defined by the $\mathsf{getDeps}(C, P)$ function, which returns a *dependence set* $D \in \mathcal{P}(\mathtt{Lab} \times \mathtt{Lab})$ that relates labels of commands in sequential block between which there is a dependence. There are three kinds of dependences that must be determined in order to safely optimize a program. These are heap-carried dependences, stack-carried dependences and dependences due to dynamic memory allocation. The $\mathsf{getDeps}$ function returns the union of all three kinds of dependences in the sequential block.

Definition 3.5 $\mathsf{getDeps}$ *function For a formula* $P \in \mathcal{P}(\mathtt{LSH})$ *and sequential block* C*, we have*

$$\mathsf{getDeps}(C, P) = \mathsf{getStackDeps}(C) \cup \mathsf{getHeapDeps}(C, P) \cup \mathsf{getAllocDeps}(C)$$

Stack dependences are easily determined from the syntax of programs by observing the variables accessed by commands, so that

$$\mathsf{getStackDeps}(C) = \{(l_i, l_j) \mid l_i : C_i \text{ and } l_j : C_j \text{ access common stack variables}\}$$

The main difficulty is in determining heap-carried dependences, which is the focus of this chapter. We first describe our method for determining heap-carried dependences, and will then discuss the

Algorithm 1 $\mathsf{Exec}(C, (P, D))$

```
1: if $C = empty$ then
2:     return $(P, D)$ ;
3: else if $C = l : c$ then
4:     return $\mathsf{ExecAtm}(l : c, (P, D))$ ;
5: else if $C = l : \mathtt{com}[\mathcal{T}]$ then
6:     return $\mathsf{ExecSpec}(l : \mathtt{com}[\mathcal{T}], (P, D))$ ;
7: else if $C = l : C_1; C'$ then
8:     return $\mathsf{Exec}(C', \mathsf{Exec}(l : C_1, (P, D)))$ ;
9: end if
```

allocation dependences.

The heap-dependence detection method is based on a symbolic execution of the program. Given a sequential block C, a pre-condition $P \in \mathcal{P}(\mathtt{LSH})$, and an initial dependence set $D \in \mathcal{P}(\mathtt{Lab} \times \mathtt{Lab})$, the symbolic execution is performed by the function $\mathsf{Exec}(C, (P, D))$, which detects heap dependences by tracking the labels of component commands through the execution, starting from the given pre-condition. If successful, the function returns a pair (P', D') where $P' \in \mathcal{P}(\mathtt{LSH})$ is the post-condition formula and $D' \in \mathcal{P}(\mathtt{Lab} \times \mathtt{Lab})$ contains any heap dependences between the commands in the sequential block. The Exec function is defined in algorithm 1. It uses the subroutines ExecAtm and ExecSpec in the case of atomic and specified commands respectively. We describe each of these cases next.

Definition 3.6 (getHeapDeps**)** *For a formula $P \in \mathcal{P}(\mathtt{LSH})$ and sequential block C, we define*

$$\mathsf{getHeapDeps}(C, P) = D$$

where D is the set of dependences obtained from $\mathsf{Exec}(C, (P, \emptyset)) = (P', D)$.

3.3.1 Executing atomic commands

The ExecAtm function symbolically executes atomic commands by transforming the symbolic states according to a set of inference rules. These rules mirror the imperative update of the concrete heap state during program execution. They are based on the original symbolic execution rules from [3] and are extended here for labelled symbolic heaps and dependence detection.

The inference rules for symbolic execution are divided into *command application* and *rearrangement* rules, which are displayed in figure 3.3. Read from top to bottom, each rule transforms a symbolic state in the premise to a symbolic state in the conclusion, where a symbolic state is of the form (H, D) with H a labelled symbolic heap and D a set of dependences collected so far. The rules can be understood by appeal to operational intuition. The command application rule for each atomic command describes the effect of the command on the labelled symbolic heap according to the operation of the command on concrete heaps, adds the label of the command to any heap region that is accessed, and records new dependence information to the dependence set. For instance, reading from top to bottom, the rule for the $l : \texttt{dispose}(x)$ command states that if initially the symbolic heap is $\Pi \wedge \Sigma * \langle E \mapsto [\rho]\rangle_L$ and the dependence set is D, then after executing the command the cell at E is disposed to give the heap $\Pi \wedge \Sigma$. The dependence set is updated to $D \cup \{(l, l') \mid l' \in L\}$ since there is a heap-carried dependence between this command and all the commands that previously accessed the heap cell at E, the labels of which were collected in the set L.

The rule for the $l : \texttt{new}(x)$ command creates a new heap cell in the symbolic heap, assigns its address to x, and uses the new existential variable x' to keep track of the old value of x. The new cell is given label set $\{l\}$ since this is the only command that has so far accessed the cell. This is recorded in the label set so that dependences between this command and commands that may subsequently access this cell can be determined. The dependence set is unchanged in this case since there are no commands that have previously accessed the new cell.

The rule for the assignment command $l : x := E$ updates the value of x and uses the new existential variable x' to keep track of the old value of x. The dependence set is again unchanged since the command does not make any heap access in this case. The rules for mutation ($l : E \to f := F$) and lookup ($l : x := E \to f$) use the following definitions:

$$\mathit{mutate}(\rho, f, F) = \begin{cases} f : F, \rho' & \text{if } \rho = f : E, \rho' \\ f : F, \rho & \text{if } f \notin \rho \end{cases} \qquad \mathit{lookup}(\rho, f) = \begin{cases} \rho, E & \text{if } \rho = f : E, \rho' \\ (\rho, f : x), x & \text{if } f \notin \rho \text{ and } x \text{ fresh} \end{cases}$$

The fresh variable returned in the lookup case corresponds to the idea that if a record expression does not give a value for a particular field then that value is unknown at that point in the analysis, and hence a fresh variable is introduced to denote the value. In both the mutate and lookup rules, the label of the command is placed in the label set of the accessed heap cell in the post-condition, and the dependence set is updated as in the case of the dispose command. The soundness of these symbolic rules is based on an over-approximation of the concrete execution semantics of commands, which we give in 3.6 when we formally demonstrate soundness.

The command application rules are not sufficient on their own. This is because, when commands access the heap, the pre-condition is expected to be in a certain form in which the heap cell that is to be accessed is explicitly present. For instance, if the symbolic heap in the premise is

$$x = z \wedge \langle y \mapsto [f : z] \rangle_{L_1} * \langle z \mapsto [f : x'] \rangle_{L_2}$$

then the rule for the mutation command $l : x \to f := y$ cannot be applied because the symbolic heap should explicitly have $x \mapsto [\rho]$ as a spatial conjunct, for some ρ. For this reason, symbolic execution has a separate *rearrangement* phase, which attempts to put the precondition in the proper form for a command application rule to fire. For instance, in the example just given we can observe that the pre-condition is equivalent to

$$x = z \wedge \langle y \mapsto [f : z] \rangle_{L_1} * \langle x \mapsto [f : x'] \rangle_{L_2}$$

which is in a form that allows the mutate rule to fire. The first rearrangement rule in figure 3.3 makes use of equalities to recognize that a dereferencing step is possible and makes appropriate variable substitutions. For instance, in the example above, the equality $x = z$ in the formula is used to change the conjunct $\langle z \mapsto [f : x'] \rangle_{L_2}$ to $\langle x \mapsto [f : x'] \rangle_{L_2}$. Notice that substituting different variables in the spatial conjunct does not change the label set of the conjunct, because the conjunct is still describing the same heap location as before.

The other two rearrangement rules are for list segments and trees, which expose $\mapsto$ facts by unrolling the inductive definitions when there is enough information to conclude that the list segment or tree is nonempty. A list segment is non-empty when the start and end points are different (side condition $F = F'$ in the rule) and a tree is non-empty when the root is not nil (side condition $F \neq \texttt{nil}$ in the rule). Besides unrolling the inductive definition, some matching is also included using the equality $E = F$ in the side-conditions. Each spatial conjunct in the unrolling of the date structure formula is given the label set of the original formula, since the described heap region remains the same. Notice that every rearrangement rule gives a valid entailment between the labelled symbolic heaps in the premise and conclusion (in accordance with definition 3.2), and that the dependence set remains unchanged in all the rearrangement rules since no new dependence information is obtained.

Definition 3.7 (ExecAtm**)** *For an atomic command $l : c$, a symbolic heap H, and dependence*

COMMAND APPLICATION RULES

$$\frac{(\Pi \wedge \Sigma * \langle E \mapsto [\rho]\rangle_L, D)}{(\Pi \wedge \Sigma, D \cup \{(l, l') \mid l' \in L\})}\ l : \mathtt{dispose}(x)$$

$$\frac{(\Pi \wedge \Sigma, D)}{((\Pi \wedge \Sigma)[x'/x] * \langle x \mapsto []\rangle_{\{l\}}, D)}\ l : \mathtt{new}(x), x'\ \mathit{fresh}$$

$$\frac{(\Pi \wedge \Sigma, D)}{(x = E[x'/x] \wedge (\Pi \wedge \Sigma)[x'/x], D)}\ l : x := E, x'\mathit{fresh}$$

$$\frac{(\Pi \wedge \Sigma * \langle E \mapsto [\rho]\rangle_L, D)}{(\Pi \wedge \Sigma * \langle E \mapsto [\rho']\rangle_{L \cup \{l\}}, D \cup \{(l, l') \mid l' \in L\})}\ l : E \to f := F, \mathit{mutate}(\rho, f, F) = \rho'$$

$$\frac{(\Pi \wedge \Sigma * \langle E \mapsto [\rho]\rangle_L, D)}{(x = F[x'/x] \wedge (\Pi \wedge \Sigma * \langle E \mapsto [\rho']\rangle_{L \cup \{l\}})[x'/x], D \cup \{(l, l') \mid l' \in L\})}\ l : x := E \to f, x'\ \mathit{fresh}, \mathit{lookup}(\rho, f) = (\rho', F)$$

REARRANGEMENT RULES

$$\frac{(\Pi \wedge \Sigma * \langle F \mapsto [\rho]\rangle_L, D)}{(\Pi \wedge \Sigma * \langle E \mapsto [\rho]\rangle_L, D)}\ \Pi \vdash E = F$$

$$\frac{(\Pi \wedge \Sigma * \langle \mathtt{ls}(F, F')\rangle_L, D)}{(\Pi \wedge \Sigma * \langle E \mapsto [n : x']\rangle_L * \langle \mathtt{ls}(x', F')\rangle_L, D)}\ \Pi \vdash F \neq F' \wedge E = F \text{ and } x' \text{ fresh}$$

$$\frac{(\Pi \wedge \Sigma * \langle \mathtt{tree}(F)\rangle_L, D)}{(\Pi \wedge \Sigma * \langle E \mapsto [l : x', r : y']\rangle_L * \langle \mathtt{tree}(x')\rangle_L * \langle \mathtt{tree}(y')\rangle_L, D)}\ \Pi \vdash F \neq \mathtt{nil} \wedge E = F \text{ and } x', y' \text{ fresh}$$

Figure 3.3: Label tracking symbolic execution rules

set D, the $\mathsf{ExecAtm}(l : c, (H, D))$ *function symbolically executes the command using the rules in figure 3.3. If c is a command accessing the heap at E (lookup, mutate or dispose), then the rearrangement rules are first applied to make E explicit before the command application rule is applied. The command application rule for $l : c$ is then applied with premise (H, D) and the function returns the conclusion of the rule (H', D'). The* $\mathsf{ExecAtm}$ *function is undefined on an input state (H, D) if the state is not a valid pre-condition for the symbolic execution rules.*

The $\mathsf{ExecAtm}$ *function is lifted to formulae as follows. For a formula $P \in \mathcal{P}(\texttt{LSH})$, we have* $\mathsf{ExecAtm}(l : c, (P, D))$ *is undefined if* $\mathsf{ExecAtm}(l : c, (H, D))$ *is undefined for any $H \in P$. Otherwise we have* $\mathsf{ExecAtm}(l : c, (P, D)) = (P_1, D_1)$ *where*

$$P_1 = \{H' \mid (H', D') = \mathsf{ExecAtm}(l : c, (H, D)), H \in P\}$$

$$D_1 = \bigcup \{D' \mid (H', D') = \mathsf{ExecAtm}(l : c, (H, D)), H \in P\}$$

3.3.2 Executing specified commands

The $\mathsf{ExecSpec}$ function defines the symbolic execution for specified commands. In the case of specified commands, the command's spec table determines the transformation of the symbolic heap in the execution. For example, assume we are given the command $l : \texttt{com}[\mathcal{T}]$, which is a procedure that traverses a tree at x and writes certain values at every tree node. In this case the spec table $\mathcal{T}$ has the single pre-condition with unlabelled symbolic heap $\texttt{tree}(x)$ and the post-condition $\mathcal{T}(\texttt{tree}(x)) = \{\texttt{tree}(x)\}$. Assume that this command is to be executed on the symbolic state (H, D), where the call-site assertion H is

$$z = w \ \wedge \ \langle z \mapsto [] \rangle_{L_1} * \langle \texttt{tree}(x) \rangle_{L_2} * \langle \texttt{ls}(y, \texttt{nil}) \rangle_{L_3}$$

Matching the pre-condition of the spec table, we know that the command only accesses the heap described by $\langle \texttt{tree}(x) \rangle_{L_2}$. Also, because this is the part that is accessed, we know that the command can only have heap-dependences with commands that have labels in L_2. To execute the command, we can use the spec table to replace the part of the heap that matches the pre-condition with the post-condition of the command, and the other parts of the formula are unaffected. The labels assigned to the post-condition formulae are all the labels in the pre-condition as well as the label of l of the command, since these are the only commands that may have accessed this heap region. Hence, after the execution the symbolic state we get is (H', D'), where

$$H' \equiv \quad z = w \ \wedge \ \langle z \mapsto []\rangle_{L_1} * \langle \mathtt{tree}(x)\rangle_{L_2 \cup \{l\}} * \langle \mathtt{ls}(y, \mathtt{nil})\rangle_{L_3}$$

$$D' \equiv \quad D \cup \{(l, l') \mid l' \in L_2\}$$

There are three things we need to know about how to use the specification to execute the command. Firstly, the assertion at the calling site of the command may be spatially larger than the pre-condition in the spec table, since the pre-condition only describes the part of the heap that is accessed by the command, and the calling site of the command may have other allocated regions which the command does not access. For this reason, we need to infer the *spatial frame* assertion, which is the part of the call-site heap that is not in the pre-condition of the command. In the above example the spatial frame is $\langle z \mapsto []\rangle_{L_1} * \langle \mathtt{ls}(y, \mathtt{nil})\rangle_{L_3}$. Secondly, we need to infer the *pure frame* which describes the variables that have not been modified by the command, which in the above example is the pure formula $z = w$. Lastly, we need to know how labels should be propagated in the execution of the command. Since the pre- and post-conditions in the spec table are only given in terms of *unlabelled* symbolic heaps, we need to infer the *accessed labels set*, which are the labels in the part of the call-site assertion that is accessed by the command. In the above example the accessed labels set is L_2.

Given this information, the symbolic state after execution is obtained by combining the frame assertions with the post-conditions given in the spec table. These post-condition formulae are given the accessed labels set plus the label l of the executing command, as in the example above. Dependences are determined between the executing command and all the commands in the accessed labels set.

Formally, the frame assertions and accessed labels set are generated by a function $\mathsf{Frm}(H, U, V)$, where H is a labelled symbolic heap (the call-site assertion), U is an unlabelled symbolic heap (command pre-condition from the spec table), and $V \subseteq \mathtt{Var}$ is the set of program variables that are modified by the command. In the following, we use the notation that, for any unlabelled spatial formula $\Psi = S_1 * \cdots * S_n$ and label set L, we shall write $\Psi[L]$ for the labelled formula $\langle S_1\rangle_L * \ldots \langle S_n\rangle_L$.

Let $U = \Pi \wedge \Psi$. If successful, the $\mathsf{Frm}(H, U, V)$ function returns a triple (Π_F, Σ_F, L_A) of a pure formula Π_F (the pure frame), a labelled spatial formula Σ_F (the spatial frame), and a label set L_A (the accessed labels set) such that

$$H \quad \vdash \quad \Pi_F \ \wedge \ \Pi \ \wedge \ \Sigma_F \ * \ \Psi[L_A]$$

where entailment is as defined in definition 3.2, and Π_F and Σ_F do not mention any variables in V. Given the call-site assertion in this form, we know that executing the command on this state will not change Π_F (since it does not mention any variables modified by the command) and Σ_F (since it is not part of the heap affected by the command). Hence, these frame formulae can be combined with the given post-condition $\mathcal{T}(U)$ of the command to get the following set of symbolic heaps after execution:

$$\{ \ \Pi_F \wedge \Pi' \wedge \Sigma_F * \Psi'[L_A \cup \{l\}] \quad | \quad \Pi' \wedge \Psi' \ \in \ \mathcal{T}(U) \ \}$$

The heap after execution satisfies the spatial frame and the post-condition of the command, and the variable stack satisfies the pure frame and the post-condition. The spatial formulae in the post-condition describe the part of the heap affected by the command so they are given the label set $L_A \cup \{l\}$ and the spatial frame preserves is labels from the call-site assertion because it is not accessed. The dependences $\{ \ (l, l') \quad | \quad l' \in L_A \ \}$ are added to the dependence set since the command l may depend on any of the commands in set L_A. The formal soundness argument for this symbolic transformation is given in section 3.6.

Before describing how the Frm function works, we first give the formal definition of the $\mathsf{ExecSpec}$ function. For a labelled symbolic heap H and dependence set D, the $\mathsf{ExecSpec}(l : \mathtt{com}[\mathcal{T}], (H, D))$ function, shown in algorithm 2, defines the execution of the specified command $\mathtt{com}[\mathcal{T}]$ on the state (H, D). The function first tests every U in the pre-condition $dom(\mathcal{T})$ of the specification to see if the call-site assertion H can be matched with it, by calling the $\mathsf{Frm}(H, U, \mathtt{MV}(l : \mathtt{com}[\mathcal{T}]))$ function. If this function is successful and returns a result (Π_F, Σ_F, L_A), then the set of symbolic heaps after the execution of the command are obtained by combining the frame formulae with every post-condition given in $\mathcal{T}(U)$ and the dependence set is updated to show that command l depends on all commands in set L_A. The $\mathsf{ExecSpec}$ function fails if none of the heaps in the pre-condition in the spec table can be matched with the call-site assertion.

Algorithm 2 $\mathsf{ExecSpec}(l : \mathtt{com}[\mathcal{T}], (H, D))$

```
1: for all $U \in dom(\mathcal{T})$ do
2:   if $\mathsf{Frm}(H, U, \mathtt{MV}(l : \mathtt{com}[\mathcal{T}])) = (\Pi_F, \Sigma_F, L_A)$ then
3:     $P \;:=\; \{\, \Pi_F \wedge \Pi \wedge \Sigma_F * \Psi[L_A \cup \{l\}] \;\mid\; \Pi \wedge \Psi \;\in\; \mathcal{T}(U) \,\}$ ;
4:     $D' \;:=\; D \cup \{\, (l, l') \;\mid\; l' \in L_A \,\}$ ;
5:     return $(P, D')$ ;
6:   end if
7: end for
8: return failure;
```

Definition 3.8 ($\mathsf{ExecSpec}(l : \mathtt{com}[\mathcal{T}], (P, D))$**)** *For a formula* $P \in \mathcal{P}(\mathtt{LSH})$*, we have* $\mathsf{ExecSpec}(l : \mathtt{com}[\mathcal{T}], (P, D))$ *fails if* $\mathsf{ExecSpec}(l : \mathtt{com}[\mathcal{T}], (H, D))$ *fails for any* $H \in P$*. Otherwise we have* $\mathsf{ExecSpec}(l : \mathtt{com}[\mathcal{T}], (P, D)) = (P_1, D_1)$ *where*

$$P_1 = \bigcup \{P' \mid (P', D') = \mathsf{ExecAtm}(l : c, (H, D)), H \in P\}$$

$$D_1 = \bigcup \{D' \mid (H', D') = \mathsf{ExecAtm}(l : c, (H, D)), H \in P\}$$

Inferring the frame assertion and accessed labels set We now describe how the Frm function infers the pure and spatial frames and the accessed labels set. As we discussed above, when given a labelled call-site symbolic heap H, an unlabelled pre-condition $U = \Pi \wedge \Psi$ and the set of variables V that are accessed by the command, the $\mathsf{Frm}(H, U, V)$ function computes a triple (Π_F, Σ_F, L_A) such that

$$H \quad \vdash \quad \Pi_F \;\wedge\; \Pi \;\wedge\; \Sigma_F \;*\; \Psi[L_A]$$

where Π_F and Σ_F do not contain any variables in V. The function is based on an adaptation of the frame inference method from [3], but extended for our purposes so that the frame assertion preserves its labels from the original call-site assertion and the accessed labels set is also inferred. To do this, we use a set of inference rules for entailments between labelled symbolic heaps, which are an extension of the rules for standard unlabelled symbolic heaps from [3]. The rules are shown in figure 3.4, which uses the following notation:

- the expression $\mathtt{op}(E)$ is an abbreviation for $E \mapsto [\rho]$, $\mathtt{ls}(E, F)$, or $\mathtt{tree}(E)$.
- the guard $\mathtt{G}(\mathtt{op}(E))$ asserts that the heap is non-empty, and is defined as

$$\mathtt{G}(E \mapsto [\rho]) \triangleq \mathtt{true} \qquad \mathtt{G}(\mathtt{ls}(E, F)) \triangleq E \neq F \qquad \mathtt{G}(\mathtt{tree}(E)) \triangleq E \neq \mathtt{nil}$$

NORMALISATION RULES

$$\frac{\Pi[E/E'] \wedge \Sigma[E/E'] \vdash \Pi'[E/E'] \wedge \Sigma'[E/E']}{\Pi \wedge E' = E \wedge \Sigma \vdash \Pi' \wedge \Sigma'} \qquad \frac{\Pi \wedge \Sigma \vdash \Pi' \wedge \Sigma'}{\Pi \wedge E = E \wedge \Sigma \vdash \Pi' \wedge \Sigma'}$$

$$\frac{\Pi \wedge \Sigma \vdash \Pi' \wedge \Sigma'}{\Pi \wedge \Sigma * \langle \mathtt{ls}(E,E)\rangle_L \vdash \Pi' \wedge \Sigma'} \qquad \frac{\Pi \wedge \Sigma \vdash \Pi' \wedge \Sigma'}{\Pi \wedge \Sigma * \langle \mathtt{tree}(\mathtt{nil})\rangle_L \vdash \Pi' \wedge \Sigma'}$$

$$\frac{\Pi \wedge \mathtt{G}(\mathtt{op}(E)) \wedge E \neq \mathtt{nil} \wedge \langle \mathtt{op}(E)\rangle_L * \Sigma \vdash \Pi' \wedge \Sigma'}{\Pi \wedge \mathtt{G}(\mathtt{op}(E)) \wedge \langle \mathtt{op}(E)\rangle_L * \Sigma \vdash \Pi' \wedge \Sigma'} \; E \neq \mathtt{nil} \notin \Pi \wedge \mathtt{G}(\mathtt{op}(E))$$

$$\frac{\Pi \wedge E_1 \neq E_2 \wedge \langle \mathtt{op}_1(E_1)\rangle_{L_1} * \langle \mathtt{op}_2(E_2)\rangle_{L_2} * \Sigma \vdash \Pi' \wedge \Sigma'}{\Pi \wedge \langle \mathtt{op}_1(E_1)\rangle_{L_1} * \langle \mathtt{op}_2(E_2)\rangle_{L_2} * \Sigma \vdash \Pi' \wedge \Sigma'} \; \begin{array}{l} \mathtt{G}(\mathtt{op}_1(E_1)), \mathtt{G}(\mathtt{op}_2(E_2)) \in \Pi \\ E_1 \neq E_2 \notin \Pi \end{array}$$

SUBTRACTION RULES

$$\frac{\Pi \wedge \Sigma \vdash \Pi' \wedge \Sigma'}{\Pi \wedge \Sigma \vdash \Pi' \wedge E = E \wedge \Sigma'} \quad \frac{\Pi \wedge \pi \wedge \Sigma \vdash \Pi' \wedge \Sigma'}{\Pi \wedge \pi \wedge \Sigma \vdash \Pi' \wedge \pi \wedge \Sigma'} \quad \frac{\Pi \wedge \Sigma \vdash \Pi' \wedge \Sigma'}{\Pi \wedge \Sigma \vdash \Pi' \wedge \langle \mathtt{ls}(E,E)\rangle_L * \Sigma'} \quad \frac{\Pi \wedge \Sigma \vdash \Pi' \wedge \Sigma'}{\Pi \wedge \Sigma \vdash \Pi' \wedge \langle \mathtt{tree}(E,\mathtt{nil})\rangle_L * \Sigma'}$$

$$\frac{\Pi \wedge \Sigma \vdash \Pi' \wedge \Sigma'}{\Pi \wedge \langle S\rangle_L * \Sigma \vdash \Pi' \wedge \langle S'\rangle_{L'} * \Sigma'} \; S \preccurlyeq S',\ L \subseteq L',\ \mathtt{Var}'(S) \cap \mathtt{Var}'(\Pi \wedge \Sigma) = \emptyset,\ \mathtt{Var}'(S') \cap \mathtt{Var}'(\Pi' \wedge \Sigma') = \emptyset$$

$$\frac{\Pi \wedge \langle E \mapsto [l:E_1, r:E_2, \rho]\rangle_{L_1} * \Sigma \vdash \Pi' \wedge \langle E \mapsto [l:E_1, r:E_2, \rho]\rangle_{L_2} * \langle \mathtt{tree}(E_1)\rangle_{L_2} * \langle \mathtt{tree}(E_2)\rangle_{L_2} * \Sigma'}{\Pi \wedge \langle E \mapsto [l:E_1, r:E_2, \rho]\rangle_{L_1} * \Sigma \vdash \Pi' \wedge \langle \mathtt{tree}(E)\rangle_{L_2} * \Sigma'} \; \langle E \mapsto [l:E_1, r:E_2, \rho]\rangle_{L_2} \notin \Sigma'$$

$$\frac{\Pi \wedge E_1 \neq E_3 \wedge \langle E_1 \mapsto [n:E_2, \rho]\rangle_{L_1} * \Sigma \vdash \Pi' \wedge \langle E_1 \mapsto [n:E_2, \rho]\rangle_{L_2} * \langle \mathtt{ls}(E_2, E_3)\rangle_{L_2} * \Sigma'}{\Pi \wedge E_1 \neq E_3 \wedge \langle E_1 \mapsto [n:E_2, \rho]\rangle_{L_1} * \Sigma \vdash \Pi' \wedge \langle \mathtt{ls}(E_1, E_3)\rangle_{L_2} * \Sigma'} \; \langle E_1 \mapsto [n:E_2, \rho]\rangle_{L_1} \notin \Sigma'$$

$$\frac{\Pi \wedge \langle \mathtt{ls}(E_1, E_2)\rangle_{L_1} * \Sigma \;\vdash\; \Pi' \wedge \langle \mathtt{ls}(E_1, E_2)\rangle_{L_2} * \langle \mathtt{ls}(E_2, \mathtt{nil})\rangle_{L_2} * \Sigma'}{\Pi \wedge \langle \mathtt{ls}(E_1, E_2)\rangle_{L_1} * \Sigma \;\vdash\; \Pi' \wedge \langle \mathtt{ls}(E_1, \mathtt{nil})\rangle_{L_2} * \Sigma'}$$

$$\frac{\Pi \wedge \mathtt{G}(\mathtt{op}(E_3)) \wedge \langle \mathtt{ls}(E_1, E_2)\rangle_{L_1} * \langle \mathtt{op}(E_3)\rangle_{L_2} * \Sigma \;\vdash\; \Pi' \wedge \langle \mathtt{ls}(E_1, E_2)\rangle_{L_3} * \langle \mathtt{ls}(E_2, E_3)\rangle_{L_3} * \Sigma'}{\Pi \wedge \mathtt{G}(\mathtt{op}(E_3)) \wedge \langle \mathtt{ls}(E_1, E_2)\rangle_{L_1} * \langle \mathtt{op}(E_3)\rangle_{L_2} * \Sigma \;\vdash\; \Pi' \wedge \langle \mathtt{ls}(E_1, E_3)\rangle_{L_3} * \Sigma'}$$

Figure 3.4: Rules for labelled entailment

- we write $S \preccurlyeq S'$ either $S = S'$ or for some E, ρ and ρ', we have $S = E \mapsto [\rho, \rho']$ and $S' = E \mapsto [\rho]$.

These rules are sound from top to bottom, in the sense that if the entailment in the premise holds according to definition 3.2 then so does the entailment in the conclusion. However, in practice the rules are applied upwards starting from a given entailment, until we arrive at the axiom $\Pi \wedge \mathtt{emp} \vdash \mathtt{true} \wedge \mathtt{emp}$, at which point we have constructed a valid proof of the given entailment.

The rules are categorized as either normalization or subtraction rules. The normalization rules simplify the left hand side of entailments and make information explicit for the subtraction rules to be applied. The first two normalization rules get rid of equalities as soon as possible so that the forthcoming rules can be formulated using simple pattern matching (i.e., we can use $\langle E \mapsto [\rho]\rangle_L$ directly rather than $\langle F \mapsto [\rho]\rangle_L$ and $E = F$ derivable). The first rule performs variable substitution after removing the equality and the second removes redundant equalities. The next two rules remove inconsistent data structure formulae from the left hand side of the entailment. The last two normalization rules make derivable inequalities explicit on the left hand side, based on the properties of the spatial points-to formula and the data structure formulae.

The second group of rules, the subtraction rules, work by simplifying and explicating information on the right hand side of entailments in order to eventually reduce to the axiom $\Pi \wedge \texttt{emp} \vdash \texttt{true} \wedge \texttt{emp}$. The first two rules eliminate redundant equalities from the right hand side, and the next two rules eliminate inconsistent data structure formulae from the right hand side. The fifth rule, which we call *-introduction, is the main subtraction rule which helps reach the desired axiom. It matches and eliminates simple spatial formulae on both sides of the entailment when applied upwards. Since $S \preccurlyeq S'$ and $L \subseteq L'$, we have $\langle S \rangle_L \vdash \langle S' \rangle_{L'}$. However, we need to be careful with the existential interpretation of primed variables, which was not the case in [3]. The addditional side condition that the new spatial conjuncts do not share primed variables with the existing formulae ensures the soundness of the rule from top to bottom.

The last four subtraction rules are based on properties of the inductive tree and list segment predicates. The first two rules unfold the predicates on the right hand side according to their inductive definition. The last two rules are specific for the list segment predicate, since it is possible to unfold a list segment from the middle to obtain two list segments. We do not have similar rules for the tree predicate since unfolding a tree in the middle does not give whole trees. In the last subtraction rule, the $\texttt{G}(\texttt{op}(E_3)) \wedge \texttt{op}(E_3)$ part of the left-hand side ensures that E_3 does not occur within the segments from E_1 to E_2, which is necessary for appending list segments, since they are required to be acyclic.

As an example of the use of these rules, if we want to check the validity of the entailment

$$z = y \wedge \langle x \mapsto [] \rangle_{L_1} * \langle \texttt{tree}(y) \rangle_{L_2} \vdash \langle x \mapsto [] \rangle_{L_1} * \langle \texttt{tree}(z) \rangle_{L_2 \cup L_3}$$

then we can apply the rules upwards to construct the following derivation:

$$\cfrac{\cfrac{\cfrac{\texttt{emp} \vdash \texttt{emp}}{\langle \texttt{tree}(z) \rangle_{L_2} \vdash \langle \texttt{tree}(z) \rangle_{L_2 \cup L_3}}}{z = y \wedge \langle \texttt{tree}(y) \rangle_{L_2} \vdash \langle \texttt{tree}(z) \rangle_{L_2 \cup L_3}}}{z = y \wedge \langle x \mapsto [] \rangle_{L_1} * \langle \texttt{tree}(y) \rangle_{L_2} \vdash \langle x \mapsto [] \rangle_{L_1} * \langle \texttt{tree}(z) \rangle_{L_2 \cup L_3}}$$

where we have applied the *-introduction rule (fifth subtraction rule), the first normalization rule and the *-introduction rule again to reach the axiom at the top.

This is fine if we only need to check the validity of entailments, but in our case we will use these rules to infer missing information about the frame assertion and the accessed labels set. We first state the following definition and results, which we shall use to define the Frm function.

Definition 3.9 *For a labelled symbolic heap $H = \Pi \wedge \langle S_1 \rangle_{L_1} * \cdots * \langle S_n \rangle_{L_n}$, we write $labsets(H)$ for the* multiset $\{L_1, \ldots, L_n\}$ *of the label sets of all spatial conjuncts in H.*

Lemma 3.1 *Suppose we apply the inference rules upwards from an entailment $H \vdash H'$ and get the following derivation*

$$\begin{array}{c} H_1 \vdash H_1' \\ \vdots \\ H \vdash H' \end{array}$$

Then $labsets(H_1) \subseteq labsets(H)$.

Proof: *The proof is by induction on the size of the derivation. For the base case where the premise and conclusion are identical and there are no rule applications, the result follows trivially. For the inductive case we assume that the result holds for all derivations of size $n-1$ and that the given derivation is of size n and of the form*

$$\begin{array}{c} H_1 \vdash H_1' \\ \vdots \\ H_2 \vdash H_2' \\ \hline H \vdash H' \end{array}$$

where the number of steps from $H_2 \vdash H_2'$ to the premise is $n-1$. Each of the rules in figure 3.4 guarantees that $labsets(H_2) \subseteq labsets(H)$, and we have by the induction hypothesis that $labsets(H_1) \subseteq labsets(H_2)$. Hence $labsets(H_1) \subseteq labsets(H)$. ■

The following theorem describes how, when given a labelled symbolic heap (the call-site assertion) and an unlabelled symbolic heap (the command pre-condition), we can use the proof rules to infer the missing information about the frame assertion and accessed labels set.

Theorem 3.2 *Assume we are given a labelled symbolic heap H and an unlabelled symbolic heap $\Pi \wedge \Psi$ such that H and $\Pi \wedge \Psi$ do not have any common primed variables. Suppose that we apply* *the inference rules upwards from the entailment $H \vdash \Pi \wedge \Psi[\mathtt{Lab}]$ to get the derivation:*

$$\begin{array}{c} \Pi' \wedge \Sigma \vdash \mathtt{true} \wedge \mathtt{emp} \\ \vdots \\ H \vdash \Pi \wedge \Psi[\mathtt{Lab}] \end{array}$$

Let L be any superset of $\bigcup(labsets(H) - labsets(\Sigma))$. Then the entailment $H \vdash \Pi \wedge \Psi[L] * \Sigma$ is valid.

Proof: We show that the given derivation implies the existence of a valid proof of the entailment $H \vdash \Pi \wedge \Psi[L] * \Sigma$. We first prove the following claim:

claim 1. In the derivation given in the assumption of the theorem, if we replace all the label sets $\mathtt{Lab}$ on the right hand side of entailments with L to get:

$$\begin{array}{c} \Pi' \wedge \Sigma \vdash \mathtt{true} \wedge \mathtt{emp} \\ \vdots \\ H \vdash \Pi \wedge \Psi[L] \end{array}$$

then this is a derivation that can be made using the proof rules in figure 3.4. We prove this claim by induction on the size of the derivation. For the base case when the premise and conclusion are identical and there are no rule applications, the result follows trivially since there are no spatial conjuncts in Ψ. For the inductive case we assume that the result holds for all derivations of size $n-1$ and that the given derivation is of size n and of the form

$$\begin{array}{c} \Pi' \wedge \Sigma \vdash \mathtt{true} \wedge \mathtt{emp} \\ \vdots \\ \dfrac{H_1 \vdash H_2}{H \vdash \Pi \wedge \Psi[\mathtt{Lab}]} \end{array}$$

where the number of steps from $H_1 \vdash H_2$ to the premise is $n-1$. The step from the conclusion $H \vdash \Pi \wedge \Psi[\mathtt{Lab}]$ to $H_1 \vdash H_2$ is an application of one of the inference rules from figure 3.4. For each of the inference rules it can be checked that H_2 is of the form $\Pi_2 \wedge \Psi_2[\mathtt{Lab}]$ for some Π_2 and Ψ_2. We have assumed L is a superset of $\bigcup(labsets(H) - labsets(\Sigma))$, so it is also a superset of $\bigcup(labsets(H_1) - labsets(\Sigma))$ since $labsets(H_1) \subseteq labsets(H)$ by lemma 3.1. Therefore, by the induction hypothesis, there exists the valid derivation

$$\begin{array}{c} \Pi' \wedge \Sigma \vdash \mathtt{true} \wedge \mathtt{emp} \\ \vdots \\ H_1 \vdash \Pi_2 \wedge \Psi_2[L] \end{array}$$

Now to complete the proof of claim 1 we have to show that we can make the inference step

$$\frac{H_1 \vdash \Pi_2 \wedge \Psi_2[L]}{H \vdash \Pi \wedge \Psi[L]} \qquad (\dagger)$$

We already know from the assumption of the inductive case that some rule in figure 3.4 can be applied to make the inference

$$\frac{H_1 \vdash \Pi_2 \wedge \Psi_2[\mathtt{Lab}]}{H \vdash \Pi \wedge \Psi[\mathtt{Lab}]} \qquad (\ddagger)$$

In every rule in figure 3.4 except the *-introduction rule (which eliminates spatial conjuncts upwards), there are no constraints imposed on the value of the label sets on the right hand side of the entailment, so we can replace every label set Lab with L on the right hand side of entailments in (‡) to achieve the required inference (†). In the case of the *-introduction rule, assume that in (‡) we have $H = \Pi_H \wedge \langle S \rangle_{L'} * \Sigma_H$ and $H_1 = \Pi_H \wedge \Sigma_H$. To achieve the inference (†), we need to check that it is possible to replace all occurrences of Lab with L on the right hand side of entailments in (‡). For this it is sufficient to check that $L' \subseteq L$, in accordance with the side condition of the *-introduction rule. We know that $\mathit{labsets}(H) = \{L'\} \uplus \mathit{labsets}(H_1)$ and that $\mathit{labsets}(\Sigma) \subseteq \mathit{labsets}(H_1)$ by lemma 3.1. Hence $L' \in (\mathit{labsets}(H) - \mathit{labsets}(\Sigma))$, and thus $L' \subseteq L$ by definition of L. This completes the proof of claim 1.

The derivation shown to exist by claim 1 can be transformed into a complete proof of the entailment $H \vdash \Pi \wedge \Psi[L] * \Sigma$ as follows. We first add Σ to the heap on the right hand side of the entailment at every proof step so that the proof becomes:

$$\begin{array}{c} \Pi' \wedge \Sigma \vdash \mathtt{true} \wedge \Sigma \\ \vdots \\ H \vdash \Pi \wedge \Psi[L] * \Sigma \end{array}$$

One thing to check is that all the *-introduction steps can still be applied after appending Σ to the right hand side (because of the side condition about disjointness of primed variables in this rule). All of these steps are possible because the primed variables of Σ are disjoint from the primed variables of $\Pi \wedge \Psi[L]$. This is because Σ only contains primed variables from H, and $\mathtt{Var}'(H)$ is disjoint from $\mathtt{Var}'(\Pi \wedge \Psi[L])$ by assumption of the theorem.

The premise of this proof is the entailment $\Pi' \wedge \Sigma \vdash \mathtt{true} \wedge \Sigma$, which is a valid entailment, and therefore the conclusion is also valid by soundness of the rules. ■

Using theorem 3.2, we infer the accessed labels set, the spatial frame and the pure frame as follows. For a call-site assertion H and unlabelled pre-condition $U = \Pi \wedge \Psi$, we first rename primed variables in H to ensure that they are disjoint from U. Then the theorem tells us that if we apply the inference rules upwards from the entailment $H \vdash \Pi \wedge \Psi[\mathsf{Lab}]$ and get $\Pi' \wedge \Sigma \vdash \mathtt{true} \wedge \mathtt{emp}$, then $H \vdash \Pi \wedge \Psi[L] * \Sigma$ holds, where $L = \bigcup(\mathit{labsets}(H) - \mathit{labsets}(\Sigma))$. The set L includes all the labels from the part of the call-site heap that is in the pre-condition of the command, so this gives us the accessed labels set. The formula Σ describes the part of the call-site heap that is not in the command pre-condition. But before we can use it as a frame assertion, we need to ensure that it does not mention any variables that are modified by the command (so that it can be safely combined with the post-condition of the command). Hence we existentially quantify any variables in Σ that are modified by the command. Similarly, the pure frame is taken as the pure part of H in which variables that are modified by the command are replaced by the existential ones used for the spatial frame.

Definition 3.10 ($\mathsf{Frm}(H, U, V)$) *Given a labelled call-site symbolic heap $H = \Pi_H \wedge \Sigma_H$, an unlabelled pre-condition $U = \Pi \wedge \Psi$ and the set of variables $V = \{x_1, \ldots, x_n\}$, the $\mathsf{Frm}(H, U, V)$ function first renames primed variables in H to make them disjoint from U. It then applies the proof rules from figure 3.4 to search for a derivation of the form:*

$$\begin{array}{c} \Pi' \wedge \Sigma \vdash \mathtt{true} \wedge \mathtt{emp} \\ \vdots \\ H \vdash \Pi \wedge \Psi[\mathsf{Lab}] \end{array}$$

If it is able to find such a derivation then it returns the triple (Π_F, Σ_F, L_A), where, for fresh primed variables $x'_1, \ldots, x'_n$,

$$\Pi_F = \Pi_H[x'_1/x_1, \ldots x'_n/x_n]$$

$$\Sigma_F = \Sigma[x'_1/x_1, \ldots x'_n/x_n]$$

$$L_A = \bigcup(\mathit{labsets}(H) - \mathit{labsets}(\Sigma))$$

Corollary 3.3 (Soundness of frame inference) *If we have $\mathsf{Frm}(H, U, V) = (\Pi_F, \Sigma_F, L_A)$ then*

$$H \quad \vdash \quad \Pi_F \ \wedge \ \Pi \ \wedge \ \Sigma_F \ * \ \Psi[L_A]$$

where Π_F and Σ_F do not mention any variables in V.

Proof: *Assume that $H = \Pi_H \wedge \Sigma_H$, $U = \Pi \wedge \Psi$ and that applying the proof rules upwards from $H \vdash \Pi \wedge \Psi[\mathtt{Lab}]$ gives $\Pi' \wedge \Sigma \vdash \mathtt{true} \wedge \mathtt{emp}$. Then, by theorem 3.2 and definition 3.10, we have*

$$H \;\vdash\; \Pi \wedge \Psi[L_A] * \Sigma$$

Since $H \;\vdash\; \Pi_H$, we have $H \;\vdash\; \Pi_H \wedge \Pi \wedge \Psi[L_A] * \Sigma$. Now, since the function existentially quantifies variables in Π_H and Σ to get Π_F and Σ_F, we have

$$\Pi_H \wedge \Pi \wedge \Psi[L_A] * \Sigma \;\vdash\; \Pi_F \wedge \Pi \wedge \Psi[L_A] * \Sigma_F$$

and therefore $H \;\vdash\; \Pi_H \wedge \Pi_F \wedge \Psi[L_A] * \Sigma_F$. The frames Π_F and Σ_F do not mention variables in V since they become existentially quantified. ■

As an example, assume the call-site assertion is

$$x_1 = y \wedge x_2 = z \wedge \langle x \mapsto []\rangle_{L_1} * \langle \mathtt{tree}(y)\rangle_{L_2} * \langle \mathtt{ls}(z, \mathtt{nil})\rangle_{L_3} * \langle \mathtt{ls}(w, \mathtt{nil})\rangle_{L_1}$$

the command pre-condition is $\mathtt{tree}(x_1) * \mathtt{ls}(w, \mathtt{nil})$ and the set of variables accessed by the command is $\{x_1, w, z\}$. The Frm function first makes the following derivation:

$$\cfrac{\cfrac{\cfrac{x_2 = z \wedge \langle x \mapsto []\rangle_{L_1} * \langle \mathtt{ls}(z, \mathtt{nil})\rangle_{L_3} \vdash \mathtt{emp}}{x_2 = z \wedge \langle x \mapsto []\rangle_{L_1} * \langle \mathtt{tree}(x_1)\rangle_{L_2} * \langle \mathtt{ls}(z, \mathtt{nil})\rangle_{L_3} \vdash \langle \mathtt{tree}(x_1)\rangle_{\mathtt{Lab}}}}{x_1 = y \wedge x_2 = z \wedge \langle x \mapsto []\rangle_{L_1} * \langle \mathtt{tree}(y)\rangle_{L_2} * \langle \mathtt{ls}(z, \mathtt{nil})\rangle_{L_3} \vdash \langle \mathtt{tree}(x_1)\rangle_{\mathtt{Lab}}}}{x_1 = y \wedge x_2 = z \wedge \langle x \mapsto []\rangle_{L_1} * \langle \mathtt{tree}(y)\rangle_{L_2} * \langle \mathtt{ls}(z, \mathtt{nil})\rangle_{L_3} * \langle w \mapsto []\rangle_{L_1} \vdash \langle \mathtt{tree}(x_1)\rangle_{\mathtt{Lab}} * \langle w \mapsto []\rangle_{\mathtt{Lab}}}$$

The function computes the accessed labels set $L_A = \bigcup(\{L_1, L_2, L_3, L_1\} - \{L_1, L_3\}) = L_2 \cup L_1$. Notice how it is important to consider the multiset of label sets in the call-site formula, since the label set L_1 occurs both in the command pre-condition and outside it. The Frm function chooses fresh variables x_1', x_2', x_3' for x_1, w, z, and computes the pure frame as $\Pi_F \;\equiv\; x_1' = y \wedge x_2 = x_3'$ and the spatial frame as $\Sigma_F \;\equiv\; \langle x \mapsto []\rangle_{L_1} * \langle \mathtt{ls}(x_3', \mathtt{nil})\rangle_{L_3}$. Notice how the spatial frame now has a list at the existential variable x_3' because the frame cannot mention the variable z, as the command may modify this variable. However, the list is still accessible for future commands because the pure frame retains the equality $x_2 = x_3'$, so the list can still be accessed through the program variable x_2. This demonstrates the importance of inferring both a pure frame and a spatial frame.

3.4 Allocation dependences

In this section we describe the third kind of dependence that we need to take into account, which is due to dynamic memory allocation. Program optimizations have often been proposed based on the *independence assumption* that if two commands access separate heap and variables in all possible executions, then they can be parallelized or reordered to give an equivalent program [21, 31]. We describe here how the independence assumption actually does not hold in the presence of dynamic memory allocation, and optimizations based on this assumption can produce results significantly different from the original program. A simple example is the following program:

$$l_1 : \texttt{new}(x);$$

$$l_2 : \texttt{new}(y);$$

$$l_3 : \texttt{dispose}(x);$$

$$l_4 : \texttt{if}(x = y)\texttt{then}\{z := 0\}\texttt{else}\{z := 1\};$$

At l_4, we have the $x \neq y$ because x and y cannot be allocated the same address, so the original program never sets $z := 0$. Now statements l_2 and l_3 are independent in that they access separate heap cells and variables, but reordering them may allow x and y to be equal if the address allocated in l_1 is re-used for l_2. So the optimized program will possibly set $z := 0$, and thus new behaviour can result. One may note that it is a widely accepted standard that programs should not read or test the values of dangling pointers [34], and it may hence be argued that the above program is somehow not a ‘proper’ program in the first place. But the problem persists even if we disallow such programs, as in the following example:

l_1 : $\texttt{new}(x)$;

l_2 : $\texttt{new}(y)$;

l_3 : $f := x$;

l_4 : $\texttt{if}(f = y)\texttt{then}\{z := 0\}\texttt{else}\{z := 1\}$;

l_5 : $\texttt{dispose}(x)$;

l_6 : $\texttt{dispose}(y)$;

This program does not perform any reading or testing of dangling pointers. However, it is possible to optimize such that the statement sequence l_1, l_3, l_5 is executed even before y is allocated in l_2! Thus again, y may get the same address as x, and so the true branch in l_4 can fire in the optimization but not in the original program. Such programs are perfectly reasonable as it is often the case that one checks that certain expected conditions are satisfied (such as whether two variables point to different allocated objects at a certain point in the program), and performs error handling otherwise. However, the optimized program introduces the possible scenario where the equality in l_4 is satisfied because of the allocator's reuse of the same address for the two intentionally different objects.

To guard against such incorrect optimizations, we introduce certain dependences between commands due to allocation. For a sequential block $l_1 : C_1; \ldots ; l_n : C_n$, there is an allocation dependence between any two commands $l_i : C_i$ and $l_j : C_j$ if one of them performs allocation and the other performs deallocation. The getAllocDeps function returns the set of all allocation dependences in the sequential block. Note that we have not required a dependence when both commands perform allocation or when both perform deallocation. The dependences we have required are sufficient to guarantee safe optimizations, as we shall show in the soundness proof.

3.5 Examples and Experiments

In this section we discuss examples to illustrate the dependence detection algorithm, and present experimental results on performance improvements. We begin with an example of a list segment traversal program $\texttt{listInit}(x, y)$, which traverses a linked list segment from x to y, setting the fields f_1 and f_2 in every cell to $\texttt{nil}$. The specification is given by spec table $\mathcal{T}$ with $dom(\mathcal{T}) =$

```
         if($(x \neq y)$){
           $(\{x{\neq}y \wedge \langle \texttt{ls}(x,y)\rangle_{\emptyset}\}, \emptyset)$
           $(\{x{\neq}y \wedge \langle x \mapsto [n : x']\rangle_{\emptyset} * \langle \texttt{ls}(x',y)\rangle_{\emptyset}\}, \emptyset)$
$l_1$ :    $x_1 := x \rightarrow n;$
           $(\{x_1{=}x' \wedge x{\neq}y \wedge \langle x \mapsto [n : x']\rangle_{\{l_1\}} * \langle \texttt{ls}(x',y)\rangle_{\emptyset}\}, \emptyset)$
$l_2$ :    $x \rightarrow f_1 := \texttt{nil};$
           $(\{x_1{=}x' \wedge x{\neq}y \wedge \langle x \mapsto [n : x', f_1 : \texttt{nil}]\rangle_{\{l_1,l_2\}} * \langle \texttt{ls}(x',y)\rangle_{\emptyset}\}, \{(l_2,l_1)\})$
$l_3$ :    $x \rightarrow f_2 := \texttt{nil};$
           $(\{x_1{=}x' \wedge x{\neq}y \wedge \langle x \mapsto [n : x', f_1 : \texttt{nil}, f_2 : \texttt{nil}]\rangle_{\{l_1,l_2,l_3\}} * \langle \texttt{ls}(x',y)\rangle_{\emptyset}\}, \{(l_2,l_1),(l_3,l_2),(l_3,l_1)\})$
$l_4$ :    listInit($x_1, y$);
           $(\{x_1{=}x' \wedge x{\neq}y \wedge \langle x \mapsto [n : x', f_1 : \texttt{nil}, f_2 : \texttt{nil}]\rangle_{\{l_1,l_2,l_3\}} * \langle \texttt{ls}(x_1,y)\rangle_{\{l_4\}}\}, \{(l_2,l_1),(l_3,l_2),(l_3,l_1)\})$
         }
```

Figure 3.5: Dependence detection for $\texttt{listInit}(x, y)$

$\{\texttt{ls}(x, y)\}$ and $\mathcal{T}(\texttt{ls}(x, y)) = \{\texttt{ls}(x, y)\}$, which gives both the pre- and post-condition of the procedure as a list segment from x to y. The procedure body and dependence analysis is shown in figure 3.5. The ExecAtm performs the symbolic execution for the atomic command l_1, first unfolding the pre-condition using the rearrangement rule for list segments from figure 3.3. It then executes the next two atomic commands l_2 and l_3. At this point the ExecSpec function is called, which finds the spatial frame $\langle x \mapsto [n : x', f_1 : \texttt{nil}, f_2 : \texttt{nil}]\rangle_{\{l_1,l_2,l_3\}}$, the pure frame $x_1{=}x' \wedge x \neq y$ and the accessed labels set $\emptyset$, since no command has yet accessed the list $\texttt{ls}(x', y)$, which is the pre-condition of the recursive call. This pre-condition is replaced by the post-condition from the spec table, and the label of the command is added to it. The dependence set obtained at the end does not contain any dependences between the recursive call l_4 and the two statements l_3 and l_2, and hence the recursive call can be executed in parallel with these statements. This program is representative of the general pattern in list processing programs in which some 'work' is done at every node of a list and then the program is recursively called on the rest of a list. If the work at each node does not depend on previous nodes then it may be done in parallel with the rest of the list, as our algorithm has detected in this case.

Previous *reachability* based approaches such as [26] would be unable to detect this opportunity for optimization. Such approaches depend on reachability properties of data structures to detect dependences, e.g., statements referring to the left and right subtrees of a tree can be determined to be independent since no heap location is reachable from both of them. In the case of the list segment traversal example, a reachability analysis will be unable to detect the independence found by our algorithm because the list segment may in fact be part of a larger cyclic data structure.

```
      if $(x = \texttt{nil})$ $\{y := \texttt{nil};\}$else$\{$
        $(\{x{\neq}\texttt{nil} \wedge \langle \texttt{ls}(x,\texttt{nil})\rangle_{\emptyset}\}, \emptyset)$
$l_1$ :   split$(x, x_1, x_2)$;
        $(\{x{\neq}\texttt{nil} \wedge \langle \texttt{ls}(x_1,\texttt{nil})\rangle_{\{l_1\}} * \langle \texttt{ls}(x_2,\texttt{nil})\rangle_{\{l_1\}}\}, \emptyset)$
$l_2$ :   mergesort$(x_1, y_1)$;
        $(\{x{\neq}\texttt{nil} \wedge \langle \texttt{ls}(y_1,\texttt{nil})\rangle_{\{l_1,l_2\}} * \langle \texttt{ls}(x_2,\texttt{nil})\rangle_{\{l_1\}}\}, \{(l_2,l_1)\})$
$l_3$ :   mergesort$(x_2, y_2)$;
        $(\{x{\neq}\texttt{nil} \wedge \langle \texttt{ls}(y_1,\texttt{nil})\rangle_{\{l_1,l_2\}} * \langle \texttt{ls}(y_2,\texttt{nil})\rangle_{\{l_1,l_3\}}\}, \{(l_2,l_1),(l_3,l_1)\})$
$l_4$ :   merge$(y, y_1, y_2)$;
        $(\{x{\neq}\texttt{nil} \wedge \langle \texttt{ls}(y,\texttt{nil})\rangle_{\{l_1,l_2,l_3,l_4\}}\}, \{(l_2,l_1),(l_3,l_1),(l_4,l_3),(l_4,l_2),(l_4,l_1)\})$
      }
```

Figure 3.6: Dependence detection for $\texttt{mergesort}(x, y)$

In contrast, our approach is based on detecting the cells that are actually accessed rather than those that are reachable by a statement. We also make a comparison with the proof-rewriting method of [31], where there is difficulty in comparing statements that are not consecutive in a sequential composition, such as determining that statement l_4 is independent of l_2. Not being able to compare non-consecutive statements significantly limits potential optimizations, especially if there are numerous statements processing a node in the list, all of which could potentially be executed in parallel with the recursive call. The dependence detection in figure 3.5 shows how, in our case, the label-tracking mechanism provides a simple and natural method for comparing distant statements.

Next, we give an example of a divide-and-conquer style algorithm. Figure 3.6 shows the analysis for the standard mergesort procedure for linked lists, which breaks the list into two lists, recursively sorts each of them, and then merges the two. The $\texttt{mergesort}(x, y)$ procedure takes a list at x and a returns a sorted version of the list at y. Its specification is given with pre-condition $\{\texttt{ls}(x, \texttt{nil})\}$ and post-condition $\{\texttt{ls}(y, \texttt{nil})\}$. The $\texttt{split}(x, x_1, x_2)$ procedure has pre-condition $\{\texttt{ls}(x, \texttt{nil})\}$ and post-condition $\{\texttt{ls}(x_1, \texttt{nil}) * \texttt{ls}(x_2, \texttt{nil})\}$ and $\texttt{merge}(y, y_1, y_2)$ has pre-condition $\{\texttt{ls}(y_1, \texttt{nil}) * \texttt{ls}(y_2, \texttt{nil})\}$ and post-condition $\{\texttt{ls}(y, \texttt{nil})\}$. Note that these specifications do not describe the sorting performed by the procedures, but only the shapes of the lists in the pre- and post-conditions. This shape information is all that is required for our method to analyse the heap accesses made in the program. The analysis shown in figure 3.6 determines that the second recursive call l_3 does not depend on l_2, which allows us to convert the sequential version of mergesort into the parallel version.

Finally, we give an example with trees, which is the tree rotation program $\texttt{rotateTree}(x)$ based

```
      if$(x \neq \texttt{nil})\{$
        $(\{x \neq \texttt{nil} \wedge \langle \texttt{tree}(x)\rangle_\emptyset\}, \emptyset)$
        $(\{x \neq \texttt{nil} \wedge \langle x \mapsto [l{:}x', r{:}y']\rangle_\emptyset * \langle \texttt{tree}(x')\rangle_\emptyset * \langle \texttt{tree}(y')\rangle_\emptyset\}, \emptyset)$
$l_1$ :   $x_1 := x \to l;$
        $(\{x_1 = x' \wedge x \neq \texttt{nil} \wedge \langle x \mapsto [l{:}x', r{:}y']\rangle_{\{l_1\}} * \langle \texttt{tree}(x')\rangle_\emptyset * \langle \texttt{tree}(y')\rangle_\emptyset\}, \emptyset)$
$l_2$ :   $x_2 := x \to r;$
        $(\{x_2 = y' \wedge x_1 = x' \wedge x \neq \texttt{nil} \wedge \langle x \mapsto [l{:}x', r{:}y']\rangle_{\{l_1,l_2\}} * \langle \texttt{tree}(x')\rangle_\emptyset * \langle \texttt{tree}(y')\rangle_\emptyset\}, D_1)$
$l_3$ :   $x \to l := x_2;$
        $(\{x_2 = y' \wedge x_1 = x' \wedge x \neq \texttt{nil} \wedge \langle x \mapsto [l{:}x_2, r{:}y']\rangle_{\{l_1,l_2,l_3\}} * \langle \texttt{tree}(x')\rangle_\emptyset * \langle \texttt{tree}(y')\rangle_\emptyset\}, D_2)$
$l_4$ :   $x \to r := x_1;$
        $(\{x_2 = y' \wedge x_1 = x' \wedge x \neq \texttt{nil} \wedge \langle x \mapsto [l{:}x_2, r{:}x_1]\rangle_{\{l_1,l_2,l_3,l_4\}} * \langle \texttt{tree}(x')\rangle_\emptyset * \langle \texttt{tree}(y')\rangle_\emptyset\}, D_3)$
$l_5$ :   rotateTree$(x_1);$
        $(\{x_2 = y' \wedge x_1 = x' \wedge x \neq \texttt{nil} \wedge \langle x \mapsto [l{:}x_2, r{:}x_1]\rangle_{\{l_1,l_2,l_3,l_4\}} * \langle \texttt{tree}(x_1)\rangle_{\{l_5\}} * \langle \texttt{tree}(y')\rangle_\emptyset\}, D_3)$
$l_6$ :   rotateTree$(x_2);$
        $(\{x_2 = y' \wedge x_1 = x' \wedge x \neq \texttt{nil} \wedge \langle x \mapsto [l{:}x_2, r{:}x_1]\rangle_{\{l_1,l_2,l_3,l_4\}} * \langle \texttt{tree}(x_1)\rangle_{\{l_5\}} * \langle \texttt{tree}(x_2)\rangle_{\{l_6\}}\}, D_3)$
      $\}$
```

Figure 3.7: Dependence detection for `rotateTree`(x)

on the main example discussed by Hendren *et al* in [26]. The procedure takes a tree at x and rotates it by recursively swapping its left and right subtrees. The spec table gives the pre-condition $\{\texttt{tree}(x)\}$ and post-condition $\{\texttt{tree}(x)\}$. The procedure body and dependence detection is shown in figure 3.7, where we have the dependence sets $D_1 = \{(l_2, l_1)\}$, $D_2 = \{(l_3, l_2), (l_3, l_1)\} \cup D_1$ and $D_3 = \{(l_4, l_3), (l_4, l_2), (l_4, l_1)\} \cup D_2$. The final dependence set D_3 shows that the two recursive calls l_5 and l_6 are independent. Similar examples are given by other divide-and-conquer programs such as the `copyTree` and `disposeTree` procedures from [3], in which our algorithm determines the recursive calls to be independent and parallelizable.

Implementation and experimental evaluation We have implemented our dependence analysis in the THOR analyzer [37, 35], which performs shape analysis of C programs using separation logic. We experimented with this implementation in the application area of *C-to-gates synthesis*, where the aim is to translate high level programs written in C into hardware circuits, bringing the power of hardware-based acceleration to mainstream programmers. Recent work in this area has made it possible to handle programs that manipulate dynamically-allocated pointer data structures, by finding a symbolic bound on the memory usage of the program [13]. However, the efficiency of the resulting circuits is very poor, because circuit optimizations cannot be performed in the absence of information about the heap dependences in the high level program. In joint work with Cook, Magill, Simsa and Singh [14], we used the THOR implementation of the dependence analysis to develop an optimizing C-to-VHDL compiler for circuit synthesis from programs with heap.

Design	**Latency reduction (%)**	**Throughput increase (%)**
Prio	45	110
Merge	65	282
Huffman	45	417

Figure 3.8: Latency and Throughput Measurements

This uses the computed dependences and adapts optimization techniques from [15] to synthesize optimized circuits.

Figure 3.8 shows the performance improvement, in terms of latency reduction and throughput increase, for three sample designs. The *Prio* program implements a priority queue, which repeatedly inputs a number of values from an input stream into a dynamically allocated linked list, sorts the values in the list, and then outputs them to an output stream. The *Merge* program repeatedly inputs values from two different streams, and then combines the two sequences into a single sorted list, before sending these values to an output stream. The *Huffman* program implements a tree data structure for binary encoding of symbols. It repeatedly inputs symbols and frequencies through two input streams to build a Huffman encoder tree. It then inputs values from a third input stream, computes their binary encodings using the encoder tree, and then outputs the encodings through an output stream.

The significant improvement in performance shown in Figure 3.8 is largely due to the detection of heap dependencies. For instance, by detecting the disjointness of list structures in the Merge program, the reading in of the two input sequences can be executed in parallel with one another, and a pipeline stage can also be introduced in which the outputting of the sorted list is done in parallel with the reading in of the next two input sequences.

3.6 Soundness

In this section we demonstrate soundness of the analysis, showing that any optimization based on the dependences we detect produces the same results as the original program. We do this by using an action trace semantics of commands. Our semantics is based on the trace semantics of Calcagno, O'Hearn and Yang presented in [11], where a trace is a sequential composition of atomic actions, and an atomic action is a function that transforms states. In our case we define a state as a triple of the form $(s, h, d) \in \texttt{Stacks} \times \texttt{Heaps} \times \mathcal{D}$ where

$$\texttt{Heaps} = \texttt{Loc} \rightharpoonup_{fin} ((\texttt{Fields} \to \texttt{Val}) \times P(\texttt{Lab}))$$

$$\texttt{Stacks} = (\texttt{Var} \cup \texttt{Var}') \to \texttt{Val}$$

$$\mathcal{D} = \mathcal{P}(\texttt{Lab} \times \texttt{Lab})$$

$$\texttt{States} = \texttt{Stacks} \times \texttt{Heaps} \times \mathcal{D}$$

The stack s and heap h are as defined in the section 3.2, where every heap cell has a label set containing the labels of commands that have accessed the cell. The element $d \in \mathcal{D}$ is a dependence relation between labels that will collect the dependences between commands as execution proceeds. Note that the label sets of cells and the dependence relation will be used as additional bookkeeping structures in the states, and without them we will have a standard stack and heap semantics of commands.

These concrete states will be related to the symbolic states used in the dependence analysis algorithm described in section 3.3. Recall that these states were of the form (P, D), where P is a formula and D is a dependence set computed by the analysis. We will use a notion of satisfaction between concrete and symbolic states, in which the formula P and dependence set D give an over-approximation of the concrete state.

Definition 3.11 (satisfaction) *For a state $(s, h, d) \in \texttt{States}$, a formula $P \in \mathcal{P}(\texttt{LSH})$ and a dependence set $D \in \mathcal{D}$, we define $(s, h, d) \models (P, D)$ if $s, h \models P$ and $d \subseteq D$. We write $[\![(P, D)]\!]$ for the set of all states that satisfy (P, D).*

Traces of programs are constructed by a sequential composition of atomic actions of the form $l : a$, where $l \in \texttt{Lab}$ is a label associated with the atomic action a. As in the semantics of commands described in section 2.1, an atomic action is semantically modelled as a total function of the form $f : \texttt{States} \to \mathcal{P}(\texttt{States})^\top$, where the $\top$ element represents a faulting execution (dereferencing a null pointer or an unallocated region of the heap), and the function maps to the powerset in order to account for possible non-determinism. Such functions are lifted to the topped powerset so that for a function f and $S \in \mathcal{P}(\texttt{States})$ we have

$$f(S) = \bigcup_{(s,h,d) \in S} f(s, h, d)$$

$l\colon a$	$[\![l : a]\!](s,h,d)$
$l\colon x := E$	$\{(s[x \mapsto [\![E]\!]s], h, d)\}$
$l\colon x := E \rightarrow f$	$\begin{cases} \{(s[x \mapsto v], h[\ell \mapsto (r, L \cup \{l\})], d \cup (\{l\} \times L))\} & \text{if } [\![E]\!]s = \ell, h(\ell) = (r,L), r(f) = v \\ \top & \text{otherwise} \end{cases}$
$l\colon E_1 \rightarrow f := E_2$	$\begin{cases} \{(s, h[\ell \mapsto (r', L \cup \{l\})], d \cup (\{l\} \times L))\} & \text{if } [\![E_1]\!]s = \ell, [\![E_2]\!]s = v, h(\ell) = (r,L), r' = r[f \rightarrow v] \\ \top & \text{otherwise} \end{cases}$
$l\colon \texttt{new}(x)$	$\{(s[x \mapsto \ell], h * \ell \mapsto (r, \{l\}), d) \mid \ell \in \texttt{Loc}\}$ where $r(f) = \texttt{nil}$ for all $f \in \texttt{Fields}$
$l\colon \texttt{dispose}(x)$	$\begin{cases} \{(s, h', d \cup (\{l\} \times L))\} & \text{if } [\![x]\!]s = \ell \text{ and } h = h' * \ell \mapsto (r,L) \\ \top & \text{otherwise} \end{cases}$
$l\colon \texttt{assume}(b)$	$\begin{cases} \{(s,h,d)\} & \text{if } [\![b]\!]s \\ \emptyset & \text{otherwise} \end{cases}$

Figure 3.9: Denotational semantics of primitive actions

where $S \cup \top = \top$ for any $S \in \mathcal{P}(\texttt{States})^\top$ and $f(\top) = \top$.

The denotational semantics of the primitive actions is given in figure 3.9. It is the standard stack and heap semantics of commands, but with additional dependence information recorded in the label sets of heap cells and the dependence relation. For heap-accessing commands, the label of the command is recorded in the label set of the heap cell that it accesses. The dependence relation accumulates discovered heap-carried dependences between commands. For example, in the case of the dispose command, although the cell that is accessed by the command is disposed, the dependences between this command and all previous commands that accessed the cell is recorded in the dependence relation. The $\texttt{assume}(b)$ action filters out states that do not satisfy the boolean b. Assume actions are used to model branching executions. For example, a conditional $\textit{if}\ (b)\ \textit{then}\ a_1\ \textit{else}\ a_2$ has possible traces $\texttt{assume}(b); a_1$ and $\texttt{assume}(\neg b); a_2$, representing the two possible execution branches.

Definition 3.12 (Action trace) *An action trace τ is a finite sequential composition of atomic actions*

$$\tau ::= (l : a) \mid \tau;\tau$$

Denotational semantics of action traces is given by the sequential composition of actions, which is defined inductively using figure 3.9:

$$[\![(l : a);\ \tau]\!](s,h) = [\![\tau]\!]([\![l : a]\!](s,h))$$

$$\begin{array}{rcl}
T(l:a) & = & \{l:a\} \\
T(l\!:\!\mathtt{com}(\mathcal{T})) & = & \{\tau \mid \tau = l\!:\!a_1;\cdots;l\!:\!a_n \text{ and } \mathtt{RV}(\tau) \subseteq \mathtt{RV}(l\!:\!\mathtt{com}[\mathcal{T}]), \mathtt{MV}(\tau) \subseteq \mathtt{MV}(l\!:\!\mathtt{com}(\mathcal{T})) \\
& & \quad \text{and } \forall U \in dom(\mathcal{T}).\ [\![\tau]\!]([\![(U,\mathcal{D})]\!]) \subseteq [\![(\mathcal{T}(U),\mathcal{D})]\!]\} \\
T(C_1;C_2) & = & \{\tau_1;\tau_2 \mid \tau_1 \in T(C_1), \tau_2 \in T(C_2)\}
\end{array}$$

Figure 3.10: Action trace semantics of commands

.

The action trace sets of programs are shown in Figure 3.10. For atomic commands the trace is the atomic command itself. For specified commands, any trace that satisfies the specification given for the command is taken as a possible execution trace of the command. Although this may permit more traces than the actual command may have, an over-approximation of possible executions is sufficient to demonstrate soundness. Thus, for a specified command $l : \mathtt{com}(\mathcal{T})$, we first require that any trace τ only reads/modifies variables that are read/modified by the command. Second, the trace may only transform the concrete state according to the spec table. Note that, since the spec table does not make assertions about dependences, the concrete states may have any possible dependence relations in the input states $[\![(U,\mathcal{D})]\!]$ and output $[\![(\mathcal{T}(U),\mathcal{D})]\!]$. Every atomic action in the trace of the specified command is given the label l of the command, since the action forms part of the execution of the command.

Given this definition of trace sets, for any sequential block $l_1 : C_1;\ldots;l_n : C_n$, we have that every trace is of the form $(\tau_1);\ldots;(\tau_n)$, where $\tau_k \in T(l_k : C_k)$ and every atomic action in τ_k has the label l_k, since it is part of the execution of the command with label l_k. Our aim is to show that, given the set of dependences computed by our analysis, then for any trace τ of the sequential block, if we perform a reordering of atomic actions in τ that satisfies the computed dependences, then the resulting trace will produce the same output states as τ.

Definition 3.13 (Dependence) *The set of heap dependences in a trace τ from an initial state (s,h,d) is defined as*

$$hDep(\tau,(s,h,d)) = \bigcup\{d' \mid (s',h',d') \in [\![\tau]\!](s,h,d)\}$$

There is a stack dependence between labels l and l' in τ if there exist actions $l : a$ and $l' : a'$ in τ that access common stack variables. There is an allocation dependence between l and l' if *there exist actions $l : a$ and $l' : a'$ in τ such that one is* `new` *and the other is* `dispose`*. We write $sDep(\tau)$ and $aDep(\tau)$ for the set of all stack and allocation dependences respectively. We write $dep(\tau, (s, h, d))$ for the set of all stack, heap and allocation dependences in τ from initial state (s, h, d).*

Lifting to a set of states S, we write $dep(\tau, S)$ (or $hDep(\tau, S)$) for the set of all dependences (or heap dependences) in τ from any state in S.

The next lemma describes a locality property of traces that is similar to the one given in definition 2.5, except that in this case the locality is only with respect to the heap, rather than all components of the concrete state.

Lemma 3.4 (heap locality) *If we have $[\![\tau]\!](s, h, d) = S$ and $S \neq \top$ then $[\![\tau]\!](s, h * h_f, d) \subseteq \{(s', h' * h_f, d') \mid (s', h', d') \in S\}$*

Proof: The proof is by induction on τ. For the base case where τ is an atomic action, the result can be checked for each of the actions in figure 3.9. For the inductive case, assume $\tau = \tau'; (l : a)$. Let $[\![\tau']\!](s, h, d) = S'$ and $[\![l : a]\!](S') = S$. By induction hypothesis we have $[\![\tau']\!](s, h * h_f, d) \subseteq \{(s', h' * h_f, d') \mid (s', h', d') \in S'\}$. Let $(s', h' * h_f, d') \in [\![\tau']\!](s, h * h_f, d)$ and $(s', h', d') \in S'$. By the base case we have $[\![l : a]\!](s', h' * h_f, d') \subseteq \{(s'', h'' * h_f, d'') \mid (s'', h'', d'') \in S\}$. Hence functional composition gives us $[\![\tau'; (l : a)]\!](s, h * h_f, d) \subseteq \{(s'', h'' * h_f, d'') \mid (s'', h'', d'') \in S\}$. ■

Our next result relates the symbolic execution of the analysis with the concrete trace semantics, showing that the dependences computed by the analysis are an over-approximation of any possible dependence relation resulting in any concrete execution of the program.

Lemma 3.5 (dependence detection soundness) *Given a sequential block C and pre-condition $P \in \mathcal{P}(\texttt{LSH})$, assume we have $\mathsf{getDeps}(C, P) = D$ (where the $\mathsf{getDeps}$ function is as defined in section 3.3). Then for any trace τ of C and any state s, h that satisfies P, we have $dep(\tau, (s, h, \emptyset)) \subseteq D$*

Proof: We have $sDep(\tau) \subseteq \mathsf{getStackDeps}(C)$ and $aDep(\tau) \subseteq \mathsf{getAllocDeps}(C)$ by definition of the $\mathsf{getStackDeps}$ and $\mathsf{getAllocDeps}$ functions. The main thing is to show that $hDep(\tau, (s, h, \emptyset)) \subseteq \mathsf{getHeapDeps}(C, P)$.

For this it is sufficient to show that, if $\mathsf{Exec}(C, (P_1, D_1)) = (P_2, D_2)$, then $[\![\tau]\!]([\![(P_1, D_1)]\!]) \subseteq [\![(P_2, D_2)]\!]$. This is shown by induction on C, where the base cases are when C is an atomic command or a specified command. For atomic commands, the result follows by checking the soundness of each of the symbolic execution rules in figure 3.3.

For a specified command $l : \mathtt{com}[\mathcal{T}]$, assume $H \in P_1$ and $s_1, h_1 \models H$ and $d_1 \subseteq D_1$. As defined in algorithm 2, the $\mathsf{ExecSpec}$ function finds some $U \in dom(\mathcal{T})$ such that $\mathsf{Frm}(H, U, V) = (\Pi_F, \Sigma_F, L_A)$ and returns the result (P_2, D_2) where

$$P_2 = \{\ \Pi_F \wedge \Pi \wedge \Sigma_F * \Psi[L_A \cup \{l\}] \ \mid\ \Pi \wedge \Psi \in \mathcal{T}(U)\ \}$$

$$D_2 = D \cup (\{l\} \times L_A)$$

Our aim is to show that $[\![\tau]\!](s_1, h_1, d_1) \subseteq [\![(P_2, D_2)]\!]$. Let $U = \Pi_U \wedge \Psi_U$. By soundness of frame inference (Corollary 3.3) we know that $H \vdash \Pi_F \wedge \Pi_U \wedge \Sigma_F * \Psi_U[L_A]$ and therefore $s_1, h_1 \models \Sigma_F * \Psi_U[L_A]$. Let $h_1 = h_f * h_a$ such that $s_1, h_f \models \Sigma_F$ and $s_1, h_a \models \Psi_U[L_A]$.

By definition of the trace set of specified commands in figure 3.10, we have $[\![\tau]\!](s_1, h_a, d_1) \subseteq [\![(\mathcal{T}(U), \mathcal{D})]\!]$. Assume we have some output state $(s', h', d') \in [\![\tau]\!](s_1, h_a, d_1)$. Since every heap cell in h_a has labels contained in L_A and every action in τ has label l, we know that the resulting heap h' can only have labels in $L_A \cup \{l\}$ and the resulting dependence set d' can only accumulate dependences between l and labels in L_A. We therefore have $d' \subseteq d_1 \cup (\{l\} \times L_A)$. This gives us

$$[\![\tau]\!](s_1, h_a, d_1) \subseteq [\![(\{\Pi \wedge \Psi[L_A \cup \{l\}] \mid \Pi \wedge \Psi \in \mathcal{T}(U)\}, D_1 \cup (\{l\} \times L_A)]\!] \qquad (\dagger)$$

We then have

$\llbracket\tau\rrbracket(s_1, h_1, d_1)$

$= \llbracket\tau\rrbracket(s_1, h_f * h_a, d_1)$

$\subseteq \{(s', h_f * h', d') \mid (s', h', d') \in \llbracket\tau\rrbracket(s_1, h_a, d_1)\}$ (by heap locality lemma 3.4)

$\subseteq \{(s', h_f * h', d') \mid (s', h_f) \models \Pi_F \wedge \Sigma_F$ and $(s', h', d') \in \llbracket\tau\rrbracket(s_1, h_a, d_1)\}$

(since $(s_1, h_f) \models \Pi_F \wedge \Sigma_F$ and τ does not modify variables appearing in $\Pi_F \wedge \Sigma_F$ (by soundness of frame inference and trace semantics of specified commands in figure 3.10))

$\subseteq \{(s', h'_f * h', d') \mid (s', h'_f) \models \Pi_F \wedge \Sigma_F$ and $(s', h', d') \in \llbracket\tau\rrbracket(s_1, h_a, d_1)\}$

$= \llbracket(\{\ \Pi_F \wedge \Pi \wedge \Sigma_F * \Psi[L_A \cup \{l\}] \ \mid \ \Pi \wedge \Psi \ \in \ \mathcal{T}(U)\ \}, D_1 \cup (\{l\} \times L_A))\rrbracket$ (by †)

$= \llbracket P_2, D_2)\rrbracket$

For the inductive case, let the induction hypothesis hold for C_1 and C_2 and let $\tau = \tau_1; \tau_2$ be a trace of $C_1; C_2$ such that τ_i is a trace of C_i. Let $\mathsf{Exec}(C_1; C_2, (P_1, D_1)) = (P_2, D_2)$. We have by definition of the Exec function that $(P_2, D_2) = \mathsf{Exec}(C_2, \mathsf{Exec}(C_1, (P_1, D_1)))$. By the induction hypothesis, we have $\llbracket\tau_1\rrbracket(\llbracket(P_1, D_1)\rrbracket) \subseteq \llbracket\mathsf{Exec}(C_1, (P_1, D_1))\rrbracket$ and $\llbracket\tau_2\rrbracket(\llbracket\mathsf{Exec}(C_1, (P_1, D_1))\rrbracket) \subseteq \llbracket(P_2, D_2)\rrbracket$. Hence by functional composition of traces we get $\llbracket\tau_1; \tau_2\rrbracket(\llbracket(P_1, D_1)\rrbracket) \subseteq \llbracket(P_2, D_2)\rrbracket$

■

So far we have related the symbolic execution semantics with the concrete semantics. However, the notion of dependence given in definition 3.13 has been formulated with respect to the dependence relation that is accumulated by the concrete semantics. It remains for us to show that these accumulated dependences indeed satisfy the property that reordering actions with respect to these dependences will produce the same behaviour as the original trace. This final result will use the next lemma, which shows that if there are no dependences between an action and all the actions before it, then the action can be commuted above those actions to give an (extensionally) equivalent trace.

Lemma 3.6 (Reordering) *Let $\tau = \tau'; (l : a)$ and $\llbracket\tau\rrbracket(s, h, d) \neq \emptyset$. If for all actions $(l' : a')$ in τ' we have $(l, l') \notin dep(\tau, (s, h, d))$, then $\llbracket(l : a); \tau'\rrbracket(s, h, d) = \llbracket\tau\rrbracket(s, h, d)$.*

Proof: The proof is by induction on τ'. For the base case we assume that τ is an atomic action $(l' : a')$. It can then be checked for all combinations of atomic actions in figure 3.9 that if $\llbracket(l' :$

$a') : (l : a)]\!](s,h,d) = S$, $S \neq \emptyset$ and $(l,l') \notin \mathit{dep}((l' : a'); (l : a), (s,h,d))$ then $[\![(l : a) : (l' : a')]\!](s,h,d) = S$.

For the inductive case, we assume that $\tau' = \tau''; (l' : a')$ and that $[\![\tau''; (l' : a'); (l : a)]\!](s,h,d) = S$ and $S \neq \emptyset$. Let $[\![\tau'']\!](s,h,d) = S'$. We have $[\![(l' : a'); (l : a)]\!](S') = S$. We know that $(l,l') \notin \mathit{dep}(\tau, (s,h,d))$, which means that $(l,l') \notin \mathit{sDep}((l' : a'); (l : a))$ and $(l,l') \notin \mathit{aDep}((l' : a'); (l : a))$. Also, by definition of heap dependence, we have $(l,l') \notin \bigcup\{d' \mid (s',h',d') \in S\}$. Since $[\![(l' : a'); (l : a)]\!](S') = S$, by definition of heap dependence we have $(l,l') \notin \mathit{hDep}((l' : a'); (l : a), S')$. Therefore $(l,l') \notin \mathit{dep}((l' : a')(l : a), S')$ and by the base case we have $[\![(l : a); (l' : a')]\!](S') = S$. This gives us $[\![\tau''; (l : a); (l' : a')]\!](s,h,d) = S$.

In order to show that $[\![(l : a); \tau''; (l' : a')]\!](s,h,d) = S$, it is sufficient to show $[\![\tau''; (l : a)]\!](s,h,d) = [\![(l : a); \tau'']\!](s,h,d)$. Let $(l'' : a'')$ be an action in τ''. We know that $(l,l'') \notin \mathit{dep}(\tau, (s,h,d))$, which means that $(l,l'') \notin \mathit{sDep}(\tau''; (l : a))$ and $(l,l'') \notin \mathit{aDep}(\tau''; (l : a))$. By definition of heap dependence, we have $(l,l'') \notin \bigcup\{d' \mid (s',h',d') \in S\}$. Since $[\![\tau''; (l : a); (l' : a')]\!](s,h,d) = S$, by definition of heap dependence we have $(l,l'') \notin \mathit{hDep}(\tau''; (l : a); (l' : a'), (s,h,d))$. This means that $(l,l'') \notin \mathit{hDep}(\tau''; (l : a), (s,h,d))$. So we have $(l,l'') \notin \mathit{dep}(\tau''; (l : a), (s,h,d))$, which by the induction hypothesis implies that $[\![\tau''; (l : a)]\!](s,h,d) = [\![(l : a); \tau'']\!](s,h,d)$

■

Theorem 3.7 (Soundness) *Given a sequential block C and pre-condition $P \in \mathcal{P}(\mathtt{LSH})$, assume we have* $\mathsf{getDeps}(C,P) = D$. *Let τ be any trace of C and τ' be any reordering of actions in τ that respects all the dependences in D. Then for any $(s,h) \models P$ such that $[\![\tau]\!](s,h,\emptyset) \neq \emptyset$, we have $[\![\tau]\!](s,h,\emptyset) = [\![\tau']\!](s,h,\emptyset)$.*

Proof: Let s,h be a state satisfying the pre-condition P and let $[\![\tau]\!](s,h,\emptyset) = S$ where $S \neq \emptyset$. By Lemma 3.5 we have $\mathit{dep}(\tau, (s,h,\emptyset)) \subseteq D$. So we let τ' be any reordering of actions in τ respecting dependences in $\mathit{dep}(\tau, (s,h,\emptyset))$, and it is sufficient to show $[\![\tau']\!](s,h,\emptyset) = S$. We show this by induction on τ. The base case is when τ is an atomic action, in which case $\tau = \tau'$ and we are done.

For the inductive case, assume that $\tau = \tau_1; (l : a)$ and that $[\![\tau_1]\!](s,h,\emptyset) = S_1$. We know that $S_1 \neq \emptyset$ since $S \neq \emptyset$. Now let $\tau' = \tau''; (l : a); \tau'''$ be the dependency respecting reordering of

τ where, for all actions $(l' : a')$ in τ''', we have $(l, l') \notin dep(\tau, (s, h, \emptyset))$. We have that $\tau''; \tau'''$ is a reordering of τ_1. We also have $dep(\tau_1, (s, h, \emptyset)) \subseteq dep(\tau, (s, h, \emptyset))$ because the dependence relation can only get bigger if another action is added to the trace. Therefore, since $\tau''; \tau'''$ respects all dependences in $dep(\tau, (s, h, \emptyset))$, it also respects all dependences in $dep(\tau_1, (s, h, \emptyset))$. Hence by the induction hypothesis we have $[\![\tau''; \tau''']\!](s, h, \emptyset) = S_1$.

Now let $[\![\tau'']\!](s, h, \emptyset) = S'$. We have $[\![\tau''']\!](S') = S_1$ and $[\![l : a]\!](S_1) = S$, which gives us $[\![\tau'''; (l : a)]\!](S') = S$. To complete the proof, we just need to show that $[\![(l : a); \tau''']\!](S') = S$, since that will give us $[\![\tau''; (l : a); \tau''']\!](s, h, \emptyset) = S$.

Claim: $[\![(l : a); \tau''']\!](S') = S$. To prove this claim, let $(l' : a')$ be an action in τ'''. We know that $(l, l') \notin dep(\tau, (s, h, \emptyset))$, which means that $(l, l') \notin sDep(\tau'''; (l : a))$ and $(l, l') \notin aDep(\tau'''; (l : a))$. Also, by definition of heap dependence, we have $(l, l') \notin \bigcup\{d' \mid (s', h', d') \in S\}$. Since $[\![\tau'''; (l : a)]\!](S') = S$, by definition of heap dependence we have $(l, l') \notin hDep(\tau'''; (l : a), S')$. So there are no stack, allocation or heap dependences between any action in τ''' and $l : a$ when executed from any state in S'. Hence by the reordering lemma 3.6 we have $[\![(l : a); \tau''']\!](S') = S$.
■

3.7 Conclusion

In this chapter we have introduced labelled separation logic and demonstrated its application to the optimization of programs. By annotating formulae with labels to keep track of memory regions that are accessed by commands, labelled separation logic permits the analysis of memory separation properties throughout a program's lifetime. Apart from the ability to detect heap dependences in this way, we have also identified the notion of dependences due to dynamic memory allocation and how it is important to account for such dependences to ensure the safety of optimizations, which we have formally demonstrated. Although we presented examples and initial experiments, there is much more to explore in terms of practical applications.

Much progress has been made in automated analysis with separation logic, which provides our dependence detection method with many potential advantages over traditional methods. For example, the important issue of scalability of the shape analysis, which has been an obstacle for heap detection methods in general, may be addressed with the use of join operators such as the ones described in [55]. Compositionality of the shape analysis has also been achieved with the Bi-abduction method of [8]. This permits procedures to be analysed independently of their callers, which would allow dependence detection and optimization for program components with unknown calling contexts. We would also like to extend the dependence detection method to handle more arbitrary data structures, such as composite heap-structures (e.g. nested lists) [1] which commonly appear in industrial programs, and to use AI techniques to infer data structure definitions automatically, as in [24].

We have briefly described the application of our method to the automated synthesis of hardware circuits, but so far we have only conducted preliminary experiments to demonstrate the potential benefits. With the extensions described above, we hope to explore applications to a greater class of industrial software that stands to benefit from fast execution, or execution with reduced energy consumption.

Chapter 4

Ownership Inference for Concurrent Programs

In this chapter we turn our attention to the sharing of resources in concurrent programs. The analysis of concurrent programs is receiving much interest in the multi-core age, but is a difficult problem because of the need to consider possible interleavings between concurrent processes, which becomes even more complicated in the presence of aliasing in the heap. This is especially true in the case of ‘daring’ concurrent programs, where resources may be accessed by concurrent processes outside of critical regions, and *ownership* of resources is dynamically transferred during program execution. Inferring how this ownership transfer occurs is the key to the successful analysis and verification of such programs. Concurrent separation logic (CSL) [42], which we described in the introduction, achieves modular reasoning about dynamic ownership transfer with the use of *resource invariants* that describe the ownership of shared resources. The problem of ownership inference can then be posed as the ability to automatically infer the resource invariants so that a program proof in CSL can be automated.

Existing approaches for automating CSL include the reachability-based fixpoint method of Gotsman et al. [23] and the more recent bi-abduction method of Calcagno et al. [9]. Both of these approaches fail on some simple programs due to problems with ownership inference. The difficulty is that, as observed in [9], ownership is a global property of the program based on how resources are accessed at arbitrary points in the program, and hence cannot be determined by analysing the critical regions in isolation.

In this chapter we present an algorithm which addresses the ownership inference problem using labelled separation logic. We extend the method of [23] to track ownership of heap state through the program proof, and infer transfers of ownership based on heap accesses that may be made at possibly arbitrary program points. Such tracking is done using a labelling technique similar in nature to the one we used for the dependence analysis in chapter 3. However, inferring ownership is different from detecting command footprints, because there is not a static distinction between the part of the heap that is shared and the part that belongs to each thread. Instead, ownership is a *dynamic* property in the sense that heap cells may move in and out of the shared state and the local states of different threads at different points in program execution. O'Hearn refers to this phenomenon as "ownership is in the eye of the asserter" since the asserter has to choose the right resource invariant in order to construct a proof in CSL. However, our algorithm shows that the ownership policy can be inferred by tracking how resources are accessed in the program. We demonstrate how the algorithm verifies programs which both of the previous approaches (reachability-based fixpoint and bi-abduction) cannot handle. Also unlike the previous methods, our algorithm does not require user annotations about ownership distribution in the pre-condition of the concurrent program, as it infers this automatically.

We start in the next section with a description of the background on the programming language, CSL reasoning and the reachability-based fixpoint method of [23] for inferring resource invariants. In the following section we give an informal description of our label-tracking approach to ownership inference. In section 4.3 we describe the formulae that are used in our analysis, and in section 4.4 we present the algorithm in detail. In section 4.5 we demonstrate the algorithm on examples that could not be handled by previous methods, and illustrate how our method handles different aspects of dynamic ownership transfer. Finally, we address ownership inference for programs with while loops in section 4.6.

4.1 Background

The parallel programming language we use is adapted from [42, 9], where a concurrent program consists of an initialization phase, a resource declaration, and a single parallel composition of sequential commands

$$\begin{aligned} Prg \;\; ::= \;\; & init; \\ & \texttt{resource}\; r_1(\text{variable list}), \ldots, r_m(\text{variable list}) \\ & C_1 \parallel \cdots \parallel C_N \end{aligned}$$

The initialisation phase is some sequential code to set values of variables or create data structures. The resource declaration $r_1(\text{variable list}), \ldots, r_m(\text{variable list})$ introduces a finite number of resource names that will be used in the program. We let $Res = \{r_1, \ldots, r_m\}$ be the set of resource names used in the program, and $\texttt{Var}$ a set of program variables ranged over by $x, y, z, \ldots$. Every resource name is associated with a set of program variables, and we write $\texttt{Var}(r)$ for the set of variables of resource r, and $\texttt{Var}(\texttt{Res})$ for the set of variables that are associated with any resource.

The resource declaration is followed by a single parallel composition of the threads $C_1, \ldots, C_N$ with thread identifiers $1, \ldots, N$ respectively. Both the initialisation code and the threads are constructed with standard sequential commands, and threads also use a synchronization construct. The grammar for these commands is given as

$$\begin{array}{lcll} E & \in & \texttt{Var} \cup \{\texttt{nil}\} & \textit{expressions} \\ B & ::= & E = E \mid E \neq E & \textit{boolean expressions} \\ C & ::= & x := E \mid x := [y] \mid [x] := E \mid \texttt{new}(x) \mid \texttt{dispose}(x) \mid \texttt{assume}(B) & \textit{commands} \\ & & \mid C;C \mid C + C \mid \texttt{with}\; r\; \texttt{when}\; B\; \texttt{do}\; C & \end{array}$$

For the discussion in this chapter, we assume a single field in heap cells for simplicity, although multiple fields may also be permitted. The heap look-up command ($x := [y]$) sets x to the value of the field in the cell at y, and the mutation command ($[x] := E$) updates the field in the cell at x to the expression E. We also have primitive commands for variable assignment (x:=E), dynamic allocation ($\texttt{new}(x)$), deallocation ($\texttt{dispose}(x)$) and $\texttt{assume}(B)$ commands, which block if condition B is not satisfied. We shall sometimes write c for primitive commands that do not access the heap, and $c[E]$ for any command that accesses the heap at E (look-up, mutation or dispose).

For simplicity, the only sequential constructs are sequential composition and non-deterministic choice. Conditionals $\texttt{if}\; B\; \texttt{then}\; C_1\; \texttt{else}\; C_2$ can then be implemented as $\texttt{assume}(B); C_1 + \texttt{assume}(\neg B); C_2$. For the moment, we shall illustrate the concepts on loop-free programs, and will describe the extension of our method to while loops in section 4.6. Synchronization between threads is implemented with the conditional critical region (CCR) command $\texttt{with}\ r\ \texttt{when}\ B\ \texttt{do}\ C$, which we discussed in the introduction. It waits until condition B is true and no other CCR for r is currently executing, and then executes the body C in mutual exclusion with all other CCRs for r.

Programs in this language have well-formedness constraints that a variable belongs to at most one resource, a variable belonging to a resource can only appear in a critical region for that resource, and if a variable is modified in one thread, then it cannot appear in another unless it belongs to a resource. We write $\texttt{MV}(C_i)$ for the set of variables that are modified in thread C_i.

CSL reasoning We now give the formulation of CSL reasoning for our programming language. Firstly, each resource name r is associated a formula $I(r)$ which is the resource invariant. The invariant should satisfy the condition that any variable occurring free in the formula must belong to resource r. When reasoning about each individual thread C_i, the inference rule for critical regions is given as:

$$\frac{\{(P * I(r)) \wedge B\}\, C\, \{Q * I(r)\}}{\{P\}\, \texttt{with}\ r\ \texttt{when}\ B\ \texttt{do}\ C\, \{Q\}}\ \texttt{free}(P, Q) \cap \texttt{MV}(C_j) = \emptyset \text{ for all } j \neq i$$

The invariant is assumed to hold separately from the local state P when a thread enters a critical region, and at the end of the CCR body, the thread must re-establish the resource invariant, along with some other post-condition Q. When the thread exits the CCR, it gives up ownership of the shared resources, and so the resource invariant is hidden in the inferred specification for the CCR. The side condition requires that no other thread modifies the free variables in the thread's local pre- and post-conditions. In this way, outside the CCR the reasoning can proceed independently of shared resources, so the parallel composition rule simply combines the pre- and post-conditions for each thread:

$$\frac{\{P_1\}\, C_1\, \{Q_1\} \cdots \{P_n\}\, C_n\, \{Q_n\}}{\{P_1 * \cdots * P_n\}\, C_1 \parallel \cdots \parallel C_n\, \{Q_1 * \cdots * Q_n\}}\ \texttt{free}(P_i, Q_i) \cap \texttt{MV}(C_j) = \emptyset \text{ when } j \neq i$$

To infer an overall specification of the program, we have the rule for complete programs:

$$\frac{\{P\}\, \mathit{init}\, \{I(r_1) * \cdots * I(r_m) * P'\} \quad \{P'\}\, C_1 \parallel \cdots \parallel C_N\, \{Q\}}{\{P\}\, \mathit{Prg}\, \{Q * I(r_1) * \cdots * I(r_m)\}}$$

```
                    c := nil;
                   resource r(c)
with r when true do {               ||  with r when true do {
  if c ≠ nil then dispose(c);       ||    if c ≠ nil then dispose(c);
  new(c);                           ||    new(c);
}                                   ||  }
```

Figure 4.1: Replace buffer

In a complete program the resource invariants must be separately established by the initialization code, along with a separate pre-condition P' for the parallel composition of threads. As the parallel composition is specified independently of the shared resources, the rule for complete programs brings back the resource invariants in the overall post-condition of the program. The soundness of CSL has been shown in [6], where it is also shown that a program proof in CSL guarantees that the program is memory safe and data race free. We illustrate reasoning with CSL on the example program shown in figure 4.1, which illustrates dynamic ownership transfer. In this program there is a one-place buffer at the heap cell c, and access to the buffer is synchronized with resource name r. There are two identical threads, and each thread has a single CCR in which the thread first gains ownership of the cell at c when it enters the critical region. It then disposes the cell and allocates a new cell for the buffer, and then gives up ownership of this cell when it exits the critical region. This is a simplification of concurrent programs where different processes may compute some value and place it in a shared buffer, so that other processes may then use the value and then replace it with a new one. To reason about this program, the resource invariant we can use for resource r is

$$I(r) \overset{def}{=} \{(\mathtt{emp} \wedge c = \mathtt{nil}) \vee (c \mapsto \mathtt{nil})\}$$

which states the buffer is either empty and c is nil, or c points to a single cell in the buffer. The CSL proof of the program is shown in figure 4.2, where application of the proof rules is displayed in a linear fashion through the structure of the program. The initial pre-condition of the program is the empty heap $\{\mathtt{emp}\}$, and after executing the initialisation code we have the post-condition $\{\mathtt{emp} \wedge c = \mathtt{nil}\}$. In accordance with the rule for complete programs, we split this formula into a pre-condition for each of the threads and the resource invariant, so that each thread gets pre-condition emp.

Each thread is then proven independently with its own pre-condition, and in this case the proofs are

$$\{\mathtt{emp} \wedge c = \mathtt{nil}\}$$
$$\{\mathtt{emp} * \mathtt{emp} * (\mathtt{emp} \wedge c = \mathtt{nil})\}$$
$$\{\mathtt{emp} * \mathtt{emp} * I(r)\}$$

```
{emp}                                      ||  {emp}
with r when true do {                      ||  with r when true do {
  {emp * I(r)}                             ||    {emp * I(r)}
  {(emp ∧ c = nil) ∨ (c↦nil)}              ||    {(emp ∧ c = nil) ∨ (c↦nil)}
  if c ≠ nil then dispose(c);              ||    if c ≠ nil then dispose(c);
  {emp}                                    ||    {emp}
  new(c);                                  ||    new(c);
  {c↦nil}                                  ||    {c↦nil}
  {emp * I(r)}                             ||    {emp * I(r)}
}                                          ||  }
{emp}                                      ||  {emp}
```

$$\{\mathtt{emp} * \mathtt{emp} * I(r)\}$$

Figure 4.2: CSL proof of replace buffer

identical for the two threads. When we enter the CCR, we add the resource invariant in accordance with the CCR rule, and then apply the standard rules of separation logic to get to the end of the CCR body. At this point, we re-establish the resource invariant as required by the CCR rule: the new cell allocated by the thread becomes part of the resource invariant and the remaining post-condition is empty. This post-condition is retained outside the CCR in the thread's local state, and the resource invariant disappears with application of the CCR rule. Having proven each thread, the parallel composition rule allows us to combine the specifications of the threads to get the empty pre- and post-condition for the parallel composition, and the rule for complete programs adds the resource invariant back to the overall post-condition of the whole program. This is an example of a program proof in CSL, with which we have shown that there are no memory errors or data races in the program.

The Reachability Method for Resource Invariant Synthesis We now outline the first method for automatically synthesizing resource invariants, which was introduced by Gotsman et al. in [23]. The approach requires the user to provide the post-condition of the initialisation phase, and also to specify how this is separated into the pre-condition of each thread and the initial resource invariants for each resource name. Thus in the case of the buffer program in figure 4.1, the user would provide the pre-condition $\{\mathtt{emp}\}$ for each of the two threads, and the initial resource

```
{emp}
with $r$ when true do {
  {emp $* I_0(r)$}
  {emp $\wedge c =$ nil}
  if $c \neq$ nil then dispose($c$);
  {emp}
  new($c$);
  {$c \mapsto$ nil}
  {emp $* c \mapsto$ nil}  //new RI disjunct {$c \mapsto$ nil}
}
{emp}
```

||

```
{emp}
with $r$ when true do {
  {emp $* I_1(r)$}
  {(emp $\wedge c =$ nil) $\vee$ ($c \mapsto$ nil)}
  if $c \neq$ nil then dispose($c$);
  {emp}
  new($c$);
  {$c \mapsto$ nil}
  {emp $* c \mapsto$ nil}  //new RI disjunct {$c \mapsto$ nil}
}
{emp}
```

Figure 4.3: Reachability-based invariant synthesis for replace buffer

invariant $I_0(r) = \{\texttt{emp} \wedge c = \texttt{nil}\}$. Given this information, the method proceeds by attempting to construct a CSL proof of each thread, using the existing approximation for the resource invariant.

When the proof reaches a CCR, the current resource invariant is added to the state, in accordance with the CCR rule. When the proof reaches the end of the CCR body, the formula needs to be split into the thread local state and the shared state, which is the point at which the question of ownership transfer comes in. At this point, the method uses the heuristic that the part of the formula that is *reachable* from the resource variables is assumed to be part of the resource invariant. This part is added as a new disjunction to the current approximation of the resource invariant, and the remaining part is kept as the thread's local state, and the proof continues. This process continues until enough disjunctions are gathered that a true resource invariant for the program is found.

In the case of the buffer program from figure 4.1, the first iteration of the method is shown in figure 4.3. Starting with the left thread, when we enter the CCR the initial approximation $I_0(r)$ is added. At the end of the CCR body the formula is $\{c \mapsto \texttt{nil}\}$. Now we apply the reachability heuristic to obtain the part reachable from resource variable c, which gives us the formula $\{c \mapsto \texttt{nil}\}$ as the new disjunct for the resource invariant and emp as the thread's local state. We refine the resource invariant approximation to get $I_1 = I_0 \vee \{c \mapsto \texttt{nil}\}$ and the thread local state emp is obtained outside the CCR. When we analyse the next thread, the CCR yields the same disjunct for the resource invariant, so the new approximation is $I_2 = \ \{(\texttt{emp} \wedge c = \texttt{nil}) \vee (c \mapsto \texttt{nil})\}$. We have therefore reached the valid resource invariant for the program which was used in the CSL proof in figure 4.2.

```
                    f := 0;
                 resource r(c, f)

new(x);                       ||  with r when f = 1 do {
with r when f = 0 do {        ||     y := c;
   c := x;                    ||     f := 0;
   f := 1;                    ||  }
}                             ||  dispose(y);
```

Figure 4.4: Put-get buffer

This method provides an elegant way of synthesizing resource invariants, by collecting disjuncts based on the program's execution. However, it is missing a method for inferring how ownership transfers occur in the program. Firstly, one may have noted the initial annotation burden for the user, since the initial *splitting* for the pre-conditions and resource invariants must be given. This involves some kind of ownership inference on the part of the user in order to determine how the splitting must be done. For example, consider the case where the initialisation code for the buffer is:

$$\texttt{new}(x); c = x;$$

So the buffer is initialised with some heap cell at c rather than the empty heap. In this case the post-condition of the initialisation code is $\{c = x \wedge x \mapsto \texttt{nil}\}$. This can be spatially split to either yield $c \mapsto \texttt{nil}$ or $\texttt{emp}$ as the initial resource invariant I_0, and it is up to the user to give us the correct one (which is the first).

A more significant problem is the use of a heuristic to decide ownership transfers occurring during program execution. We illustrate this with the put-get buffer example shown in figure 4.4. In this case, instead of each thread replacing the buffer cell, we have one thread which puts the cell in and another which takes it out of the buffer. The variable c points to the buffer cell and flag variable f indicates whether the buffer is full or empty, and is initialised to false. The left thread first creates a new cell at x. It then waits until the buffer is empty, then enters the CCR and sets c to point to the cell and sets f to true. Ownership of the cell has now moved into the buffer (the shared state). The right thread waits until the buffer is full, and then sets the variable y to c and f to false. After exiting the CCR, the thread has obtained ownership of the heap cell, and it then disposes the cell. To do a CSL proof of this program, the correct resource invariant is given by:

$$\{(f = 0 \wedge \texttt{emp}) \vee (f = 1 \wedge c \mapsto \texttt{nil})\}$$

```
{emp}
new(x);
{$x \mapsto$ nil}
with r when f = 0 do {
  {$f = 0 \wedge x \mapsto$ nil $* (f = 0 \wedge$ emp$)$}
  $c := x$;
  $f := 1$;
  {$f = 1 \wedge c = x \wedge x \mapsto$ nil}
}    // new RI disjunct {$f = 1 \wedge c \mapsto$ nil}
{emp}
```

```
{emp}
with r when f = 1 do {
  {$f = 1 \wedge$ emp $* ((f = 0 \wedge$ emp$) \vee (f = 1 \wedge c \mapsto$ nil$))$}
  $y := c$;
  $f := 0$;
  {$f = 0 \wedge y = c \wedge c \mapsto$ nil}
}    // new RI disjunct {$f = 0 \wedge c \mapsto$ nil}
{emp}
dispose(y);
{?}    // analysis fails
```

Figure 4.5: Failed analysis of put-get buffer

which states that either the buffer is empty and f is false, or the buffer has a single cell at c and f is true.

We now describe how the reachability method would proceed in this example. Initially there is no heap allocated so the initial resource invariant is $f = 0 \wedge \mathtt{emp}$ and the the thread pre-condition is emp. The analysis proceeds as shown in figure 4.5. When the first thread enters the CCR, the initial resource invariant is added. When it reaches the end of the CCR, we have the formula $\{f = 1 \wedge c = x \wedge x \mapsto \mathtt{nil}\}$. At this point, since the cell is reachable from resource variable c, the resource invariant gets ownership of the cell with the new disjunct $\{f = 1 \wedge c \mapsto \mathtt{nil}\}$ and the thread gets emp in its local state. We then analyse the right thread and add the resource invariant at the beginning of the CCR. At the end of the CCR, we again apply the reachability heuristic, which now gives the disjunct $\{f = 0 \wedge c \mapsto \mathtt{nil}\}$ for the resource invariant. This says that the buffer is not full, and yet there is a cell in it. Because of this splitting, the thread's local state becomes $\{\mathtt{emp}\}$. The algorithm then fails when it attempts to dispose the cell, as the thread does not have the required ownership. Hence, the reachability heuristic caused the resource invariant to incorrectly keep ownership of resource which belonged to the thread.

4.2 Label Tracking for Ownership Inference

We now give an informal introduction to our label-tracking approach to address the ownership inference problem, which both alleviates the need for initial user annotations about ownership and also infers ownership transfers during execution without using heuristics. The difficulty in ownership inference is that when a decision has to be made about how to spatially split a formula to distribute ownership, this choice cannot be based solely on the existing information about the formula. It actually depends on how resources may be accessed in the future, at arbitrary program points in any thread in the program. For example, in the analysis of the put-get buffer shown in figure 4.5, at the end of the second CCR there is no way of knowing whether the cell should go to the shared state or be kept in the local state, until one considers the point at which the cell is actually accessed when the thread attempts to dispose it.

Since ownership is a global property of the program, our approach *delays* ownership decisions until heap accesses are actually made. We do this by associating an *ownership constraint* with each spatial conjunct, which represents the condition under which ownership of the conjunct may be assumed. An ownership constraint is a relation that relates *labels* to *owners*, where a label represents a spatial conjunct and an owner is a resource name or a thread identifier. So at the end of a CCR, for example, we do not immediately decide for each spatial conjunct whether we should send it to the resource invariant or keep in the thread local state. Instead, we create a fresh label for the conjunct, and then send the conjunct to the invariant, but add a constraint relating its label to the resource, and also keep the conjunct in the local state, but with a constraint relating its label to the thread ID. But a label can only have a single owner, and it is when heap accesses are actually made that the ownership questions are resolved: when a heap cell is accessed, it is discovered that the ownership constraints it carries must be satisfied for the program to execute safely.

We illustrate the intuition behind our approach with the put-get buffer example on which the reachability method fails. The analysis is shown in figure 4.6. The user only provides the pre-condition of the whole program, which is $\{\mathtt{emp}\}$ in this case. In this example, since the post-condition of the initialization phase is also empty, each of the thread pre-conditions is empty and the initial resource invariant is also empty with f false. We will discuss examples with non-empty pre-conditions later on.

In the first thread, the cell that is allocated at x is given an empty ownership constraint as there is no question about the thread having ownership of this cell. When we enter the CCR, as before we add the initial approximation of the invariant. At the end of the CCR we come to the ownership decision, where it is not known whether the cell should go to the resource invariant or stay in the local state, and so we introduce the label l_1 to represent this choice. The cell is then sent as a new disjunct for the resource invariant with a new ownership constraint relating l_1 to the resource r,

```
{emp}
new($x$);
{$\langle x \mapsto \mathtt{nil}\rangle_{\emptyset}$}
with $r$ when f = 0 do {
  {$f = 0 \wedge \langle x \mapsto \mathtt{nil}\rangle_{\emptyset} * (f = 0 \wedge \mathtt{emp})$}
  $c := x$;
  $f := 1$;
  {$f = 1 \wedge c = x \wedge \langle x \mapsto \mathtt{nil}\rangle_{\emptyset}$}
} //new RI disjunct {$f = 1 \wedge \langle c \mapsto \mathtt{nil}\rangle_{\{(l_1, r)\}}$}
{$\langle x \mapsto \mathtt{nil}\rangle_{\{(l_1, 1)\}}$}
```

||

```
{emp}
with $r$ when f = 1 do {
  {$f = 1 \wedge \mathtt{emp} * (f = 1 \wedge \langle c \mapsto \mathtt{nil}\rangle_{\{(l_1, r)\}})$}
  $y := c$;
  $f := 0$;
  {$f = 0 \wedge y = c \wedge \langle c \mapsto \mathtt{nil}\rangle_{\{(l_1, r)\}}$}
} //new RI disjunct {$f = 0 \wedge \langle c \mapsto \mathtt{nil}\rangle_{\{(l_1, r), (l_2, r)\}}$}
{$\langle y \mapsto \mathtt{nil}\rangle_{\{(l_1, r), (l_2, 2)\}}$}
{$\langle y \mapsto \mathtt{nil}\rangle_{\emptyset}$} //discover constraint $\{(l_1, r), (l_2, 2)\}$
dispose($y$);
{emp}
```

Figure 4.6: Analysis of put-get buffer using label-tracking

and the cell is also kept in the local state with a constraint relating l_1 to the thread identifier 1. The question of which owner l_1 actually relates to is resolved when we analyse the second thread.

When we enter the CCR in the second thread we add the new disjunct $\{f = 1 \wedge \langle c \mapsto \mathtt{nil}\rangle_{\{(l_1, r)\}}\}$, and when we reach the end of the CCR, we have another ownership decision. As before we send the cell to both the invariant and the local state, but add new constraints with fresh label l_2. Now when we come outside the CCR, we encounter the dispose command, which is making a heap access. At this point, in order to get concrete ownership of the cell, the constraint associated with the cell must hold. So we infer that the constraint $\{(l_1, r), (l_2, 2)\}$ relating l_1 to r and l_2 to thread 2 is valid, which gives concrete ownership of the cell. Having discovered this constraint, we also apply it to refine our existing resource invariant approximation. Thus the disjunct $\{f = 1 \wedge \langle c \mapsto \mathtt{nil}\rangle_{\{(l_1, r)\}}\}$ refines to $\{f = 1 \wedge \langle c \mapsto \mathtt{nil}\rangle_{\emptyset}\}$ since we have discovered that l_1 does relate to r. In the case of the other disjunct $\{f = 0 \wedge \langle c \mapsto \mathtt{nil}\rangle_{\{(l_1, r), (l_2, r)\}}\}$, the constraint associated with the spatial conjunct relates l_2 to r. Since the discovered constraint relates l_2 to thread 2, this disjunct cannot have ownership of the cell, and so the disjunct refines to $\{f = 0 \wedge \mathtt{emp}\}$. In this way, the ownership constraint we discover at the heap access gives us the correct resource invariant for the program.

4.3 Formulae with Ownership constraints

In this section we describe the formulae that are used in the ownership inference algorithm. The formulae are based on symbolic heaps extended with a notion of labelling on the spatial conjuncts, which is similar to the labelling method used in the previous chapter to detect dependences. However, instead of recording the labels of accessing commands, in this case the spatial conjuncts record *ownership constraints* that describe conditional ownership of parts of the formula. The algorithm marks spatial conjuncts with such constraints in order to track ownership of heap state between threads and resources.

We use the term *owner* to refer in general to either a thread identifier or a resource name: in the context of CSL, an owner is anything that can have ownership of some heap state at some point in program execution.

Definition 4.1 (Ownership constraint) *The set of owners, ranged over by $\omega, \omega', \ldots$, is defined as $\Omega \overset{def}{=} \mathtt{Res} \cup \{1, \ldots, N\}$, where N is the number of threads in the parallel composition. Let $\mathtt{Lab}$ be an infinite set of labels ranged over by $l, l', \ldots$. An ownership constraint $L \in \mathcal{P}(\mathtt{Lab} \times \Omega)$ is a relation relating labels with owners.*

An ownership constraint is said to be consistent when it does not associate more than one owner with any label.

Definition 4.2 (Consistent constraint) *An ownership constraint L is consistent, which is written $consistent(L)$, if it is not that case that $(l, \omega) \in L$ and $(l, \omega') \in L$ for $\omega' \neq \omega$ and $l \in \mathtt{Lab}$.*

The formulae we use are from the standard symbolic heap fragment defined in figure 3.1 in the last chapter. However, since we are assuming a single field for heap cells, the only simple spatial formulae we consider here are:

$$S ::= E \mapsto F \mid \mathtt{ls}(E, F)$$

where $E \mapsto F$ describes the heap cell at E in which the field has value F, and $\mathtt{ls}(E, F)$ is a linked list from E to F. We extend these formulae by associating an ownership constraint with every simple spatial conjunct, as follows:

$$\begin{array}{lll} \Sigma ::= & \mathtt{emp} \mid \langle S \rangle_L \mid \Sigma * \Sigma & \text{labelled spatial formulae} \\ H ::= & \Pi \wedge \Sigma & \text{labelled symbolic heaps} \end{array}$$

We may sometimes write $\langle S \rangle_L \in H$ to mean that $\langle S \rangle_L$ is a conjunct in H. We let $\mathtt{Var}(H)$ be the set of all program variables in H and let $\mathtt{LSH}$ be the set of all labelled symbolic heaps. A general formula $P \in \mathtt{LSH}$ is a set of labelled symbolic heaps.

The ownership constraint for every simple spatial conjunct in a labelled symbolic heap describes the condition under which ownership of the heap state described by the conjunct can be assumed. For example, in the labelled symbolic heap $\Sigma * \langle S \rangle_L$, the heap described by S is only owned if the constraint L is satisfied, and otherwise the formula is equivalent to Σ. We refer to a formula as concrete if we have unconditional ownership over all parts of the formula.

Definition 4.3 (Concrete formula) *A labelled symbolic heap H is* concrete, *written $isConc(H)$, if every spatial conjunct in H has an empty ownership constraint. We define $conc(H)$ as the concrete portion of H, which is the heap H without any spatial conjuncts that have non-empty ownership constraints. Similarly, for a formula P we define $conc(P) \overset{def}{=} \{conc(H) \mid H \in P\}$ and $isConc(P)$ if every $H \in P$ is concrete.*

Because ownership constraints introduce conditional information about ownership, a formula with ownership constraints is interpreted with respect to a specific valuation of the labels in its constraints. This is formalised by the $\mathsf{Upd}(P, L)$ function, which resolves the constraints in P with respect to the valuation given by the constraint $L \in \mathcal{P}(\mathtt{Lab} \times \Omega)$. The update function does two things. First, it removes any conjuncts in P whose ownership constraint is inconsistent with L. For the remaining conjuncts, which all have ownership constraints consistent with L, the labels that are in common with L are removed from the conjunct, as these labels are now redundant.

Definition 4.4 (Upd function) *Assume we are given a labelled symbolic heap H and a label constraint L. Let $H = \Pi \wedge \langle S_1 \rangle_{L_1} * \cdots * \langle S_n \rangle_{L_n} * \Sigma$, where Σ contains all the $\langle S \rangle_{L'} \in H$ such that $\neg consistent(L' \cup L)$. We then have*

$$\mathsf{Upd}(H, L) \overset{def}{=} \Pi \wedge \langle S_1 \rangle_{L'_1} * \cdots * \langle S_n \rangle_{L'_n}$$

where $L'_i = L_i \setminus L$. For a formula P, we define $\mathsf{Upd}(P, L) \overset{def}{=} \{\mathsf{Upd}(H, L) \mid H \in P\}$.

For example, a formula $P = \{\langle x \mapsto \mathtt{nil} \rangle_{\{(l_1,2)\}}\}$ can be updated to the following concrete formulae depending on the valuation of labels:

$$\mathsf{Upd}(P, \{(l_1, r)\}) = \{\mathtt{emp}\}$$

$$\mathsf{Upd}(P, \{(l_1, 2)\}) = \{\langle x \mapsto \mathtt{nil} \rangle_{\emptyset}\}$$

Algorithm 3 InvSynth(Prg, P_{in})

```
1: $P_{cmp} := \mathsf{SeqExec}(init, P_{in})$ ;
2: $\phi := \mathsf{Split}(P_{cmp}, \Omega)$ ;
3: $Pre := \phi|_{\{1 \ldots N\}}$ ;
4: $I_0 := \phi|_{\texttt{Res}}$ ;
5: $\mathcal{I} := \{(I_0, \emptyset)\}$ ;
6: while true do
7:    for $i = 1$ to $N$ do
8:       $T := \{(\mathsf{Upd}(Pre(i), L), I, L) \mid (I, L) \in \mathcal{I}\}$ ;
9:       $\mathcal{I} := \{(I, L) \mid (P, I, L) \in \mathsf{Exec}(i, C_i, T)\}$ ;
10:   end for
11:   if $\mathcal{I} = \emptyset$ then
12:      return failure ;
13:   end if
14:   $\mathcal{I} := \{(\mathsf{abs}(I), L) \mid (I, L) \in T\}$ ;
15:   for all $(I, L) \in T$ do
16:      $I_c := \{H \in I \mid isConc(H)\}$ ;
17:      if $\mathsf{Test}(Prg, P_{in}, I_c)$ then
18:         return $I_c$ ;
19:      end if
20:   end for
21: end while
```

A formula with non-empty ownership constraints therefore represents a collection of possible concrete formulae depending on the specific valuation of labels that we use to interpret it. Concrete formulae have the standard interpretation of unlabelled formulae given in figure 3.2 in the last chapter.

4.4 Invariant Synthesis

We now describe the resource invariant synthesis procedure, which is shown in Algorithm 3. The InvSynth procedure takes as arguments the program Prg (consisting of the initialisation code, resource declaration and parallel composition), and a concrete formula P_{in} which is the given pre-condition of the whole program. If successful, the procedure returns a resource invariant for every resource in the program, such that a proof of the program exists in CSL using these resource invariants. The resource invariants are returned as a function $I : \texttt{Res} \to \mathcal{P}(\texttt{LSH})$, which maps every resource name used in the program to a concrete formula.

We now explain how the InvSynth procedure works, using the running example of the put-get buffer program from figure 4.4, which we discussed informally in section 4.2. Given this program and the overall pre-condition $\{\texttt{emp}\}$, the InvSynth procedure returns the correct resource invariant

$\{(f = 0 \wedge \texttt{emp}) \vee (f = 1 \wedge c \mapsto \texttt{nil})\}$ for the resource r. The $\mathsf{InvSynth}$ procedure can be described in three phases: initialisation, thread execution, and testing of invariants.

4.4.1 Initialisation

In the first line in Algorithm 3, the $\mathsf{SeqExec}$ function performs a standard symbolic execution of the initialisation code on the given pre-condition to get the post-condition P_{cmp}. For our running example we have $P_{cmp} = \{f = 0 \wedge \texttt{emp}\}$.

At this point in a CSL proof, the formula P_{cmp} is spatially separated as a pre-condition for each of the threads and the resource invariant, in accordance with the rule for complete programs. But since it is not known how this splitting should be done, the algorithm gives all possible owners as much resource as possible, but introduces new ownership constraints. The constraints use fresh labels such that any possible valuation of the new labels to the owners represents a specific choice of ownership distribution. These constraints will be resolved later on when the heap is accessed by the right owners. The algorithm calls the $\mathsf{Split}(P_{cmp}, \Omega)$ function to determine the formula assigned to each owner at line 2, which returns a mapping $\phi : \Omega \to \mathcal{P}(\texttt{LSH})$ that maps each owner to its assigned formula.

For a given formula and a set of owners, the Split function first creates a fresh label for every spatial conjunct. It then creates a copy of the formula for every owner, but adds ownership constraints associating the new labels to that owner. The part of the formula that is reachable from the owner's variables is then returned as the heap assigned to that owner, with variables not belonging to the owner becoming existential (as required by the variable conditions of CSL). In the following definition, for a symbolic heap H and a set of variables V, we write $restrict(H, V)$ for the heap H in which every variable not in V is replaced by a fresh primed variable. We also write $nonjunk(H)$ to be the part of H that is reachable from some program variable.

Definition 4.5 (Split **function)** *Assume we are given a set of owners $O \subseteq \Omega$ and a labelled symbolic heap $H = \Pi \wedge \langle S_1 \rangle_{L_1} * \cdots * \langle S_n \rangle_{L_n}$. We assume fresh labels $l_1, \ldots, l_n$. For each $\omega \in O$, let*

$$H_\omega = \Pi \wedge \langle S_1 \rangle_{L_1 \cup \{(l_1, \omega)\}} * \cdots * \langle S_n \rangle_{L_n \cup \{(l_n, \omega)\}}$$

If $\omega \in \{1, \ldots, N\}$ then let $\texttt{Var}(\omega) = \texttt{Var} \setminus \big(\texttt{Var}(\texttt{Res}) \cup \bigcup_{i \neq \omega} \texttt{MV}(C_i)\big)$. The function $\mathsf{Split}(H, O)$:

$O \rightarrow \texttt{LSH}$ is defined as

$$\mathsf{Split}(H,O)(\omega) = \mathit{nonjunk}(\mathit{restrict}(H_\omega, \texttt{Var}(\omega)))$$

For a formula P, the function $\mathsf{Split}(P,O) : O \rightarrow \mathcal{P}(\texttt{LSH})$ is defined as

$$\mathsf{Split}(P,O)(\omega) = \{\mathsf{Split}(H,O)(\omega) \mid H \in P\}$$

In our running example, the formula P_{cmp} is spatially empty, so $\mathsf{Split}(P_{cmp}, \{r,1,2\}) = \phi$, where

$$\begin{array}{rcl} \phi(r) & = & \{f = 0 \wedge \texttt{emp}\} \\ \phi(1) & = & \{f_1' = 0 \wedge \texttt{emp}\} \;=\; \{\texttt{emp}\} \\ \phi(2) & = & \{f_2' = 0 \wedge \texttt{emp}\} \;=\; \{\texttt{emp}\} \end{array}$$

where the resource variable f becomes existential in the thread pre-conditions. As another example, if the pre-condition of the program was given as $\{c = x \wedge \langle x \mapsto \texttt{nil}\rangle_\emptyset\}$ then the Split function would give us

$$\begin{array}{rcl} \phi(r) & = & \{f = 0 \wedge c = x_1' \wedge \langle c \mapsto \texttt{nil}\rangle_{\{(l_1,r)\}}\} \;=\; \{f = 0 \wedge \langle c \mapsto \texttt{nil}\rangle_{\{(l_1,r)\}}\} \\ \phi(1) & = & \{f_1' = 0 \wedge c_1' = x \wedge \langle x \mapsto \texttt{nil}\rangle_{\{(l_1,1)\}}\} \;=\; \{\langle x \mapsto \texttt{nil}\rangle_{\{(l_1,1)\}}\} \\ \phi(2) & = & \{f_2' = 0 \wedge c_2' = x \wedge \langle x \mapsto \texttt{nil}\rangle_{\{(l_1,2)\}}\} \;=\; \{\langle x \mapsto \texttt{nil}\rangle_{\{(l_1,1)\}}\} \end{array}$$

where the resource variables become existential in the thread pre-conditions and non-resource variable x becomes existential in the resource invariant.

At lines 3 and 4 in the algorithm, the mapping of formulae returned by the Split function is separated into the thread pre-conditions $\mathit{Pre} : \{1 \ldots N\} \rightarrow \mathcal{P}(\texttt{LSH})$, and the initial approximation of the resource invariants $I_0 : \texttt{Res} \rightarrow \mathcal{P}(\texttt{LSH})$. In our example this gives $\mathit{Pre}(1) = \mathit{Pre}(2) = \{\texttt{emp}\}$ and $I_0(r) = \{f = 0 \wedge \texttt{emp}\}$.

Next, at line 5 the algorithm initializes the candidate resource invariants and ownership constraints that will be discovered by the algorithm. The set $\mathcal{I}$ is a set of tuples of the form (I, L), where $I : \texttt{Res} \rightarrow \mathcal{P}(\texttt{LSH})$ is a possible candidate for the resource invariant, and $L \in \mathcal{P}(\texttt{Lab} \times \Omega)$ contains all the ownership constraints that have been discovered by the algorithm. The reason that we maintain a *set* of tuples is that the algorithm may sometimes encounter a choice on how to make an ownership inference, which leads to different candidate resource invariants and associated ownership constraints, as we shall discuss in examples. The set $\mathcal{I}$ is initialised at line 5 in the algorithm with the single tuple in which the resource invariant candidate is I_0 and the ownership constraint is empty, as nothing has been discovered yet.

4.4.2 Thread execution

We enter the main loop of the algorithm at line 6. At every iteration, the first thing is to refine the existing resource invariant candidates in the set $\mathcal{I}$, by executing each thread on its pre-condition using the existing invariants. This is done in the for loop (line 7 to 10). For each thread i, before executing the thread, the thread pre-condition is first updated with respect to each of the candidate ownership constraints that have been determined so far (line 8). In our running example, each thread has the empty pre-condition, so it does not change with the update. The set of triples T is the input to the thread execution function Exec, which is called at line 9.

The Exec function symbolically executes a thread in order to discover new ownership constraints and to improve the resource invariant approximation. The function takes as parameters a thread identifier i where $1 \leq i \leq N$, the thread body C_i, and a set of triples of the form (P, I, L). The formula $P \in \mathcal{P}(\texttt{LSH})$ is the pre-condition of the thread, $I : \texttt{Res} \rightarrow \mathcal{P}(\texttt{LSH})$ is the current approximation of the resource invariants, and $L \in \mathcal{P}(\texttt{Lab} \times \Omega)$ is the accumulation of discovered ownership constraints. We let $\mathcal{T}$ be the set of all triples of the form (P, I, L).

The function returns a set of triples from $\mathcal{T}$, the non-determinism in the output of the function being due to possible branching on ownership decisions. The definition of the Exec function is given in figure 4.7, and we now discuss each of the cases shown in the figure.

Primitive commands The first two cases in figure 4.7 are for primitive commands, which use the application and rearrangement functions App and Rng for transforming symbolic heaps. These functions are defined using the symbolic execution rules for command application and rearrangement shown in figure 4.8, which define the symbolic heap transformations and the propagation of label constraints. The symbolic heap transformations are standard, as in the previous chapter. The label constraints are propagated such that under any valuation of labels to concretize the symbolic heaps, each rule is a sound inference from premise to conclusion.

$$\mathsf{Exec}(i, c, \{(P, I, L)\}) \overset{def}{=} \Big\{(\mathsf{App}(c, P), I, L)\Big\}$$

$$\mathsf{Exec}(i, c[E], \{(P, I, L)\}) \overset{def}{=}$$
$$\Big\{(\mathsf{App}(c[E], \mathsf{Upd}(P', L')), \mathsf{Upd}(I, L'), L \cup L') \;\mid\; P' = \mathsf{Rng}(P, E),\; L' \in \mathsf{GetCons}(P', E)\Big\}$$

$$\mathsf{Exec}(i, C1; C2, \{(P, I, L)\}) \overset{def}{=} \mathsf{Exec}(i, C2, \mathsf{Exec}(i, C1, (P, I, L)))$$

$$\mathsf{Exec}(i, C1 + C2, \{(P, I, L)\}) \overset{def}{=}$$
$$\Big\{((\mathsf{Upd}(P', L'') \cup P''), I'', L'') \;\mid\; (P', I', L') \in \mathsf{Exec}(i, C1, (P, I, L)),\; (P'', I'', L'') \in \mathsf{Exec}(i, C2, (\mathsf{Upd}(P, L'), I', L'))\Big\}$$

$$\mathsf{Exec}(i, \mathtt{with}\ r\ \mathsf{when}\ B\ \mathsf{do}\ C, \{(P, I, L)\}) \overset{def}{=}$$
$$\Big\{(\phi(i), I'[r \mapsto I'(r) \cup \phi(r)], L') \;\mid\; (P', I', L') \in \mathsf{Exec}(i, C, (B \wedge P * I(r), I, L)),\; \phi = \mathsf{Split}(P', \{i, r\})\Big\}$$

$$\mathsf{Exec}(i, C, T_1 \cup T_2) = \mathsf{Exec}(i, C, T_1 \cup T_2) \cup \mathsf{Exec}(i, C, T_1 \cup T_2)$$

Figure 4.7: Thread execution function

Definition 4.6 (Application and Rearrangement Functions) *For a non-heap-accessing primitive command c and formula P, the function $\mathsf{App}(c, P)$ applies the command application rule for command c from figure 4.8 to every symbolic heap $H \in P$ that is consistent* [1] *and returns the resulting set of symbolic heaps that are consistent.*

For a heap-accessing command $c[E]$, the function $\mathsf{App}(c[E], P)$ is undefined if there exists $H \in P$ such that there is no $\langle E \mapsto F \rangle_{\emptyset} \in H$. Otherwise, $\mathsf{App}(c[E], P)$ applies the command application rule for $c[E]$ to all consistent symbolic heaps $H \in P$, and returns the resulting set of heaps that are consistent.

The rearrangement function $\mathsf{Rng}(P, E)$ applies the rearrangement rules from figure 4.8 to every symbolic heap $H \in P$, to all conjuncts in H to which the rules apply, and returns the resulting set of heaps.

The first case in figure 4.7 is that of non-heap accessing command primitive command c. In this case the thread state is simply transformed according to the command application rule, and everything else stays the same. The second case is that of a primitive command $c[E]$ accessing the heap cell at E. This is the point at which ownership decisions are made, because the heap is actually accessed. In this case, every symbolic heap in the pre-conditon P needs to be brought

[1] a symbolic heap is inconsistent if its concrete portion is not satisfiable and equivalent to false (e.g. it contains $E = F \wedge E \neq F$, or it contains $\langle E \mapsto F \rangle_{\emptyset} * \langle E \mapsto G \rangle_{\emptyset}$). Inconsistent heaps do not represent concrete state and are therefore ignored. Consistency of concrete heaps can be checked using methods such as the one described in [17].

COMMAND APPLICATION RULES

$$\frac{\Pi \wedge \Sigma}{x = E[x'/x] \wedge (\Pi \wedge \Sigma)[x'/x]}\ x := E, x' \mathit{fresh} \qquad \frac{\Pi \wedge \Sigma * \langle E \mapsto F' \rangle_\emptyset}{\Pi \wedge \Sigma * \langle E \mapsto F \rangle_\emptyset}\ [E] := F$$

$$\frac{\Pi \wedge \Sigma}{(\Pi \wedge \Sigma)[x'/x] * \langle x \mapsto \mathtt{nil} \rangle_\emptyset}\ \mathtt{new}(x), x'\ \mathit{fresh} \qquad \frac{\Pi \wedge \Sigma * \langle E \mapsto F \rangle_\emptyset}{\Pi \wedge \Sigma}\ \mathtt{dispose}(E)$$

$$\frac{\Pi \wedge \Sigma * \langle E \mapsto F \rangle_\emptyset}{x = F[x'/x] \wedge (\Pi \wedge \Sigma * \langle E \mapsto F \rangle_\emptyset)[x'/x]}\ x := [E], x'\ \mathit{fresh} \qquad \frac{\Pi \wedge \Sigma}{B \wedge \Pi \wedge \Sigma}\ \mathtt{assume}(B)$$

REARRANGEMENT RULES

$$\frac{\Pi \wedge \Sigma * \langle F \mapsto F' \rangle_L}{\Pi \wedge \Sigma * \langle E \mapsto F' \rangle_L}\ \Pi \vdash E = F \qquad \frac{\Pi \wedge \Sigma * \langle \mathtt{ls}(F, F') \rangle_L}{\Pi \wedge \Sigma * \langle E \mapsto x' \rangle_L * \langle \mathtt{ls}(x', F') \rangle_L}\ \Pi \vdash F \neq F' \wedge E = F \text{ and } x' \text{ fresh}$$

Figure 4.8: Command Application and Rearrangement Rules

into a form where it contains concrete ownership of the heap cell at E: that is, it contains a spatial conjunct of the form $\langle E \mapsto F \rangle_\emptyset$ so that the appropriate command application rule from figure 4.8 can be applied.

This is done by first applying the rearrangement function to get $P' = \mathsf{Rng}(P, E)$. Then the function $\mathsf{GetCons}(P', E)$ returns the set of all possible label constraints under which every heap in P' contains concrete ownership of a heap cell at E. Each element of this set of constraints represents a different choice of ownership. We will discuss a program where such choice is encountered in example 9.

Definition 4.7 (GetCons**)** *Let E be an expression, H a symbolic heap, and $P' = \{H_1, \ldots, H_n\}$ a formula with n symbolic heaps. We define*

$$\mathsf{GetCons}(H, E) \overset{\mathit{def}}{=} \{L \mid \langle E \mapsto F \rangle_L \in H\}$$

$$\mathsf{GetCons}(P', E) \overset{\mathit{def}}{=} \{L_1 \cup \cdots \cup L_n \ \mid L_i \in \mathsf{GetCons}(H_i, E),\ \mathit{consistent}(L_1 \cup \cdots \cup L_n)\}$$

Note that if there is a symbolic heap in P' that does not contain a conjunct of the form $\langle E \mapsto F \rangle_L$, then $\mathsf{GetCons}(P', E) = \emptyset$, and hence $\mathsf{Exec}(i, c[E], \{(P, I, L)\}) = \emptyset$. This means that I is eliminated as a possible resource invariant for the program because execution could not proceed further.

Sequential and choice composition The next two cases in figure 4.7 are for sequential composition and non-deterministic choice, which are handled in the standard way. Sequential composition applies the second command to the output of the first command. For non-deterministic choice, both commands are executed on the pre-condition and the post-conditions are disjunctively combined, but the resource invariants and ownership constraints refined by the execution of the first command are used in the execution of the second. In Example 9 we describe the analysis of a program with non-deterministic choice.

Critical regions The final case in figure 4.7 is that of the conditional critical region command $\texttt{with}\, r\, \texttt{when}\, B\, \texttt{do}\, C$, which is where new questions about ownership arise. Given the input state (P, I, L), the body C of the CCR is first executed with pre-condition $B \wedge P * I(r)$, in accordance with the CSL rule for critical regions. For every output state (P', I', L') that this returns, we need to split the post-condition P' into the part that goes into the resource invariant and the part that stays in the thread local state outside the CCR. This is done by calling the Split function with two possible owners: the resource r and the thread identifier i. The part going to the resource invariant is added as a new disjunct to the resource invariant, and the part going to the thread is the post-condition of the CCR execution.

We now illustrate the thread execution function on our running example of the put-get buffer. Recall from the end of section 4.4.1 that the thread pre-conditions are $\mathit{Pre}(1) = \mathit{Pre}(2) = \{\texttt{emp}\}$ and the initial resource invariant is $I_0(r) = \{f = 0 \wedge \texttt{emp}\}$. The execution of the two threads in the first iteration of the algorithm is shown in figure 4.9. In the left thread, the command application rule is first applied to allocate the new cell. The CCR is then executed, which again involves applying the command application rules for the two commands. At the end of the CCR a new disjunct for the resource invariant is obtained using the Split function, which introduces fresh label l_1. Execution of the right thread starts with the CCR, in which the command application rules are applied for the two commands in the body. At the end of the CCR, another resource invariant disjunct is obtained with new label l_2. At this point, the heap accessing dispose command is executed, in which the $\mathsf{GetCons}$ function returns the set with single constraint $\{(l_1, r), (l_2, 2)\}$. Applying this constraint gives concrete ownership of the heap cell, which allows the command application rule for dispose to be applied. The resource invariant is also updated with respect to the discovered constraint, as defined by the execution of primitive heap-accessing commands.

```
$\{(\{\texttt{emp}\}, I_0, \emptyset)\}$
new($x$);
$\{(\{\langle x \mapsto \texttt{nil}\rangle_\emptyset\}, I_0, \emptyset)\}$
with $r$ when f = 0 do {
  $\{(\{f = 0 \wedge \langle x \mapsto \texttt{nil}\rangle_\emptyset * I_0\}, I_0, \emptyset)\}$
  $c := x; f := 1;$
  $\{(\{f = 1 \wedge c = x \wedge \langle x \mapsto \texttt{nil}\rangle_\emptyset\}, I_0, \emptyset)\}$
}
$\{(\{\langle x \mapsto \texttt{nil}\rangle_{\{(l_1,1)\}}\}, I_1, \emptyset)\}$
```

```
$\{(\{\texttt{emp}\}, I_1, \emptyset)\}$
with $r$ when f = 1 do {
  $\{(\{f = 1 \wedge \texttt{emp} * I_1\}, I_1, \emptyset)\}$
  $\{(\{f = 1 \wedge \langle c \mapsto \texttt{nil}\rangle_{\{(l_1,r)\}}\}, I_1, \emptyset)\}$
  $y := c; f := 0;$
  $\{(\{f = 0 \wedge y = c \wedge \langle c \mapsto \texttt{nil}\rangle_{\{(l_1,r)\}}\}, I_1, \emptyset)\}$
}
$\{(\{\langle y \mapsto \texttt{nil}\rangle_{\{(l_1,r),(l_2,2)\}}\}, I_2, \emptyset)\}$
dispose($y$);
$\{(\emptyset,\ I_3,\ \{(l_1,r),(l_2,2)\})\}$
```

$$I_1(r) = I_0(r) \cup \{f = 1 \wedge \langle c \mapsto \texttt{nil}\rangle_{\{(l_1,r)\}}\}$$
$$I_2(r) = I_1(r) \cup \{f = 0 \wedge \langle c \mapsto \texttt{nil}\rangle_{\{(l_1,r),(l_2,r)\}}\}$$
$$I_3(r) = \mathsf{Upd}(I_2(r),\ \{(l_1,r),(l_2,2)\}) = \{f = 0 \wedge \texttt{emp},\ f = 1 \wedge \langle c \mapsto \texttt{nil}\rangle_\emptyset\}$$

Figure 4.9: Thread executions for put-get buffer

4.4.3 Testing of invariants

After executing all of the threads, at line 11 the algorithm tests if the set $\mathcal{I}$ of candidate resource invariants is empty. This set can be empty because, as we discussed in the previous section, the Exec function eliminates candidate resource invariants if execution of heap-accessing commands cannot proceed. If $\mathcal{I}$ is empty then there are no more resource invariant candidates left, and so the algorithm fails at this point.

At line 14, the algorithm applies some standard abstraction heuristics to the resource invariant candidates in order to generalize the formulae to help reach an invariant. This is done with the $\mathsf{abs}(I)$ function, which applies the abstraction heuristics for symbolic heaps presented in [17] to the concrete parts of all the formulae in I. These heuristics make generalizations such as removing existential variables from the middle of list segments: for example, a formula $\langle \texttt{ls}(x, x')\rangle_\emptyset * \langle \texttt{ls}(x', y)\rangle_\emptyset$ will be abstracted to $\langle \texttt{ls}(x, y)\rangle_\emptyset$. We give an example of where this kind of generalization is required in Example 10 in the next section.

The for loop at line 15 tests each of the candidates to see if enough ownership decisions have been resolved that a resource invariant has been found. The formula I_c at line 16 consists only of the concrete disjuncts in I. At line 17, the algorithm checks if these concrete formulae provide a resource invariant with which a CSL proof can be constructed. The Test function is a standard CSL specification checker, such as the one described in [2], which takes a concurrent program, a concrete pre-condition and a concrete resource invariant, and checks if a CSL proof of the program can be constructed with the given resource invariant. If the test succeeds on any of the candidates, then the algorithm returns the resource invariant and we are done. If the test does not succeed on any of the candidates, then we exit the for loop and go to the next iteration to further refine the candidate resource invariants. Since the algorithm constructs a CSL proof using the synthesized resource invariants, soundness follows from the soundness of CSL.

In the case of our running example of the put-get buffer, at the end of the thread executions in the first iteration (shown in figure 4.9), we have the single candidate resource invariant I_3 such that

$$I_3(r) = \{f = 0 \wedge \texttt{emp},\ f = 1 \wedge \langle c \mapsto \texttt{nil} \rangle_{\emptyset}\}$$

We therefore have $\mathcal{I} = \{I_3\}$, and since both formulae are concrete, we have $I_c = I_3$ at line 16. This is a valid resource invariant for the put-get buffer program, and so our algorithm successfully completes in one iteration.

4.5 Examples and Comparison with Previous Methods

In this section we discuss some more examples to illustrate different aspects of the invariant synthesis algorithm. For our first example, we discuss the phenomenon which O'Hearn describes as "ownership is in the eye of the asserter". Consider the program in figure 4.10, which is an address-transferring put-get buffer, as opposed to the cell-transferring buffer from figure 4.4. We briefly discussed the contrast between these two programs in section 1.2.3 in the introduction. While the initialisation phase and the *put* and *get* CCRs are exactly the same in both cases, the difference is that in the address-transferring case, the left thread keeps ownership of the heap cell rather than transferring it into the buffer. This is because the left thread disposes the cell outside the CCR and, in agreement with this ownership policy, the right thread does not access the cell and only reads its address. The ownership policy for the address-transferring buffer can be formalised with a different choice of resource invariant, which is the formula $(f = 1 \wedge \texttt{emp}) \vee (f = 0 \wedge \texttt{emp})$. O'Hearn refers to this as ownership being in the "eye of the asserter" [42], since one can verify the two programs with a different choice of resource invariant. However, our algorithm illustrates how this choice can actually be inferred from how threads access the heap, as shown in the next example.

```
                    c := nil;
                    f := 0;
                 resource r(c, f)

new(x);                       ||
with r when f = 0 do {        ||  with r when f = 1 do {
   c := x; f := 1;            ||     y := c; f := 0;
}                             ||  }
dispose(x);                   ||
```

Figure 4.10: Address transferring put-get buffer

Example 8 (Address transferring put-get buffer) *For the program in figure 4.10, we are given the overall pre-condition $\{\texttt{emp}\}$. The initialisation phase is identical to the cell-transferring buffer described in section 4.4, and so we have $Pre(1) = Pre(2) = \{\texttt{emp}\}$ and $I_0(r) = \{f=0 \wedge \texttt{emp}\}$. The first iteration proceeds as follows:*

```
({emp}, I_0, ∅)                                   ||
new(x);                                           ||  ({emp}, I_2, L)
({⟨x ↦ nil⟩_∅}, I_0, ∅)                            ||  with r when f = 1 do {
with r when f = 0 do {                            ||    ({f = 1 ∧ emp * I_2}, I_2, L)
  ({f = 0 ∧ ⟨x ↦ nil⟩_∅ * I_0}, I_0, ∅)            ||    ({f = 1 ∧ emp}, I_2, L)
  c := x; f := 1;                                 ||    y := c; f := 0;
  ({f = 1 ∧ c = x ∧ ⟨x ↦ nil⟩_∅}, I_0, ∅)          ||    ({f = 0 ∧ y = c ∧ emp}, I_2, L)
}                                                 ||  }
({⟨x ↦ nil⟩_{(l_1,1)}}, I_1, ∅)                    ||  ({emp}, I_2, L)
dispose(x);                                       ||
({emp}, I_2, L)                                   ||
```

The new disjunct obtained at the end of the first CCR is added to the invariant to get $I_1(r) = I_0(r) \cup \{f = 1 \wedge \langle c \mapsto \texttt{nil}\rangle_{\{(l_1,r)\}}\}$. When the dispose command is encountered, the constraint $L = \{(l_1, 1)\}$ is discovered, which causes an update of I_1 to get $I_2(r) = \{f = 0 \wedge \texttt{emp},\ f = 1 \wedge \texttt{emp}\}$. In the second thread, the new invariant is added when we enter the CCR, and at the end of the CCR the heap is empty. Hence the disjunct $\{f = 0 \wedge \texttt{emp}\}$ is added to the invariant but it already contains this disjunct and so it remains unchanged. At the end of the iteration, the invariant I_2 is checked to be the valid resource invariant for the program.

We next discuss an example where the algorithm encounters a choice about how to make an ownership inference. The program is shown in figure 4.11, where this time there are two buffers

$$
\begin{array}{c}
c_1 := \texttt{nil};\ c_2 := \texttt{nil};\\
f_1 := 0;\ f_2 := 0;\\
\texttt{resource}\ r_1(c_1, f_1),\ \ r_2(c_2, f_2)\\
\begin{array}{l||l}
 & \texttt{get}_1(y);\\
\texttt{new}(x); & \texttt{get}_2(z);\\
\texttt{put}_1(x); & \texttt{if}\ (z = y)\ \texttt{then}\ \{[z] := w\ ;\}\\
\texttt{put}_2(x); & \texttt{dispose}(y);
\end{array}
\end{array}
$$

Figure 4.11: Two buffer program

represented by resource names r_1 and r_2. For resource r_i, the $\texttt{put}_i$ and $\texttt{get}_i$ commands are the put and get CCRs using cell and flag variables c_i and f_i. The left thread creates a single new cell and puts it into both buffers. But ownership of the cell can only move into one of them, and in this program the buffer r_1 acts as a cell-transferring buffer while r_2 acts as an address-transferring buffer. This can be seen in the right thread, where ownership of the cell comes out from the $\texttt{get}_1(y)$ command because the thread disposes the cell at y in the last line. Hence the $\texttt{get}_2(z)$ command only obtains the address of the cell and not the cell itself. In the following example we show how the algorithm encounters a choice at a certain point about how to make the correct ownership inference, which it then resolves later on.

Example 9 (Ownership choice in two buffer program) *The algorithm analyses the program in figure 4.11 in one iteration. Firstly, note that, as described previously, we implement the conditional in the right thread as the non-deterministic composition* $(\texttt{assume}(z = y);\ [z] := w;) + (\texttt{assume}(z \neq y);\ \texttt{skip};)$. *For the initial stage, the* Split *function gives us* $I_0(r_1) = \{f_1 = 0 \wedge \texttt{emp}\}$ *and* $I_0(r_2) = \{f_2 = 0 \wedge \texttt{emp}\}$ *as the initial resource invariants and* $\{\texttt{emp}\}$ *as precondition of each thread. For the left thread we have the execution:*

$$
\begin{array}{l}
(\{\texttt{emp}\}, I_0, \emptyset)\\
\texttt{new}(x);\\
(\{\langle x \mapsto \texttt{nil}\rangle_{\emptyset}\}, I_0, \emptyset)\\
\texttt{put}_1(x);\\
(\{\langle x \mapsto \texttt{nil}\rangle_{\{(l_1,1)\}}\}, I_1, \emptyset)\\
\texttt{put}_2(x);\\
(\{\langle x \mapsto \texttt{nil}\rangle_{\{(l_1,1),(l_2,1)\}}\}, I_2, \emptyset)
\end{array}
$$

After executing the first CCR we have $I_1(r_1) = I_0(r_1) \cup \{f_1 = 1 \wedge \langle c_1 \mapsto \texttt{nil}\rangle_{\{(l_1,r_1)\}}\}$ *and*

$I_1(r_2) = I_0(r_2)$. After the second CCR we have $I_2(r_1) = I_1(r_1)$ and $I_2(r_2) = \{f_2 = 1 \wedge \langle c_2 \mapsto \mathtt{nil}\rangle_{\{(l_1,1),(l_2,r_2)\}}\}$. Then for the right thread we have the following execution:

$(\{\mathtt{emp}\}, I_2, \emptyset)$

$\mathtt{get}_1(y);$

$(\{\langle y \mapsto \mathtt{nil}\rangle_{\{(l_1,r_1),(l_3,2)\}}\}, I_3, \emptyset)$

$\mathtt{get}_2(z);$

$(\{\langle y \mapsto \mathtt{nil}\rangle_{\{(l_1,r_1),(l_3,2)\}} * \langle z \mapsto \mathtt{nil}\rangle_{\{(l_1,1),(l_2,r_2),(l_4,2)\}}\}, I_4, \emptyset)$

$$\left(\begin{array}{l}(\{\langle y \mapsto \mathtt{nil}\rangle_{\{(l_1,r_1),(l_3,2)\}} * \langle z \mapsto \mathtt{nil}\rangle_{\{(l_1,1),(l_2,r_2),(l_4,2)\}}\}, I_4, \emptyset)\\ \mathtt{assume}(z = y);\\ (\{z = y \wedge \langle y \mapsto \mathtt{nil}\rangle_{\{(l_1,r_1),(l_3,2)\}} * \langle z \mapsto \mathtt{nil}\rangle_{\{(l_1,1),(l_2,r_2),(l_4,2)\}}\}, I_4, \emptyset)\\ [z] := w;\\ (\{z = y \wedge \langle y \mapsto w\rangle_{\emptyset}\}, I_5, L_1) \quad (\{z = y \wedge \langle z \mapsto w\rangle_{\emptyset}\}, I_6, L_2)\end{array}\right)$$

$$+$$

$$\left(\begin{array}{l}(\{\langle y \mapsto \mathtt{nil}\rangle_{\emptyset}\}, I_5, L_1) \quad (\{\langle z \mapsto \mathtt{nil}\rangle_{\emptyset}\}, I_6, L_2)\\ \mathtt{assume}(z \neq y);\\ (\{z \neq y \wedge \langle y \mapsto \mathtt{nil}\rangle_{\emptyset}\}, I_5, L_1) \quad (\{z \neq y \wedge \langle z \mapsto \mathtt{nil}\rangle_{\emptyset}\}, I_6, L_2)\\ \mathtt{skip};\\ (\{z \neq y \wedge \langle y \mapsto \mathtt{nil}\rangle_{\emptyset}\}, I_5, L_1) \quad (\{z \neq y \wedge \langle z \mapsto \mathtt{nil}\rangle_{\emptyset}\}, I_6, L_2)\end{array}\right)$$

$(\{z = y \wedge \langle y \mapsto w\rangle_{\emptyset},\ z \neq y \wedge \langle y \mapsto \mathtt{nil}\rangle_{\emptyset}\}, I_5, L_1) \quad (\{z = y \wedge \langle z \mapsto w\rangle_{\emptyset},\ z \neq y \wedge \langle z \mapsto \mathtt{nil}\rangle_{\emptyset}\}, I_6, L_2)$

$\mathtt{dispose}(y);$

$(\{z = y \wedge \mathtt{emp}, z \neq y \wedge \mathtt{emp}\}, I_5, L_1)$

The first CCR gives $I_3(r_1) = I_2(r_1) \cup \{f_1 = 0 \wedge \langle c \mapsto \mathtt{nil}\rangle_{\{(l_1,r_1),(l_3,r_1)\}}\}$ and $I_3(r_2) = I_2(r_2)$. The next CCR gives $I_4(r_1) = I_3(r_1)$ and $I_4(r_2) = I_3(r_2) \cup \{f_2 = 0 \wedge \langle c \mapsto \mathtt{nil}\rangle_{\{(l_1,1),(l_2,r_2),(l_4,r_2)\}}\}$. We then enter the first part of the non-deterministic composition, and after executing the assume statement we encounter the heap access on the cell at z. Because of the equality $z = y$, after applying the rearrangement function both conjuncts can possibly give the heap cell required to execute the command. We therefore obtain two possible constraints after executing the heap mutation, which are $L_1 = \{(l_1, r_1), (l_3, 2)\}$ and $L_2 = \{(l_1, 1), (l_2, r_2), (l_4, 2)\}$. We branch on these two constraints with the two triples in the output of the Exec *command, one updated with respect to L_1 and the other with respect to L_2.*

We then execute the second part of the non-deterministic composition, where the initial pre-condition of the non-deterministic composition is updated in the left and the right triple with the newly discovered constraints L_1 and L_2 respectively. When we exit the non-deterministic composition, for each triple, the post-condition of the composition is obtained as the union of the post-conditions of the two parts of the composition. Finally, with these two triples we then encounter the $\texttt{dispose}(y)$ *command. At this point we find that the triple with the L_2 constraint cannot provide the heap cell at y in the disjunct where $z \neq y$, and so the L_2 branch fails and disappears. The L_1 branch is able to proceed safely, and this resolves the ownership decision. Constraint L_1 gives us the updated resource invariant $I_5 = \mathsf{Upd}(I_4, L_1)$, where $I_5(r_1) = \{f_1 = 0 \wedge \texttt{emp},\ f_1 = 1 \wedge \langle c_1 \mapsto \texttt{nil} \rangle_{\emptyset}\}$ and $I_5(r_2) = \{f_2 = 0 \wedge \texttt{emp},\ f_2 = 1 \wedge \texttt{emp}\}$, which is the correct resource invariant for the program.*

We also remark that the two-buffer program in figure 4.11 is an example which cannot be analysed by the bi-abduction method of [9]. In this method, resource invariants are built up spatially by inferring missing pieces of state using bi-abduction [8]. For example, in the put-get buffer program in figure 4.4, the resource invariant is initialised to the spatially smallest formula $\{(f = 1 \wedge \texttt{emp}) \vee (f = 0 \wedge \texttt{emp})\}$, and a proof is attempted. The proof fails in the second thread when the dispose command requires ownership of the heap cell. At this point, the last CCR, which is the get CCR, is checked to see if the cell could have come from the resource invariant if the formula had more state. The answer is yes, and the spatially bigger invariant $\{(f = 1 \wedge c \mapsto \texttt{nil}) \vee (f = 0 \wedge \texttt{emp})\}$ is obtained through bi-abduction, which allows the proof to succeed.

However, this method does not address the ownership inference problem in full. Although bi-abduction is useful in finding out what state is required, it does not answer the other part of the question, which is *where* the ownership is coming from. It addresses this with the heuristic of checking the *last* CCR from the point at which the proof fails. This heuristic works in the simple put-get buffer where there is only one previous CCR in the thread, but does not work in general. An example where the method would fail is the two-buffer program in figure 4.11. In this case the method will initialise the resource invariants for r_1 and r_2 to $\{(f_1 = 1 \wedge \texttt{emp}) \vee (f_1 = 0 \wedge \texttt{emp})\}$ and $\{(f_2 = 1 \wedge \texttt{emp}) \vee (f_2 = 0 \wedge \texttt{emp})\}$ respectively. The proof will then fail when the heap access is made in the conditional in the left thread. In this case the last CCR $\texttt{get}_2(z)$ would be able to provide the required heap cell, but the cell should actually come from the $\texttt{get}_1(y)$ CCR before that, because r_1 is the cell-transferring buffer. This way the method generates the incorrect

```
                         c := nil;
                         f := 0;
                       list := nil;
                resource r(c, f),  r'(list)

               alloc(x1);      ||
               alloc(x2);      ||   get(y);
               put(x1);        ||   dealloc(y);
               dealloc(x2);    ||

alloc(x)  ≝  with r' when true do {          dealloc(y)  ≝  with r' when true do {
               if (list = 0) then {new(x);}                     [y] := list;
               else{x := list; list := [x];}                    list := y;
             }                                                }
```

Figure 4.12: Combined buffer and memory manager

resource invariants $\{(f_1 = 1 \wedge \texttt{emp}) \vee (f_1 = 0 \wedge \texttt{emp})\}$ and $\{(f_2 = 1 \wedge c \mapsto \texttt{nil}) \vee (f_2 = 0 \wedge \texttt{emp})\}$ for r_1 and r_2 respectively. In contrast, our method avoids this problem as it uses label-tracking to keep track of where ownership is coming from.

As our final example, we choose a complex program that combines the use of two different resources: the put-get buffer and a memory manager. The example shows how our algorithm determines the resource invariants for both resources, and also shows how the abstraction function helps to reach a valid resource invariant for memory manager. This is also an example where the algorithm requires more than one iteration to reach the correct resource invariants.

The program is shown in figure 4.12. The buffer is represented by resource name r and the memory manager by resource name r'. The memory manager internally maintains a linked list of all the free cells, where the head of the list is maintained in the resource variable $list$. This list is initialised to the empty list in the initialisation phase. The manager has two CCRs: alloc(x) which removes an element from the head of the list or calls the system allocator if the list is empty, and dealloc(y) which moves the cell at y to the head of the free list and back into the resource. The main program has two threads. The left one allocates two cells and puts one in the buffer and disposes the other. The other thread gets the cell from the buffer and then disposes it. For this program the algorithm determines that the resource invariant for the memory manager is a singly linked list at variable $list$, and determines the standard invariant for the buffer.

Example 10 (Combined buffer and memory manager) *We are given the program in figure 4.12 and the precondition* $\{\texttt{emp}\}$. *The* Split *function gives us* $I_0(r) = \{f = 0 \wedge \texttt{emp}\}$ *and* $I_0(r') = \{\mathit{list} = \texttt{nil} \wedge \texttt{emp}\}$ *as the initial resource invariants and* $\{\texttt{emp}\}$ *as pre-condition of each thread. The first iteration proceeds as follows:*

$(\{\texttt{emp}\}, I_0, \emptyset)$	
$\texttt{alloc}(x_1);$	
$(\{\langle x_1 \mapsto \texttt{nil}\rangle_\emptyset\}, I_0, \emptyset)$	$(\{\texttt{emp}\}, I_2, \emptyset)$
$\texttt{alloc}(x_2);$	$\texttt{get}(y);$
$(\{\langle x_1 \mapsto \texttt{nil}\rangle_\emptyset\} * \langle x_2 \mapsto \texttt{nil}\rangle_\emptyset\}, I_0, \emptyset)$	$(\{\langle y \mapsto \texttt{nil}\rangle_{\{(l_1,r),(l_3,2)\}}\}, I_3, \emptyset)$
$\texttt{put}(x_1);$	$\texttt{dealloc}(y);$
$(\{\langle x \mapsto \texttt{nil}\rangle_{\{(l_1,1)\}} * \langle x_2 \mapsto \texttt{nil}\rangle_\emptyset\}, I_1, \emptyset)$	$(\{\langle y \mapsto x'\rangle_{\{(l_4,2)\}} * \langle x' \mapsto \texttt{nil}\rangle_{\{(l_2,r'),(l_5,2)\}}, \langle y \mapsto \texttt{nil}\rangle_{\{(l_6,2)\}}\}, I_4, L_1)$
$\texttt{dealloc}(x_2);$	
$(\{\langle x \mapsto \texttt{nil}\rangle_{\{(l_1,1)\}} * \langle x_2 \mapsto \texttt{nil}\rangle_{\{(l_2,2)\}}\}, I_2, \emptyset)$	

After the first two allocations, there are no new disjuncts for the r' *resource invariant because the system allocator is used. Then the put CCR gives us* $I_1 = I_0[r \rightarrow I_0(r) \cup \{f = 1 \wedge \langle c \mapsto \texttt{nil}\rangle_{\{(l_1,r)\}}\}]$, *and the* $\texttt{dealloc}(y)$ *CCR gives* $I_2 = I_1[r' \rightarrow I_1(r') \cup \{\langle \mathit{list} \mapsto \texttt{nil}\rangle_{\{(l_2,r')\}}\}]$. *In the second thread, the get CCR gives* $I_3 = I_2[r \rightarrow I_2(r) \cup \{f = 0 \wedge \langle c \mapsto \texttt{nil}\rangle_{\{(l_1,r),(l_3,r)\}}\}]$. *Then inside the* $\texttt{dealloc}(y)$ *CCR, the cell at* y *is accessed, and hence we discover the constraint* $L_1 = \{(l_1, r), (l_3, 2)\}$, *which updates the invariant to give us* $I_4(r) = \{f = 0 \wedge \texttt{emp},\ f = 1 \wedge \langle c \mapsto \texttt{nil}\rangle_\emptyset\}$. *The new disjunct added to the invariant for* r' *gives us*

$$I_4(r') = I_3(r') \cup \{\langle \mathit{list} \mapsto x'\rangle_{\{(l_4,r')\}} * \langle x' \mapsto \texttt{nil}\rangle_{\{(l_2,r'),(l_5,r')\}},\ \langle \mathit{list} \mapsto \texttt{nil}\rangle_{\{(l_6,r')\}}\}$$

At the end of this iteration, all the concrete disjuncts for the buffer give the correct resource invariant. However, the concrete disjuncts for the memory manager collected so far do not give a valid resource invariant for the memory manager, and so we go to the next iteration. This starts as follows:

$$(\{\texttt{emp}\}, I_4, L_1)$$

$$\texttt{alloc}(x_1);$$

$$(\{\langle x_1 \mapsto \texttt{nil}\rangle_\emptyset\}, I_5, L_2)$$

$$\texttt{alloc}(x_2);$$

$$(\{\langle x_1 \mapsto \texttt{nil}\rangle_\emptyset\} * \langle x_2 \mapsto \texttt{nil}\rangle_\emptyset\}, I_6, L_3)$$

In the first CCR, the head of the free list is accessed, and so we discover the constraint $L_2 =$

$L_1 \cup \{(l_2, r'), (l_4, r'), (l_6, r')\}$. *Updating with respect to this constraint gives us*

$$\mathsf{Upd}(I_4(r'), L_2) = \{\mathit{list} = \mathtt{nil} \wedge \mathtt{emp},\ \langle \mathit{list} \mapsto \mathtt{nil}\rangle_{\emptyset},\ \langle \mathit{list} \mapsto x'\rangle_{\emptyset} * \langle x' \mapsto \mathtt{nil}\rangle_{\{(l_5, r')\}}\}$$

When we exit the CCR, we get $I_5(r') = I_4(r') \cup \{\langle \mathit{list} \mapsto \mathtt{nil}\rangle_{\{(l_5,r'),(l_7,r')\}}\}$. *In the next CCR, again the head of the list is accessed and so the latest disjunct gives us the constraint* $L_3 = L_2 \cup \{(l_5, r'), (l_7, r')\}$. *Updating with respect to this constraint and adding the new disjunct we get*

$$I_6(r') = \{\mathit{list} = \mathtt{nil} \wedge \mathtt{emp},\ \langle \mathit{list} \mapsto \mathtt{nil}\rangle_{\emptyset},\ \langle \mathit{list} \mapsto x'\rangle_{\emptyset} * \langle x' \mapsto \mathtt{nil}\rangle_{\emptyset},\ \langle \mathit{list} \mapsto \mathtt{nil}\rangle_{\{(l_8, r')\}}\}$$

Now the third disjunct in this set is concrete, and so at the end of this iteration when the abstraction heuristics are applied, the existential variable x' *in the third disjunct is thrown away to give the general formula* $\mathtt{ls}(\mathit{list}, \mathtt{nil})$. *This is a list segment from* list *to* `nil`, *which is the valid resource invariant for the memory manager.*

4.6 Extension to Loops

So far we have not discussed how to handle programs with while loops. Programs with loops usually require abstraction on the symbolic states in order to generalise the states so that a loop invariant can be reached. This causes a complication when tracking ownership transfer between loop iterations, since it is not known how to perform abstraction without knowing the distribution of ownership. This is a general problem for resource invariant synthesis methods that attempt to infer ownership, such as in the case of the bi-abduction method [9] where the problem is not addressed. In the case of the reachability method of [23], loops do not present a complication since there is no ownership inference being performed, as it assumed that the reachability heuristic provides the correct ownership distribution. This method can therefore proceed with standard abstraction techniques to reach loop invariants.

In this section we present a general technique for ownership inference in the presence of loops, which is parametric in an invariant synthesis method for loop free programs. The technique is based on analysing finite loop-free *unfoldings* of the concurrent program. We extend the programming language with the while loop command `while` B C which executes the body C until the

Algorithm 4 ExtInvSynth(Prg, P_{in})

```
1: $i := 1$;
2: while true do
3:     $I :=$ InvSynth$(Prg_{<i>}, P_{in})$;
4:     if TestInv$(Prg, P_{in}, I)$ then
5:         return $I$;
6:     end if
7:     $i := i + 1$;
8: end while
```

condition B is false. Note that the loop body may contain CCRs and ownership transfers may occur between iterations of the loop. Given a program $C = \texttt{while}\ B\ C'$ we define the i^{th} unfolding $C_{<i>}$ of C as

$$C_{<0>} \overset{def}{=} \texttt{assume}(\neg B)$$

$$C_{<i+1>} \overset{def}{=} \texttt{assume}(B); C'; C_{<i>} + C_{<i>}$$

For a given concurrent program *Prg* containing while loops, we define $Prg_{<i>}$ to be *Prg* in which every while loop C is replaced by $C_{<i>}$. We show the extended method for handling concurrent programs with while loops in algorithm 4. The algorithm progressively unfolds all the loops in the program and infers a resource invariant for the unfolded program using the InvSynth procedure from Algorithm 3. The abstraction function applied in the InvSynth procedure help to reach a generalised resource invariant formula which is applicable to an *arbitrary* number of loop unfoldings. The algorithm uses the TestInv function to check if such a formula has been reached after every iteration.

Note is that this is a generic method for ownership inference in the presence of loops, in that our technique can be used with any invariant inference method that works for loop-free programs. For example, instead of using the InvSynth procedure in algorithm 4, we may use some other invariant synthesis method such as the bi-abduction method of [9].

Examples 4.1 *The program on the left in figure 4.13 shows the producer-consumer pattern in which a producer thread continuously creates a new cell and waits for the buffer to empty before placing the cell in the buffer, and the consumer thread waits until the buffer is full and then takes the cell out of the buffer and disposes it. In this case, one unfolding of the program is the put-get buffer program from figure 4.4, for which the* InvSynth *procedure generates the resource invariant* $\{f = 0 \wedge \texttt{emp},\ f = 1 \wedge c \mapsto \texttt{nil}\}$. *This is the valid resource invariant for the producer-consumer program and so the* ExtInvSynth *procedure finds the correct invariant in one unfolding.*

```
          c := nil;                                   c := nil;
          f := 0;                                     f := 0;
       resource r(c, f)                             list := nil;
                                              resource r(c, f)  r'(list)
while (true) {    ||  while (true) {
  new(x);         ||    get(y);          while (true) {    ||  while (true) {
  put(x);         ||    dispose(y);        alloc(x);       ||    get(y);
}                 ||  }                    put(x);         ||    dealloc(y);
                                         }                 ||  }
```

Figure 4.13: Producer-consumer (left) and producer-consumer with memory manager (right)

The program on the right in figure 4.13 shows the producer-consumer pattern that uses the memory manager we described in example 10. In this case again, for one unfolding of the program, the InvSynth *procedure discovers the resource invariant* $\mathtt{ls}(list, \mathtt{nil})$ *for the memory manager and the standard one for the buffer, which are the correct resource invariants for the actual program with loops.*

4.7 Conclusion

We have presented an algorithm that synthesizes resource invariants for automating CSL proofs. The method is based on a form of label tracking in which ownership constraints are propagated through a program proof, and ownership transfers are determined from heap accesses at arbitrary points in the execution. We have demonstrated how the method is able to analyse programs which could not be handled previously, and also does not require user annotations about ownership distribution.

Although the approach that we have presented intuitively does not rely on any heuristic decisions about ownership inference (as was the case in previous methods), this property has not yet been demonstrated in a formal sense. It remains for us to explore some kind of *completeness* result about our algorithm, perhaps in a restricted setting which avoids the other standard sources of imprecision such as the abstraction functions.

Another next step is to investigate ownership inference for fine-grained concurrent programs, which use synchronization methods that avoid mutually exclusive access to entire data structures, and therefore achieve greater parallelism. Such programs require more advanced analyses than

CSL, such as the recent rely-guarantee based methods of [53, 12]. In these analyses the state of shared resources may change under specified *actions* rather than always satisfying an invariant. However, as with the inference of resource invariants, existing methods for inferring actions [52] resort to heuristic ownership inference techniques, which may possibly be avoided with the incorporation of label tracking.

Chapter 5

Conclusion and Future Directions

This thesis has extended resource reasoning with separation logic in the areas of modular program specification, program optimization, and concurrency verification. The general theory of relevance footprints developed in the setting of abstract separation logic has formalised the essentiality and sufficiency properties of footprints in local reasoning. We have also introduced a new semantic model of heaps which reestablishes the correspondence between safety and relevance footprints, and we identified the general property of determinism constancy which guarantees this correspondence in arbitrary resource models. We have yet to explore practical applications of the new semantic heap model which, apart from resolving the footprint problems, also permits the modelling of deterministic and bounded-memory allocation, unlike the standard model.

The second part of the thesis introduced labelled separation logic, which is based on the idea that, while spatial separation delivers local reasoning and tractable verification of programs, the stronger notion of *labelled separation* allows us to track deeper properties of program executions. We first described the use of labelled separation logic for detecting dependences between program statements, which is needed for program optimizations such as parallelization. The method has yielded promising initial results in the area of hardware synthesis, and with the incorporation of various recent techniques to improve the underlying shape analysis, we aim to explore its application to a wider class of industrial programs.

In the third part of the thesis we applied the notion of resource labelling to the problem of inferring ownership of shared resources in concurrent programs. We presented a new ownership inference algorithm based on label-tracking which can verify concurrent programs that could not be handled by previous methods. As part of next steps, we would like to formally demonstrate the non-heuristic nature of the ownership inference method, as well as to extend it to the verification of fine-grained concurrent programs.

We end with a discussion of some further directions in which the resource reasoning techniques presented in this thesis may be developed and applied in the longer term.

Dependence analysis at higher levels of abstraction Traditional analyses only test dependence based on actual memory accesses, but sometimes commands may access the same memory yet still be 'independent' with respect to the abstract specification required of a program. For example, the W3C DOM library for updating web pages is specified in terms of operations that act locally on an abstract tree structure, so that two updates to disjoint nodes in the DOM tree would be considered to be independent at the abstract level. However, depending on the specific implementation, these operations may interfere on the underlying heap structure that represents the abstract structure, such as performing global traversals to reach the relevant nodes. It will be interesting to explore recent approaches such as context logic [10, 20] or abstract predicates [44] to adapt labelled resource reasoning to higher levels of abstraction, which may lead to more powerful forms of optimizations.

Object-oriented programming Due to a range of difficulties, such as the extensive presence of aliasing, higher-order features and inheritance, the analysis of object-oriented programs presents many challenges. In this thesis we have focused on simple imperative programming languages with heap-manipulating commands, but significant advances have also been made in using separation logic for the verification of object-oriented programs [44, 45, 18]. Exploring the possible combination of these techniques with the methods presented in this thesis may be a first step toward addressing optimization and concurrency analysis for object-oriented programs.

Futures-annotated programs Another area that we are currently exploring are programs with *futures* annotations [40]. Futures are synchronization constructs that permit the result of a computation to be computed in parallel with the rest of the execution. We aim to use the resource tracking provided by labelled separation logic to verify the important safety property that resources are shared correctly between the future computation and the continuation, so that the annotated program can be guaranteed to produce the same results as its sequential counterpart. This is unlike memory-safety checking for concurrent programs, where the aim is only to show the absence of memory errors, rather than to show equivalence with some sequential program.

Concurrency optimizations Since concurrent programs are difficult to write, programmers often make over-conservative choices to ensure safety, such as unneeded synchronization or larger than needed critical regions. The ownership inference method of chapter 4 provides information about how resources are being used and shared in a concurrent program. Apart from verifying safety properties, this information may also help in using resources more efficiently in concurrent programs. For example, together with a dependence analysis like the one described in chapter 3, it may enable various synchronization optimizations in concurrent programs [16], such as synchronization elimination, critical region expansion (to minimize overhead of acquires and releases), critical region reduction (to increase concurrency), and data replication to reduce synchronized access to shared data.

Bibliography

[1] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *Computer Aided Verification*, pages 178–192. Springer-Verlag, 2007.

[2] J. Berdine, C. Calcagno, and P. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *International Symposium on Formal Methods for Components and Objects*, pages 115–137. Springer-Verlag, 2005.

[3] J. Berdine, C. Calcagno, and P. O'Hearn. Symbolic execution with separation logic. In *Asian Symposium on Programming Languages and Systems*, pages 52–68. Springer-Verlag, 2005.

[4] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *Symposium on Principles of Programming Languages*, pages 259–270. ACM, 2005.

[5] R. Bornat, C. Calcagno, and H. Yang. Variables as resource in separation logic. In *Conference on the Mathematical Foundations of Programming Semantics*, pages 247–276. Elsevier ENTCS, 2005.

[6] Stephen Brookes. A semantics for concurrent separation logic. *Theoretical Computer Science*, 375:227–270, 2007.

[7] C. Calcagno, D. Distefano, P. OHearn, and H. Yang. Footprint analysis: A shape analysis that discovers preconditions. In *Static Analysis Symposium*, pages 402–418. Springer-Verlag, 2007.

[8] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional shape analysis by means of Bi-abduction. In *Symposium on Principles of Programming Languages*, pages 289–300. ACM, 2009.

[9] C. Calcagno, D. Distefano, and Viktor Vafeiadis. Bi-abductive resource invariant synthesis. In *Asian Symposium on Programming Languages and Systems*, pages 259–274. Springer-Verlag, 2009.

[10] C. Calcagno, P. Gardner, and U. Zarfaty. Context logic and tree update. In *Symposium on Principles of Programming Languages*, pages 271–282. ACM, 2005.

[11] C. Calcagno, P. O'Hearn, and H. Yang. Local action and abstract separation logic. In *Symposium on Logic in Computer Science*, pages 366–378. IEEE Computer Society, 2007.

[12] C. Calcagno, M. Parkinson, and V. Vafeiadis. Modular safety checking for fine-grained concurrency. In *Static Analysis Symposium*, pages 233–248. Springer-Verlag, 2007.

[13] B. Cook, Ashutosh Gupta, S. Magill, Andrey Rybalchenko, Jiri Simsa, Satnam Singh, and Viktor Vafeiadis. Finding heap-bounds for hardware synthesis. In *Formal Methods in Computer-Aided Design*, pages 205–212. IEEE, 2010.

[14] B. Cook, S. Magill, M. Raza, J. Simsa, and S. Singh. Making fast hardware with separation logic. In preparation, 2010.

[15] J. B. Dennis. First version of a data flow procedure language. In *Programming Symposium, Proceedings Colloque sur la Programmation*, pages 362–376, London, UK, 1974. Springer-Verlag.

[16] P. Diniz and M. Rinard. Synchronization transformations for parallel computing. In *Principles of Programming Languages*, pages 187–200. ACM, 1997.

[17] D. Distefano, P. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS: Tools and Algorithms for the Construction and Analysis of Systems*, pages 287–302. Springer-Verlag, 2006.

[18] D. Distefano and M. Parkinson. jstar: Towards practical verification for java. In *Object Oriented Programming Systems Languages and Applications*, pages 213–226. ACM, 2008.

[19] Robert W. Floyd. Assigning meanings to programs. In *Symposium on Applied Mathematics*, pages 19–32. American Mathematical Society, 1967.

[20] P. Gardner, G. Smith, M. Wheelhouse, and U. Zarfaty. Local hoare reasoning about DOM. In *Principles of Database Systems*, pages 261–270. ACM, 2008.

[21] R. Ghiya, L. Hendren, and Y. Zhu. Detecting parallelism in C programs with recursive data structures. In *Conference of Compiler Construction*, pages 159–173. Springer-Verlag, 1998.

[22] A. Gotsman, J. Berdine, and B. Cook. Interprocedural shape analysis with separated heap abstractions. In *Static Analysis Symposium*, pages 240–260. Springer-Verlag, 2006.

[23] A. Gotsman, J. Berdine, B. Cook, and M. Sagiv. Thread-modular shape analysis. In *Programming Language Design and Implementation*, pages 266–277. ACM, 2007.

[24] B. Guo, N. Vachharajani, and D. August. Shape analysis with inductive recursion synthesis. In *Programming Language Design and Implementation*, pages 256–265. ACM, 2007.

[25] R. Gupta, S. Pande, K. Psarris, and V. Sarkar. Compilation techniques for parallel systems. In *Parallel Computing*, pages 1741–1783. Elsevier Science Publishers, 1999.

[26] L. J. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. In *IEEE Transactions on Parallel and Distributed Systems*, pages 35–47. IEEE Press, 1990.

[27] C. A. R Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.

[28] T. Hoare and P. O' Hearn. Separation logic semantics of communicating processes. In *Foundations of Informatics*, pages 3–25. Elsevier ENTCS, 2008.

[29] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. In *Programming Language Design and Implementation*, pages 28–40. ACM, 1989.

[30] J. Hummel, L. Hendren, and A. Nicolau. A general data dependence test for dynamic, pointer-based data structures. In *Programming Language Design and Implementation*, pages 218–229. ACM, 1994.

[31] C. Hurlin. Automatic parallelization and optimization of programs by proof rewriting. In *Static Analysis Symposium*, pages 52–68. Springer-Verlag, 2009.

[32] S. Isthiaq and P. O' Hearn. BI as an assertion language for mutable data structures. In *Principles of Programming Languages*, pages 14–26. ACM, 2001.

[33] N.D. Jones. and S.S Muchnick. Flow analysis and optimization of Lisp-like structures. In *Principles of Programming Languages*, pages 244 – 256. ACM, 1979.

[34] B. Kernighan and D. Ritchie. *The C programming language*. Prentice Hall, 1988.

[35] S. Magill, J. Berdine, E. Clarke, and B. Cook. Arithmetic strengthening for shape analysis. In *Static Analysis Symposium*, pages 419–436. Springer-Verlag, 2007.

[36] S. Magill, A. Nanevski, E. Clarke, and P. Lee. Inferring invariants in separation logic for imperative list-processing programs. In *Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management*, 2006.

[37] S. Magill, M. Tsai, P. Lee, and Y. Tsay. THOR: A tool for reasoning about shape and arithmetic. In *Computer Aided Verification*, pages 428–432. Springer-Verlag, 2008.

[38] M. Marron, D. Stefanovic, D. Kapur, and M. Hermenegildo. Identification of heap-carried data dependence via explicit store heap models. In *Languages and Compilers for Parallel Computing*, pages 94–108. Springer-Verlag, 2008.

[39] C. Morgan. The specification statement. In *ACM Transactions on Programming Languages and Systems*, pages 403–419. ACM, 1988.

[40] A. Navabi and S. Jagannathan. Exceptionally safe futures. In *Coordination Models and Languages*, pages 47–65. Springer-Verlag, 2009.

[41] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Computer Science Logic*, pages 1–19. Springer-Verlag, 2001.

[42] P. W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375:271–307, 2007.

[43] M. Parkinson. *Local Reasoning for Java*. University of Cambridge (Ph.D. Thesis), 2005.

[44] M. Parkinson and G. Bierman. Separation logic and abstraction. In *Principles of Programming Languages*, pages 247–258. ACM, 2005.

[45] M. Parkinson and G. Bierman. Separation logic, abstraction and inheritance. In *Principles of Programming Languages*, pages 75–86. ACM, 2008.

[46] M. Parkinson, R. Bornat, and C. Calcagno. Variables as resource in Hoare logics. In *Logic in Computer Science*, pages 137–146. IEEE Computer Society, 2006.

[47] M. Parkinson, R. Bornat, and P. O'Hearn. Modular verification of a non-blocking stack. In *Principles of Programming Languages*, pages 297–302, 2007.

[48] M. Raza, C. Calcagno, and P. Gardner. Automatic parallelization with separation logic. In *European Symposium on Programming*, pages 348–362, 2009.

[49] M. Raza and P. Gardner. Footprints in local reasoning. In *Foundations of Software Science and Computation Structures*, pages 201–215. Springer-Verlag, 2008.

[50] M. Raza and P. Gardner. Footprints in local reasoning (with regaining of safety footprints). *Logical Methods in Computer Science*, 5(2), 2009.

[51] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS: Symposium on Logic in Computer Science*, pages 55–74. IEEE Computer Society, 2002.

[52] V. Vafeiadis. RGSep action inference. In *Verification, Model Checking, and Abstract Interpretation*, pages 345–361. Springer-Verlag, 2010.

[53] V. Vafeiadis and M. Parkinson. A marriage of rely/guarantee and separation logic. In *Conference on Concurrency Theory*, pages 256–271. Springer-Verlag, 2007.

[54] H. Yang. An example of local reasoning in BI pointer logic: the Schorr-Waite graph marking algorithm. In *SPACE*, 2001.

[55] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. OHearn. Scalable shape analysis for systems code. In *Computer Aided Verification*, pages 385–398. ACM, 2008.

[56] H. Yang and P. O'Hearn. A semantic basis for local reasoning. In *Foundations of Software Science and Computational Structures*, pages 402–416. Springer-Verlag, 2002.