Technical Report

UCAM-CL-TR-917
ISSN 1476-2986

Number 917

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Formal verification-driven parallelisation synthesis

Matko Botinčan

March 2018

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted September 2013 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Trinity College.

ISSN 1476-2986

ABSTRACT

Concurrency is often an optimisation, rather than intrinsic to the functional behaviour of a program, i.e., a concurrent program is often intended to achieve the same effect of a simpler sequential counterpart, just faster. Error-free concurrent programming remains a tricky problem, beyond the capabilities of most programmers. Consequently, an attractive alternative to manually developing a concurrent program is to automatically *synthesise* one.

This dissertation presents two novel formal verification-based methods for safely transforming a sequential program into a concurrent one. The first method—an instance of *proof-directed synthesis*—takes as the input a sequential program and its safety proof, as well as annotations on where to parallelise, and produces a correctly-synchronised parallelised program, along with a proof of that program. The method uses the sequential proof to guide the insertion of synchronisation barriers to ensure that the parallelised program has the same behaviour as the original sequential version. The sequential proof, written in separation logic, need only represent shape properties, meaning we can parallelise complex heap-manipulating programs without verifying every aspect of their behaviour.

The second method proposes *specification-directed synthesis*: given a sequential program, we extract a rich, stateful specification compactly summarising program behaviour, and use that specification for parallelisation. At the heart of the method is a learning algorithm which combines dynamic and static analysis. In particular, dynamic symbolic execution and the computational learning technique *grammar induction* are used to conjecture input-output specifications, and counterexample-guided abstraction refinement to confirm or refute the equivalence between the conjectured specification and the original program. Once equivalence checking succeeds, from the inferred specifications we synthesise code that executes speculatively in parallel—enabling automated parallelisation of irregular loops that are not necessary polyhedral, disjoint or with a static pipeline structure.

Acknowledgements

There are many people to whom I am indebted for their help and support, and without whom this thesis would never have seen the light of the day.

Firstly, I would like to express my deepest gratitude to my extraordinary supervisors, Matthew Parkinson and Mike Gordon, for their invaluable guidance, letting me plot my own research path while gently keeping me on track, and to whom I could always have turned for assistance. Matthew, who engaged me into separation logic, initially my primary supervisor, and who became my industrial supervisor afterwards, provided holistic mentorship at every stage of my degree and has been the best supervisor a PhD student could ever ask for—inspiring, patient and fun to work with. Mike, who took over as my primary supervisor when Matthew moved to industry, provided careful feedback from a broader verification perspective and has always offered a friendly ear whenever I was in a need of advice.

I would like to thank my PhD examiners, Alan Mycroft and Hongseok Yang, for their insightful remarks in the initial version of the thesis and helpful comments and suggestions that made it better than it initially was.

I am thankful to have had the privilege of working with many collaborators throughout my PhD—this thesis owes much to them. In alphabetical order (as no other would do justice), they were: Domagoj Babić, Byron Cook, Dino Distefano, Mike Dodds, Alastair F. Donaldson, Radu Grigore, Suresh Jagannathan, Stephen Magill, Daiva Naudžiūnienė, Matthew Parkinson, and Wolfram Schulte. It has been wonderful exchanging thoughts, materialising them into theorems and proofs, and writing computer code together. I am also grateful to have shared my office with the exquisite minds of Eric Koskinen (despite occasional disapproval of his choice of music) and John Wickerson (despite his loud typing), and am appreciative of the Programming, Logic, and Semantics Group's environment fostering academic brilliance. I have benefited a lot from stimulating interactions with the people who at some point worked (or still work) at Microsoft Research Cambridge, including Josh Berdine, Samin Ishtiaq, Mohammad Raza, Andrey Rybalchenko, Dimitrios Vytiniotis and Christoph M. Wintersteiger, and members of the loosely-defined group once known as the *East London Massive*, including Richard Bornat, Philippa Gardner, Peter W. O'Hearn and Hongseok Yang. Finally, I also thank all the support staff of the Computer Laboratory, and particularly Lise Gough who handled administrative obstacles with great care and made them go away.

I also thank my former supervisors of my previous degrees at Department of Mathematics at the University of Zagreb, Robert Manger, Dean Rosenzweig and Mladen Vuković, for a great start into the fields of mathematical logic and concurrency, as well as all the other professors who fueled my first research steps and engraved into me the mathematical way of thinking.

I gratefully acknowledge the scholarship and financial support from Gates Cambridge Trust that enabled my PhD study at the University of Cambridge, and the travel grants received from Computer Laboratory that helped covering the costs of attending conferences and workshops. I am also thankful to Trinity College for all the support received from their end, and for providing me with accommodation, meals and friendly environment during my stay at Cambridge.

Lastly, and most importantly, I save the biggest thanks to my family. I thank my parents and my sister for their unwavering encouragement of my endeavours every step of the way, and for having done their best to make sure I never lacked for anything. Finally, none of this would have been possible if not for my amazing wife Agata, who has been the greatest source of inspiration and support throughout my entire PhD journey. I am forever indebted to her for all the sacrifices she has made, for always believing in me and driving me to achieve more than I thought I could, and ultimately, for making all the efforts worthwhile. Thank you for bringing the sunshine into my life, Bubica 🐞!

CONTENTS

Chapter 1

Introduction

The immense impact of computer technology on all spheres of life requires software to be developed more rapidly and reliably than ever before. Despite great advancements in programming methods and tools, computer programs often do not behave as intended. There are many sources of program errors. For this dissertation we are particularly interested in those introduced by program optimisation. Programs are optimised for various goals—e.g., so that they execute faster, operate with smaller amount of memory or drain less power—which are not intrinsic to their functional behaviour.[1] And so, while optimised programs do not add new features, it can be much harder to check them for errors as the lower-level optimisation details obfuscate the higher-level programmer's intentions.

Over the last decade, significant progress has been made in formal analysis and verification of software systems, yielding methods for precise, mathematical reasoning about properties of program behaviour. The practicality of these methods owes much to the maturing automated theorem proving technology (enabling automation) and abstraction mechanisms (enabling scaling to sizeable and realistic classes of programs). Yet, even with the state-of-the art techniques, the complexity of verifying optimised programs is often at odds with the high-level intentions of the programmer, expressed in the original, unoptimised program. Consequently, an attractive alternative to *verifying* an optimised program is to automatically *synthesise* one.

This dissertation is concerned with a particular kind of program optimisation—*parallelisation*. Indeed, in most cases a concurrent version of the program is intended to achieve the same effect of a simpler sequential counterpart, but faster (e.g., by exploiting the many cores now supplied by chip manufacturers). Error-free concurrent programming remains a tricky problem, beyond the capabilities of most programmers; consequently, automatically synthesising a concurrent program is an appealing proposition over manually parallelising it. In such approach, the programmer writes a sequential program, which is then automatically transformed into a concurrent one, exhibiting the same behaviour. Programmers can work in the simpler, more reliable sequential setting, yet reap the performance benefit of concurrency.

In this dissertation, we propose ways for automated synthesis of parallelised pro-

[1]The commonly acknowledged wisdom "The First Rule of Program Optimisation: Don't do it. The Second Rule of Program Optimisation (for experts only!): Don't do it yet." (Michael A. Jackson) does not render program optimisation void; "Optimisation is a standard part of the software development process, along with testing, debugging, and quality assurance, and it should be treated as such." [Hyd09].

grams from sequential ones. The central thesis of the dissertation is that

> *formal verification methods, originally developed for the analysis and verification of sequential programs, can be profitably exploited to create useful methods for automated synthesis of program parallelisation.*

To support this thesis, the dissertation contributes methods of two kinds:

Proof-directed parallelisation synthesis, in which a programmer supplies a sequential program and a lightweight sequential proof, written in separation logic, and as a result obtains a correctly-synchronised parallelised program, along with a proof of that program; and

Specification-directed parallelisation synthesis, in which a rich, stateful specification of the sequential program is extracted by means of formal verification, and that specification, compactly summarising the program behaviour, is then used to execute the program in parallel.

As we will see, the two approaches to parallelisation synthesis are built using radically different formal verification methods, but they share a common trait: they leverage the sequential program as an evidence of programmer's intentions and delegate the burden of parallelisation to an automated algorithm, ensuring safety of the parallelised program. In a sense, the parallelised program is verified by construction, with respect to the original sequential program.

1.1 Relation to the state-of-the-art

Many traditional compiler analyses have similar goals as our parallelisation methods. While our methods are not tied to a specific structure of the input sequential program, to give the reader some intuition about how they relate to prior methods, we briefly illustrate the effect of our methods in the context of loop parallelisation. This context was commonly the focus of prior methods since loops are in many cases the most abundant source of potential parallelism in programs.

Proof-directed parallelisation synthesis. Assume we give our proof-directed parallelisation method a sequential for-loop iteratively running some function `fun`

```
for (...) {
  fun(...);
}
```

and a sequential proof of `fun` (unspecified here). The particular details of `fun` are unimportant, but we note that calls to `fun` from different loop iterations may perform operations on shared variables or data structures. Our method would transform this sequential loop into a *parallelised* loop of the form

```
for (...) {
  fork(fun'(...));
}
```

in which each loop iteration runs a synchronised version (`fun'`) of `fun` in a separate thread. The synchronisation mechanisms injected in `fun'` by our algorithm guarantee that the behaviour of sequential loop iterations will be maintained when they are run concurrently in the parallelised loop.

Traditional compiler analyses typically allow such parallelisation when loop iterations can statically be shown to operate on disjoint portions of memory or when dependencies between iterations are constrained in some way (e.g., by requiring that indices into array elements are affine). Our method differs in that it does not restrict the kind of objects shared across iterations (e.g., these could be heap locations, dynamic data structures, system resources, etc.), while permitting arbitrary dependencies. The degree of parallelisation inserted is governed by the accompanying sequential proof—a finer-grained proof may discover finer-grained dependencies that allow making the program more parallel.

Specification-directed parallelisation synthesis. This method for parallelisation synthesis aims to unlock data-parallelism in programs that interact with the environment by means of streams. For instance, let us consider the following for-loop which repeatedly runs a computation `io_fun` reading some number of data items from an input stream and writing result(s) to an output stream:

```
for (...) {
  io_fun(...);
}
```

Again, the particular details of `io_fun` are unimportant, although our focus is on fairly simple computations that generate output(s) using only a bounded window of data items from the past. Our method would extract a symbolic specification `spec` which compactly represents the input-output behaviour of the computation and then use `spec` to parallelise the computation on successive portions of the input.

The key aspect of specifications synthesised by our method is that they symbolically represent behaviour of the program using a finite set of states. Thus instead of protecting relevant dependencies in the original program we simply speculatively compute all possible outputs associated with the states of the specification. That is, at the expense of redundant computations we exploit the available concurrency without introducing additional synchronisation. The resulting parallelised loop is then roughly of the following form:

```
pfor (...) {
  spec(1, ...) || ... || spec(n, ...);
}
```

where **`pfor`** denotes that iterations operating on different portions of input may be run in parallel, `||` denotes that the specification `spec` may be run in parallel from each state, and `n` is the number of states in `spec`.

1.2 Contributions

Contributions of Part I.

1. We present a proof-directed parallelisation synthesis method which given a sequential program augmented with annotations indicating where parallelisation

might be profitable, and a sequential proof of the program, synthesises a parallel version of the program. In cases we consider, the proof would be generated automatically by a tool, which may require lightweight specifications to define relevant loop invariants.

2. We leverage tools of separation logic (in particular, abduction and frame inference) to define a path- and context-sensitive program analysis capable of identifying resources that are *needed*—resources that must be held to avoid faulting between two program locations (and thus must be held by any thread that would execute this block of code); and, *redundant*—resources that are not needed between two program locations (and thus would no longer be used by the executing thread).

3. We use information from the above analysis to inject synchronisation barriers into the original program—their semantics enable resource transfer from redundant to needed resources in the parallelised program. The analysis also constructs a safety proof in separation logic which validates the correctness of the barrier injections.

4. We prove that our transformation is specification-preserving: the parallelised program is guaranteed to satisfy the same specification as the sequential program. Moreover, for terminating programs that do not deallocate memory, we also show that the parallelised program preserves the *behaviour* of the original.

Contributions of Part II.

1. We present a method for synthesising symbolic specifications of a program's input-output behaviour. The key insight of our method is that such specifications can be synthesised by simultaneously exploiting techniques for under-approximating and over-approximating program traces.

2. We introduce a carefully limited variant of symbolic specifications, with which it is possible to faithfully represent input-output behaviour of still-interesting classes of programs. We show that such specifications are useful in practice through three case studies.

3. We develop algorithms for synthesising over-approximations and generalising under-approximations with grammar induction, both of which represent program's input-output behaviour using our limited variant of symbolic specifications.

4. We prove that for a certain class of programs, by repeatedly refining specifications we converge to a sound and complete specification of the program. Since our approach to synthesising over-approximations depends on predicate abstraction of the program, completeness holds as long as the abstraction refinement eventually yields a sufficiently strong set of predicates.

Collaboration. Part I is based on the POPL 2012 paper [BDJ12] and its TOPLAS 2013 extension [BDJ13], both co-authored by Mike Dodds and Suresh Jagannathan; and, to a smaller extent on the manuscript [BDM15] co-authored by Mike Dodds and Stephen Magill. Part II is based on the POPL 2013 paper [BB13] co-authored by Domagoj Babić.

Part I

Proof-directed parallelisation synthesis

Part I is concerned with a method for proof-directed parallelisation synthesis. We propose an analysis that, guided by a lightweight proof of the sequential program, safely transforms a sequential program into a concurrent one. We assume that the programmer annotates points in the sequential program where concurrency might be profitably exploited, without supplying any additional concurrency control or synchronisation. The result after applying the transformation is a correctly-synchronised concurrent program, along with a safety proof.

The proposed transformation ensures that the input-output behaviour of the sequential program is preserved by requiring that the concurrent program respects sequential data dependencies. That is, the transformation guarantees that the way threads access and modify shared resources in the concurrent program never results in a behaviour that would not be possible in the sequential program. To enforce these dependencies, we inject *synchronisation barriers*, signalling operations that regulate when threads are allowed to read or write shared state. These barriers can be viewed as resource-transfer operations that acquire and release access to shared resources such as shared-memory data structures when necessary.

The method is built on separation logic [Rey02]. The input safety proof, written in separation logic, is used to discover how resources are demanded and consumed within the program, and to synthesise barriers to precisely control access to these resources. The analysis relies on frame inference [BCO05b] and abduction [CDOY11], two techniques that generate fine-grained information capable of describing when resources are necessary and when they are redundant. This information enables optimisations that depend on deep structural properties of the resource—for example, we can split a linked list into dynamically-sized segments and transmit portions between threads piece-by-piece.

The input proof supplied to the analysis must state all resources that are used by the program such as the *shape* of all data-structures (for example, that a data-structure is a linked list). However, it need not establish the functional correctness of the algorithm. The analysis protects all dependencies that are relevant to the program's input-output behaviour, not just those specified in the proof. Thus we can apply the analysis to parallelise complex heap-manipulating programs without verifying every aspect of their behaviour.

Structure of Part I. An overview of the analysis and the necessary background, along with motivating examples are given in Chapter 2. Formal technical background concerning the program representation and the sequential proof is given in Chapter 3. Chapter 4 presents formalisation of the analysis and a proof that the parallelising transformation is specification-preserving, along with a discussion of its complexity, soundness and limitations. Observations about the way the choice of parameters affects the analysis are given in Chapter 5. A formal semantics of the programming language we use to develop the analysis is given in §6.1 and the soundness results—behaviour preservation and termination—are formalised in §6.2 and §6.3. Chapter 7 discusses related work.

CHAPTER 2

OVERVIEW

This chapter gives an informal overview of our method for proof-directed parallelisation synthesis. We start by describing the goal of the parallelisation process on a simple example (§2.1). As our method is built on separation logic, we explain basic separation logic concepts such as resource, separation, frame inference and abduction, along with the notions of dependency inherent to our method (§2.2). With these main elements we then give an overview (§2.3) of how our method parallelises the simple example from §2.1. At the end of the chapter, we discuss how the basic approach developed for the simple example is extended to handle unbounded iterations and dynamic data structures (§2.4).

2.1 Overview of the parallelisation process

The objective of our proof-directed parallelisation synthesis is to take a sequential program, annotated to indicate segments that might be feasible candidates to execute in parallel, and a sequential proof of the program, and to produce a *parallelised* program. This new program should be observationally equivalent to the sequential original; i.e. it should have identical input-output behaviour.

In the simplest scenario, the input sequential program is just a sequential composition $C_1; C_2$ of two sub-programs C_1 and C_2. We assume the programmer has identified C_1 and C_2 as candidates for parallelisation. The intended semantics is that C_1 and C_2 may run in parallel, but that the visible behaviour will be identical to running them in sequence; this semantics provides a simple but useful form of data parallelism in which concurrently executing computations may nonetheless access shared state.

To ensure behaviour preservation, we often need to insert synchronisation operations.

Example 2.1. Let us consider the following example program:

```
        f(1); f(2)
```

where `f()` is a procedure manipulating two distinct memory cells pointed to by `x` and `y`:[1]

[1]Throughout Part I we use C-style syntax which merely resembles the C language but is not actually C. Where the intended semantics differs from the usual C semantics, an additional clarification will be provided.

```
f(int i) {
  int v = *x;
  if (v>=i) *y = v;
  else *x = 0;
}
```

How can we parallelise this program without introducing any unwanted new behaviours, i.e., behaviours not possible under sequential execution? Naïvely, we might simply run both calls in parallel, without synchronisation, i.e.:

```
f(1) || f(2)
```

In some situations this parallelisation is good, and introduces no new observable behaviours (e.g., if the initial value pointed to by `x` is `0`). But, under others, the second call to `f()` may write to a memory location that is subsequently read by the first call to `f()`, violating the intended sequential ordering. For example, consider the case when the value pointed to by `x` is initially `1`; sequential execution would produce a final result in which the locations pointed to by `x` and `y` resp. are `0` and `1`, while executing the second call to completion before the first would yield a result in which the location pointed to by `y` is not updated.

Definition 2.1. We say one portion of a parallelised program occurs *logically earlier* than another if the two portions were sequentially ordered in the original sequential program. For instance, the call `f(1)` in Example 2.1 occurs logically earlier than `f(2)`.

To be sure that no new observable behaviour is introduced in the parallelisation, we must ensure that:

- a computation cannot read from a location that was already written to by a computation that occurs logically later; and
- a computation cannot write to a location that was already read from or written to by a computation that occurs logically later.

Note that a computation that occurs logically later from another need not always wait for the earlier one—synchronisation is only needed when reads and writes to a particular memory location could result in out-of-order behaviour. In other words, we want to enforce only sequential dependencies, while allowing beneficial race-free concurrency.[2]

Dependency-enforcing barriers. In order to enforce the logical ordering in the parallelised program, our method inserts barriers enforcing the logical order over the shared memory locations.

For the program in Example 2.1, our method would generate modified procedures $\texttt{f}_1\texttt{()}$ and $\texttt{f}_2\texttt{()}$, which would lead to the safely parallelised program

$$\texttt{f}_1\texttt{(1)}\ \ ||\ \ \texttt{f}_2\texttt{(2)}$$

[2]In a sequential program execution, a *dependency* between two events exists if the effect of one event can affect the behaviour of a subsequently executed one; these effects can be based on control-flow (e.g., the outcome of a conditional expression) or data-flow (e.g., assigning a value to a location that is subsequently read). If new dependencies are introduced or removed during parallelisation, the input-output behaviour of the program may change.

```
f₁(i){                                  f₂(i){
 int v = *x;                             wait(wx);
 if (v>=i) {                             int v = *x;
   grant(wx);                            if (v>=i) {
   *y = v;                                 wait(wy);
   grant(wy);                              *y = v;
 }                                       }
 else {                                  else {
   grant(wy);                              *x = 0;
   *x = 0;                                 wait(wy);
   grant(wx);                            }
 }                                      }
}
```

Figure 2.1: Parallelisation of the two calls to `f()`.

To enforce the ordering between calls to `f()`, we use barriers associated with memory locations (or, generally, resources) that are shared between concurrently executing computations ([NZJ08, DJP11]):

- `grant()`, a barrier that signals that the resource it protects can safely be accessed by a computation that occurs logically afterwards; and
- `wait()`, a barrier that blocks until the associated resource becomes available from a computation that occurs logically before it.

How do we use these barriers to enforce sequential dependencies? The exact pattern of parallelisation depends on the granularity of our parallelisation analysis. In the best case, there are no dependencies between successive calls (e.g., each invocation operates on a different portion of a data structure). In this case, we need no barriers, and all invocations run independently. In our example program, however, the locations `x` and `y` are shared between calls to `f()`, meaning barriers are required.

In the simplest parallelisation of our example, the call to `f`$_1$`()` would end with a call to `grant()`, while the call to `f`$_2$`()` would begin with a call to `wait()`—this would enforce a total sequential ordering on the invocations. A better (although still relatively simple) parallelisation is shown in Fig. 2.1. Two channels, `wx` and `wy`, are used to mediate signalling between `f`$_1$ and `f`$_2$—`wx` (resp. `wy`) is used to signal that the later thread can read and write to the heap location referred to by `x` (resp. `y`).

Our method inserts a `grant()` barrier when the associated resource will no longer be accessed. Similarly, it injects a `wait()` barrier to wait for the resource to become available before it is accessed. Intuitively, `grant()` barriers are inserted as early as possible in `f`$_1$`()` and `wait()` barriers as late as possible in `f`$_2$`()` so that every control flow path in `f`$_1$ (resp. `f`$_2$) calls `grant()` (resp. `wait()`) exactly once for each channel. Note that the call to `wait(wy)` in the else-branch of `f`$_2$`()` is not strictly necessary since there is no further access of the location `y`, however, to simplify the semantics, we require that along all control flow paths, each channel that has been granted on is eventually finalised with `wait()`.

How does our method generate these synchronisation barriers? Example 2.1 is simple, but in general the method must cope with complex dynamically-allocated resources such as linked lists. It should deal with the partial ownership (for example, read-access), and with manipulating portions of a dynamic data structure (for example, just the head of a linked list, rather than the entire list). We therefore need a means to express complex, dynamic patterns of resource management and transfer.

To achieve this, our method demands a sequential *proof*, written in separation logic, rather than just an undecorated program. This proof need not capture full functional correctness—it is sufficient that it just proves that the program does not fault. We exploit the dependencies expressed within this proof to determine the resources that are needed at a program point, and when they can be released. Our method inserts barriers to enforce the sequential dependencies represented in this proof. As the proof system we use is sound, these dependencies faithfully represent those in the original sequential program.

2.2 Resources, separation and dependency

Our method works by calculating the ways that resources are used in the source sequential program. By *resource*, we mean a set of elements of a mutable state that can generate dependencies between program statements. For example: heap cells in memory; objects built from heap cells such as linked lists; portions of objects, such as linked-list segments; and other kinds of mutable objects such as channels. If two program statements access the same resource, there may be a dependency between them. If they do not, then a dependency cannot exist.

At the heart of our method is automated reasoning using separation logic [Rey02, O'H07]. The power of separation logic lies in its ability to cleanly represent and compose resources. Indeed, resources in our analysis are simply separation logic assertions. Two assertions can be composed using the $*$ connective (the *separating conjunction*) to give a larger assertion. An assertion $P * F$ is satisfied if P and F hold, and the resources denoted do not overlap.[3] Loosely speaking, if one program statement *only* accesses the resource P and another F, there can be no dependency between them. In the case of heap resources, separation by $*$ corresponds to disjointness of heap address spaces.

As well as establishing the absence of faults, specifications in separation logic also establish the resource bounds of a program. A specification $\{P\}\ C\ \{Q\}$ can be read as saying: "*if C is run with a resource satisfying P, then (1) it will not fault, (2) if it terminates then it will give a resource satisfying Q, and (3) it will only access resources held in P, acquired through resource transfer or allocated during execution*".

This property—that all resources be described in preconditions or acquired explicitly—is an essential feature of separation logic (it is sometimes called the *tight interpretation* of specifications).

Example 2.2. The following specification is invalid in separation logic:

$$\{\mathtt{x} \mapsto v_1'\}\ \mathtt{*y} = 5\ \{\mathtt{x} \mapsto v_2'\}$$

[3]More formally, we define satisfaction for $*$ as follows (here $\oplus$ is a union of resources, and $\perp$ denotes that two resources are disjoint):

$$\sigma \models P * F \quad \overset{\text{def}}{\iff} \quad \exists \sigma_1, \sigma_2.\, (\sigma_1 \models P) \wedge (\sigma_2 \models F \wedge \sigma_1 \perp \sigma_2 \wedge \sigma_1 \oplus \sigma_2 = \sigma)$$

Loosely, this says that a resource σ satisfies a conjunction $P * Q$ if the resource can be broken down into two disjoint sub-resources σ_1 and σ_2, satisfying P and Q respectively.

(We write $a \mapsto b$ to indicate that the address a maps to the value b in the heap. By convention, primed variables are implicitly existentially quantified.) This specification is invalid because `x` and `y` might point to different locations, meaning the program might write to a location not mentioned in its precondition. On the other hand, the following specification is valid:

$$\{\mathtt{x} \mapsto v_1' \wedge \mathtt{x} = \mathtt{y}\}\ {*}\mathtt{y} = 5\ \{\mathtt{x} \mapsto v_2' \wedge \mathtt{x} = \mathtt{y}\}$$

The tight interpretation is essential for the soundness of our analysis. Suppose we prove a specification for a program (or portion thereof); resources outside of the precondition cannot be accessed by the program, and cannot affect its behaviour. If two program statements access disjoint resources, then there can be no dependencies between the two statements.[4] Consequently, such resources can be safely transferred to other program segments that are intended to be executed in parallel.

The key proof rule of separation logic is the *frame rule*, which allows small specifications to be embedded into a larger context.

$$\frac{\{P\}\ C\ \{Q\}}{\{P * F\}\ C\ \{Q * F\}}\ \textsc{Frame}$$

The concurrent counterpart of the frame rule is the *parallel rule*. This allows two threads that access non-overlapping resources to run in parallel, without affecting each other's behaviour.

$$\frac{\{P_1\}\ C_1\ \{Q_1\} \qquad \{P_2\}\ C_2\ \{Q_2\}}{\{P_1 * P_2\}\ C_1 \| C_2\ \{Q_1 * Q_2\}}\ \textsc{Parallel}$$

The parallel rule enforces the absence of data races between C_1 and C_2. The two threads can consequently only affect each other's behaviour by communicating through built-in synchronisation mechanisms, such as locks or barriers, that act as shared resources.

Assertion language. As well as the separating conjunction $*$ and the standard first-order logical operators, $\vee$, $\wedge$, $\exists$, we assume a number of basic assertions for representing heap cells. The simplest is emp, which asserts that the heap is empty. We have already encountered the points-to assertion $a \mapsto v$, which says that the heap cell with address a holds value v. Assertions in separation logic represent ownership of heap cells, so this assertion also says that the thread is allowed to read and write to address a freely. Where the object at an address can have more than one field, we write $a.f \mapsto v$ to represent an individual field f [Par05]. By convention, primed variables are implicitly existentially quantified (as in Example 2.2).

We also assume a *fractional* points-to assertion in the style of [BCOP05] to allow a location to be shared between threads. An assertion $a \overset{f}{\mapsto} v$ for some $f \in (0, 1]$ means that the current thread holds permission to read f on address a. If $f = 1$, the current thread also holds permission to write to the address. Fractional permissions can be split and recombined according to the following rules:

$$a \xmapsto{f_1+f_2} v \iff a \overset{f_1}{\mapsto} v * a \overset{f_2}{\mapsto} v \qquad\qquad a \mapsto v \iff a \overset{1}{\mapsto} v$$

[4]However, there is an important caveat regarding memory allocation and disposal—see §4.4 for a discussion.

Using fractional permissions, read-only access can be shared between several threads in the system. Read-write permission can be recovered if some thread collects together all of the fractional permissions for the address.

To represent lists, we assume a predicate $\mathsf{lseg}(x,t)$, which asserts that a segment of a linked list exists with head x and tail-pointer t. We write $\mathsf{node}(x,y)$ to represent a single node in the list:

$$\mathsf{node}(x,y) \quad \triangleq \quad x.\mathsf{val} \mapsto v' * x.\mathsf{nxt} \mapsto y$$

We then define lseg as the least separation logic predicate satisfying the following recursive equation:

$$\mathsf{lseg}(x,t) \quad \triangleq \quad (x = t \wedge \mathsf{emp}) \vee (\mathsf{node}(x,y') * \mathsf{lseg}(y',t))$$

The behaviour of our analysis depends strongly on the choice of these resource predicates. We discuss alternatives to lseg, node, etc. in §5.

Automated inference. Automated reasoning and symbolic execution in separation logic revolves around two kinds of inference questions: *frame inference* and *abduction*. These questions form the basis of symbolic execution with separation logic [BCO05a, CDOY11]. Intuitively, frame inference lets us reason forwards, while abduction lets us reason backwards.

The first question, *frame inference*, calculates the portion of an input assertion which is 'left over', once a desired assertion has been satisfied. Given an input assertion S and desired assertion P, an assertion $F_?$ is a valid frame if $S \vdash P * F_?$. We write such inference question as follows:

$$S \vdash P * [F_?]$$

(Throughout the text, we use square brackets to indicate the portions of an entailment that should be computed.)

Frame inference is used during forwards symbolic execution to calculate the effects of commands. Suppose we have a symbolic state represented by an assertion S, and a command C with specification $\{P\}\, C\, \{Q\}$. If we calculate the frame in $S \vdash P * [F_?]$, the symbolic state after executing C must be $Q * F_?$.

Example 2.3. Consider the following specification:

$$\{\mathtt{x} \mapsto v'\} \;\; \mathtt{*x\ =\ 42} \;\; \{\mathtt{x} \mapsto 42\}$$

With initial assertion $S = \mathtt{x} \mapsto 5 * \mathtt{y} \mapsto 7$ and desired precondition $P = \mathtt{x} \mapsto v'$ we obtain the frame $\mathtt{y} \mapsto 7$. By combining this with the command's postcondition, we get the assertion after executing the command, $\mathtt{x} \mapsto 42 * \mathtt{y} \mapsto 7$.

The second kind of inference question, *abduction*, calculates an *antiframe*—the 'missing' assertion that must be combined with an input assertion in order to satisfy some desired assertion. Given an input assertion S and a desired assertion P, an assertion $A_?$ is a valid antiframe if $S * A_? \vdash P$. We write such inference question as follows:

$$S * [A_?] \vdash P$$

Abduction is used during a backwards analysis to calculate what extra resources are necessary to execute the command safely. Suppose we have a symbolic state represented by an assertion S, and a command C with specification $\{P\}\,C\,\{Q\}$. It might be that S does not include sufficient resources to execute C. In this case, if we calculate the antiframe $S * [A_?] \vdash P$, the assertion $S * A_?$ will be sufficient to execute C safely.

Example 2.4. Suppose we have the following specification:

$$\{\mathtt{x} \mapsto v'_1 * \mathtt{y} \mapsto v'_2\}\ \texttt{\textbf{if} (*y==0) *x = 42}\ \{\mathtt{x} \mapsto v'_3 * \mathtt{y} \mapsto v'_4\}$$

With assertion $S = \mathtt{x} \mapsto 5 * \mathtt{z} \mapsto 7$ and desired precondition $P = \mathtt{x} \mapsto v'_1 * \mathtt{y} \mapsto v'_2$, abduction will obtain the antiframe $A_? = \mathtt{y} \mapsto v'$. The combined assertion $\mathtt{x} \mapsto 5 * \mathtt{z} \mapsto 7 * \mathtt{y} \mapsto v'$ suffices to execute the command safely.

The tight interpretation of specifications is necessary for this kind of reasoning. Because a specification must describe all the resources affected by a thread, any resources in the frame must be unaccessed. Conversely, if we calculate the antiframe $S * [A_?] \vdash P$, it must be the case that before executing C, we must first acquire the additional resource $A_?$ (as well as S).

Redundant and needed resources. In a separation logic proof, an assertion at a particular program point may describe resources that are not needed until much later in the program (or that are not needed by any subsequent command). Different commands access different resources, but the tight interpretation means that each assertion expresses all the resources used by the program at any point during its execution (aside from those gained and lost by resource transfer). Thus at a given point in the program, many resources that are represented in the proof will be redundant.

Resources that become redundant will not be accessed by the current thread, and thus can be accessed by a parallel thread without generating data races.[5] Redundant resources can be seen as over-approximating lack of dependence: a resource that is redundant at a particular point in a thread cannot subsequently generate dependencies between it and other threads. Our parallelisation analysis uses the input separation logic proof to identify resources that are definitely redundant, and uses `grant()` and `wait()` signalling to transfer them to the logically next point at which they are needed.

Thus our analysis is likely to be most effective for programs with complex specifications encompassing many resources, and with programs which modify resources in very few places, and where the resources accessed change over the course of the program. The `move()` example we discuss below in §2.4.2 has precisely these characteristics.

We use frame inference to determine *redundant resources*—resources that will not be accessed in a particular portion of the program, and which can thus be safely transferred to other threads. Conversely, we use abduction to determine *needed resources*—resources that must be held for a particular portion of the program to complete (in that some control flow path uses them), and which must thus be acquired before the program can proceed safely.

In our analysis, we calculate the redundant resource from the current program point to the end of the current parallelised program segment. This resource can be transferred to the logically next program segment with a `grant()` barrier. We calculate the needed resource from the start of current segment to the current program point. This resource

[5]Similarly, for garbage collection, only live data needs to be preserved and dead data can be collected early [ASKM14].

must be acquired from the logically preceding segment using a `wait()` barrier before execution can proceed further.

Note that these two resources need not be disjoint. A resource may be used early in a segment, in which case it will be both needed up to the current point, and redundant from the current point, or e.g., it may become redundant due to being used in only one of the branches of an if-then-else block. Note also that redundant and needed resources need not cover the entire state—some resource described in the proof may never be accessed or modified by the program.

2.3 Algorithm overview

We assume the user supplies a sequential program with parallelisable code segments identified, as well as a sequential proof written in separation logic establishing relevant memory safety properties of the program. The high-level structure of our algorithm is as follows:

1. A *resource usage analysis* uses abduction and frame inference to discover redundant and needed resources for different portions of the program.
2. A *parallelising transformation* consisting of the following three parts leverages this information to construct a parallelised program and associated separation logic proof of correctness:
   (i) The *parallelisation* phase converts the sequential program into a concurrent program by replacing all the identified parallelisable segments with parallel threads.
   (ii) The *resource matching* phase matches redundant resources in each parallel thread with needed resources in the logically following parallel thread.
   (iii) The *barrier insertion* phase modifies each parallel thread to insert `grant()` and `wait()` barriers consistent with the discovered resource transfer.

Algorithm properties. Our algorithm does not guarantee optimal parallelisation; there may be programs that it does not find which are more parallel. However, the resulting parallelised program is guaranteed to be race free and memory safe, and the analysis constructs a separation logic proof that the parallelised program satisfies the same safety specification as the original sequential program. This new proof differs from the proof of the original sequential proof, reflecting parallelisation and injected synchronisation. If the original sequential program is terminating and does not dispose memory, then parallelisation is behaviour-preserving: every input-output behaviour of the parallelised program is also a behaviour of the original sequential program.

2.3.1 Resource usage analysis

Consider again the program that we introduced in Example 2.1 and the following specification for `f()`, covering any input value `i`:

$$\{\mathtt{x} \mapsto a * \mathtt{y} \mapsto b\} \quad \mathtt{f(i)} \quad \{(a \geq \mathtt{i} \wedge \mathtt{x} \mapsto a * \mathtt{y} \mapsto a) \vee (a < \mathtt{i} \wedge \mathtt{x} \mapsto 0 * \mathtt{y} \mapsto b)\}$$

```
1  $\{x \mapsto a * y \mapsto b\}$
2  void f(i) {
3    int v = *x;
4   $\{v = a \wedge x \mapsto a * y \mapsto b\}$
5    if (v >= i) {
6     $\{v = a \wedge v \geq i \wedge x \mapsto a * y \mapsto b\}$
7      *y = v;
8    }
9    else {
10    $\{v = a \wedge v < i \wedge x \mapsto a * y \mapsto b\}$
11     *x = 0;
12   }
13 }
14 $\{(a \geq i \wedge x \mapsto a * y \mapsto a) \vee (a < i \wedge x \mapsto 0 * y \mapsto b)\}$
```

Figure 2.2: Separation logic proof of the function `f()`.

The precondition of this specification says that `f(i)` accesses two distinct memory locations, `x` and `y`. As is standard in Hoare logic, we use auxiliary variables a and b to record the values stored in `x` and `y`. The postcondition gives two possibilities: if a is greater than `i`, then both `x` is unmodified and `y` are set to a; otherwise, `x` is set to 0 and `y` unmodified. Fig. 2.2 shows a proof of this specification.

Needed resources. At the core of the analysis is computing *needed* resources for program segments induced by pairs of program points.

Definition 2.2. The needed resource for a particular program interval is the resource that must be held to execute from the start of the interval to the end without faulting.

Needed resources are used to work out when `wait`-barriers must be injected. Resources must be acquired before they are used in a program segment. Thus, needed resources are calculated *from* the start of the segment, *to* every other program point (such calculation constitutes a forward program analysis).

Example 2.5. In our example, `wait()` barriers are injected into the second call to `f()`. These needed resources are shown in Fig. 2.3. For example, the resource calculated for the start of the **`else`**-branch (Fig. 2.3, line 11) is:

$$a < \mathtt{i} \wedge \mathtt{x} \mapsto a$$

That is, to reach this program point from the start of `f()`, the thread only requires access to the heap cell `x`, provided $a <$ `i`. Access to `x` is needed because it is dereferenced in line 3.

Redundant resources. To calculate where `grant()` barriers can be inserted, the analysis computes *redundant* resources for particular program intervals.

```
n:{emp}
void f(i) {
  int v = *x;
  n:{x ↦ a}
  if (v >= i) {
    n:{a ≥ i ∧ x ↦ a}
    *y = v;
    n:{a ≥ i ∧ x ↦ a ∗ y ↦ b}
  }
  else {
    n:{a < i ∧ x ↦ a}
    *x = 0;
    n:{a < i ∧ x ↦ a}
  }
}
n:{(a ≥ i ∧ x ↦ a ∗ y ↦ b) ∨ (a < i ∧ x ↦ a)}
```

Figure 2.3: Needed resources, indicated by the prefix `n`, from the start of `f()` to all other program points.

Definition 2.3. The redundant resource in a program interval is the portion of the currently-held resource that is *not* needed to execute from the start to the end of the interval.

Resources that are redundant to the end of the segment can be transferred to logically-later program segments using `grant()`. Consequently, the redundant resource is calculated *from* each program point, *to* the end of the segment (such calculation, in a way, constitutes a backward program analysis).

To calculate a redundant resource, the analysis first calculates the needed resource for the interval. It then subtracts using frame inference the needed resource from the resource described in the sequential proof.

Example 2.6. The redundant resources, along with the needed resources used to calculate them, for the first call to `f()` are shown in Fig. 2.4. For example, the needed resource at the start of the **`else`**-branch (Fig. 2.4, line 11) is:

$$\mathtt{x} \mapsto a$$

According to the sequential proof in Fig. 2.2, such a thread *actually* holds the following resource at this point:

$$\mathtt{v} = a \land \mathtt{v} < \mathtt{i} \land \mathtt{x} \mapsto a * \mathtt{y} \mapsto b$$

To calculate the redundant resource, we pose the following frame inference question, computing the frame $F_?$.

$$\mathtt{v} = a \land \mathtt{v} < \mathtt{i} \land \mathtt{x} \mapsto a * \mathtt{y} \mapsto b \quad \vdash \quad \mathtt{x} \mapsto a * [F_?]$$

The resulting redundant resource is as follows:

$$F_? : \quad \mathtt{v} < \mathtt{i} \land \mathtt{y} \mapsto b$$

```
n: $\{\mathtt{x} \mapsto a * (a \geq \mathtt{i} \land \mathtt{y} \mapsto b)\}$
r: $\{\mathsf{emp}\}$
void f(i) {
  int v = *x;
  n: $\{(\mathtt{v} \geq \mathtt{i} \land \mathtt{y} \mapsto b) \lor (\mathtt{v} < \mathtt{i} \land \mathtt{x} \mapsto a)\}$
  r: $\{(\mathtt{v} = a \land \mathtt{v} \geq \mathtt{i} \land \mathtt{x} \mapsto a) \lor (\mathtt{v} < i \land \mathtt{y} \mapsto b)\}$
  if (v >= i) {
    n: $\{\mathtt{y} \mapsto b\}$
    r: $\{\mathtt{v} = a \land \mathtt{v} \geq \mathtt{i} \land \mathtt{x} \mapsto a\}$
    *y = v;
    n: $\{\mathsf{emp}\}$
    r: $\{\mathtt{v} = a \land \mathtt{v} \geq \mathtt{i} \land \mathtt{x} \mapsto a * \mathtt{y} \mapsto b\}$
  }
  else {
    n: $\{\mathtt{x} \mapsto a\}$
    r: $\{\mathtt{v} < \mathtt{i} \land \mathtt{y} \mapsto b\}$
    *x = 0;
    n: $\{\mathsf{emp}\}$
    r: $\{\mathtt{v} = a \land \mathtt{v} < \mathtt{i} \land \mathtt{x} \mapsto a * \mathtt{y} \mapsto b\}$
  }
}
n: $\{\mathsf{emp}\}$
r: $\{\mathtt{x} \mapsto a * \mathtt{y} \mapsto b\}$
```

Figure 2.4: Redundant resources from all program points to the end of `f()`, along with the needed resources used to calculate them. Needed resources are prefixed with `n` and redundant resources are prefixed with `r`.

In other words, the thread no longer requires access to the heap cell `y` beyond line 11 in `f()`, and we also know that $v < i$. Note however, that we do not know that `v` is the same as the value pointed to by `x`, because this does not necessarily hold at every following point in the segment.

2.3.2 Parallelising transformation

As described above, the parallelising transformation has three steps. We now describe how these steps are applied in parallelisation of our example 2.1.

Step (i). First, the program is divided into two parallel threads.

```
f₁(1) || f₂(2)
```

The transformation specialises `f()` into distinct functions $\texttt{f}_1\texttt{()}$ and $\texttt{f}_2\texttt{()}$ because each thread has different synchronisation.

Step (ii). Next, the transformation looks for needed resources in the second thread $\texttt{f}_2\texttt{()}$, and matches them against redundant resources in the first thread $\texttt{f}_1\texttt{()}$. In this case, the transformation identifies two distinct needed resources, nx and ny:

$$\mathsf{nx}\colon x \mapsto v'_1 \qquad\qquad \mathsf{ny}\colon y \mapsto v'_2$$

Resource nx becomes needed in $\texttt{f}_2\texttt{()}$ at line 3. It becomes redundant in $\texttt{f}_1\texttt{()}$ at lines 6 and 13 of Fig 2.4. Resource ny becomes needed at lines 6 and 13. It becomes redundant at lines 8 and 11.

Step (iii). The final stage of the transformation is to inject synchronisation corresponding to these discovered resources. Specifically, `grant`-barriers are injected at the points that the resources become redundant, while `wait`-barriers are injected at the points the resources become needed. Synthesising barriers follows a four-step process:

1. Identify a needed resource r at some point in the later segment. This point may be chosen with assistance from the programmer, or heuristically, for example by favouring bigger needed resources – see §5.2 for a discussion.

2. Down all other control-flow paths in the later segment, find a program point such that the needed resources is covered by r. This step ensures that the identified resources r is received down all possible paths in the program.

3. Down all control-flow paths in the earlier segment, find a program point such that the redundant resource covers r. This step ensures that the resource r is satisfied down all possible paths in the program.

4. Synthesise a channel name and insert `wait()` into the later segment, and `grant()` into the earlier segment, at the identified program points.

After each iteration of this process, the needed and redundant resources are updated to reflect the new synchronisation. The process terminates when no needed resources remain.

The parallelised version of `f()` is shown in Fig. 2.1. The needed resources associated with barriers are selected heuristically. The transformation could also safely choose a single needed resource:

$$x \mapsto v'_1 * y \mapsto v'_2$$

This resource "overapproximates" (i.e., entails with a non-empty frame) both nx and ny. However, this resource would need to be acquired at a dominating program point for nx and ny, which would unfortunately sequentialise the program by forcing the `wait`-barrier to the start of the second thread.

As there are only two heap locations, it is clear that nx and ny are a sensible choice of needed resource. Selecting needed resources becomes more tricky when resources are dynamic structures, such as linked lists (see §2.4.2).

2.4 Generalising the method

In the previous section we informally presented the core of our method. In this section, we discuss how it can be generalised to handle an unbounded number of threads (§2.4.1), and dynamic (heap-allocated) data structures (§2.4.2).

2.4.1 Unbounded iterations: parallel-for

Until this point we have assumed that the programmer identifies a fixed number of static program segments for parallelisation—in Example 2.1, these were the two calls to `f()`. However, our method can support unbounded segments running in parallel. To do this, we introduce the construct **`pfor`** (*parallel-for*) for annotating the sequential code. The intended semantics of **`pfor`** is identical to a standard **`for`**-loop, but in which all iterations may run in parallel, with our method injecting the synchronisation across iterations as required.

Example 2.7. Let us consider the following program:

```
pfor (i=1; i++; i<=n) {
  f(i);
}
```

where `f()` is as in Example 2.1. The externally-visible input-output behaviour of this program should be the same as if we did the sequential composition:

```
f(1); f(2); f(3); ... f(n);
```

Internally, however, each call to `f()` may run concurrently.

Parallelisation. Fig. 2.5 shows the parallelisation of the **`pfor`**-annotated program from example 2.7, as realised by our method. A **`pfor`** is translated to a sequential **`for`** loop, in which each iteration forks a new copy of the parallelised loop body. Resources are transferred between the threads in order of thread-creation. That is, the nth iteration of the **`pfor`** acquires resources from the logically-preceding $(n-1)$th iteration, and releases resources to the logically-succeeding $(n+1)$th iteration.

This ordering is implemented through *shared channels*; a thread shares with its predecessor and successor a set of channels for receiving (i.e., waiting) and sending (i.e., granting) resources, resp. The number of threads—and so, the required number of channels—is potentially decided at run-time. Consequently, channels are dynamically allocated in the main **`for`**-loop using the `newchan()` operation [DJP11]. Each iteration creates a set of new channels, and passes the prior and new set to the forked thread.

```
chan wx = newchan();
chan wy = newchan();
chan wx', wy';
grant(wx);
grant(wy);
for (i=1; i++; i<=n) {
  wx' = newchan();
  wy' = newchan();
  fork(f(i,wx,wy,wx',wy'));
  wx = wx';
  wy = wy';
}
wait(wx);
wait(wy);
```

```
f(i, wxp, wyp, wx, wy){
  wait(wxp);
  int v = *x;
  if (v >= i) {
    grant(wx);
    wait(wyp);
    *y = v;
    grant(wy);
  }
  else {
    wait(wyp);
    grant(wy);
    *x = 0;
    grant(wx);
  }
}
```

Figure 2.5: Parallelisation of `f()`, generalised to deal with unbounded iteration. Global variables `x` and `y` point to distinct memory locations.

The parallelised version of `f()` (Fig. 2.5) takes four channel arguments, a pair of each for `x` and `y`. The prior channels are used for resource transfer with the logically-preceding thread (here `wx`, `wy`), and the new channels are used to communicate resource transfer with the logically-following thread (here, `wx'`, `wy'`). As each iteration calls both `wait()` and `grant()`, to receive and send resources, the injected synchronisation in Fig. 2.5 merges the synchronisation from both the first and second threads in Fig. 2.1.

2.4.2 Loops and dynamic data structures

Up to this point, we have dealt with injecting channels transferring single heap locations into straight-line code. Our method can in fact handle more complicated heap-allocated data structures such as linked lists, and control-flow constructs such as loops.

Example 2.8. To illustrate, consider the function `move()`, shown in Fig. 2.6. This function searches a non-empty linked list `s` for the node at position `l` in the list and destructively moves that node to the end of the list.

We assume that the following specification of `move()` holds:

$$\{\mathsf{node}(\mathtt{s}, n_1') * \mathsf{node}(n_1', n_2') * \mathsf{lseg}(n_2', \mathsf{null})\}$$

```
move(s, l)
```

$$\{\mathsf{node}(\mathtt{s}, n_3') * \mathsf{node}(n_3', n_4') * \mathsf{lseg}(n_4', \mathsf{null})\}$$

This specification is simple: all it says is that executing `move()` on a list of length two or more results in a list of length two or more. A proof of this specification is given in Fig. 2.7.

An important feature of our approach is that the input safety proof need not specify all the relevant sequential properties (e.g. it does not specify that the node moved to the end of the list was previously at position `l`); all sequential dependencies are still

```
move(s, l) {
  int i = 0;
  node x = s.nxt;
  node p = s;
  while (i<l && x.nxt!=null) {
    i++;
    p = x;
    x = x.nxt;
  }
  while (x.nxt!=null)
    x = x.nxt;
  if (p.nxt!=x) {
    node tmp = p.nxt.nxt;
    p.nxt.nxt = null;
    x.nxt = p.nxt;
    p.nxt = tmp;
  }
}
```

Figure 2.6: Example function `move()`, a list-manipulating program whose automated parallelisation requires reasoning over complex assertions and predicates.

enforced in the parallelised program. Thus we can view this example program a s representative of a class of algorithms and implementations that traverse a list from its head, then mutate the tail. With minor modifications, the same proof pattern would cover assigning to the values in the list, or sorting the tail of the list.

We consider the parallelisation of a sequential program consisting of pair of calls to `move()`:

```
move(s,a); move(s,b);
```

The parallelised version of this program consists of two parallel calls to modified versions of `move`:

$\texttt{move}_1\texttt{(s,a)}\ ||\ \texttt{move}_2\texttt{(s,b)}$

(We could also parallelise `move()` to deal with unbounded iterations (as in §2.4.1), but we choose to simplify the exposition at this stage. The formal description of our method in Chapter 4 describes the general case.)

The `wait()` and `grant()` barriers in $\texttt{move}_1\texttt{()}$ and $\texttt{move}_2\texttt{()}$ must enforce the following properties:

- $\texttt{move}_2\texttt{()}$ must not write to a list node until $\texttt{move}_1\texttt{()}$ has finished both reading from and writing to it. Consequently, $\texttt{move}_2\texttt{()}$ must wait for $\texttt{move}_1\texttt{()}$ to finish traversing a segment of the list before moving a node inside the segment.
- $\texttt{move}_2\texttt{()}$ must not read from a list node until $\texttt{move}_1\texttt{()}$ has finished writing to it. Consequently, $\texttt{move}_2\texttt{()}$ must wait for $\texttt{move}_1\texttt{()}$ to finish moving a node before it traverses over that point in the list.

This example is substantially more subtle to analyse and parallelise than our earlier one, because the list is not divided into segments that can be discovered from the

```
$\{\mathsf{node}(\mathtt{s}, n'_1) * \mathsf{node}(n'_1, n'_2) * \mathsf{lseg}(n'_2, \mathsf{null})\}$
move(s, l) {
  int i = 0;
  node x = s.nxt;
  node p = s;
  $\{\mathsf{node}(\mathtt{s}, \mathtt{x}) * \mathsf{node}(\mathtt{x}, n') * \mathsf{lseg}(n', \mathsf{null}) \wedge \mathtt{s} = \mathtt{p}\}$
  while (i<l && x.nxt!=null) {
    $\{\mathsf{lseg}(\mathtt{s}, \mathtt{p}) * \mathsf{node}(\mathtt{p}, \mathtt{x}) * \mathsf{node}(\mathtt{x}, n') * \mathsf{lseg}(n', \mathsf{null})\}$
    i++;
    p = x;
    x = x.nxt;
  }
  $\{\mathsf{lseg}(\mathtt{s}, \mathtt{p}) * \mathsf{node}(\mathtt{p}, \mathtt{x}) * \mathsf{node}(\mathtt{x}, n') * \mathsf{lseg}(n', \mathsf{null})\}$
  while (x.nxt!=null) {
    $\{\mathsf{lseg}(\mathtt{s}, \mathtt{p}) * \mathsf{node}(\mathtt{p}, n'_1) * \mathsf{lseg}(n'_1, \mathtt{x}) * \mathsf{node}(\mathtt{x}, n'_2) * \mathsf{lseg}(n'_2, \mathsf{null})\}$
    x = x.nxt;
  }
  $\{\mathsf{lseg}(\mathtt{s}, \mathtt{p}) * \mathsf{node}(\mathtt{p}, n') * \mathsf{lseg}(n', \mathtt{x}) * \mathsf{node}(\mathtt{x}, \mathsf{null})\}$
  if (p.nxt!=x) {
    $\{\mathsf{lseg}(\mathtt{s}, \mathtt{p}) * \mathsf{node}(\mathtt{p}, n'_1) * \mathsf{node}(n'_1, n'_2) * \mathsf{lseg}(n'_2, \mathtt{x}) * \mathsf{node}(\mathtt{x}, \mathsf{null})\}$
    node tmp = p.nxt.nxt;
    p.nxt.nxt = null;
    x.nxt = p.nxt;
    p.nxt = tmp;
    $\{\mathsf{lseg}(\mathtt{s}, \mathtt{p}) * \mathsf{node}(\mathtt{p}, \mathtt{tmp}) * \mathsf{lseg}(\mathtt{tmp}, \mathtt{x}) * \mathsf{node}(\mathtt{x}, n') * \mathsf{node}(n', \mathsf{null})\}$
  }
}
$\{\mathsf{node}(\mathtt{s}, n'_1) * \mathsf{node}(n'_1, n'_2) * \mathsf{lseg}(n'_2, \mathsf{null})\}$
```

Figure 2.7: Sequential separation logic proof of `move()`.

```
move1(s, l) {
  int i = 0;
  node x = s.nxt;
  node p = s;
  while (i<l && x.nxt!=null) {
    i++;
    p = x;
    x = x.nxt;
  }
  *pr = p;
  grant(i1);
  while (x.nxt!=null)
    x = x.nxt;
  if (p.nxt!=x) {
    ... // omitted
  }
  grant(i2);
}
```

```
move2(s, l) {
  int i = 0;
  wait(i1);
  if (s==*pr || s.nxt==*pr) wait(i2);
  node x = s.nxt;
  node p = s;
  while (i<l && x.nxt!=null) {
    if (x.nxt==*pr) wait(i2);
    i++;
    p = x;
    x = x.nxt;
  }
  while (x.nxt!=null) {
    if (x.nxt==*pr) wait(i2);
    x = x.nxt;
  }
  if (p.nxt!=x) {
    ... // omitted
  }
}
```

Figure 2.8: Parallelisation of `move()` for two threads.

static structure of the program. (A simple points-to analysis would be insufficient to discover the salient partitioning, for example.) In the worst case, a `wait()` at the start of move$_2$() and a `grant()` at the end of move$_1$() enforces sequential order. However, by reasoning about the structure of the manipulated list using the safety proof given in Fig. 2.7, our approach can do considerably better.

The parallelised program synthesised by our algorithm is shown in Fig. 2.8. This parallelisation divides the list into two segments, consisting of the portions read and modified by move$_1$(). A shared location `pr`, introduced by our algorithm, stores the address of the starting node of the portion modified by move$_1$(). The thread move$_2$() uses `pr` to synchronise access to the second segment of the list. We discuss how the analysis generates `pr` below, under "Value materialisation".

Handling dynamic structures means dealing with allocation and disposal. Fortunately, separation logic handles both straightforwardly. As already argued, updates to the data structure and object allocation are reflected in the invariants of the original sequential proof. Thus updates and allocations are also reflected in the invariants which our analysis constructs to represent the contents of channels. However, introducing allocation and disposal affects the behaviour-preservation result discussed in §4.4; this result ensures the program behaviour is unaffected by the translation (i.e., the translation enforces deterministic parallelism).

Handling list segments. We run our resource-usage analysis over the program to determine redundant and needed resources. The results of the analysis are shown in Fig. 2.9.

Consider the program point in Fig. 2.9 just before the start of the second while loop. Our analysis discovers that only the resource $\mathsf{lseg}(\mathsf{p}, \mathsf{null})$ is needed to execute

```
move$_1$(s, l) {
  $\mathsf{r}: \{\mathsf{emp}\}$
  int i = 0;
  node x = s.nxt;
  node p = s;
  while (i<l && x.nxt!=null) {
    $\mathsf{r}: \{\mathsf{lseg}(\mathtt{s}, \mathtt{p})\}$
    i++;
    p = x;
    x = x.nxt;
  }
  $\mathsf{r}: \{\mathsf{lseg}(\mathtt{s}, \mathtt{p})\}$
  while (x.nxt!=null) {
    $\mathsf{r}: \{\mathsf{lseg}(\mathtt{s}, \mathtt{p})\}$
    x = x.nxt;
  }
  $\mathsf{r}: \{\mathsf{lseg}(\mathtt{s}, \mathtt{p})\}$
  if (p.nxt!=x) {
    ... // omitted
  }
}
$\mathsf{r}: \{\mathsf{node}(\mathtt{s}, n_1') * \mathsf{node}(n_1', n_2') * \mathsf{lseg}(n_2', \mathsf{null})\}$
```

```
move$_2$(s, l) {
  int i = 0;
  node x = s.nxt;
  $\mathsf{n}: \{\mathsf{node}(\mathtt{s}, n')\}$
  node p = s;
  while (i<l && x.nxt!=null) {
    $\mathsf{n}: \{\mathsf{lseg}(\mathtt{s}, n_1') * \mathsf{node}(n_1', \mathtt{x}) * \mathsf{node}(\mathtt{x}, n_2')\}$
    i++;
    p=x;
    x=x.nxt;
  }
  $\mathsf{n}: \{\mathsf{lseg}(\mathtt{s}, n_1') * \mathsf{node}(n_1', \mathtt{x}) * \mathsf{node}(\mathtt{x}, n_2')\}$
  while (x.nxt!=null) {
    $\mathsf{n}: \{\mathsf{lseg}(\mathtt{s}, n_1') * \mathsf{node}(n_1', \mathtt{x}) * \mathsf{node}(\mathtt{x}, n_2')\}$
    x=x.nxt;
  }
  $\mathsf{n}: \{\mathsf{lseg}(\mathtt{s}, n') * \mathsf{node}(n', \mathtt{x}) * \mathsf{node}(\mathtt{x}, \mathsf{null})\}$
  if (p.nxt!=x) {
    ... // omitted
  }
}
$\mathsf{n}: \{\mathsf{lseg}(\mathtt{s}, n') * \mathsf{node}(n', \mathtt{x}) * \mathsf{node}(\mathtt{x}, \mathsf{null})\}$
```

Figure 2.9: Left: redundant resources (from the indicated program points to the end) for `move()`. Right: needed resources (from start to the indicated other program points) for `move()`.

from the start of this loop to the end of the function. Comparing this resource to the corresponding invariant in the sequential proof (Fig. 2.7) reveals that the resource $\mathsf{lseg}(\mathtt{s},\mathtt{p})$ is redundant at this point. This assertion represents the segment of the list that has already been traversed by `move₁()`, from the head of the list to `p`.

Injecting barriers. Barriers `grant()` and `wait()` are injected into `move()` as discussed above, in §2.3.2. Barriers `wait()` and `grant()` should only be called once for any given channel in the program. For simplicity of exposition, in Fig. 2.8, we have rewritten `move₂()` using a `wait()` controlled by conditionals. However, in general this approach requires us to modify loop invariants, which is often difficult. In the formal definition of the parallelising transformation, we only inject barriers into loop-free code, and perform syntactic loop-splitting to expose points in loops where barriers can be called conditionally. Details are given in §4.2.4.

Value materialisation. In order to safely transfer a redundant resource to logically later program segments we need the assertion to be expressed in global variables, and to be invariant from the point in time that the resource is released, to the point it is received in the subsequent thread. The assertion $\mathsf{lseg}(\mathtt{s},\mathtt{p})$ initially generated by the analysis satisfies neither condition, because it is partly expressed in terms of the local variable `p`, which changes during execution of `move₁()`. The current thread may invalidate the assertion by changing the value stored in `p`.

To satisfy this requirement, we could simply existentially quantify the offending variable, `p`, giving the redundant resource

$$\exists y.\ \mathsf{lseg}(\mathtt{s}, y)$$

However, such a weakening loses important information, in particular the relationship between the resource, and the list tail beginning at `p`. To retain such dependency relationships, our parallelisation algorithm stores the current value of `p` into a global location `pr` shared between `move₁()` and `move₂()`. (We call storing a snapshot of a local value in this way *materialisation*). An assignment is injected into `move₁()` at the start of the second loop:

```
...
*pr = p;
while (x.nxt!=null)
...
```

After the assignment, the proof needs to be modified to accommodate the new global location `pr`. In this case, this amounts to simply modifying the invariant to add an extra points-to assertion representing the new location:

$$\mathsf{lseg}(\mathtt{s},\mathtt{p}) * \mathsf{node}(\mathtt{p},\mathtt{x}) * \mathsf{node}(\mathtt{x},n') * \mathsf{lseg}(n',\mathsf{null}) * \mathtt{pr} \mapsto \mathtt{p}$$

The redundant state just after the assignment `*pr = p` can now be described as follows:

$$\mathsf{lseg}(\mathtt{s},y') * \mathtt{pr} \overset{1/2}{\longmapsto} y'$$

The assertion `pr` $\overset{1/2}{\longmapsto} y'$ represents fractional, read-only permission on the shared location `pr`. This binds together the head of the list and the remainder of the list when they are recombined, and thus helps preserve the list invariant.

When traversing the list, $\texttt{move}_2\texttt{()}$ compares its current position with `pr`, which represents the boundary of the list already processed by $\texttt{move}_1\texttt{()}$. If it reaches the pointer stored in `pr`, it must wait to receive the second, remainder segment of the list from $\texttt{move}_1\texttt{()}$.

CHAPTER 3

TECHNICAL BACKGROUND

In the previous chapter we gave an overview of our proof-directed parallelisation method. In this and the next chapter we turn to a formal description of the method. We start with a technical background about the programming language (§3.1) we use in the formalisation, followed by our assertion language and automated reasoning therein (§3.2), and finish with the representation of the sequential proof (§3.3). This chapter is not a new research, but presents the relevant concepts coherently from the literature.

3.1 Programming language and program representation

We present the formalisation of our parallelisation method for programs written in a simple heap-manipulating language without procedures[1], defined by:

$$\begin{array}{rcll} e & ::= & \mathtt{x} \mid \mathtt{nil} \mid t(\bar{e}) \mid \ldots & \textit{(expressions)} \\ b & ::= & \mathtt{true} \mid \mathtt{false} \mid e = e \mid e \neq e \mid \ldots & \textit{(booleans)} \\ a & ::= & \mathtt{x} := e \mid \mathtt{x} := *e \mid *\mathtt{x} := e \mid \mathtt{x} := \mathtt{alloc}() \mid \ldots & \textit{(primitive commands)} \\ C & ::= & C; C \mid \ell\colon \mathtt{skip} \mid \ell\colon a \mid \ell\colon \mathtt{if}(b)\,\{\,C\,\}\,\mathtt{else}\,\{\,C\,\} \mid \ell\colon \mathtt{while}(b)\,\{\,C\,\} & \end{array}$$

, where t ranges over constructors of pure expressions such as $e + e$, $e - e$, etc. To avoid an extra construct, we define $\mathtt{for}(C_1;C_2;b)\{C_3\}$ as $C_1;\mathtt{while}(b)\{C_2;C_3\}$. The choice of primitive commands is given for illustrative purposes; our method is independent of this choice. We postpone the formal operational semantics of the language to §6.1 as it is not crucial for the development of the method.

We formalise our method on an example program structure using the multi-iteration construct **pfor**, which we introduced in §2.4.1. For clarity of the exposition, throughout §3 and §4 we assume a fixed input program $\mathbb{P}$ of the following form:

```
global r̄;              main_ℙ(void) {
work(x̄) {                pfor(C1; C2; b) {
  local ȳ;                 work(ē);
  C;                     }
}                      }
```

[1]An extension involving procedures is considered in §5.4.

```
void f(i) {
ℓ1:  int v = *x;
ℓ2:  if (v >= i) {
ℓ3:  *y = v;
    }
    else {
ℓ4:  *x = 0;
    }
}
```

$$\begin{aligned}
\mathtt{f_s}&: \mathtt{x} \mapsto a * \mathtt{y} \mapsto b \\
\ell_1&: \mathtt{x} \mapsto a * \mathtt{y} \mapsto b \\
\ell_2&: \mathtt{v} = a \land \mathtt{x} \mapsto a * \mathtt{y} \mapsto b \\
\ell_2\ell_3&: \mathtt{v} = a \land \mathtt{v} \geq \mathtt{i} \land \mathtt{x} \mapsto a * \mathtt{y} \mapsto b \\
\ell_2(\ell_2)^{\mathtt{t}}_{\mathtt{e}}&: \mathtt{v} = a \land \mathtt{v} \geq \mathtt{i} \land \mathtt{x} \mapsto a * \mathtt{y} \mapsto a \\
\ell_2\ell_4&: \mathtt{v} = a \land \mathtt{v} < \mathtt{i} \land \mathtt{x} \mapsto a * \mathtt{y} \mapsto b \\
\ell_2(\ell_4)^{\mathtt{f}}_{\mathtt{e}}&: \mathtt{v} = a \land \mathtt{v} < \mathtt{i} \land \mathtt{x} \mapsto 0 * \mathtt{y} \mapsto b \\
\mathtt{f_e}&: (a \geq \mathtt{i} \land \mathtt{x} \mapsto a * \mathtt{y} \mapsto a) \lor \\
&\quad (a < \mathtt{i} \land \mathtt{x} \mapsto 0 * \mathtt{y} \mapsto b)
\end{aligned}$$

Figure 3.1: Left: labels for commands in function `f()` from §2. Right: associated assertions in the sequential proof of `f()` from Fig. 2.2.

where `global` and `local` indicate the scope of variables used in C, C_1, C_2, b and $\bar{e}$, and `work` embodies the work performed iteratively by the program, exposed for potential parallelisation. Our parallelisation method generates a new program $\mathbb{P}_{par}$ that allows safe execution of a transformed version of `work` in separate threads.

Labelling of commands. We assume that every command in C is indexed by a unique label $\ell \in \mathsf{Label}$; the entry and the exit point of `work` are also treated as labels. We identify a particular command by its label, and when needed denote by $\mathsf{cmd}(\ell)$ the command at the label ℓ. Functions pred and succ: $\mathsf{Label} \rightharpoonup \mathsf{Label}$ return the label of the previous (the next, resp.) command in a block of sequential compositions. For a label ℓ corresponding to a `while` loop, the predecessor of the first command and successor of the last command in the block are denoted by $\ell_{\mathtt{s}}$ and $\ell_{\mathtt{e}}$, respectively. For ℓ corresponding to `if-else`, $\ell^{\mathtt{t}}_{\mathtt{s}}$ ($\ell^{\mathtt{t}}_{\mathtt{e}}$) and $\ell^{\mathtt{f}}_{\mathtt{s}}$ ($\ell^{\mathtt{f}}_{\mathtt{e}}$) are labels of the predecessor of the first (the successor of the last) commands in the `if` and `else` branches, respectively.

Example 3.1. The left-hand side of Fig. 3.1 shows a labelling of the function `f()` from Example 2.1 in §2.

Program representation. We assume that the program's `work` function is represented by a variant of abstract syntax tree (formally, an ordered forest) $\mathcal{T}$ with $\mathsf{Label} \cup \{\mathtt{work_s}, \mathtt{work_e}\}$ as the set of nodes and as the set of edges all pairs (ℓ, ℓ') where ℓ is a label of an `if-else` block and ℓ' a label of a command within the encompassed `if` or `else` branches. Labels ℓ and ℓ' that belong to the same block are siblings in the tree. They are ordered $\ell < \ell'$ if there exists $n \geq 1$ such that $\ell' = \mathsf{succ}^n(\ell)$. We denote by $\ell_\uparrow$ and $\ell_\downarrow$ the smallest (resp. largest) label in the block containing ℓ. We write $[\ell, \ell'\rangle$ to denote the sequence of labels between ℓ inclusively and ℓ' exclusively, and $\mathbb{P}_{[\ell,\ell'\rangle}$ for the corresponding program fragment from ℓ (inclusively) to ℓ' (exclusively).

Program paths. A *program path* is any finite non-empty sequence of nodes in the abstract syntax tree $\mathcal{T}$. Each program path determines a sequence of conditionals that need to be traversed in order to reach a particular point in the program. We denote the set of all program paths by Paths.

We often want to manipulate and compare program paths. For a program path $\gamma = \ell_1 \ldots \ell_n \in \mathsf{Paths}$, we write $\gamma[i]$ to denote ℓ_i, $\gamma[i..j]$ to denote $\ell_i \ldots \ell_j$ and $|\gamma|$ to denote the length n. For program paths γ and γ', $\gamma \curlywedge \gamma'$ denotes their longest common prefix, i.e., for $k = |\gamma \curlywedge \gamma'|$ we have $\forall j \leq k, \gamma \curlywedge \gamma' = \gamma[j] = \gamma'[j]$ and if $|\gamma|, |\gamma'| > k$ then $\gamma[k+1] \neq \gamma'[k+1]$. We define a partial order $\prec$ on Paths as follows. We say that $\gamma \prec \gamma'$ iff $\gamma \curlywedge \gamma' = \gamma$ and $|\gamma| < |\gamma'|$, or for $k = |\gamma \curlywedge \gamma'|$ we have $|\gamma|, |\gamma'| > k$ and $\gamma[k+1] < \gamma'[k+1]$. (Intuitively, two paths are related in this way if they share a prefix, and one takes predecessor branch to the other in the same block). We say that $\gamma \preceq \gamma'$ iff $\gamma \prec \gamma'$ or $\gamma = \gamma'$.

Lemma 3.1. $(\mathsf{Paths}, \preceq)$ is a lattice with least element $\mathtt{work_s}$ and greatest element $\mathtt{work_e}$.

For $\Gamma \subseteq \mathsf{Paths}$ we define $\max(\Gamma)$ as $\max(\Gamma) \triangleq \{\gamma \in \Gamma \mid \neg\exists\gamma' \in \Gamma \,.\, \gamma \prec \gamma'\}$. We define $\min(\Gamma)$ analogously. For $\Gamma, \Gamma' \subseteq \mathsf{Paths}$ we write $\Gamma \preceq \Gamma'$ if for all $\gamma \in \Gamma$, there exists $\gamma' \in \Gamma'$ such that $\gamma \preceq \gamma'$.

3.2 Assertion language

We assume that the assertions in the program proof are expressed using a class of separation logic formulae called *symbolic heaps*.

Definition 3.2. A symbolic heap Δ is a formula of the form $\exists\bar{x}\,.\,\Pi \wedge \Sigma$ where Π (the pure part) and Σ (the spatial part) are defined by:

$$\begin{array}{rcl} \Pi & ::= & \mathsf{true} \mid e = e \mid e \neq e \mid p(\bar{e}) \mid \Pi \wedge \Pi \\ \Sigma & ::= & \mathsf{emp} \mid s(\bar{e}) \mid \Sigma * \Sigma \end{array}$$

Here $\bar{x}$ are auxiliary (logical) variables, e ranges over expressions, $p(\bar{e})$ is a family of pure (first-order) predicates (such as e.g., arithmetic inequalities, etc), and $s(\bar{e})$ a family of spatial predicates (such as e.g., points-to $x \mapsto e$, singly-linked list $\mathsf{lseg}(e, e)$, doubly-linked lists, trees, etc). For readability, the existential quantification of auxiliary variables in Δ is usually omitted.

We refer to the pure and the spatial part of an assertion Δ as Δ^{Π} and Δ^{Σ} respectively. We denote by $\Pi_{\vee}$ the set of all quantifier-free first-order formulae built in the same way as Π but also allowing the $\vee$ connective.

We often need to substitute auxiliary variables in symbolic heaps, for example when recasting an assertion into a different calling context. If $\varrho = \bar{x} \mapsto \bar{e}$ is a mapping from variables in $\bar{x}$ to $\bar{e}$ then $\Delta[\varrho]$ denotes the formula obtained by substituting every occurrence of x_i in Δ with the corresponding e_i. We denote by δ^{-1} the inverse variable mapping, if δ is injective[2].

We treat a disjunction of symbolic heaps as a set and interchangeably use the $\cup$ and $\vee$ operators. Such disjunctions will often arise in assertions at join points in the program. The set of all symbolic heaps is denoted by SH and the set of all disjunctive symbolic

[2]In our framework, substitutions are always guaranteed to be injective because the variables being substituted correspond to heap locations and channel resources whose denotations are guaranteed to be distinct; if the substitution involves values, then they must be implicitly existentially quantified, and can therefore be assumed to be distinct.

heaps by $\mathcal{P}(\mathsf{SH})$. We overload the $\wedge$ and $*$ operators in a natural way: for $\Delta_i = \Pi_i \wedge \Sigma_i$, $i = 1, 2$, and $\Delta' = \Pi' \wedge \Sigma'$, we define

$$\begin{aligned} \Delta_1 * \Delta_2 &\triangleq (\Pi_1 \wedge \Pi_2) \wedge (\Sigma_1 * \Sigma_2) \\ \Pi \wedge \Delta' &\triangleq (\Pi \wedge \Pi') \wedge \Sigma' \\ \Sigma * \Delta' &\triangleq \Pi \wedge (\Sigma * \Sigma') \end{aligned}$$

Operators $\wedge$ and $*$ distribute over $\vee$, thus we allow use of these operations on disjunctive symbolic heaps and furthermore use the same notation Δ to refer to both symbolic and disjunctive symbolic heaps.

Automation. Our resource-usage analysis relies on an (automated) theorem prover for separation logic assertions that can deal with three types of inference queries involving disjunctive symbolic heaps:

frame inference: $\Delta_1 \vdash \Delta_2 * [\Delta_F]$,
i.e., given Δ_1 and Δ_2 find the frame Δ_F such that $\Delta_1 \vdash \Delta_2 * \Delta_F$ holds;

abduction: $\Delta_1 * [\Delta_A] \vdash \Delta_2$,
i.e., given Δ_1 and Δ_2 find the "missing" assumption (antiframe) Δ_A such that $\Delta_1 * \Delta_A \vdash \Delta_2$ holds;

bi-abduction: [3] $\Delta_1 * [\Delta_A] \vdash \Delta_2 * [\Delta_F]$,
i.e., given Δ_1 and Δ_2 find Δ_A and Δ_F such that $\Delta_1 * \Delta_A \vdash \Delta_2 * \Delta_F$ holds.

As before, square brackets denote the portion of the entailment to be computed. We sometimes write $[\_]$ for a portion that should be computed but that is existentially quantified and will not be used further.

None of these queries has a unique answer in general. However, for the soundness of our resource-usage analysis any answer is acceptable (though some will give rise to a better parallelisation than the others). Existing separation logic tools generally supply only a single answer. In our implementation, we handle disjunctions using backtracking as in [DP08], while gracefully loosing disjunctions in (bi-)abductive inference similarly to [CDOY11].

We note that the theorem prover support for these inference queries is required independently of how the supplied sequential input program proof has been obtained. That is, our resource-usage analysis crucially depends on the theorem prover providing answers to these queries but not on the program analysis (or manual effort) that constructed the original sequential proof.

3.3 Sequential proof

We assume that the separation logic proof of the function `work` is represented as a map $\mathfrak{P}\colon \mathsf{Label} \to \mathcal{P}(\mathsf{SH})$.[4] Assertions in the proof have the property that for any label ℓ

[3]The bi-abduction query can be solved by using independent proof systems for frame inference and abduction, invoked separately in an external procedure (see [CDOY11]). In our implementation, we solve the bi-abduction query using a single proof system, *within* the theorem prover (see [BDD+11]).

[4]We use the disjunctive completion of symbolic heaps (the powerset) as the domain of proof assertions to allow use of the logical $\vee$ as the join operation. We made this design choice to the sake of simplicity, however, more elaborate join operations (e.g. [YLB+08]) could be accommodated.

executing the program from a state satisfying $\mathfrak{P}(\ell)$ up to some subsequent label ℓ' will result in a state satisfying $\mathfrak{P}(\ell')$, and will not fault. Our approach is agnostic to the method by which the proof is created: it can be discovered automatically (e.g., by a tool such as Abductor [CDOY11]), prepared by a proof assistant, or written manually.

We assume that the structure of the proof is *modular*, i.e., that primitive commands and loops at each label use specifications which are given a priori "in isolation", inherent only to the underlying programming language and agnostic to the particular context in which the specification is used. (This property is a standard feature of separation logic proofs). We assume functions $\mathsf{Pre}, \mathsf{Post} \colon \mathsf{Label} \rightharpoonup \mathcal{P}(\mathsf{SH})$ associating labels of primitive commands with such modular pre- and post-conditions, respectively. That is, the command at label ℓ has a specification

$$\{\mathsf{Pre}(\ell)\}\ \mathsf{cmd}(\ell)\ \{\mathsf{Post}(\ell)\},$$

which is applied in the actual proof by using the frame rule and appropriate variable substitutions. We also assume a function $\mathsf{Inv} \colon \mathsf{Label} \rightharpoonup \mathcal{P}(\mathsf{SH})$ associating `while` labels with loop invariants, which are used similarly as specifications.

Specifications and loop invariants are formally instantiated in the proof by using a mapping Ω from labels to variable substitutions such that $\Omega(\ell) = \bar{x} \mapsto \bar{x'}$ maps formal variables $\bar{x}$ in the specification assertion to actual variables $\bar{x'}$ in the proof assertion at a label ℓ. We write $\Delta[\Omega(\ell)]$ to represent the heap constructed by applying the substitutions defined by $\Omega(\ell)$ to the assertion Δ.

We also assume a function $\mathfrak{F} \colon \mathsf{Label} \rightharpoonup \mathcal{P}(\mathsf{SH})$ returning the portion of the assertion framed away at a particular program label. The function $\mathfrak{F}$ satisfies, for each label ℓ of a primitive command,

$$\mathfrak{P}(\ell) \vdash \mathsf{Pre}(\ell)[\Omega(\ell)] * \mathfrak{F}(\ell)$$

and

$$\mathsf{Post}(\ell)[\Omega(\ell)] * \mathfrak{F}(\ell) \vdash \mathfrak{P}(\mathsf{succ}(\ell)).$$

If ℓ is a `while` label then $\mathsf{Pre}(\ell)$ and $\mathsf{Post}(\ell)$ are replaced by a single $\mathsf{Inv}(\ell)$.

We extend functions $\mathfrak{P}$ and $\mathfrak{F}$ from labels to program paths by taking the assertion associated with the last label. That is, if $\gamma = \ell_1 \ldots \ell_n$ then we define $\mathfrak{P}(\gamma) \triangleq \mathfrak{P}(\ell_n)$ and $\mathfrak{F}(\gamma) \triangleq \mathfrak{F}(\ell_n)$.

Example 3.2. The right-hand side of Fig. 3.1 shows a proof for the function `f()` from §2 laid out with respect to the associated program paths.

CHAPTER 4

PARALLELISATION ALGORITHM

We now formally define our proof-directed parallelisation algorithm. Given an input program $\mathbb{P}$, the goal of our algorithm is to construct a program $\mathbb{P}_{par}$, which is a correctly-synchronised parallelised version of $\mathbb{P}$. We assumed in §3 that $\mathbb{P}$ is of a fixed form in which the program's $\texttt{main}_{\mathbb{P}}$ function repeatedly calls a function `work` in a loop. Consequently, our parallelisation algorithm generates $\mathbb{P}_{par}$ containing new functions $\texttt{work}'$ and $\texttt{main}_{\mathbb{P}_{par}}$ such that

```
main_Ppar(void) {
  // channel initialisation.
  for(C1; C2; b) {
    // channel creation.
    fork(work', ...);
  }
  // channel finalisation.
}
```

has the same input-output behaviour as the original $\texttt{main}_{\mathbb{P}}$.

As described in the intuitive development in §2.3, our parallelisation algorithm is comprised of the resource usage analysis and parallelising transformation. The parallelising transformation is heuristic in nature and should be viewed as just one application of the resource usage analysis; more optimised parallelising transformations are certainly possible. With this in mind, the overall aim of this chapter is to present a framework for resource-sensitive dependency-preserving analyses—with a particular instantiation of the parallelisation backend.

The rest of this chapter is structured as follows. In §4.1 we formalise the resource usage analysis and in §4.2 the parallelising transformation. Then in §4.3 we give remarks about our implementation and the computational complexity of algorithms. We discuss soundness informally in §4.4. We conclude with a discussion of limitations (§4.5).

4.1 Resource usage analysis

The resource usage analysis computes two maps:

$$\begin{aligned} \mathsf{needed}\colon \mathsf{Paths} \times \mathsf{Paths} &\rightharpoonup \mathcal{P}(\mathsf{SH}) \\ \mathsf{redundant}\colon \mathsf{Paths} \times \mathsf{Paths} &\rightharpoonup \mathcal{P}(\mathsf{SH}) \end{aligned}$$

For $\gamma_s, \gamma_e \in \mathsf{Paths}$, such that $\gamma_s \prec \gamma_e$, $\mathsf{needed}(\gamma_s, \gamma_e)$ gives the resources that might be accessed during execution from γ_s to γ_e. In parallelisation, these are the resources that must be acquired before execution of the current thread can proceed. Similarly, $\mathsf{redundant}(\gamma_s, \gamma_e)$ gives the resources that are guaranteed not to be accessed by the program between γ_s and γ_e. In parallelisation, these are the resources that can safely be transferred to other parallel threads.

Algorithm 1: Computing needed resources between labels within the same program block using backward symbolic execution.

```
Needed-Block(($\ell', \ell''$): Label × Label, $\Delta$: $\mathcal{P}(\mathsf{SH})$)
begin
    $\ell := \ell''$;
    while $\ell \neq \ell'$ do
        $\ell := \mathsf{pred}(\ell)$;
        if $\mathsf{cmd}(\ell)$ matches if($b$) { $C$ } else { $C'$ } then
            $\Delta := \mathfrak{P}(\ell)^{\Pi} \wedge$
              $($Needed-Block(($\ell^{\mathbf{t}}_{\mathbf{s}}, \ell^{\mathbf{t}}_{\mathbf{e}}$), $\Delta$) $\cup$ Needed-Block(($\ell^{\mathbf{f}}_{\mathbf{s}}, \ell^{\mathbf{f}}_{\mathbf{e}}$), $\Delta$)$)$;
        end
        else if $\mathsf{cmd}(\ell)$ matches while($b$) { $C$ } then
            $\mathsf{Inv}(\ell)[\Omega(\ell)] * [\Delta_A] \vdash \Delta * [\_]$;
            $\Delta := \mathfrak{P}(\ell)^{\Pi} \wedge (\mathsf{Inv}(\ell)[\Omega(\ell)] * \Delta_A)$;
        end
        else
            $\mathsf{Post}(\ell)[\Omega(\ell)] * [\Delta_A] \vdash \Delta * [\_]$;
            $\Delta := \mathfrak{P}(\ell)^{\Pi} \wedge (\mathsf{Pre}(\ell)[\Omega(\ell)] * \Delta_A)$;
        end
    end
    return $\Delta$;
end
```

4.1.1 Needed resources computation

We calculate needed resources in a program by using two functions, defined below: `Needed-Block`, which computes needed resources between labels within the same program block, and `Needed`, which computes needed resources between arbitrary program paths, spanning possibly different blocks. Intuitively, the needed resources computation can be seen as a *liveness* analysis for resources—a resource R which is needed between program paths γ_s and γ_e is *live* in the sense that on the way from γ_s to γ_e some command might access R by virtue of command's specification mentioning R.

`Needed-Block`. The function `Needed-Block` (Alg. 1) uses backward symbolic execution to compute needed resources between a pair of labels (ℓ', ℓ'') located within the same program block. The execution process starts by an arbitrary symbolic heap Δ, representing the resources we have at the end of the block. By pushing Δ backwards along the preceding commands, `Needed-Block` successively discovers missing resources that are needed to execute the commands in the block. The algorithm poses bi-abduction queries to determine the sufficient precondition of primitive commands and loops (using the loop invariants as specifications) and dives recursively into the blocks of `if` and `else` branches. The existing pure components of the sequential proof are used to strengthen the abduced solutions returned by the prover. Upon completion, `Needed-Block` returns a symbolic heap sufficient to execute the program block $\mathbb{P}_{[\ell',\ell'')}$.

The key step of the algorithm is performed on lines 14 and 15. These two steps "simulate" backward execution of the command at label ℓ, $\mathsf{cmd}(\ell)$, by using its specification. The goal of such process is to discover the eventually missing resources that $\mathsf{cmd}(\ell)$ needs for safe execution. Let us explain how these two steps work in more detail.

The command at label ℓ has a specification $\{\mathsf{Pre}(\ell)\}\ \mathsf{cmd}(\ell)\ \{\mathsf{Post}(\ell)\}$. Let Δ be the symbolic heap representing the needed resources for safe execution of commands below label ℓ that we have computed in previous iterations. For $\mathsf{cmd}(\ell)$ to have been executed safely, it must have been able to establish $\mathsf{Post}(\ell)$ upon completion. Since in general we do not know in which capacity Δ contains the resources described by $\mathsf{Post}(\ell)$, we pose a bi-abduction query

$$\mathsf{Post}(\ell)[\Omega(\ell)] * [\Delta_A] \vdash \Delta * [\_],$$

forcing the post-state after execution of $\mathsf{cmd}(\ell)$ to become $\mathsf{Post}(\ell)[\Omega(\ell)] * \Delta_A$. By framing Δ_A away and using the specification of $\mathsf{cmd}(\ell)$, we obtain $\mathsf{Pre}(\ell)[\Omega(\ell)] * \Delta_A$ as the pre-state before the execution of $\mathsf{cmd}(\ell)$. Finally, we enrich the computed pre-state with the pure assertion $\mathfrak{P}(\ell)^\Pi$ from the sequential proof and store the resulting symbolic heap as the new Δ.

The use of existing pure assertions from the sequential proof (rather than synthesizing new ones) is essential in practice—abducing missing pure assertions is much harder than abducing spatial resources and the prover would likely often fail to compute a solution. Pure parts of the sequential proof also control the precision of conditions for if-then-else blocks and loops. For instance, we could have updated Δ in line 7 by always requiring b and $\neg b$ in the respective disjuncts:

$$\begin{gathered}\big(\mathfrak{P}(\ell)^\Pi \wedge b \wedge \texttt{Needed-Block}((\ell^{\mathsf{t}}_{\mathsf{s}},\ell^{\mathsf{t}}_{\mathsf{e}}),\Delta)\big) \cup \\ \big(\mathfrak{P}(\ell)^\Pi \wedge \neg b \wedge \texttt{Needed-Block}((\ell^{\mathsf{f}}_{\mathsf{s}},\ell^{\mathsf{f}}_{\mathsf{e}}),\Delta)\big)\,.\end{gathered}$$

However, if b ($\neg b$) is necessary in the proof of the `if` (resp. `else`) branch then b (resp. $\neg b$) is already implied by $\mathfrak{P}(\ell^{\mathsf{t}}_{\mathsf{s}})$ (resp. $\mathfrak{P}(\ell^{\mathsf{t}}_{\mathsf{s}})$), and thus will be included by construction.

Lemma 4.1. For every Δ, the symbolic heap returned by `Needed-Block`$((\ell',\ell''),\Delta)$ is a sufficient precondition for $\mathbb{P}_{[\ell',\ell'')}$.

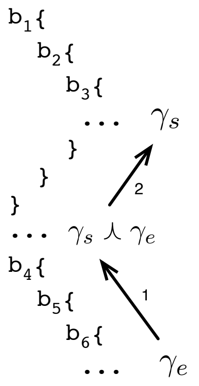
Proof. Follows by iteratively applying the disjunctive version of the frame rule:

$$\frac{\{P\}\, C\, \{\bigvee_{i\in\mathcal{I}} Q_i\} \qquad \Delta * \bigvee_{j\in\mathcal{J}} \Delta^A_j \vdash P * \bigvee_{k\in\mathcal{K}} \Delta^F_k}{\{\bigvee_{j\in\mathcal{J}}(\Delta * \Delta^A_j)\}\, C\, \{\bigvee_{i\in\mathcal{I},k\in\mathcal{K}}(Q_i * \Delta^F_k)\}}$$

and Hoare's rule of composition. □

`Needed`**.** The function `Needed` (Alg. 2) lifts `Needed-Block` from within blocks to between blocks and calculates sufficient precondition between arbitrary program paths. Given two assertion points represented as program paths γ_s and γ_e, `Needed`(γ_s, γ_e) works by successively pushing backwards the assertion $\mathfrak{P}(\gamma_e)^{\Pi} \wedge \mathsf{emp}$ from γ_e to γ_s. In the first while loop (line 5), the algorithm steps backwards from γ_e towards the nearest ancestor block containing both γ_s and γ_e, designated by their longest common prefix. After stepping through that block, in the second while loop (line 10) the algorithm keeps stepping backwards, proceeding inwards towards γ_s. Both phases of the algorithm use Alg. 1 to compute the needed resources in-between program blocks.

The following diagram illustrates the behaviour of Alg. 2:



Arrow 1 corresponds to the first while loop in the algorithm (line 5), while arrow 2 corresponds to the second while loop (line 10).

Algorithm 2: Computing needed resources between blocks.

```
1  Needed($\gamma_s$: Paths, $\gamma_e$: Paths)
2  begin
3      $k := |\gamma_e|$;
4      $\Delta := \mathfrak{P}(\gamma_e)^{\Pi} \wedge \mathsf{emp}$;
5      while $k > |\gamma_s \curlywedge \gamma_e| + 1$ do
6          $\Delta :=$ Needed-Block(($\gamma_e[k]_{\uparrow}, \gamma_e[k]$), $\Delta$);
7          $k := k - 1$;
8      end
9      $\Delta :=$ Needed-Block(($\gamma_e[k], \gamma_s[k]$), $\Delta$);
10     while $k < |\gamma_s|$ do
11         $k := k + 1$;
12         $\Delta :=$ Needed-Block(($\gamma_s[k], \gamma_s[k]_{\downarrow}$), $\Delta$);
13     end
14     return $\Delta$;
15 end
```

We tabulate the results computed by `Needed()` into a map needed as follows:

- if $\mathfrak{P}(\gamma_s) \vdash$ `Needed`$(\gamma_s, \gamma_e) * [\_]$, then $\mathsf{needed}(\gamma_s, \gamma_e) := \mathfrak{P}(\gamma_s)^{\Pi} \wedge$ `Needed`(γ_s, γ_e);
- otherwise, $\mathsf{needed}(\gamma_s, \gamma_e) := \mathfrak{P}(\gamma_s)$.

	ℓ_2	$\ell_2\ell_3$	$\ell_2(\ell_2)^{\mathtt{t}}_{\mathtt{e}}$	$\ell_2\ell_4$	$\ell_2(\ell_2)^{\mathtt{f}}_{\mathtt{e}}$	$\mathtt{f}_{\mathtt{e}}$
ℓ_1	$\mathtt{x} \mapsto a$	$a \geq \mathtt{i} \wedge \mathtt{x} \mapsto a$	$a \geq \mathtt{i} \wedge \mathtt{x} \mapsto a * \mathtt{y} \mapsto b$	$a < \mathtt{i} \wedge \mathtt{x} \mapsto a$	$a < \mathtt{i} \wedge \mathtt{x} \mapsto a$	$(a \geq \mathtt{i} \wedge \mathtt{x} \mapsto a * \mathtt{y} \mapsto b) \vee (a < \mathtt{i} \wedge \mathtt{x} \mapsto a)$
ℓ_2		$\mathtt{v} = a \wedge \mathtt{v} \geq \mathtt{i}$	$\mathtt{v} = a \wedge \mathtt{v} \geq \mathtt{i} \wedge \mathtt{y} \mapsto b$	$\mathtt{v} = a \wedge \mathtt{v} < \mathtt{i}$	$\mathtt{v} = a \wedge \mathtt{v} < \mathtt{i} \wedge \mathtt{x} \mapsto a$	$(\mathtt{v} = a \wedge \mathtt{v} \geq \mathtt{i} \wedge \mathtt{y} \mapsto b) \vee (\mathtt{v} = a \wedge \mathtt{v} < \mathtt{i} \wedge \mathtt{x} \mapsto a)$
$\ell_2\ell_3$			$\mathtt{v} = a \wedge \mathtt{v} \geq \mathtt{i} \wedge \mathtt{y} \mapsto b$			$\mathtt{v} = a \wedge \mathtt{v} \geq \mathtt{i} \wedge \mathtt{y} \mapsto b$
$\ell_2(\ell_2)^{\mathtt{t}}_{\mathtt{e}}$						$\mathtt{v} = a \wedge \mathtt{v} \geq \mathtt{i}$
$\ell_2\ell_4$					$\mathtt{v} = a \wedge \mathtt{v} < \mathtt{i} \wedge \mathtt{x} \mapsto a$	$\mathtt{v} = a \wedge \mathtt{v} < \mathtt{i} \wedge \mathtt{x} \mapsto a$
$\ell_2(\ell_2)^{\mathtt{f}}_{\mathtt{e}}$						$\mathtt{v} = a \wedge \mathtt{v} < \mathtt{i}$

Figure 4.1: The needed map for the function `f()` from §2. (an empty cell means not defined)

Lemma 4.2. needed(γ_s, γ_e) is a sufficient precondition to execute $\mathbb{P}$ from γ_s to γ_e.

For further steps of the parallelisation algorithm, we must ensure that the needed map is monotonic with respect to program paths, i.e., that $\forall \gamma, \gamma', \gamma'' \in$ Paths such that $\gamma \prec \gamma' \prec \gamma''$ we have needed$(\gamma, \gamma'') \vdash$ needed$(\gamma, \gamma') * [_]$. This ensures that the resources needed to reach a particular point include all the resources needed to reach preceding points. If the underlying theorem prover behaves consistently with respect to failing and precision then this property will hold. If that is not the case, we post-process the map and insert additional assertions from the sequential proof as needed.

Example 4.1. Consider the function `f()` from example 2.1 in §2. The map obtained by running the algorithm `Needed` is shown in Fig. 4.1.

4.1.2 Redundant resources computation

We now turn to computing redundant resources. The redundant resource between two program paths is a portion of the proof assertion in $\mathfrak{P}$ that is not required by the needed map. We calculate this portion by frame inference, as shown in function `Redundant` (Alg. 3). We tabulate redundant resources in the map redundant by setting redundant$(\gamma_s, \gamma_e) :=$ `Redundant` (γ_s, γ_e).

Lemma 4.3. If $\mathfrak{P}(\gamma_s) \vdash$ redundant$(\gamma_s, \gamma_e) * [\Delta]$ then Δ is a sufficient precondition to execute $\mathbb{P}$ from γ_s to γ_e.

Example 4.2. Fig. 4.2 shows the redundant map computed by running the algorithm `Redundant` with respect to the program path $\mathtt{f}_{\mathtt{e}}$ for the function `f()` from §2.

Algorithm 3: Computing redundant resources

```
1 Redundant($\gamma_s$: Paths, $\gamma_e$: Paths)
2 begin
3     $\mathfrak{P}(\gamma_s) \vdash$ Needed($\gamma_s$, $\gamma_e$) $*$ $[\Delta_R]$;
4     return $\Delta_R$;
5 end
```

	$\mathtt{f_e}$
ℓ_1	$a < \mathtt{i} \wedge \mathtt{y} \mapsto b$
ℓ_2	$(\mathtt{v} = a \wedge \mathtt{v} \geq \mathtt{i} \wedge \mathtt{x} \mapsto a) \vee (\mathtt{v} = a \wedge \mathtt{v} < \mathtt{i} \wedge \mathtt{y} \mapsto b)$
$\ell_2\ell_3$	$\mathtt{v} = a \wedge \mathtt{v} \geq \mathtt{i} \wedge \mathtt{x} \mapsto a$
$\ell_2(\ell_2)^{\mathtt{t}}_{\mathtt{e}}$	$\mathtt{v} = a \wedge \mathtt{v} \geq \mathtt{i} \wedge \mathtt{x} \mapsto a * \mathtt{y} \mapsto b$
$\ell_2\ell_4$	$\mathtt{v} = a \wedge \mathtt{v} < \mathtt{i} \wedge \mathtt{y} \mapsto b$
$\ell_2(\ell_2)^{\mathtt{f}}_{\mathtt{e}}$	$\mathtt{v} = a \wedge \mathtt{v} < \mathtt{i} \wedge \mathtt{x} \mapsto a * \mathtt{y} \mapsto b$

Figure 4.2: The redundant map with respect to the program path $\mathtt{f_e}$ for the function `f()` from §2.

4.2 Parallelising transformation

We now formalise our parallelising transformation—an instance of a parallelisation backend based on the resource-usage analysis. As described in §2.3, the construction proceeds in three steps. First, we generate the skeleton of the parallelised program by running each sequential iteration in a separate thread. Then we compute program points for resource transfer and the associated resources that will be transferred between threads. Finally, we inject `grant` and `wait` barriers to realise this resource transfer. The resource transfer mechanism transfers a resource from one invocation of the `work` function to another. Along with the parallelised program, we generate its proof of correctness in separation logic with disjoint concurrency.

4.2.1 Parallelised program skeleton

In the first phase we take the sequential program $\mathbb{P}$, consisting of a **`pfor`**-loop repeatedly invoking `work` as a function (i.e. waiting for completion in each iteration), and generate a parallelised program $\mathbb{P}_{par}$ with a corresponding **`for`**-loop in which each iteration forks a new thread invoking a function `work`$'$ (without waiting for completion):

```
main_Ppar(void) {
  // channel initialisation.
  for(C1; C2; b) {
    // channel creation.
    fork(work', ...);
  }
  // channel finalisation.
}
```

The function `work`$'$ is a modified version of the function `work`, "patched" with appropriate barrier synchronisation, and is synthesised (along with the channel initialisation, creation and finalisation) as a result of the next two phases.

4.2.2 Resource matching

In the second phase we determine resources that should be released and acquired at particular points in the parallelised program. We first show general conditions that have to hold for such resource transfer to be valid. We then give a heuristic algorithm to add `grant`/`wait` pairs which satisfy conditions.

Conditions on released and acquired resources. Released resources cannot be revoked, i.e., each released resource should be included in the redundant map from the point of the release to the end of the `work` function—this way we know the resource will not be needed further. Acquired resources are held by the executing thread until released. Resources that are acquired along a sequence of program paths should contain what is prescribed by the needed map between each of the program paths.

We represent the resource transfer realised by the released and acquired resources via the following maps:

- $\mathsf{resource}\colon \mathsf{ResId} \to \mathcal{P}(\mathsf{SH})$, denoting resource identifiers that identify released and acquired resources selected by the algorithm;
- $\mathsf{released}\colon \mathsf{Paths} \rightharpoonup \mathsf{ResId} \times \mathsf{Subst}$, representing resources that are going to be released at a program path together with the variable substitution applied at that point;
- $\mathsf{acquired}\colon \mathsf{Paths} \rightharpoonup \mathsf{ResId}$, representing resources that are going to be acquired at a program path.

In order for these maps to represent a *feasible* resource transfer, we require that they satisfy the following well-formedness properties:

1. $\forall \gamma \in \mathrm{dom}(\mathsf{released})\,.$
$$\forall \gamma' \succ \gamma\,.\,\mathsf{released}(\gamma) = (r, \rho) \Rightarrow (\mathsf{redundant}(\gamma, \gamma') \vdash \mathsf{resource}(r)[\rho] * [\_]);$$
2. $\forall \gamma \in \mathsf{Paths}\,.$
$$\circledast_{r \in \mathrm{dom}(\mathsf{resource})}\{\mathsf{resource}(r) \mid \exists \gamma' \prec \gamma \wedge \mathsf{acquired}(\gamma') = r\} \vdash \mathsf{needed}(\gamma, \mathtt{work}_{\mathbf{e}}) * [\_];$$
3. $\forall \gamma \in \mathrm{dom}(\mathsf{released})\,.$
$$\circledast_{r \in \mathrm{dom}(\mathsf{resource})}\{\mathsf{resource}(r) \mid \exists \gamma' \prec \gamma \wedge \mathsf{acquired}(\gamma') = r\} \vdash \mathsf{released}(\gamma) * [\_].$$

The first property states we can release only resources that are redundant between the given program path and any subsequent one. The second property states that the resources needed at a program path must have already been acquired. The third property states that only the resources that have been previously acquired can be released.[1]

In general, there could be many feasible resource transfers satisfying properties 1–3. For instance, there is always a trivial transfer that acquires the resource $\mathsf{needed}(\mathtt{work}_{\mathbf{s}}, \mathtt{work}_{\mathbf{e}})$ before the first command and releases it after the last, causing each invocation of `work` to be blocked until the preceding invocation finishes the last command. Of course, some transfers are better than others. Determining a resource transfer's practical quality a priori would need to take into account the runtime behaviour of synchronisation, and that kind of analysis is beyond the scope of this work. However, to demonstrate how one might compute resource transfers in practice, we present a heuristic algorithm that works well on our examples.

Computing released and acquired resources. Algorithm 4 constructs $\mathsf{released}$, $\mathsf{acquired}$ and $\mathsf{resource}$ maps satisfying properties 1–3. Each iteration of the algorithm heuristically picks a needed resource from some subset of program paths, and then iteratively searches for matching redundant resources along all preceding paths. The algorithm maintains a set $\mathcal{C}$ of all program paths up to which no more resources are needed. It terminates once no unsatisfied needed resources remain (line 3).

At the start of the main loop (line 4) the algorithm picks a still-needed resource between a program path in $\mathcal{C}$ and some further program path. The picking of the needed resource is governed by a heuristic function *choose*, which picks a pair of program paths (γ, γ') such that $\gamma \in \mathcal{C}$, $\gamma' \in \mathsf{Paths}$ and $\gamma \prec \gamma'$. We do not make any specific assumption about how these program paths are being picked. For our examples, we defined a simple partial order reflecting the size of heap resources and picked program paths that lead to largest resources with respect to that order being picked first. We discuss the resource selection heuristic further in §5.2.

The key step of the algorithm is performed on line 8:

$$\mathcal{C}_{\mathsf{R}} := \min\{\gamma \in \mathsf{Paths} \,|\, \mathsf{R}(\gamma, \mathtt{work}_{\mathbf{e}})^{\Sigma} \vdash \Sigma_r' * [\_] \quad \wedge \quad \exists \gamma' \in \mathcal{C}_{\mathsf{N}}.\, \gamma' \preceq \gamma\}$$

Here $\mathsf{R}(\gamma, \mathtt{work}_{\mathbf{e}})$ is the redundant resource from γ to the end of the `work` function, Σ_r' is the candidate resource that we want to acquire, and $\mathcal{C}_{\mathsf{N}}$ the set of paths along which the resource Σ_r needs to be acquired. The constructed set $\mathcal{C}_{\mathsf{R}}$ is a set of program paths along which we can satisfy the candidate needed resource. In line 9, the algorithm checks whether $\mathcal{C}_{\mathsf{R}}$ covers all paths from $\mathcal{C}_{\mathsf{N}}$.

Resources stored in needed contain various first-order conditions embedded in the pure part of the symbolic heap. Since we can transfer resources between potentially different program paths, we only take the spatial part of the resource into consideration when asking entailment questions; this is denoted by a superscript Σ. Moreover, since the acquired resource is being sent to a different function invocation, we substitute a fresh set of variables (line 6).

[1]We could relax the third requirement if we extended our barriers to support *renunciation* [DJP11], the ability to release a resource without first acquiring it. Renunciation allows a resource to 'skip' iterations, giving limited out-of-order signalling.

Algorithm 4: A heuristic algorithm for computing released and acquired resources.

```
N := needed; R := redundant;
$\mathcal{C} := \max\{\gamma \in \mathsf{Paths} \mid \mathsf{N}(\mathtt{work}_{\mathbf{s}}, \gamma) = \mathsf{emp}\}$;
while $\mathcal{C} \neq \{\mathtt{work}_{\mathbf{e}}\}$ do
    $\Sigma_r := \mathsf{N}(\mathit{choose}(\{(\gamma, \gamma') \mid \gamma \in \mathcal{C} \ \wedge\ \gamma' \in \mathsf{Paths} \ \wedge\ \gamma \prec \gamma'\}))^{\Sigma}$;
    $\mathcal{C}_{\mathsf{N}} := \{\gamma \in \mathcal{C} \mid \exists \gamma' \in \mathsf{Paths}.\, \mathsf{N}(\gamma, \gamma') \vdash \Sigma_r * [\_]\}$;
    $\varrho := \bar{\mathrm{x}} \mapsto \bar{\mathrm{x}}'$, where $\bar{\mathrm{x}}'$ fresh;
    $\Sigma_r' := \Sigma_r[\varrho]$;
    $\mathcal{C}_{\mathsf{R}} := \min\{\gamma' \in \mathsf{Paths} \mid \mathsf{R}(\gamma', \mathtt{work}_{\mathbf{e}})^{\Sigma} \vdash \Sigma_r' * [\_] \ \wedge\ \exists \gamma \in \mathcal{C}_{\mathsf{N}}.\, \gamma \preceq \gamma'\}$;
    if $\mathcal{C}_{\mathsf{N}} \preceq \mathcal{C}_{\mathsf{R}}$ then
        $r$ := fresh resource id;
        $\mathsf{resource}(r) := \Sigma_r'$;
        forall $\gamma \in \mathcal{C}_{\mathsf{R}}$ do
            $\mathsf{released}(\gamma) := (r, \varrho)$;
            forall $\gamma'$ s.t. $\gamma \preceq \gamma'$ do
                $\mathsf{R}(\gamma', \mathtt{work}_{\mathbf{e}})^{\Sigma} \vdash \Sigma_r * [\Delta]$;
                $\mathsf{R}(\gamma', \mathtt{work}_{\mathbf{e}}) := \Delta$;
            end
        end
        forall $\gamma \in \mathcal{C}_{\mathsf{N}}$ do
            $\mathsf{acquired}(\gamma) := r$;
            forall $\gamma', \gamma''$ s.t. $\gamma \preceq \gamma' \preceq \gamma''$ do
                $\mathsf{N}(\gamma', \gamma'') * [\_] \vdash \Sigma_r' * [\Delta]$;
                $\mathsf{N}(\gamma', \gamma'') := \Delta$;
            end
        end
        $\mathcal{C} := \max\{\gamma' \in \mathsf{Paths} \mid \exists \gamma \in \mathcal{C}.\, \mathsf{N}(\gamma, \gamma') = \mathsf{emp}\}$;
    end
end
```

If the check $\mathcal{C}_\mathsf{N} \preceq \mathcal{C}_\mathsf{R}$ succeeds, the remainder of the algorithm is devoted to constructing the new resource (line 11), and updating released (lines 12–18), acquired (lines 19–25), and $\mathcal{C}$ (line 26).

Lemma 4.4. Maps resource, released and acquired computed by Algorithm 4 satisfy properties 1–3.

Example 4.3. Consider the run of the Algorithm 4 on our running example (Fig. 4.1 and Fig. 4.2). If *choose* picks (ℓ_1, ℓ_2) in the first iteration and $(\ell_2\ell_3, \ell_2(\ell_2)^{\mathtt{t}}_{\mathbf{e}})$ in the second iteration, then the end result of Alg. 4 is resource $= \{r_1 \mapsto (\mathtt{x} \mapsto a), r_2 \mapsto (\mathtt{y} \mapsto b)\}$, released $= \{\ell_2\ell_3 \mapsto (r_1, \emptyset), \ell_2(\ell_2)^{\mathtt{f}}_{\mathbf{e}} \mapsto (r_1, \emptyset), \ell_2(\ell_2)^{\mathtt{t}}_{\mathbf{e}} \mapsto (r_2, \emptyset), \ell_2\ell_4 \mapsto (r_2, \emptyset)\}$ and acquired $= \{\ell_1 \mapsto r_1, \ell_2\ell_3 \mapsto r_2, \ell_2\ell_4 \mapsto r_2\}$.

4.2.3 Inserting grant and wait barriers

In the final phase we synthesise the function work′ in the parallelised program $\mathbb{P}_{par}$ by inserting grant and wait barriers into work′. The inserted barriers realise the resource transfer established by the computed released and acquired resources, while respecting loop-carried dependencies.

We generate the function $\mathtt{work}'(\bar{\mathtt{i}}_r^{(p)}, \bar{\mathtt{i}}_r, \mathtt{env}^{(p)}, \mathtt{env})$ from work as follows:

1. To each $r \in$ ResId we assign a unique channel name $\mathtt{i}_r$. Denote by $\mathtt{i}_r^{(p)}$ the corresponding channel of the previous thread in the sequence.
2. Let env be an associative array that for each channel maps local variable names to values. Let $\mathtt{env}^{(p)}$ be such map from the previous thread in the sequence. env and $\mathtt{env}^{(p)}$ are used for *materialisation* (see below).
3. For each $\gamma \in \mathrm{dom}(\mathsf{acquired})$ such that $\mathsf{acquired}(\gamma) = r$ we insert a wait barrier $\mathtt{wait}(\mathtt{i}_r^{(p)})$ between program paths $\mathsf{pred}(\gamma)$ and γ.
4. For each $\gamma \in \mathrm{dom}(\mathsf{released})$ such that $\mathsf{released}(\gamma) = (r, \_)$, between program paths $\mathsf{pred}(\gamma)$ and γ we insert a sequence of assignments of the form $\mathtt{env}(\mathtt{i}_r)[''\mathtt{y}''] := \mathtt{y}$ for every local variable y, followed by a grant barrier $\mathtt{grant}(\mathtt{i}_r)$.

Each invocation of work′ creates a fresh set of local variables that are bound to the scope of the function. If the structure of some shared resource depends on local variables from a previous invocation, we use the env map to transfer the values from the previous thread in a sequence to the next thread in a sequence. The preceding thread *materialises* the variables by storing them in the env map (see rule (4) above), which is passed as $\mathtt{env}^{(p)}$ to the subsequent thread. Whenever the subsequent thread needs to access a variable from a previous invocation, it refers to the variable using the $\mathtt{env}^{(p)}$ map. For instance, the variable `*pr` in the `move()` example from §2.4.2 (Fig. 2.8) would be referred to as $\mathtt{env}^{(p)}(\mathtt{i1})[''\mathtt{p}'']$.

The main function $\mathtt{main}_{\mathbb{P}_{par}}$ in $\mathbb{P}_{par}$ first creates the set of "dummy" channels; then in the **`for`**-loop repeatedly creates a set of new channels for the current iteration, forks a new thread with work′ taking the channels from the previous ($\bar{\mathtt{i}}_r^{(p)}$) and the current ($\bar{\mathtt{i}}_r$) iteration, as well as the maps $\mathtt{env}^{(p)}$ and env, and at the end of the loop body assigns the new channels to the previous channels; and, after the **`for`**-loop completes, waits on the channels in the last set.

Parallel proof. We generate the parallel proof $\mathfrak{P}_{par}$ from the sequential proof $\mathfrak{P}$ using the following specifications for newchan, grant and wait from [DJP11]:

$$\begin{array}{rcl}
\{\mathsf{emp}\} & \mathtt{i := newchan()} & \{\mathsf{req}(\mathtt{i}, R) * \mathsf{fut}(\mathtt{i}, R)\} \\
\{\mathsf{req}(\mathtt{i}, R) * R\} & \mathtt{grant(i)} & \{\mathsf{emp}\} \\
\{\mathsf{fut}(\mathtt{i}, R)\} & \mathtt{wait(i)} & \{R\}
\end{array}$$

The predicates req and fut, corresponding to the required (resp. future) resource, track the ownership of the input and output ends of each channel: $\mathsf{req}(\mathtt{i}, R)$ states that by calling grant on i when holding a resource satisfying R, the thread will lose this resource, and $\mathsf{fut}(\mathtt{i}, R)$ states that by calling wait on i, the thread will acquire a resource satisfying R. In the proof $\mathfrak{P}_{par}$, we instantiate each variable R associated with a channel $\mathtt{i}_r$ with the corresponding resource $\mathsf{resource}(r)$.

Example 4.4. Fig. 4.3 illustrates the use of req and fut predicates in the parallel proof of the parallelised function f() in Fig. 2.5 from §2.

To reason about threads, we use the standard separation logic rules for fork-join disjoint concurrency [GBC+07]:

$$\frac{\{P\}\ \mathtt{f}(\bar{\mathtt{x}})\ \{Q\}}{\{P[\bar{e}/\bar{\mathtt{x}}]\}\ \mathtt{t = fork(f,}\bar{e}\mathtt{)}\ \{\mathsf{thread}(\mathtt{t},\mathtt{f},\bar{e})\}}\ \text{FORK}$$

$$\frac{\{P\}\ \mathtt{f}(\bar{\mathtt{x}})\ \{Q\}}{\{\mathsf{thread}(\mathtt{t},\mathtt{f},\bar{e})\}\ \mathtt{join(t)}\ \{Q[\bar{e}/\bar{\mathtt{x}}]\}}\ \text{JOIN}$$

Upon forking a new thread, the parent thread obtains the assertion thread that stores information about passed arguments for program variables and gives up ownership of the precondition of the function. Joining requires that the executing thread owns the thread handle which it then exchanges for the function's postcondition.

Theorem 4.5. $\mathfrak{P}_{par}$ is a proof of the parallelised program $\mathbb{P}_{par}$, and defines the same specification for $\mathtt{main}_{\mathbb{P}_{par}}$ as $\mathfrak{P}$ does for $\mathtt{main}_{\mathbb{P}}$.

4.2.4 Loop splitting

The approach presented so far treats a loop as a single command with a specification derived from its invariant. Acquiring or releasing resources within a loop is subtle, as it changes the sequential loop invariant. In our algorithm we take a pragmatic approach that performs heuristic loop-splitting of the sequential loop before the parallel code generation so that the conditions guarding acquiring and releasing of the resources become explicit.

The move() example in §2.4.2 uses two channels to transfer the segment of the list traversed after the first and the second while loop, respectively. The resource released via channel i1 in Fig. 2.8 is $\mathsf{lseg}(\mathtt{s}, y) * \mathtt{pr} \mapsto y$. In the following iteration, the needed resource for the whole loop is $\mathsf{lseg}(\mathtt{s},\mathtt{p}) * \mathsf{node}(\mathtt{p},\mathtt{x}) * \mathsf{node}(\mathtt{x}, n')$. If we try to match the released against the needed resource, the entailment $\mathsf{R}(\gamma, \mathtt{work}_{\mathbf{e}})^{\Sigma} \vdash \Sigma'_r * [\_]$ in Algorithm 4 will fail. This is because the value of p upon exiting the loop cannot be a priori determined, meaning we cannot establish whether the released resource will cover the needed resource.

```
$\{\mathsf{fut}(\mathtt{wxp}, \mathtt{x} \mapsto a) * \mathsf{fut}(\mathtt{wyp}, \mathtt{y} \mapsto b) * \mathsf{req}(\mathtt{wx}, \mathtt{x} \mapsto a') * \mathsf{req}(\mathtt{wy}, \mathtt{y} \mapsto b')\}$
f(i, wxp, wyp, wx, wy) {
  wait(wxp);
 $\{\mathtt{x} \mapsto a * \mathsf{fut}(\mathtt{wyp}, \mathtt{y} \mapsto b) * \mathsf{req}(\mathtt{wx}, \mathtt{x} \mapsto a') * \mathsf{req}(\mathtt{wy}, \mathtt{y} \mapsto b')\}$
  int v = *x;
 $\left\{\begin{array}{l} \mathtt{v} = a \land \mathtt{x} \mapsto a * \mathsf{fut}(\mathtt{wyp}, \mathtt{y} \mapsto b) * \\ \mathsf{req}(\mathtt{wx}, \mathtt{x} \mapsto a') * \mathsf{req}(\mathtt{wy}, \mathtt{y} \mapsto b') \end{array}\right\}$
  if (v >= i) {
   $\left\{\begin{array}{l} \mathtt{v} = a \land \mathtt{v} \geq \mathtt{i} \land \mathtt{x} \mapsto a * \mathsf{fut}(\mathtt{wyp}, \mathtt{y} \mapsto b) * \\ \mathsf{req}(\mathtt{wx}, \mathtt{x} \mapsto a') * \mathsf{req}(\mathtt{wy}, \mathtt{y} \mapsto b') \end{array}\right\}$
    grant(wx);
   $\{\mathtt{v} = a \land \mathtt{v} \geq \mathtt{i} * \mathsf{fut}(\mathtt{wyp}, \mathtt{y} \mapsto b) * \mathsf{req}(\mathtt{wy}, \mathtt{y} \mapsto b')\}$
    wait(wyp);
   $\{\mathtt{v} = a \land \mathtt{v} \geq \mathtt{i} * \mathtt{y} \mapsto b) * \mathsf{req}(\mathtt{wy}, \mathtt{y} \mapsto b')\}$
    *y = v;
   $\{\mathtt{v} = a \land \mathtt{v} \geq \mathtt{i} * \mathtt{y} \mapsto a) * \mathsf{req}(\mathtt{wy}, \mathtt{y} \mapsto b')\}$
    grant(wy);
   $\{\mathtt{v} = a \land \mathtt{v} \geq \mathtt{i}\}$
  }
  else {
   $\left\{\begin{array}{l} \mathtt{v} = a \land \mathtt{v} < \mathtt{i} \land \mathtt{x} \mapsto a * \mathsf{fut}(\mathtt{wyp}, \mathtt{y} \mapsto b) * \\ \mathsf{req}(\mathtt{wx}, \mathtt{x} \mapsto a') * \mathsf{req}(\mathtt{wy}, \mathtt{y} \mapsto b') \end{array}\right\}$
    wait(wyp);
   $\left\{\begin{array}{l} \mathtt{v} = a \land \mathtt{v} < \mathtt{i} \land \mathtt{x} \mapsto a * \mathtt{y} \mapsto b * \\ \mathsf{req}(\mathtt{wx}, \mathtt{x} \mapsto a') * \mathsf{req}(\mathtt{wy}, \mathtt{y} \mapsto b') \end{array}\right\}$
    grant(wy);
   $\{\mathtt{v} = a \land \mathtt{v} < \mathtt{i} \land \mathtt{x} \mapsto a * \mathsf{req}(\mathtt{wx}, \mathtt{x} \mapsto a')\}$
    *x = 0;
   $\{\mathtt{v} = a \land \mathtt{v} < \mathtt{i} \land \mathtt{x} \mapsto 0 * \mathsf{req}(\mathtt{wx}, \mathtt{x} \mapsto a')\}$
    grant(wx);
   $\{\mathtt{v} = a \land \mathtt{v} < \mathtt{i}\}$
  }
}
$\{\mathsf{emp}\}$
```

Figure 4.3: The parallel separation logic proof of the parallelised function `f()` from Fig. 2.5.

```
...
pr:=envp(i1)["p"];
while (i<l && x.nxt!=null && x.nxt!=*pr) {
 i++;
 p=x;
 x=x.nxt;
}
if (i<l && x.nxt!=null && x.nxt==*pr) {
 i++;
 p=x;
 x=x.nxt;
 while (i<l && x.nxt!=null) {
  i++;
  p=x;
  x=x.nxt;
 }
 // remainder skipped.
 ...
else {
...
```

Figure 4.4: Loop-splitting for the `move()` example.

One way to resolve this would be to acquire the entire list before the first loop, but this would result in a poor parallelisation. Instead, we modify the structure of the loop to expose the point at which the second list segment becomes necessary:

1. We split the spatial portion of the resource needed by the whole loop into a "dead" (already traversed) and a "live" (still to be traversed) part. In our example, $\mathsf{lseg}(\mathtt{s},\mathtt{p})$ would be the "dead" and $\mathsf{node}(\mathtt{p},\mathtt{x}) * \mathsf{node}(\mathtt{x},n') * \mathsf{lseg}(n',\mathsf{null})$ the live part. This kind of splitting is specific to some classes of programs, e.g., linked list programs that do not traverse a node twice.

2. We match the resource against the "dead" part of the loop invariant and infer the condition under which the two resources are the same. In our example, after unfolding the special cases of list length one and two, the entailment between the two resources holds if $\mathtt{x.nxt} = *\mathtt{pr}$. This condition can be inferred by asking a bi-abduction question $\mathsf{R}(\gamma,\mathtt{work_e})^{\Sigma} \wedge [c] \vdash \Sigma'_r * [\_]$ with the pure fact c to be abduced.

Now we can syntactically split the loop against the inferred condition $c = (\mathtt{x.nxt} = *\mathtt{pr})$ and obtain a transformed version that ensures that after entering the true branch of the `if` statement the condition c holds. The transformation of our example is shown in Fig. 4.4.

Formally, we define splitting of a loop against a condition c inferred by $\mathsf{R}(\gamma,\mathtt{work_e})^{\Sigma} \wedge [c] \vdash \Sigma'_r * [\_]$ as given in Alg. 5. The condition c' is obtained from c by replacing a reference to every primed variable $\mathtt{y}'$ with a reference to $\mathtt{env}^{(p)}(\mathtt{i}_r^{(p)})[''\mathtt{y}'']$, where $\mathtt{i}_r$ is the channel name associated to the resource. It is not difficult to see that the accompanying proof of C can be split in a proof-preserving way against c. This transformation

Algorithm 5: Loop-splitting against c.

```
Split (while($b$) { $C'$ }; $C''$)
begin
    return
      while($b \wedge \neg c'$) { $C'$ };
      if($b \wedge c'$) { $C'$; while($b$) { $C'$ }; $C''$ }
      else { $C''$ }
end
```

can either be applied to $\mathbb{P}$ between the resource-usage analysis and parallelisation, or embedded within Alg. 4. In the case of multiple loops, the transformation can be iteratively applied for each abduced condition c associated with the "dead" part of the respective loop invariant.

4.3 Implementation

We have validated our parallelisation algorithm by crafting a prototype implementation on top of the existing separation logic tool, coreStar [BDD+11]. While our implementation is not intended to provide full end-to-end automated translation, it is capable of validating the algorithms on the examples given in this part, and automatically answering the underlying theorem proving queries.

Our parallelisation algorithm does not require a shape-invariant generator, except possibly to help construct the sequential proof. Soundness is independent of the "cleanliness" of the invariants (the analysis will always give a correct result, in the worst case defaulting to sequential behaviour). The invariants generated automatically by coreStar were sufficient to validate our examples. Other efforts [CDOY11] indicate that bi-abduction works well with automatically-generated invariants produced by shape analysis, even over very large code bases.

4.3.1 Computational complexity

Deciding entailment of symbolic heaps is solvable in polynomial time for the standard fragment of separation logic built from the points-to and the inductive linked-list predicate [CHO+11]. Although a complete syntactic proof search may require an exponential number of proofs to be explored in the worst case [BCO05a], our implementation uses heuristic proof rules that yield an algorithm having polynomial complexity (the potential incompleteness never occurred as a problem in practice). The same procedure is used for the frame inference, the prover just searches for a different kind of leaves in the proof search tree.

Abduction, on the other hand, is a fundamentally harder problem than deduction. While general abduction for arbitrary pure theories reaches the second level of polynomial hierarchy (in fact, it is Σ_2^{P}-complete) [EG95, CZ06], abduction of resources for the standard fragment of symbolic heaps is NP-complete. In our implementation we use a polynomial heuristic algorithm (similar to [CDOY11]) which may miss some solutions, but in practice has roughly the same cost as frame inference.

Overall, our resource-usage analysis (Alg. 2 and Alg. 3) asks $\mathcal{O}(|\mathsf{Paths}|^2 \cdot d \cdot s)$ frame inference and abduction queries, where d is the depth of the abstract syntax tree $\mathcal{T}$ representing the program, and s the arity (the maximum number of siblings) of nodes in $\mathcal{T}$. The heuristic algorithm for computing released and acquired resources (Alg. 4) asks roughly a cubic number (in the cardinality of Paths) of prover queries.

4.4 Soundness (informally)

A distinctive property of our parallelisation algorithm is that it enforces sequential data-dependencies in the parallelised program even if the safety proof does not explicitly reason about such dependencies. The result is that our analysis preserves the sequential behaviour of the program: any behaviour exhibited by the parallelised program is also a behaviour that could have occurred in the original sequential program. However, there are important caveats relating to *termination* and *allocation*.

Termination. If the original sequential program does not terminate, our analysis may introduce new behaviours simply by virtue of running segments of the program that would be unreachable under a sequential schedule. To see this, suppose we have a **pfor** such that the first iteration of the loop will never terminate. Sequentially, the second iteration of the loop will never execute. However, our parallelisation analysis will execute all iterations of the loop in parallel. This permits witnessing behaviours from the second (and subsequent) iterations. These behaviours were latent in the original program, and became visible only as a result of parallelisation.

Allocation and disposal. If the program both allocates and disposes memory, the parallelised program may exhibit aliasing that could not occur in the original program. To see this, consider the following sequential program:

```
x=alloc(); y=alloc(); dispose(x).
```

Parallelisation might give us the following program:

```
(x=alloc(); grant(wx); y=alloc()) || (wait(wx); dispose(x))
```

This parallelised version of the program is race-free and obeys the required sequential ordering on data. Depending upon the implementation of the underlying memory allocator, however, `x` and `y` may be aliased if the **dispose** operation was interleaved between the two allocations. Such aliasing could not happen in the original non-parallelised version.

Either kind of new behaviour might result in further new behaviours—for example, we might have an if-statement conditional on `x==y` in the second example above. These caveats are common to our analysis and others based on separation logic—for example, see the similar discussion in [CMR+10].

Behaviour preservation. In §6.2 we prove our behaviour preservation result for terminating programs (Theorem 6.18). The proof establishes that each trace of the parallelised program corresponds to a trace of the sequential program so that a simulation invariant between the states of the parallel program and the states of the sequential program is preserved, provided that the sequential program terminates.

As will be seen in the proof, one attractive property is that it in fact does not place explicit requirements on the positioning of barriers—it is sufficient that we can provide a separation logic proof of the parallelised version of the program. The caveat on memory disposition manifests in the way we establish our simulation invariant, which necessarily assumes newly allocated data is always "fresh" and thus does not alias with any accessible heap-allocated structure. The caveat on termination is necessary to ensure we can suitably reorder threads in the parallelised program to yield a "sequentialised" trace.

In addition to inserting barriers, our analysis mutates the program by materialising thread-local variables and splitting loops. Both of these can be performed as mutations on the initial sequential program, and neither affect visible behaviour. Loop-splitting is straightforwardly semantics-preserving, while materialised variables are only used to control barrier calls.

Theorem 6.18 guarantees that parallelisation does not introduce deadlocks; otherwise the simulation relation would not exist. The ordering on barriers ensures that termination in the sequential program is preserved in the resulting parallelised program.

4.5 Limitations

Loop handling. Our parallelising transformation treats loops within `work` as opaque, with a loop-splitting transformation used to allow mid-loop signalling. Since not all loops have a structure amenable to such a transformation, it will not always be possible to discover parallelisation opportunities within loop-based computations. However, one can envision analogous transformations that would split according to different phase-switching patterns in the loop body.

In the main parallel-`for` loop, we currently assume each iteration communicates with only its immediate predecessor and successor. A technique called *renunciation*, discussed in [DJP11], allows a thread to release a resource without having to first acquire it, allowing resource transfers to skip iterations. It would be relatively straightforward to fold such a technique into our analysis.

Procedures. The analysis we have given in §4.1 treats procedures in two ways: either as inlined code, or opaquely, in terms of their sequential specifications. The latter lets us deal with recursively-defined procedures, although it means that such procedures cannot have barriers injected into them. This is a similar simplification to treating loops as opaque. As with loops, we can develop a more sophisticated procedure-unrolling transformation which would exploit parallelisation opportunities across function calls. We present such an inter-procedural version of the analysis in §5.4.

Scalability. A naïve implementation of our proposed algorithm would not scale due to rapid combinatorial explosion of program paths. However, the algorithm can be tuned to avoid exploring paths that are unlikely to expose parallelism or, conversely, to explore paths that involve intense computations worth running concurrently. We note that reducing the depth of the abstract syntax tree exploration only limits the potential for parallelisation but does not lead to unsoundness.

Tool support. An automated separation logic prover for solving frame inference, abduction and bi-abduction queries is essential for our approach, and the quality of parallelisation depends strongly on the success of the prover. However, our analysis degrades gracefully with imprecise abduction giving less-precise parallelisation. In the worst case, we obtain sequentialisation of the parallel threads, reflecting our analysis being unable to prove that any other parallelisation was safe with respect to our ordering assumptions.

Non-list domains. As discussed in §5, our analysis is generic in the choice of the abstract domain; any separation logic predicate could be used in place of lseg, for example. However, the success of *automated* parallelisation is highly dependent on the power of the entailment prover in the chosen domain. The lseg domain is one of the best-developed in separation logic, and consequently automated parallelization is feasible using tools such as `coreStar`. Other domains (such as trees) are far less developed so additional automated reasoning support and special-purpose heuristics for abduction would be needed.

Alternative shape analyses. While our parallelisation method crucially relies on the separation logic proof in order to guarantee preservation of sequential data-dependencies in the parallelised program, it does not assume particular shape invariants. Shape analyses not based on separation logic gave birth to alternative shape domains for approximating deep heap updates to unbounded structures. Most notably, shape graphs have formed the basis of a flow-sensitive analysis for heap manipulation based on abstract interpretation [SRW96, SRW98], which after being subsequently refined in 3-valued logic [SRW99, SRW02] became de facto *the shape analysis* in a broader literature. One could envision a potential incorporation of such graph structures in the construction of shape invariants for our separation-logic-based analysis, however, a formal basis for such crossover of approaches is yet to be developed.

Alternative parallelisation primitives. Our use of `pfor` as the primitive parallelisation construct in §2.4.1 is a design choice, rather than an essential feature of our algorithm. We could use a more irregular concurrency annotation, for example safe futures [NZJ08]. In this case, our resource-usage analysis would be mostly unchanged, but our parallelised program would construct a set of syntactically distinct threads, rather than a pipeline of identical threads.

We emphasise that our parallelisation algorithm is targetting programs where ordering is important. As the analysis is syntactically driven by the structure of the loop and the signalling order, it may impose order between iterations that are not strictly enforced by the underlying data structure. For instance, if we would use unordered parallel-**`for`** (e.g., as in the Galois system [PNK+11]) our current analysis would generate a deterministic parallel program that respects some specific sequential order of the loop iterator (contrary to Galois that would, with its speculative parallel execution, allow nondeterminism). Removing ordering between iterations could be achieved by replacing ordered `grant`-`wait` pairs with conventional locks, but this would introduce an extra obligation to show that locks were always acquired as a set by a single thread at once.

CHAPTER 5

EXTENSIONS

Our proof-directed parallelisation algorithm of Chapter 4, assuming a fully automated theorem prover, can be seen as being parametric in (1) the abstract domain for resources, (2) the resource selection procedure, and (3) the level of path-sensitivity. In this chapter, we discuss the way the properties of the algorithm change under various choices of these parameters (§5.1, §5.2 and §5.3). We conclude the chapter with the inter-procedural version of the algorithm (§5.4).

5.1 Resource domain

Our analysis is generic in the choice of abstract domain for resources; any separation logic predicate can potentially be used in place of lseg, for example. The choice of resource predicates affects the success of the analysis particularly strongly, as resources can only be accessed in parallel if they can be expressed disjointly. Stronger domains allow the analysis to split resources more finely, and so (in some cases) making the parallelisation better. However, altering the choice of resources may also force barriers earlier, making the parallelisation worse.

In §2.4.2 we parallelised the `move()` example using the resource predicates lseg, representing list segments, $x \mapsto a$, representing individual heap cells, and $x \overset{c}{\mapsto} a$, representing fractional ownership of a cell. To see how the choice of abstract domain influences the analysis, in this section, we consider two other resource predicates: list segments with length; and list segments with membership.

List segment with length. We introduce the predicate $\mathsf{lsegni}(h, t, n, i)$ which extends lseg with a parameter $n \in \mathbb{Z}$ recording the *length* of the list segment, and a parameter $i \in (0, 1]$ recording the *permission* on the list segment. If $i = 1$ the thread has read-write access; otherwise it has read access only (this follows the behaviour of permission on individual heap cells). We define nodei and lsegni as follows:

$$\begin{aligned} \mathsf{nodei}(x, y, i) &\triangleq x.\mathrm{val} \overset{i}{\mapsto} v' * x.\mathrm{nxt} \overset{i}{\mapsto} y \\ \mathsf{lsegni}(x, t, n, i) &\triangleq (x = t \wedge n = 0 \wedge \mathsf{emp}) \vee (\mathsf{nodei}(x, y', i) * \mathsf{lsegni}(y', t, n{-}1, i)) \end{aligned}$$

We use as input proof the original sequential proof shown in Fig. 2.7. The equivalence $\mathsf{lseg}(h, t) \iff \mathsf{lsegni}(h, t, n', 1)$ allows the analysis to exploit the lsegni predicate even though the sequential proof is written using the coarser lseg predicate.

```
move₁(s, l) {
 $\{\mathsf{lsegni}(\mathtt{s}, n', \mathtt{l}, j')\}$
  i=0;
  x=s.nxt;
  p=s;
  while (i<l && x.nxt!=null) {
   $\{\mathsf{lsegni}(\mathtt{s}, n', \mathtt{l}, j')\}$
    i++;
    p=x;
    x=x.nxt;
  }
 $\{\mathsf{lsegni}(\mathtt{s}, \mathtt{p}, \mathtt{l}, 1)\}$
  while (x.nxt!=null) {
   $\{\mathsf{lsegni}(\mathtt{s}, \mathtt{p}, \mathtt{l}, 1)\}$
    x=x.nxt;
  }
 $\{\mathsf{lsegni}(\mathtt{s}, \mathtt{p}, \mathtt{l}, 1)\}$
  if (p.nxt!=x) {
   ... // omitted
  }
}
$\{\mathsf{node}(\mathtt{s}, n'_1) * \mathsf{node}(n'_1, n'_2) * \mathsf{lseg}(n'_2, \mathsf{null})\}$
```

```
move₂(s, l) {
  i=0;
  x=s.nxt;
 $\{\mathsf{nodei}(\mathtt{s}, n', j')\}$
  p=s;
  while (i<l && x.nxt!=null) {
   $\{\mathsf{lsegni}(\mathtt{s}, n', \mathtt{i}+2, j')\}$
    i++;
    p=x;
    x=x.nxt;
  }
 $\left\{\begin{array}{l}\mathsf{lsegni}(\mathtt{s}, n', \mathtt{l}+2, j') \vee \\ (\mathsf{lsegni}(\mathtt{s}, \mathsf{null}, k', j) \wedge 2 \leq k' \leq \mathtt{l}+1)\end{array}\right\}$
  while (x.nxt!=null) {
   $\left\{\begin{array}{l}\mathsf{lsegni}(\mathtt{s}, n'_1, k', j') * \\ \mathsf{nodei}(n'_1, \mathtt{x}, j') * \mathsf{nodei}(\mathtt{x}, n'_2, j')\end{array}\right\}$
    x=x.nxt;
  }
 $\left\{\begin{array}{l}\mathsf{lsegni}(\mathtt{s}, n'_1, k', j') * \\ \mathsf{nodei}(n'_1, \mathtt{x}, j') * \mathsf{nodei}(\mathtt{x}, \mathsf{null}, j')\end{array}\right\}$
  if (p.nxt!=x) {
   ... // omitted
  }
}
$\{\mathsf{node}(\mathtt{s}, n'_1) * \mathsf{node}(n'_1, n'_2) * \mathsf{lseg}(n'_2, \mathsf{null})\}$
```

Figure 5.1: Redundant and needed resources for `move()` calculated using the lsegni resource predicate.

```
move₁(s, l) {                           move₂(s, l) {
  i=0;                                    i=0;
  x=s.nxt;                                wait(i1);
  p=s;                                    if (2 >= *lr) wait(i2);
  *lr=l;                                  x=s.nxt;
  grant(i1);                              p=s;
  while (i<l && x.nxt!=null) {            while (i<l && x.nxt!=null) {
    i++;                                    if (i+2 >= *lr) wait(i2);
    p=x;                                    i++;
    x=x.nxt;                                p=x;
  }                                         x=x.nxt;
  while (x.nxt!=null)                     }
    x=x.nxt;                              if (l+2 < *lr) wait(i2);
  if (p.nxt!=x) {                         while (x.nxt!=null) {
    ... // omitted                          x=x.nxt;
  }                                       }
  grant(i2);                              if (p.nxt!=x) {
}                                           ... // omitted
                                          }
                                        }
```

Figure 5.2: Parallelisation of `move()` using lsegni.

We will now use the lsegni and node predicates when parallelising our running example `move()` function. Once again, we parallelise a pair of calls to `move()`:

```
move(s,a); move(s,b);
```

The needed and redundant resources for the first and second calls to move calculated using lsegni are shown in Fig. 5.1.

In first call to `move()`, the resource lsegni$(\mathtt{s}, n', \mathtt{l}, j')$ is redundant immediately. This is because `move()` only needs read-write access to the list at the `l`-th node. It can pass read-access to the second call to `move()`. Nodes earlier than `l` in the list can immediately be accessed by the subsequent call.

Conversely, the while-loops in `move()` only need read-only access to the list. This is reflected in the fact that the needed resources all use lsegni. It is only in the final if-statement that `move()` requires write-access to the list.

Fig. 5.2 shows a parallelisation of the example using lsegni. There are several changes over the parallelisation with lseg that we show in Fig. 2.8:

- The value `lr` shared between `move`$_1$`()` and `move`$_2$`()` has been changed from the address of a node, to the position of the node to be moved.
- In `move`$_2$`()`, the calls to `wait(i2)` are conditional on the position value stored in `lr`, rather than on the address of the `.nxt` node.
- In `move`$_1$ the call to `grant(i1)` has been pushed up past the first while-loop.
- The third call to `wait(i2)` has been moved up out of the while loop.

The first two changes make no difference to the degree of parallelisation. The third, moving the call to `grant()` upwards, makes the program more parallel: the subsequent

$$\begin{array}{lrcl}
\text{case } S = \emptyset: & \mathsf{lsegv}(e,f,\emptyset) & \triangleq & \mathsf{nodev}(e,f,\_) \lor \\
& & & (\mathsf{nodev}(e,x',\_) * \mathsf{lsegv}(x',f,\emptyset)) \\
\text{case } S = \{d\}: & \mathsf{lsegv}(e,f,\{d\}) & \triangleq & \mathsf{nodev}(e,f,\{d\}) \lor \\
& & & (\mathsf{nodev}(e,x',\{d\}) * \mathsf{lsegv}(x',f,\emptyset)) \lor \\
& & & (\mathsf{nodev}(e,x',\_) * \mathsf{lsegv}(x',f,\{d\})) \\
\text{case } |S| > 1, d \in S: & \mathsf{lsegv}(e,f,\{S\}) & \triangleq & \mathsf{nodev}(e,x',\{d\}) * \mathsf{lsegv}(x',f,S \setminus \{d\}) \lor \\
& & & \mathsf{nodev}(e,x',\_) * \mathsf{lsegv}(x',f,S)
\end{array}$$

Figure 5.3: Definition of the lsegv, a list segment predicate augmented with a multiset representing the lower bound on the number of particular elements in the segment.

call to `move₂()` needs to wait less time before it can proceed. The change in domain enables this optimisation because it allows the analysis to represent a list segment of a particular fixed length, and to work out that it can be safely transferred to the second call to `move()`.

The fourth change, moving the `wait()` upwards, makes the program less parallel. This is a consequence of sharing the position of the node, rather than its address. In the original parallelisation, the loop could conditionally wait on the address of the node. However, the second loop does not store its position in the list, meaning that it cannot check when it needs to call `wait()`. To be conservative, it must wait for the resource *before* entering the loop.

Thus, there is a subtle interplay between the choice of domain, the particular resources chosen by the analysis, and the parallelisation that can be achieved. (Of course, as the lsegni domain includes the original lseg domain, the analysis could also construct the original parallelisation). The takeaway point is that, simply making the domain richer does not automatically make the parallelisation more successful.

List segment with membership. The list predicates we have considered so far have remembered the structure of the list, but not the values held in it. However, in some cases we can get a better parallelisation if the predicate records the values stored in the list. To do this, we introduce the predicates nodev and lsegv. Respectively, these represent a list node holding a particular value, and a list segment with an associated multiset of values (see Fig. 5.3). Note that in $\mathsf{lsegv}(x, y, S)$ the multiset of values S is a sub-multiset of all the values in the list: if a value is in S, then it must be in the list, but not vice versa.

We apply nodev and lsegv to the functions `build()` and `check()`, defined in Fig. 5.4. The `build()` function constructs a linked list containing a sequence of values matching the regular expression $\mathtt{0}^+\mathtt{1}^+\mathtt{2}^+\mathtt{3}$. The `check()` function reads through the list and returns true if a `1`-value node is present, and zero otherwise. This example is inspired by similar build-and-check examples defined in the SV-Comp benchmark suite [SV-13].

We can parallelise the following program:

```
r = build(); check(r);
```

The possible parallelisations depend on the needed and redundant resources calculated by the analysis. These in turn depend on the choice of resource predicates. The needed and redundant resources calculated without and with lsegv are shown interleaved in the code in Fig. 5.5.

```
build(){                        check(s){
  s=alloc();                      x=s;
  x=s;                            while(x != null){
  do {                              if (x.val=1)
    x.val=0;                          return true;
    x.nxt=alloc();                }
    x=x.nxt;                      return false;
  } while(nondet());            }
  do {
    x.val=1;
    x.nxt=alloc();
    x=x.nxt;
  } while(nondet());
  do {
    x.val=2;
    x.nxt=alloc();
    x=x.nxt;
  } while(nondet());
  x.val=3;
  return s;
}
```

Figure 5.4: Functions `build()`, which constructs a linked list containing values $0^+1^+2^+3$, and `check()`, which checks whether a `1`-valued node is present.

Suppose we only use the lseg and node predicates: then the analysis can only establish that the resources redundant in `build()` are list segments, without expressing any knowledge of the values stored. With lseg, the analysis calculates that `check()` requires access to the whole list. If we use lsegv, then the analysis computes that redundant resources in `build()` are list segments containing particular values. It also calculates that `check()` either needs access to a list segment containing `1`, or a complete list segment—either will suffice.

The difference between parallelisations can be seen in Fig. 5.6. To make the exposition clearer, we only inject barriers into loop-free code, rather than injecting barriers inside loops. On the left, we have the parallelisation possible using lseg. The `grant`-barrier in `build()` waits until after the final loop has terminated. On the right, we have the parallelisation with lsegv. The analysis is able to work out that `check()` only needs access to a list segment containing a `1`-valued node. This needed resource will be satisfied after the second loop. The `grant`-barrier can thus be hoisted to above the third loop, resulting in a better parallelisation.

5.2 Resource selection procedure

Algorithm 4 is parameterised by a function *choose* governing resource selection. In a way, *choose* can be seen as a proxy for external knowledge (e.g., hints provided by a programmer) about likely points for parallelisation. The way how *choose* picks program paths determining the resource selection is not important for the correctness of the algorithm, but different strategies can lead to better or worse parallelisations.

```
build(){
  s=alloc();
  x=s;
  do {
    x.val=0;
    x.nxt=alloc();
    x=x.nxt;
  } while(nondet());
 1. {node(s,n') * lseg(n',x)}
 2. {lsegv(s,x,{0})}
  do {
    x.val=1;
    x.nxt=alloc();
    x=x.nxt;
  } while(nondet());
 1. {node(s,n') * lseg(n',x)}
 2. {lsegv(s,x,{0,1})}
  do {
    x.val=2;
    x.nxt=alloc();
    x=x.nxt;
  } while(nondet());
  x.val=3;
 1. {node(s,n') * lseg(n',null)}
 2. {lsegv(s,null,{0,1,2,3})}
  return s;
}
```

```
check(s){
  x=s;
 1. {node(s,n')}
 2. {node(s,n')}
  while(x != null) {
    if (x.val==1)
      return true;
  }
 1. {node(s,n') * lseg(n',null)}
 2. {(node(s,n') * lseg(n',null)) ∨ lsegv(s,n',{1})}
  return false;
}
```

Figure 5.5: Redundant and needed resources for `build()` and `check()`. Resources labelled (1) are calculated w.r.t. a logic including the default lseg predicate, while resources labelled (2) are calculated w.r.t. a logic including the lsegv predicate.

```
build(){
  s=alloc();
  x=s;
  do {
    x.val=0;
    x.nxt=alloc();
    x=x.nxt;
  } while(nondet());
  do {
    x.val=1;
    x.nxt=alloc();
    x=x.nxt;
  } while(nondet());
  do {
    x.val=2;
    x.nxt=alloc();
    x=x.nxt;
  } while(nondet());
  x.val=3;
  grant(i);
  return s;
}

check(s){
  x=s;
  wait(i);
  while(x != null) {
    if (x.val==1)
      return true;
  }
  return false;
}
```

```
build(){
  s=alloc();
  x=s;
  do {
    x.val=0;
    x.nxt=alloc();
    x=x.nxt;
  } while(nondet());
  do {
    x.val=1;
    x.nxt=alloc();
    x=x.nxt;
  } while(nondet());
  grant(i);
  do {
    x.val=2;
    x.nxt=alloc();
    x=x.nxt;
  } while(nondet());
  x.val=3;
  return s;
}

check(s){
  x=s;
  wait(i);
  while(x != null) {
    if (x.val==1)
      return true;
  }
  return false;
}
```

Figure 5.6: Left: parallelisation of `build()` and `check()` using the lseg predicate. Right: parallelisation using lsegv.

In our examples, we used a very simple, automatic greedy heuristic for the *choose* function. We introduced a simple syntactic partial order $\lhd$ reflecting the coarse-grained size of heap resources, under which:

$$\begin{array}{rcl} \mathsf{emp} & \lhd & e \mapsto f \\ \mathsf{emp} & \lhd & \mathsf{lseg}(e, f) \\ e \mapsto f & \lhd & \mathsf{lseg}(e', f') \wedge e' \neq f', \end{array}$$

and $\Sigma \lhd \Sigma'$ if for every conjunct S in Σ there exists a conjunct S' in Σ' such that $S \lhd S'$. We write $\Sigma \bowtie \Sigma'$ if $\Sigma \lhd \Sigma'$ and $\Sigma' \lhd \Sigma$. Then, given a set of program path pairs $\{(\gamma, \gamma') \mid \gamma \in \mathcal{C} \wedge \gamma' \in \mathsf{Paths} \wedge \gamma \prec \gamma'\}$, we defined a partial order $\sqsubset$ by letting $(\gamma, \gamma') \sqsubset (\delta, \delta')$ if $\mathsf{N}(\delta, \delta') \lhd \mathsf{N}(\gamma, \gamma')$ or $\mathsf{N}(\gamma, \gamma') \bowtie \mathsf{N}(\delta, \delta')$, $\gamma = \delta$ and $\gamma' \prec \delta'$. The *choose* function traversed topologically sorted program-path pairs and picked each pair in turn.

This *choose* function favoured picking "large" resource that are encountered along program paths first, which worked well for our examples. However, in some cases we might produce better parallelisations by picking "smaller" resources first. Such preference of resources can be tuned by an alternative definition of the partial order $\lhd$.

5.3 Path-sensitivity

While the resource-usage analysis (§4.1) is fully path-sensitive modulo loops (i.e. it is path sensitive for if-then-else, but conflates iterations of a loop), the parallelising transformation (§4.2) accounts for path-sensitivity only syntactically. During the execution of the parallelised program precisely one call to `wait()` and `grant()` should occur for each channel. However, barriers may be injected into conditional branches, creating the possibility of missed barriers or unwanted multiple calls to barriers. To ensure that barriers are called exactly once, the parallelising transformation follows the syntactic structure of the program, forcing the calls to `grant()` to occur at program paths that are covering the corresponding calls to `wait()` for the same channel (the condition $\mathcal{C}_\mathsf{N} \preceq \mathcal{C}_\mathsf{R}$ in line 9 of Algorithm 4). This approach is sufficient for all of the examples that we consider.

We could strengthen this approach by making the prover record explicit path conditions in the assertions of the sequential proof and using these conditions semantically. To check that all paths are covered by the points chosen for barrier injection, Algorithm 4 could then disjoin the path conditions and check for tautology. More precisely, if we let $\mathsf{pc}\colon \mathsf{Paths} \to \Pi_\mathsf{V}$ represent the path condition associated with each program path, then we would replace the condition in line 9 of Algorithm 4 with $\bigvee_{\gamma \in \mathcal{C}_\mathsf{R}} \mathsf{pc}(\gamma) \Leftrightarrow \mathsf{true}$. To check that two paths cannot be satisfied at the same time, the algorithm could check that the associated path conditions contradict.

Enriching the parallelising transformation in this way would possibly allow better placement of barriers down conditional branches. For example, the following barrier placement would be forbidden by the syntactic approach, but allowed by the semantic path-sensitive version:

```
void f(int i){
ℓ1:  int v = *x;
ℓ2:  if (v >= i){
ℓ3:  g(y, v);
     }
     else{
ℓ4:  g(x, 0);
     }
}

void g(int* p,
       int v){
ℓ5:  *p = v;
}
```

$$\begin{array}{l}
\mathtt{f_s}\colon \mathtt{x} \mapsto x * \mathtt{y} \mapsto y \\
\ell_1\colon \mathtt{x} \mapsto x * \mathtt{y} \mapsto y \\
\ell_2\colon \mathtt{v} = x \wedge \mathtt{x} \mapsto x * \mathtt{y} \mapsto y \\
\ell_3\colon \mathtt{v} = x \wedge \mathtt{v} \geq \mathtt{i} \wedge \mathtt{x} \mapsto x * \mathtt{y} \mapsto y \\
\ell_4\colon \mathtt{v} = x \wedge \mathtt{v} < \mathtt{i} \wedge \mathtt{x} \mapsto x * \mathtt{y} \mapsto y \\
\mathtt{f_e}\colon (x \geq \mathtt{i} \wedge \mathtt{x} \mapsto x * \mathtt{y} \mapsto x) \vee (x < \mathtt{i} \wedge \mathtt{x} \mapsto 0 * \mathtt{y} \mapsto y) \\
\\
\mathtt{g_s}\colon \mathtt{p} \mapsto \_ \wedge \mathtt{v} = v \\
\ell_5\colon \mathtt{p} \mapsto \_ \wedge \mathtt{v} = v \\
\mathtt{g_e}\colon \mathtt{p} \mapsto v
\end{array}$$

Figure 5.7: Left: inter-procedural version of the example from §2. Right: associated assertions in the sequential proof.

```
if (P)
  grant(i);
if (¬P)
  grant(i);
```

```
if (Q)
  wait(i);
if (¬Q)
  wait(i);
```

5.4 Inter-procedural analysis

We conclude this chapter by providing an inter-procedural version of our parallelisation algorithm from §4, which allows exploiting parallelisation opportunities across function calls.

Programming language. We assume the following heap-manipulating programming language with functions:

$$\begin{array}{rcll}
e & ::= & \mathtt{x} \mid \mathtt{nil} \mid t(\bar{e}) \mid \ldots & \textit{(expressions)} \\
b & ::= & \mathtt{true} \mid \mathtt{false} \mid e = e \mid e \neq e \mid \ldots & \textit{(booleans)} \\
a & ::= & \ldots & \textit{(primitive commands)} \\
C & ::= & C; C \mid \ell\colon \mathtt{skip} \mid \ell\colon a \mid \ell\colon \mathtt{x} := \mathtt{f}(\bar{e}) \mid \ell\colon \mathtt{return}\ e & \\
 & \mid & \ell\colon \mathtt{if}(b)\,\{\,C\,\}\,\mathtt{else}\,\{\,C\,\} \mid \ell\colon \mathtt{while}(b)\,\{\,C\,\} &
\end{array}$$

Program representation. As in §3, we represent the program by an ordered forest $\mathcal{T}$, with the addition of edges (ℓ, ℓ') such that ℓ corresponds to a label of a function call $\mathtt{x} := \mathtt{f}(\bar{e})$ and $\ell' \in \langle \mathtt{f_s}, \mathtt{f_e} \rangle$, where $\mathtt{f_s}$ and $\mathtt{f_e}$ are labels of the predecessor of the first (resp. the successor of the last) command in the body of f.

Example 5.1. The left-hand side of Fig. 5.7 shows example labelling containing a function call.

Sequential proof. The inter-procedural sequential proof $\mathfrak{P}$ assumes use of modular specification for each function. That is, each function is associated with a pre- and post-condition which are applied in the proof by using the frame rule and variable substitutions. The right-hand side of Fig. 5.7 shows a proof which is in such form.

Example 5.2. Fig. 5.7 shows example of an inter-procedural sequential proof.

To gain context-sensitivity, our analysis will dive through function calls and refer to the assertions in the local function with respect to the variables of the caller, all the way up to the top-most `work` function. We therefore define a way for lifting local assertions from a function to their global contexts.

For $\gamma \in \mathsf{Paths}$ such that $|\gamma| = n$ and $\ell_1, \ldots, \ell_n$ are the labels corresponding to the function calls (in the order of occurrence as in γ) we define the lifted ("inlined") proof assertion $\mathfrak{P}(\gamma)$ as

$$\mathfrak{F}(\ell_1) * (\mathfrak{F}(\ell_2) * ... (\mathfrak{F}(\ell_n) * \mathfrak{P}(\gamma[m])[\Omega(\ell_n)])[\Omega(\ell_{n-1})]...)[\Omega(\ell_1)]$$

Intuitively, the assertion $\mathfrak{P}(\gamma)$ represents the "global" proof state at γ (including the framed portions) in terms of `work`'s variables.

Resource-usage analysis. The inter-procedural version of the resource-usage analysis resembles the intra-procedural version, with the addition being the computation of needed and redundant resources across function calls.

The function `Needed-Func` (Alg. 6) is similar to `Needed-Block` (Alg. 1) from §4.1 but lifts the within-blocks computation to the context-sensitive inter-procedural level. Given program paths γ_s and γ_e, `Needed-Func(`γ_s, γ_e`)` works by successively pushing backwards $\mathfrak{P}(\gamma_e)^{\Pi} \wedge \mathsf{emp}$ similarly as Alg. 2. In phase A, Alg. 6 steps backwards from γ_e towards the outermost calling context in the function-invocation hierarchy. This context, represented as the longest common prefix of γ_s and γ_e, is the dominator of the two functions in which γ_s and γ_e are found in the function call graph (which is well-defined since there are no higher-order functions). Phase B of the algorithm keeps stepping backwards, but proceeds inwards into the function-invocation hierarchy towards γ_s. Both phases of Alg. 6 use Alg. 1 to compute the needed resources in-between function call boundaries: in phase A we establish the needed assertions from the dominating point to γ_e, and in phase B from γ_s to the dominating point.

Since the invariants of the input proof are written in terms of the outermost calling context, comparing locally computed specifications with these invariants requires the local specifications to be recast in terms of the outer context. In the first line of phase A we construct a variable substitution ϱ that recasts the assertion in terms of the calling context at the start of γ_e. The second line constructs $\Delta[\varrho^{-1}]$—the running state recast in terms of γ_e's starting context; this is typically the context defined by the `work()` function used in a `pfor` command. `Needed-Block` constructs a new needed state up to the start of the current block. Finally, ϱ recasts the resulting state back into the current context. When a function call is reached, we unwind the variable substitution by one call since we now have moved from the callee's context to a caller's. Operations in phase B are similar.

We calculate the redundant resources between arbitrary program paths analogously as in function `Redundant` (Alg. 3).

Algorithm 6: Computing needed resources across function calls.

```
Needed-Func($\gamma_s$: Paths, $\gamma_e$: Paths)
begin
    $k := |\gamma_e|$;
    $\Delta := \mathfrak{P}(\gamma_e)^{\Pi} \wedge \mathsf{emp}$;
    while $k > |\gamma_s \curlywedge \gamma_e| + 1$ do // Phase A
        $\varrho := \Omega(\gamma_e[1..k])$;
        $\Delta :=$ Needed-Block($(\gamma_e[k]_{\uparrow}, \gamma_e[k]), \Delta[\varrho^{-1}]$) $[\varrho]$;
        $k := k - 1$;
        if $\mathsf{cmd}(\gamma_e[k])$ is function call then $\Delta := \Delta[\Omega(\gamma_e[k])^{-1}]$;
        ;
    end
    $\varrho := \Omega(\gamma_s[1..k])$;
    $\Delta :=$ Needed-Block($(\gamma_e[k], \gamma_s[k]), \Delta[\varrho^{-1}]$) $[\varrho]$;
    while $k < |\gamma_s|$ do // Phase B
        if $\mathsf{cmd}(\gamma_s[k])$ is function call then $\Delta := \Delta[\Omega(\gamma_s[k])]$;
        ;
        $k := k + 1$;
        $\varrho := \Omega(\gamma_s[1..k])$;
        $\Delta :=$ Needed-Block($(\gamma_s[k], \gamma_s[k]_{\downarrow}), \Delta[\varrho^{-1}]$) $[\varrho]$;
    end
    return $\Delta$;
end
```

Example 5.3. Fig. 5.8 and Fig 5.9 show the needed and redundant maps computed by the intra-procedural resource-usage analysis on the example in Fig. 5.7.

Parallelising transformation. The inter-procedural version of the parallelising transformation is mainly the same as the intra-procedural version, with the difference being in how grant and wait barriers are inserted. To generate the parallel function $\mathtt{work}'(\bar{\mathtt{i}}_r^{(p)}, \bar{\mathtt{i}}_r, \mathtt{env}^{(p)}, \mathtt{env})$ in $\mathbb{P}_{par}$ we replace the intra-procedural steps (3) and (4) with the following steps (3), (4) and (5):

3. For each $\gamma = \ell_1 \ldots \ell_n \in \mathrm{dom}(\mathsf{released}) \cup \mathrm{dom}(\mathsf{acquired})$ let $\ell_{k_1}, \ldots \ell_{k_m}$ be the labels in γ corresponding to function calls. Then for each $\gamma_j := \ell_1 \ldots \ell_{k_j}$ we create in $\mathbb{P}_{par}$ an identical copy $\mathtt{f}'$ of the function $\mathtt{f}$ called at ℓ_{k_j} and replace the call to $\mathtt{f}$ with the call to $\mathtt{f}'$. Let us denote by $\mathsf{tr}(\gamma')$ the program path in $\mathbb{P}_{par}$ corresponding to γ' after this transformation has been applied for all $\gamma \in \mathrm{dom}(\mathsf{released}) \cup \mathrm{dom}(\mathsf{acquired})$.
4. For each $\gamma \in \mathrm{dom}(\mathsf{acquired})$ such that $\mathsf{acquired}(\gamma) = r$ we insert a wait barrier $\mathtt{wait}(\mathtt{i}_r^{(p)})$ between program paths $\mathsf{tr}(\mathsf{pred}(\gamma))$ and $\mathsf{tr}(\gamma)$.
5. For each $\gamma \in \mathrm{dom}(\mathsf{released})$ such that $\mathsf{released}(\gamma) = (r, \_)$, between program paths $\mathtt{tr}(\mathsf{pred}(\gamma))$ and $\mathtt{tr}(\gamma)$ we insert a sequence of assignments of the form $\mathtt{env}(\mathtt{i}_r)[''\mathrm{y}''] := \mathtt{y}$ for every local variable $\mathtt{y}$, followed by a grant barrier $\mathtt{grant}(\mathtt{i}_r)$.

	ℓ_2	$\ell_2\ell_3$	$\ell_2\ell_3\ell_5$	$\ell_2\ell_3\mathtt{g_e}$	$\ell_2\ell_4$	$\ell_2\ell_4\ell_5$	$\ell_2\ell_4\mathtt{g_e}$	$\mathtt{f_e}$
ℓ_1	$\mathtt{x} \mapsto x$	$x \geq \mathtt{i} \wedge \mathtt{x} \mapsto x$	$x \geq \mathtt{i} \wedge \mathtt{x} \mapsto x$	$x \geq \mathtt{i} \wedge \mathtt{x} \mapsto x * \mathtt{y} \mapsto y$	$x < \mathtt{i} \wedge \mathtt{x} \mapsto x$	$x < \mathtt{i} \wedge \mathtt{x} \mapsto x$	$x < \mathtt{i} \wedge \mathtt{x} \mapsto x$	$(x \geq \mathtt{i} \wedge \mathtt{x} \mapsto x * \mathtt{y} \mapsto y) \vee (x < \mathtt{i} \wedge \mathtt{x} \mapsto x)$
ℓ_2		$\mathtt{v} = x \wedge \mathtt{v} \geq \mathtt{i}$	$\mathtt{v} = x \wedge \mathtt{v} \geq \mathtt{i}$	$\mathtt{v} = x \wedge \mathtt{v} \geq \mathtt{i} * \mathtt{y} \mapsto y$	$\mathtt{v} = x \wedge \mathtt{v} < \mathtt{i}$	$\mathtt{v} = x \wedge \mathtt{v} < \mathtt{i}$	$\mathtt{v} = x \wedge \mathtt{v} < \mathtt{i} \wedge \mathtt{x} \mapsto x$	$(\mathtt{v} = x \wedge \mathtt{v} \geq \mathtt{i} * \mathtt{y} \mapsto y) \vee (\mathtt{v} = x \wedge \mathtt{v} < \mathtt{i} \wedge \mathtt{x} \mapsto x)$
$\ell_2\ell_3$			$\mathtt{v} = x \wedge \mathtt{v} \geq \mathtt{i}$	$\mathtt{v} = x \wedge \mathtt{v} \geq \mathtt{i} * \mathtt{y} \mapsto y$				$\mathtt{v} = x \wedge \mathtt{v} \geq \mathtt{i} * \mathtt{y} \mapsto y$
$\ell_2\ell_3\ell_5$				$\mathtt{v} = x \wedge \mathtt{v} \geq \mathtt{i} * \mathtt{y} \mapsto y$				$\mathtt{v} = x \wedge \mathtt{v} \geq \mathtt{i} * \mathtt{y} \mapsto y$
$\ell_2\ell_3\mathtt{g_e}$								$\mathtt{v} = x \wedge \mathtt{v} \geq \mathtt{i}$
$\ell_2\ell_4$						$\mathtt{v} = x \wedge \mathtt{v} < \mathtt{i}$	$\mathtt{v} = x \wedge \mathtt{v} < \mathtt{i} * \mathtt{x} \mapsto x$	$\mathtt{v} = x \wedge \mathtt{v} < \mathtt{i} * \mathtt{x} \mapsto x$
$\ell_2\ell_4\ell_5$							$\mathtt{v} = x \wedge \mathtt{v} < \mathtt{i} * \mathtt{x} \mapsto x$	$\mathtt{v} = x \wedge \mathtt{v} < \mathtt{i} * \mathtt{x} \mapsto x$
$\ell_2\ell_4\mathtt{g_e}$								$\mathtt{v} = x \wedge \mathtt{v} < \mathtt{i}$

Figure 5.8: The needed map computed by the algorithm `Needed` for the example in Fig. 5.7. (not shown trivial entries; an empty cell means not defined)

Example 5.4. We show the resulting two-iteration parallelisation and unbounded iteration parallelisation for function `f()` from Fig. 5.7, obtained by applying our interprocedural parallelising transformation, in Fig. 5.10 and Fig. 5.11.

In Fig. 5.11 the transformation generates two versions of function `g`, specialised to the two contexts in which it is invoked. Function `ga` executes when `v >= i`. Here, the executing thread needs write access to `y`, which it acquires by calling `wait(wyp)`. Function `gb` needs write access to `x`, but it runs in a context where the resource `x` has already been acquired. Consequently, it need not call `wait()`. Both versions release a resource to the following thread using `grant()` before returning.

	ℓ_2	$\ell_2\ell_3$	$\ell_2\ell_3\ell_5$	$\ell_2\ell_3\mathsf{g_e}$	$\ell_2\ell_4$	$\ell_2\ell_4\ell_5$	$\ell_2\ell_4\mathsf{g_e}$	$\mathtt{f_e}$
ℓ_1	$\mathtt{y} \mapsto y$	$(x \geq \mathtt{i} \wedge \mathtt{y} \mapsto y) \vee (x < \mathtt{i} \wedge \mathtt{x} \mapsto x * \mathtt{y} \mapsto y)$	$(x \geq \mathtt{i} \wedge \mathtt{y} \mapsto y) \vee (x < \mathtt{i} \wedge \mathtt{x} \mapsto x * \mathtt{y} \mapsto y)$	$x < \mathtt{i} \wedge \mathtt{x} \mapsto x * \mathtt{y} \mapsto y$	$(x \geq \mathtt{i} \wedge \mathtt{x} \mapsto x * \mathtt{y} \mapsto y) \vee (x < \mathtt{i} \wedge \mathtt{y} \mapsto y)$	$(x \geq \mathtt{i} \wedge \mathtt{x} \mapsto x * \mathtt{y} \mapsto y) \vee (x < \mathtt{i} \wedge \mathtt{y} \mapsto y)$	$(x \geq \mathtt{i} \wedge \mathtt{x} \mapsto x * \mathtt{y} \mapsto y) \vee (x < \mathtt{i} \wedge \mathtt{y} \mapsto y)$	$x < \mathtt{i} \wedge \mathtt{y} \mapsto y$
ℓ_2		$\mathtt{v} = x \wedge \mathtt{x} \mapsto x * \mathtt{y} \mapsto y$	$\mathtt{v} = x \wedge \mathtt{x} \mapsto x * \mathtt{y} \mapsto y$	$(\mathtt{v} = x \wedge \mathtt{v} \geq \mathtt{i} \wedge \mathtt{x} \mapsto x) \vee (\mathtt{v} = x \wedge \mathtt{v} < \mathtt{i} \wedge \mathtt{x} \mapsto x * \mathtt{y} \mapsto y)$	$\mathtt{v} = x \wedge \mathtt{x} \mapsto x * \mathtt{y} \mapsto y$	$\mathtt{v} = x \wedge \mathtt{x} \mapsto x * \mathtt{y} \mapsto y$	$(\mathtt{v} = x \wedge \mathtt{v} \geq \mathtt{i} \wedge \mathtt{x} \mapsto x * \mathtt{y} \mapsto y) \vee (\mathtt{v} = x \wedge \mathtt{v} < \mathtt{i} \wedge \mathtt{y} \mapsto y)$	$(\mathtt{v} = x \wedge \mathtt{v} \geq \mathtt{i} \wedge \mathtt{x} \mapsto x) \vee (\mathtt{v} = x \wedge \mathtt{v} < \mathtt{i} \wedge \mathtt{y} \mapsto y)$
$\ell_2\ell_3$			$\mathtt{v} = x \wedge \mathtt{v} \geq \mathtt{i} \wedge \mathtt{x} \mapsto x * \mathtt{y} \mapsto y$	$\mathtt{v} = x \wedge \mathtt{v} \geq \mathtt{i} \wedge \mathtt{x} \mapsto x$				$\mathtt{v} = x \wedge \mathtt{v} \geq \mathtt{i} \wedge \mathtt{x} \mapsto x$
$\ell_2\ell_3\ell_5$				$\mathtt{v} = x \wedge \mathtt{v} \geq \mathtt{i} \wedge \mathtt{x} \mapsto x$				$\mathtt{v} = x \wedge \mathtt{v} \geq \mathtt{i} \wedge \mathtt{x} \mapsto x$
$\ell_2\ell_3\mathsf{g_e}$								$\mathtt{v} = x \wedge \mathtt{v} \geq \mathtt{i} \wedge \mathtt{x} \mapsto x * \mathtt{y} \mapsto y$
$\ell_2\ell_4$						$\mathtt{v} = x \wedge \mathtt{v} < \mathtt{i} \wedge \mathtt{x} \mapsto x * \mathtt{y} \mapsto y$	$\mathtt{v} = x \wedge \mathtt{v} < \mathtt{i} \wedge \mathtt{y} \mapsto y$	$\mathtt{v} = x \wedge \mathtt{v} < \mathtt{i} \wedge \mathtt{y} \mapsto y$
$\ell_2\ell_4\ell_5$							$\mathtt{v} = x \wedge \mathtt{v} < \mathtt{i} \wedge \mathtt{y} \mapsto y$	$\mathtt{v} = x \wedge \mathtt{v} < \mathtt{i} \wedge \mathtt{y} \mapsto y$
$\ell_2\ell_4\mathsf{g_e}$								$\mathtt{v} = x \wedge \mathtt{v} < \mathtt{i} \wedge \mathtt{x} \mapsto x * \mathtt{y} \mapsto y$

Figure 5.9: The redundant map computed by the algorithm `Redundant` for the example in Fig. 5.7. (not shown trivial entries; empty entry means not defined)

```
f1(i, wx, wy) {
  local v=*x;
  if (v>=i) {
    grant(wx);
    ga1(y, v);
  }
  else {
    grant(wy);
    gb1(x, 0)
  }
}

ga1(*p, v, wy) {
  *p=v;
  grant(wy);
}


gb1(*p, v, wx) {
  *p=v;
  grant(wx);
}
```

```
f2(i, wx, wy) {
  wait(wx);
  local v=*x;
  if (v>=i) {
    ga2(y, v);
  }
  else {
    gb2(x, 0);
    wait(wy);
  }
}

ga2(*p, v, wy) {
  wait(wy);
  *p=v;
}


gb2(*p, v) {
  *p=v;
}
```

Figure 5.10: Parallelisation of two sequential calls to `f()` from Fig. 5.7.

```
void main() {
  local i, n = nondet();
  x = alloc();
  y = alloc();
  chan wx', wy';
  chan wx = newchan();
  chan wy = newchan();
  grant(wx);
  grant(wy);
  for (i = 0; i++; i < n) {
    wx' = newchan();
    wy' = newchan();
    fork(f(i,wx,wy,wx',wy'));
    wx = wx';
    wy = wy';
  }
  wait(wx);
  wait(wy);
}
```

```
f(i, wxp, wyp, wx, wy){           ga(*p, v, wyp, wy){
  wait(wxp);                         wait(wyp);
  local v = *x;                      *p = v;
  if (v >= i) {                      grant(wy);
    grant(wx);                     }
    ga(y, v, wy);
  }
  else {                           gb(*p, v, wx){
    wait(wyp);                       *p = v;
    grant(wy);                       grant(wx);
    gb(x, 0, wx);                  }
  }
}
```

Figure 5.11: Parallelisation of `f()` from Fig. 5.7 for unbounded iteration.

CHAPTER 6

SOUNDNESS

This chapter proves the soundness of our proof-directed parallelisation algorithm, i.e., that it does not introduce any new behaviours to the original program, provided the original program terminates. Essentially, we establish a simulation property which states that for any non-faulting trace of the parallelised program, all the combined resources held by channels and parallel forked threads correspond to the resource held in some sequential trace. We begin, in §6.1 by defining the operational semantics of the language which we used in the formalisation of the algorithm. In §6.2 we prove that our parallelisation is behaviour preserving, and then in §6.3 that it does not introduce non-termination to the parallelised program.

6.1 Operational semantics

We proceed to define an operational semantics of the multi-threaded core language to which the programs generated by our parallelising transformation are translated. As we have seen in §4, the syntax of this language is similar to the sequential core language described in §3.1, but is additionally equipped with threads and operations on channels (`newchan`, `grant` and `wait`).

Our semantics makes several simplifying assumptions: (1) We forbid memory disposal, and so can assume that allocation always gives fresh locations. This restriction is introduced because full memory allocation (with memory disposition and reuse) may allow parallelisation to introduce new behaviours (see §4.4); (2) to simplify the presentation, our semantics does not include function definitions or calls. Data races are interpreted as faults in this semantics.

The operational semantics has two levels, a labelled thread-local semantics ('$\rightsquigarrow$'), and a global semantics ('$\Longmapsto$') defined in terms of the local semantics. Every transition in the global semantics corresponds to a transition for some thread in the local semantics. Our semantics is *annotated* in the sense that it keeps track of the resources that are associated with threads and channels. These distinctions would not be present in the operational semantics of the real machine.

Thread-local semantics. The semantics assumes the following basic sets: Prog, programs; Tid, thread identifiers, ordered by $<$; Cid, channel identifiers; Var, variable names;

Loc, heap locations; Val, primitive values (which include locations). We assume that the sets Tid, Cid and Loc are infinite, and that $\mathsf{Tid} \uplus \mathsf{Cid} \uplus \mathsf{Loc} \subseteq \mathsf{Val}$.

A *local state* in LState consists of a *stack* mapping variables to values and a *heap* mapping locations to values:

$$\begin{aligned}\sigma_s \in{} & \mathsf{Stack} : \mathsf{Var} \rightharpoonup \mathsf{Val} \\ \sigma_h \in{} & \mathsf{Heap} : \mathsf{Loc} \rightharpoonup \mathsf{Val} \\ \sigma \in{} & \mathsf{LState} : \mathsf{Stack} \times \mathsf{Heap}\end{aligned}$$

(Note that we do not support address arithmetic.)

A *channel state* in CState consists of a pair of sets, respectively recording the channels the thread can **wait** for and **grant** on:

$$(\omega_{\mathrm{w}}, \omega_{\mathrm{g}}) \in \mathsf{CState} : \mathcal{P}(\mathsf{Cid}) \times \mathcal{P}(\mathsf{Cid})$$

A *thread state* in TState consists of a program (i.e. command), a local state, a channel state, and a set of thread identifiers of child threads:

$$\mathsf{TState} : \mathsf{Prog} \times \mathsf{LState} \times \mathsf{CState} \times \mathcal{P}(\mathsf{Tid})$$

The thread-local semantics associates each thread with a resource which can be safely manipulated by that thread. Globally visible events, meaning forking, creating channels, granting and waiting, and heap allocation are modelled by labels on the thread-local transition relation. These labels are used to ensure that global events are propagated to all threads. The set of labels, Label, is defined as follows:

$$\begin{aligned}\mathsf{Label} \triangleq{} & (\{\mathbf{fork}\} \to \mathsf{Tid} \times \mathsf{Prog} \times \mathsf{LState} \times \mathsf{CState}) \\ & \uplus (\{\mathbf{newchan}\} \to \mathsf{Cid}) \\ & \uplus (\{\mathbf{wait}, \mathbf{grant}\} \to \mathsf{Cid} \times \mathsf{Heap}) \\ & \uplus (\{\mathbf{alloc}\} \to \mathsf{Loc})\end{aligned}$$

The annotated thread-local semantics is defined by a labelled transition relation $\rightsquigarrow \in \mathcal{P}(\mathsf{TState} \times \mathsf{TState} \times \mathsf{Label})$, whose rules are given in Fig. 6.1.

The rules use a join operator on heaps $\oplus : \mathsf{Heap} \times \mathsf{Heap} \rightharpoonup \mathsf{Heap}$ by union of functions, defined only if the two heaps' domains are disjoint. The join operator $\oplus : \mathsf{LState} \times \mathsf{LState} \rightharpoonup \mathsf{LState}$ is defined as identity on stacks and join on heaps:

$$\sigma \oplus \sigma' \triangleq (\sigma_s, \sigma_h \oplus \sigma'_h) \qquad \text{if } \sigma_s = \sigma'_s$$

The semantics assumes a set of primitive commands $\mathsf{Prim} : \mathcal{P}(\mathsf{LState} \times \mathsf{LState})$. We assume for any command $c \in \mathsf{Prim}$ that:

- For any states σ_1 and σ_2 such that $(\sigma_1 \oplus \sigma_2, \sigma') \in c$, if there exists a transition $(\sigma_1, \sigma'_1) \in c$ then $\sigma'_1 \oplus \sigma_2 = \sigma'$.
- For any transition $(\sigma, \sigma') \in c$ and state σ_2, if $\sigma \oplus \sigma_2$ is well-defined, then there exists a transition $(\sigma \oplus \sigma_2, \sigma'') \in c$ and $\sigma'' = \sigma' \oplus \sigma_2$.
- For any states σ_1 and σ_2, if there exists no σ' such that such that $(\sigma_1 \oplus \sigma_2, \sigma') \notin c$, then $(\sigma_1, \sigma') \notin c$.

$$\frac{(\sigma,\sigma') \in c}{(c,\sigma,\omega,\gamma) \rightsquigarrow (\mathbf{skip},\sigma',\omega,\gamma)} \qquad \frac{\neg\exists\sigma'.\,(\sigma,\sigma') \in c}{(c,\sigma,\omega,\gamma) \rightsquigarrow \mathbf{abort}}$$

$$\frac{\sigma,\omega \not\models P * \mathsf{true}}{(\mathbf{fork}_{[P]}C,\sigma,\omega,\gamma) \rightsquigarrow \mathbf{abort}} \qquad \frac{\sigma',\omega' \models P \quad \sigma = \sigma' \oplus \sigma'' \quad \omega = \omega' \uplus \omega'' \quad t \text{ fresh}}{(\mathbf{fork}_{[P]}C,\sigma,\omega,\gamma) \overset{\mathbf{fork}\,(t,C,\sigma',\omega')}{\rightsquigarrow} (\mathbf{skip},\sigma'',\omega'',\gamma \uplus \{t\})}$$

$$\frac{s \text{ fresh}}{(x := \mathbf{newchan}\,,\sigma,\omega,\gamma) \overset{\mathbf{newchan}\,(s)}{\rightsquigarrow} (\mathbf{skip},(\sigma_s[x \mapsto s],\sigma_h),(\omega_{\mathrm{w}} \uplus \{s\},\omega_{\mathrm{g}} \uplus \{s\}),\gamma)}$$

$$\frac{\sigma,(\emptyset,\emptyset) \not\models P * \mathsf{true}}{(\mathbf{grant}_{[P]}E,\sigma,\omega,\gamma) \rightsquigarrow \mathbf{abort}} \qquad \frac{[\![E]\!]_{\sigma_{\mathrm{s}}} = s \quad s \in \omega_{\mathrm{g}} \quad \sigma = \sigma' \oplus \sigma'' \quad \sigma',(\emptyset,\emptyset) \models P}{(\mathbf{grant}_{[P]}E,\sigma,\omega,\gamma) \overset{\mathbf{grant}\,(s,\sigma')}{\rightsquigarrow} (\mathbf{skip},\sigma'',(\omega_{\mathrm{w}},\omega_{\mathrm{g}} \setminus \{s\}),\gamma)}$$

$$\frac{[\![E]\!]_{\sigma_{\mathrm{s}}} = s \quad s \in \omega_{\mathrm{w}}}{(\mathbf{wait}\,E,\sigma,\omega,\gamma) \overset{\mathbf{wait}\,(s,\sigma')}{\rightsquigarrow} (\mathbf{skip},\sigma \oplus \sigma',(\omega_{\mathrm{w}} \setminus \{s\},\omega_{\mathrm{g}}),\gamma)}$$

$$\frac{(C,\sigma,\omega,\gamma) \rightsquigarrow (C',\sigma',\omega',\gamma')}{(C;C'',\sigma,\omega,\gamma) \rightsquigarrow (C';C'',\sigma',\omega',\gamma')} \qquad \frac{}{(\mathbf{skip};C,\sigma,\omega,\gamma) \rightsquigarrow (C,\sigma,\omega,\gamma)}$$

$$\frac{(C,\sigma,\omega,\gamma) \rightsquigarrow \mathbf{abort}}{(C;C',\sigma,\omega,\gamma) \rightsquigarrow \mathbf{abort}}$$

$$\frac{[\![B]\!]_{\sigma_{\mathrm{s}}} = \mathsf{tt}}{(\mathbf{if}\,(B)\,C_1\,\mathbf{else}\,C_2,\sigma,\omega,\gamma) \rightsquigarrow (C_1,\sigma,\omega,\gamma)} \qquad \frac{[\![B]\!]_{\sigma_{\mathrm{s}}} = \mathsf{ff}}{(\mathbf{if}\,(B)\,C_1\,\mathbf{else}\,C_2,\sigma,\omega,\gamma) \rightsquigarrow (C_2,\sigma,\omega,\gamma)}$$

$$\frac{[\![B]\!]_{\sigma_{\mathrm{s}}} = \mathsf{tt}}{(\mathbf{while}\,(B)\,C,\sigma,\omega,\gamma) \rightsquigarrow (C;\mathbf{while}\,(B)\,C,\sigma,\omega,\gamma)} \qquad \frac{[\![B]\!]_{\sigma_{\mathrm{s}}} = \mathsf{ff}}{(\mathbf{while}\,(B)\,C,\sigma,\omega,\gamma) \rightsquigarrow (\mathbf{skip},\sigma,\omega,\gamma)}$$

$$\frac{l \notin \mathrm{dom}(\sigma_h)}{(x := \mathbf{alloc},\sigma,\omega,\gamma) \overset{\mathbf{alloc}(l)}{\rightsquigarrow} (\mathbf{skip},(\sigma_s[x \mapsto l],\sigma_h \uplus [l \mapsto 0]),\omega,\gamma)}$$

Figure 6.1: Annotated thread-local operational semantics.

Besides capturing a semantic relation between local states, primitive commands also have a syntactic representation (e.g., the name of the command itself). We overload these notions in the rules to allow us flexibility in characterising effects without having to choose a fixed command set *a priori*. It is straightforward to define an interpretation function that maps the syntactic representation of a command with its relational definition.

The effect of evaluating primitive command c in state $\langle c, \sigma, \omega, \gamma \rangle$ is a new state $\langle \mathbf{skip}, \sigma', \omega, \gamma \rangle$ if $(\sigma, \sigma') \in c$. If command c can effect no transition from local state σ, the thread aborts.

A thread may fork a child. We associate a syntactic assertion P with a **fork** command that captures the resources required by the child thread. (The existence of a syntactic separation logic proof means we can always annotate a *fork* command with such assertions.) If the assertion is not satisfiable within the current local state (i.e., $\sigma, \omega \not\models P * \mathsf{true}$), the thread aborts. Otherwise, we can partition the state, associating the resources (memory and channels) needed by the forked thread (denoted as σ' and ω' in the rule) from those required by the parent. The effect annotation $\mathbf{fork}\ (t, C, \sigma', \omega')$ records this action. We assume that the fresh thread identifier t created on each **fork** invocation is strictly greater (with respect to $<$) than previous ones used by any thread.

Creating a new channel with **newchan** augments the set of channel identifiers for both **wait** and **grant** actions.

In order for a thread to **grant** resources to another thread, the assertion that defines the structure of these resources must hold in the current local state; note that the only resources that can be propagated along channels are locations and values in the thread's local state—channel identifiers can only be distributed at fork-time. If the assertion indeed holds, then the resources it describes can be transferred on the channel. This semantics ensures all concurrently executing threads operate over disjoint portions of shared memory. The semantics of **wait** transfers a set of resources to the executing thread whose composition is determined by the assertion on the corresponding **grant** that communicates on channel s. The rules for sequencing and loops are standard.

Memory allocation annotates the transition with an allocation label $\mathbf{alloc}(l)$ for fresh location l.

Global semantics. The global semantics is defined as a transition relation $\Longmapsto \in \mathcal{P}(\mathsf{GState} \times \mathsf{GState})$, with GState defined as follows:

$$\begin{aligned} \delta \in{}& \mathsf{TMap} : \mathsf{Tid} \rightharpoonup \mathsf{TState} \\ \eta \in{}& \mathsf{CMap} : \mathsf{Cid} \rightharpoonup \mathsf{Heap} \uplus \{\triangle, \triangledown\} \\ \kappa \in{}& \mathsf{GState} : \mathsf{TMap} \times \mathsf{CMap} \times \mathcal{P}(\mathsf{Loc}) \end{aligned}$$

The rules of the global semantics are given in Fig. 6.2. Every transition in the global semantics corresponds to a transition for some thread t in the local semantics. The global semantics models the pool of active threads with a thread-map in TMap, and models the resources held by channels with a channel-map in CMap. The set of locations in $\mathcal{P}(\mathsf{Loc})$ represents unallocated locations in the heap. Because we never return locations to this set, all allocated locations are fresh.

Given a channel map $\eta \in \mathsf{CMap}$, a given channel $s \in \mathrm{dom}(\eta)$ can be either allocated but unused, denoted by $\eta(s) = \triangle$, in use, denoted by $\eta(s) \in \mathsf{Heap}$, or finalised, denoted

$$\frac{(C,\sigma,\omega,\gamma) \rightsquigarrow (C',\sigma',\omega',\gamma')}{([t \mapsto C,\sigma,\omega,\gamma] \uplus \delta,\eta,\mathcal{L}) \Longmapsto ([t \mapsto C',\sigma',\omega',\gamma'] \uplus \delta,\eta,\mathcal{L})}$$

$$\frac{(C,\sigma,\omega,\gamma) \overset{\mathbf{fork}\,(t_2,C_2,\sigma_2,\omega_2)}{\rightsquigarrow} (C',\sigma',\omega',\gamma')}{([t_1 \mapsto C,\sigma,\omega,\gamma] \uplus \delta,\eta,\mathcal{L}) \Longmapsto ([t_1 \mapsto C',\sigma',\omega',\gamma'] \uplus [t_2 \mapsto C_2,\sigma_2,\omega_2,\emptyset] \uplus \delta,\eta,\mathcal{L})}$$

$$\frac{(C,\sigma,\omega,\gamma) \overset{\mathbf{newchan}\,(s)}{\rightsquigarrow} (C',\sigma',\omega',\gamma') \quad s \notin \mathrm{dom}(\eta)}{([t \mapsto C,\sigma,\omega,\gamma] \uplus \delta,\eta,\mathcal{L}) \Longmapsto ([t \mapsto C',\sigma',\omega',\gamma'] \uplus \delta,\eta \uplus [s \mapsto \triangle],\mathcal{L})}$$

$$\frac{(C,\sigma,\omega,\gamma) \overset{\mathbf{grant}\,(s,\sigma)}{\rightsquigarrow} (C',\sigma',\omega',\gamma') \quad \eta(s) = \triangle}{([t \mapsto C,\sigma,\omega,\gamma] \uplus \delta,\eta,\mathcal{L}) \Longmapsto ([t \mapsto C',\sigma',\omega',\gamma'] \uplus \delta,\eta[s \mapsto \sigma],\mathcal{L})}$$

$$\frac{(C,\sigma,\omega,\gamma) \overset{\mathbf{wait}\,(s,\sigma)}{\rightsquigarrow} (C',\sigma',\omega',\gamma') \quad \eta(s) = \sigma}{([t \mapsto C,\sigma,\omega,\gamma] \uplus \delta,\eta,\mathcal{L}) \Longmapsto ([t \mapsto C',\sigma',\omega',\gamma'] \uplus \delta,\eta[s \mapsto \triangledown],\mathcal{L})}$$

$$\frac{(C,\sigma,\omega,\gamma) \overset{\mathbf{alloc}(l)}{\rightsquigarrow} (C',\sigma',\omega',\gamma') \quad l \in \mathcal{L}}{([t \mapsto C,\sigma,\omega,\gamma] \uplus \delta,\eta,\mathcal{L}) \Longmapsto ([t \mapsto C',\sigma',\omega',\gamma'] \uplus \delta,\eta,\mathcal{L} \setminus \{l\})}$$

$$\frac{(C,\sigma,\omega,\gamma) \rightsquigarrow \mathbf{abort}}{([t \mapsto C,\sigma,\omega,\gamma] \uplus \delta,\eta,\mathcal{L}) \Longmapsto \mathbf{abort}}$$

Figure 6.2: Annotated global operational semantics.

by $\eta(s) = \triangledown$. Once finalised, a channel identifier cannot be reused (fortunately we have an infinite supply of fresh channel identifiers). The semantics enforces consistency between calls to **wait**, **grant** and **newchan**. A thread can only take a local transition acquiring or releasing a resource if it is consistent with the global channel map.

Thread-local steps are mirrored in the global semantics by updating the state of the thread in the thread map. A fork operation augments the thread map by associating the thread identifier with a new thread-local configuration whose components are specified on the local transition. Creating a new channel is permissible only if the channel identifier does not already exist. A **grant** action is allowed if the channel is unused; after the action the channel is associated with the heap resources provided by the **grant**. A **wait** action is allowed if the resources demanded by the **wait** are provided by the channel (via a **grant**); if so, the channel becomes finalised. Allocation removes the location from the set of available locations.

Definition 6.1. For a global state $\kappa = \langle \delta, \eta, \mathcal{L} \rangle$ and thread $t \in \mathrm{dom}(\delta)$ such that $\delta(t) = \langle C, \sigma, (\omega_{\mathrm{w}}, \omega_{\mathrm{g}}), \gamma \rangle$, we define $\mathrm{waits}(\kappa, t) \triangleq \omega_{\mathrm{w}}$ and $\mathrm{grants}(\kappa, t) \triangleq \omega_{\mathrm{g}}$.

Definition 6.2 (well-formedness)**.** A global state $\kappa = \langle \delta, \eta, \mathcal{L} \rangle$ is *well-formed* if:

1. The set of all heap locations either allocated and referred to in δ and η is disjoint from $\mathcal{L}$, the set of unallocated heap locations.

2. The thread-local states in δ and states stored in the channel map η can be joined using $\oplus$ to give a well-defined state. Note that, as $\oplus$ is associative and commuta-

tive, this state is unique, and also that any pair of states from δ and η can be joined to give a well-defined result.

3. For any channel $c \in \mathsf{Cid}$, there exists at most one thread t_1 such that $c \in \text{grants}(\kappa, t_1)$ and at most one thread t_2 such that $c \in \text{waits}(\kappa, t_2)$. If $c \in \text{grants}(\kappa, t_1)$, then $\eta(c) = \triangle$. If $c \in \text{waits}(\kappa, t_2)$, then $\eta(c) \in \mathsf{Heap} \uplus \{\triangle\}$.

Well-formedness expresses fundamental linearity assumptions on states and channels; essentially, it ensures elements of the state can be owned by only one thread at once.

Lemma 6.3. The global transition relation $\Longmapsto$ preserves well-formedness.

Assumption 1. We assume all global states are well-formed, unless explicitly stated otherwise.

Definition 6.4 (trace)**.** A trace is a (finite or infinite) sequence of global states $\mathcal{K} = \kappa_0\kappa_1 \ldots$ such that for every i, $\kappa_i \Longmapsto \kappa_{i+1}$. A finite (infinite) trace is called terminating (nonterminating).

Definition 6.5 (child thread, thread order)**.** Each local state $\langle C, \sigma, \omega, \gamma \rangle$ includes the set γ of forked child thread identifiers. For convenience, we define $\text{child}(\kappa, t)$ as the set of children for thread t in global state κ. The children of a thread are ordered with a relation $<$ that follows the order in which they were created. That is, suppose we have a trace $\mathcal{K}$ with initial state $\langle \delta, \eta, \mathcal{L} \rangle$ such that $c_1, c_2 \notin \text{dom}(\delta)$. Given an arbitrary state $\kappa_i \in \mathcal{K}$, if $\{c_1, c_2\} \subseteq \text{child}(\kappa, t)$ and $c_1 < c_2$, then c_1 was created earlier in the trace than c_2.

We denote by $\Longmapsto^*$ the reflexive transitive closure of the global transition relation. We sometimes write $\overset{t}{\Longmapsto}$ to denote a global transition resulting from the thread t taking a step, and call it a transition over thread t.

Definition 6.6 (sequentialised trace)**.** We define the *sequentialised* transition relation $\Longmapsto_{\mathrm{s}} \subseteq \Longmapsto$ as the transition relation in which a step $\kappa \overset{t}{\Longmapsto} \kappa'$ can only be taken if all threads in $\text{child}(\kappa, t)$ have reduced to **skip**. We say that a trace $\mathcal{K}$ is *sequentialised with respect to t* if every transition over thread t is in $\Longmapsto_{\mathrm{s}}$. We denote this transition relation by $\Longmapsto_{\mathrm{s}(t)}$.

6.2 Proof of behaviour preservation

We now prove the soundness of our analysis, i.e., that our parallelising transformation does not introduce any new behaviours to the original program. We prove our result using the semantics from §6.1 for both the sequential and the parallelised program. In this setting, the sequential program is represented with a sequential thread[1] possibly executing concurrently with other sequential threads—this degenerates into the purely sequential case when there are no other runnable threads.

[1]A thread is sequential if it does not call `fork`.

6.2.1 Informal description of the proof

The proof of the behaviour preservation (Theorem 6.18) is based on a simulation argument: each trace of the parallelised program must correspond to a trace of the sequential program and preserve as an invariant a given binary relation between states of the parallelised and the sequential program. The proof is structured as follows:

1. We establish a simulation invariant between the parallelised program and the sequential program in two steps: first for a program decorated with calls to **grant**, **wait** and **newchan** and a program with these calls erased (Lemma 6.7), and then for a program additionally decorated with **fork** calls and a **fork**-erased program (Lemma 6.8). The combined invariant relates every non-faulting sequentialised trace of the fully decorated program with a trace of the sequential program. By *sequentialised*, we mean that forked children must execute to completion before their parent threads can be scheduled (Def. 6.6).

2. We show that, under the assumption that forked child threads never wait for channels granted by their parent or later-forked child threads, any terminating non-faulting trace of the parallelised program can be reordered into a sequentialised trace with the same thread-local behaviour. This is established by showing that traces of the program parallelised by our analysis have signalling barriers aligned with the thread ordering (Lemma 6.14), and that any such trace can be reordered into a sequentialised trace (Lemma 6.13).

3. Previous steps establish behaviour-preservation for non-faulting traces. Inserting **grant**, **wait**, **newchan** and **fork** may introduce aborts. However, our parallelising transformation is proof-preserving so the parallelised programs is also verified using separation logic. This establishes, for every state satisfying the program precondition, that the program cannot abort.

6.2.2 Proof details

We start with a lemma that establishes a simulation invariant between a program decorated with calls to **grant**, **wait** and **newchan** and a program with these calls replaced with **skip**.

Lemma 6.7 (wait, grant, newchan insertion). Let $[\![-]\!]^{\downarrow}_{\mathrm{ch}}$ be a program transformation which replaces calls to **newchan**, **grant** and **wait** with **skip**. Let $\kappa = \langle \delta, \eta, \mathcal{L} \rangle$ and κ' be global states. We say that the invariant $I(t, \kappa, \kappa')$ is satisfied if:

- The thread identifier t is in $\mathrm{dom}(\delta)$.
- For $\delta(t) = \langle C, \sigma, \omega, \gamma \rangle$, the program C executed by t does not contain **fork**.
- The thread waits for all channels it grants on, i.e., $\omega_{\mathrm{g}} \subseteq \omega_{\mathrm{w}}$, and all channels in $\omega_{\mathrm{w}} \setminus \omega_{\mathrm{g}}$ have already been granted on but are not yet finalised, i.e., $\forall_{c \in \omega_{\mathrm{w}} \setminus \omega_{\mathrm{g}}}.\, \eta(c) \notin \{\triangledown, \vartriangle\}$.
- Let $\sigma_{\mathrm{ch}} \triangleq \circledast_{c \in \omega_{\mathrm{w}} \setminus \omega_{\mathrm{g}}}.\, \eta(c)$ be all resources the thread does not grant but waits for. Then there exists $\eta' \subseteq \eta$ such that $\kappa' = \langle \delta[t \mapsto \langle [\![C]\!]^{\downarrow}_{\mathrm{ch}}, \sigma \oplus \sigma_{\mathrm{ch}}, (\emptyset, \emptyset), \gamma \rangle], \eta', \mathcal{L} \rangle$ and for all channels c held by threads other than t, $\eta'(c) = \eta(c)$.

For any states κ, κ', κ_2, if $I(t, \kappa, \kappa')$ and $\kappa \Longmapsto \kappa_2$, then there exists a state κ_2' such that $\kappa' \Longmapsto^* \kappa_2'$ and $I(t, \kappa_2, \kappa_2')$. This property can be alternatively represented by the following diagram:

$$\begin{tikzcd}
\kappa \arrow[r, "I(t)"] \arrow[d, Rightarrow] & \kappa' \arrow[d, Rightarrow, "*"] \\
\kappa_2 \arrow[r, "I(t)"] & \kappa_2'
\end{tikzcd}$$

Intuitively, the invariant I expresses the simulation relation between programs containing **wait**, **grant** and **newchan** (left-hand side of the diagram), and the equivalent program without them (right-hand side).

Proof. There are two cases: either $\kappa \Longmapsto \kappa_2$ is a transition over t, or it is a transition over some other thread $t' \neq t$. If $\kappa \Longmapsto \kappa_2$ is a transition over t, we proceed by structural induction over C.

- C is a primitive command in Prim. In this case, the result holds by the assumption of behavioural monotonicity for primitive commands.
- C is a **grant** or **wait** for some channel c. In both of these cases, the corresponding command $[\![C]\!]^{\downarrow}_{\mathrm{ch}}$ will be **skip**. Consequently the transition $\kappa' \Longmapsto \kappa_2'$ will be a τ-transition, meaning $\kappa' = \kappa_2'$.

  By assumption, the join of the thread-local state σ and state σ_{ch} stored in accessible channels in κ is equal to the thread-local state in κ'. Calling a **grant** takes some local state and pushes it into a channel, which preserves this property. Similarly calling a **wait** pulls some state out of an accessible channel and into local state, which also preserves the property.
- C is a **newchan**. In this case, the corresponding command also be a **skip**, so $\kappa' = \kappa_2'$. The effect of **newchan** will be to initialise a fresh channel not already in η, and add these channels to the channel state for κ_2. These channels are erased by the invariant, so the property is preserved.
- C is **alloc**. The set $\mathcal{L}$ of unallocated locations is the same in κ and κ'. Consequently, if a location can be allocated in κ, it can also be allocated in κ'.
- C is a sequential composition, loop, or conditional. The result follows trivially by appeal to the induction hypothesis.

Suppose now that the transition is over some other thread $t' \neq t$. By assumption (the third property of I), we know that the channels erased by the invariant are not shared with any other thread. We can therefore establish straightforwardly that such a transition must be replicated exactly between κ' and κ_2'. The invariant is not disturbed, because it does not mutate the content of other threads. $\square$

The next lemma establishes a simulation invariant between a program containing calls to **fork** and a program with these calls erased.

Lemma 6.8 (**fork** insertion). Let $[\![-]\!]^{\downarrow}_{\mathrm{fk}}$ be a program transformation which replaces calls to **fork** C with C. We say that the invariant $J(t, \kappa, \kappa')$ is satisfied if:

- The thread identifier t is in $\mathrm{dom}(\delta)$.
- For $\delta(t) = \langle C, \sigma, \omega, \gamma \rangle$, any calls to **fork** in C do not themselves include calls to **fork**.
- Let $\sigma_{\mathrm{fk}} = \circledast \{\sigma_c \mid c \in \mathrm{child}(t) \wedge \delta(c) = \langle C_c, \sigma_c, \omega_c, \gamma_c \rangle\}$[2] be thread-local states and $\omega_{\mathrm{fk}} = \biguplus\{\omega_c \mid c \in \mathrm{child}(t) \wedge \delta(c) = \langle C_c, \sigma_c, \omega_c, \gamma_c \rangle\}$ channel states of forked child threads conjoined. By the well-formedness condition on global states (Assumption 6.2), both of these must be well-defined.
- There exists at most one child thread $c \in \mathrm{child}(t)$ such that $\delta(c) = \langle C_c, \sigma_c, \omega_c, \gamma_c \rangle$ and $C_c \neq \mathbf{skip}$.
- Either of the following two cases holds:
  - If $\forall c \in \mathrm{child}(t).\, \delta(c) = \langle \mathbf{skip}, \sigma_c, \omega_c, \emptyset \rangle$, then $\kappa' = \langle \delta[t \mapsto \langle [\![C]\!]^{\downarrow}_{\mathrm{fk}}, \sigma \oplus \sigma_{\mathrm{fk}}, \omega \uplus \omega_{\mathrm{fk}}, \emptyset \rangle], \eta, \mathcal{L} \rangle$.[3]
    (In other words, all child threads of t have terminated, and the main thread corresponds to the original sequential state.)
  - If $\exists! c \in \mathrm{child}(t).\, \delta(c) = \langle C_c, \sigma_c, \omega_c, \gamma_c \rangle \wedge C_c \neq \mathbf{skip}$, then $C = \mathbf{skip}; C_1$ and $\kappa' = \langle \delta[t \mapsto \langle C_c; [\![C_1]\!]^{\downarrow}_{\mathrm{fk}}, \sigma \oplus \sigma_{\mathrm{fk}}, \omega \uplus \omega_{\mathrm{fk}}, \emptyset \rangle], \eta, \mathcal{L} \rangle$.
    (In other words, there exists exactly one unterminated child thread, and join of the states of the child threads and main thread correspond to the original sequential state.)

For any states κ, κ', κ_2, if $J(t, \kappa, \kappa')$, and $\kappa \Longmapsto_{\mathsf{s}(t)} \kappa_2$, then there exists a state κ_2' such that $\kappa' \Longmapsto^{*} \kappa_2'$ and $J(t, \kappa_2, \kappa_2')$. This property can be alternatively represented by the following diagram:

$$\begin{tikzcd}
\kappa \arrow[r, "J(t)"] \arrow[d, Rightarrow, "\mathsf{s}(t)"'] & \kappa' \arrow[d, Rightarrow, "*"] \\
\kappa_2 \arrow[r, "J(t)"] & \kappa_2'
\end{tikzcd}$$

The invariant J expresses the simulation relation between programs containing **fork** (left-hand side of the diagram) and the equivalent program without it (right-hand side).

Proof. Unlike with Lemma 6.7, there are *three* cases: either $\kappa \Longmapsto_{\mathsf{s}(t)} \kappa_2$ is a transition over t; or it is a transition over some member of the set $\mathrm{child}(t)$; or the transition is over some other, unrelated thread. First suppose that the transition is over t. We proceed by structural induction on C:

- $C \in \mathsf{Prim}$. By assumption, any behaviour in a small state σ will be reflected in a big one, which suffices to ensure that the invariant holds.

[2] By $\circledast \{\sigma_s \mid s \in S\}$, we mean $\circledast_{s \in S}.\, \sigma_s$.

[3] Note that $\sigma \oplus \sigma_{\mathrm{fk}}$ is well-defined because thread-local states of child threads are disjoint from the thread-local state of their parent (Fig. 6.1).

- C is **alloc**. The set $\mathcal{L}$ of unallocated locations is the same for κ and κ'. By the same argument as Lemma 6.7, the same allocation step is therefore possible in both global states.

- C is **wait** or **grant**. By the well-formedness assumption, resources held by channels must be disjoint from those held by other threads. This suffices to ensure that **wait** and **grant** transitions can take place identically in the corresponding **fork**-free program.

- C is **fork**. This will correspond to a τ-transition in the **fork**-erased program. Because the transition over the program with **fork** is sequentialised with respect to t, the transition can only be over t if all the children of t have reduced to **skip**. Calling **fork** pushes some state to the child thread, and results in exactly one active (non-**skip**) child thread.

- C is a loop, conditional, sequential composition. Property holds by appeal to the induction hypothesis.

Now suppose the transition is over some thread $c \in \mathbf{child}(t)$. By assumption (the fourth property of J), there can be at most one such thread. The state held by the child thread must be a substate of that held in the **fork**-free program. Consequently, an almost identical structural induction argument suffices to show that the simulation property holds.

Finally, suppose that the transition is over some other thread t' that is neither t nor a child of t. Such threads cannot be affected by the behaviour of t and its children. It is straightforward to show that any transition for the program can take place in the corresponding **fork**-free program. $\square$

The previous two lemmas combined establish an invariant relating non-faulting sequentialised traces of the parallelised program with traces of the sequential program. It remains to show that terminating non-faulting traces of the parallelised program can be reordered into sequentialised traces with the same thread-local behaviour. We do this by introducing a well-founded order on traces in which sequentialised traces are minimal elements, and showing that each signal-ordered trace (i.e., having its signalling barriers aligned with the thread ordering) can be reordered to a behaviourally equivalent trace smaller with respect to the well-founded order, eventually leading to a behaviourally equivalent sequentialised trace.

Definition 6.9. We say that traces $\mathcal{K}_1$ and $\mathcal{K}_2$ are *behaviourally equivalent* if, for each thread identifier t, the sequence of transitions over t in $\mathcal{K}_1$ and $\mathcal{K}_2$ are identical.

We first show that consecutive transitions of two different threads can be reordered if the second thread does not wait for any channel that the first thread grants on. We use this property in the subsequent lemma to inductively construct a behaviourally equivalent sequentialised trace.

Lemma 6.10 (trace reordering)**.** Suppose we have a two-step trace:

$$\mathcal{K} \;=\; \langle \delta_1, \eta_1, \mathcal{L}_1 \rangle \overset{t_1}{\Longmapsto} \langle \delta_2, \eta_2, \mathcal{L}_2 \rangle \overset{t_2}{\Longmapsto} \langle \delta_3, \eta_3, \mathcal{L}_3 \rangle$$

such that $t_1 \neq t_2$, and that $t_1, t_2 \in \mathrm{dom}(\delta_1)$. Let $\delta_1(t_1) = \langle C, \sigma, \omega, \gamma \rangle$ and $\delta_1(t_2) = \langle C', \sigma', \omega', \gamma' \rangle$. Suppose that t_2 does not wait for any channel that t_1 grants on, i.e., that $\omega_{\mathrm{g}} \cap \omega'_{\mathrm{w}} = \emptyset$ holds. Then there exists a thread environment η', an unallocated set $\mathcal{L}'$, and a two-step trace:

$$\mathcal{K}' \;=\; \langle \delta_1, \eta_1, \mathcal{L}_1 \rangle \overset{t_2}{\Longmapsto} \langle \delta_1[t_2 \mapsto \delta_3(t_2)], \eta', \mathcal{L}' \rangle \overset{t_1}{\Longmapsto} \langle \delta_3, \eta_3, \mathcal{L}_3 \rangle$$

The new trace $\mathcal{K}'$ is behaviourally equivalent (Def. 6.9) to $\mathcal{K}$.

Proof. We proceed by case-splitting on the shape of transitions over t_1 and t_2:

- One or both of the transition are thread-local (i.e. do not result in a labelled transition in the thread-local semantics). It is straightforward to see that neither transition can affect the other one, and the rearrangement result follows trivially.
- One or both of the transitions is a **fork**. The effect of such a transition is localised to the parent and newly-created child thread. By assumption, neither thread was created by either of these transitions. The rearrangement result follows straightforwardly.
- One or both transitions is a **newchan**. Channels can only be transferred by **fork**, and by assumption, neither transition created either of the threads. Consequently the effect of a **newchan** call is confined to the local thread.
- Both transitions are **wait** (resp. **grant**). The threads could only affect one another if both waited (resp. granted) on the same channel, but this is ruled out by the well-formedness assumption that each end of each channel is held by at most one thread.
- One transition is a **wait** and the other is a **grant**. Once again, the threads could only affect one another if they waited and granted on the same channel. However, by assumption of the lemma, the thread t_1 cannot grant on any channel on which t_2 can wait. By the structure of the semantics, the transition t_1 cannot wait for a channel that is subsequently granted. So this case cannot occur.
- One or both transitions are a **alloc**. If one operation is a **alloc** and the other another command, the threads trivially cannot affect one another. If both transitions are **alloc**, the locations allocated must be distinct, and the transitions again can be exchanged straightforwardly.

The constructed trace $\mathcal{K}'$ is trivially behaviourally equivalent to $\mathcal{K}$, since the thread-local actions for each thread are identical. □

Definition 6.11 (signal order)**.** We describe a trace $\mathcal{K}$ as *signal-ordered with respect to t* if t is a thread identifier such that for all states $\kappa = \langle \delta, \eta, \mathcal{L} \rangle \in \mathcal{K}$:

$$\begin{array}{l} \forall c \in \mathrm{child}(\kappa, t).\, \forall k \in \mathrm{waits}(\kappa, c). \\ ((\exists c' \in \mathrm{child}(\kappa, t).\, c' < c \wedge k \in \mathrm{grants}(\kappa, c')) \;\vee\; (\eta(k) \notin \{\bot, \triangledown, \vartriangle\})) \end{array}$$

(In other words, if a child of t can wait on a channel k, some earlier child must be able to grant on it, or the channel must already be filled.)

Definition 6.12 (degree of sequentialisation)**.** Let $\mathcal{K}$ be a trace. If $\mathcal{K}$ is not sequentialised with respect to t, let $c \in \text{child}(t)$ be the earliest thread in the child order with an unsequentialised transition (that is, for all other children c' with unsequentialised transitions, $c' > c$). Now, let F be the number of transitions from the originating **fork** transition for c and the end of the trace. Let K be the number of transitions over c *after* the first unsequentialised transition. Let D be the number of intervening transitions between the originating **fork** and the first unsequentialised transition.

We call such a tuple (F, K, D) the *degree of sequentialisation* with respect to t. We order such tuples lexicographically, with left-to-right priority, and lift this order to give a partial order on traces $<_t$. This order on traces is well-founded, with sequentialised traces occupying the minimum position in the order.

Using the ordering on traces introduced in Definition 6.12 we now show that a trace that is signal-ordered with respect to a thread t can be reordered to a behaviourally equivalent trace that is sequentialised with respect to t.

Lemma 6.13 (sequentialisation)**.** Let $\mathcal{K}$ be a trace such that

- $\mathcal{K}$ is signal-ordered with respect to t; and
- Let $\kappa = \langle \delta, \eta, \mathcal{L} \rangle$ be the final state in $\mathcal{K}$ and let t' be the maximum thread in $\text{child}(\kappa, t)$ with transitions in $\mathcal{K}$. For all threads $t'' \in \text{child}(\kappa, t)$ such that $t'' < t'$, $\delta(t'') = \langle \mathbf{skip}, \sigma, \omega, \gamma \rangle$ for some σ, ω, γ. *(Note this holds automatically if $\mathcal{K}$ is terminating.)*

Then there must exist a trace $\mathcal{K}'$ sequentialised with respect to t such that $\mathcal{K}$ is behaviourally equivalent to $\mathcal{K}'$.

Proof. We show that any out-of-order trace $\mathcal{K}$ can be reordered by applying Lemma 6.10, to give a behaviourally equivalent trace with a lower degree of sequentialisation w.r.t. t.

Consider a trace $\mathcal{K}$ that is signal-ordered but not sequentialised. Let t_2 be the earliest thread in the child order for t that has an unsequentialised transition. By the assumption that the transition is unsequentialised, the preceding transition in the trace must be on some other thread $t_1 \neq t_2$, and must not be the **fork** that created the out-of-order transition. Note that t_1 also must not be some child of t earlier in the child order—otherwise the transition over t_1 would be the earliest out-of-order transition.

By the definition of signal-ordering we know that $\text{wait}(\kappa, t_2) \cap \text{grant}(\kappa, t_1) = \emptyset$. Consequently, we can apply Lemma 6.10, and push the transition over t_2 earlier than the transition over t_1. Call the resulting trace $\mathcal{K}''$.

By Lemma 6.10, $\mathcal{K}''$ is behaviourally equivalent to $\mathcal{K}$. As a trivial consequence, $\mathcal{K}''$ is also signal-ordered. As a deeper consequence, $\mathcal{K}'' <_t \mathcal{K}$ (with respect to the order given in Def. 6.12). There are three cases:

- The transition shifts left, but is still unsequentialised. The number of intervening transitions from the originating **fork** decreases, while the other two measures are unchanged.
- The left shift means the transition is sequentialised, but more transitions over the same thread are unsequentialised. The number of transitions from the first unsequentialised transition decreases, while the distance from the initial **fork** to the end of the thread is unchanged.

- The left shift means all transitions for the thread are sequentialised. The next target thread must be later in the child order, meaning it is later in the trace. The number of transitions from the initial **fork** to the end of the thread decreases.

As the order is well-founded, by repeatedly applying the lemma, we get a behaviourally-equivalent trace that is sequentialised w.r.t. t. $\square$

We now establish that the pattern of barriers that we insert into the program during our parallelisation algorithm results in a trace that is signal-ordered. In the interest of clarity, we assume that the parallelised program $\mathbb{P}_{\text{par}}$ constructed by the analysis has the following simplified form:

```
ci := newchan();
grant(ci);
while(B) {
  cj := newchan();
  fork[P](C, ci, cj);
  ci := cj;
}
wait(ci);
```

Here we assume that we only need a single active pair of channels `ci/cj`—the argument generalises straightforwardly to the case with n channels. The assertion P is defined so that: $P \vdash \mathsf{fut}(\mathtt{ci}, P_1) * \mathsf{req}(\mathtt{cj}, P_2) * F$, for some P_1, P_2 and F; and $F \nvdash \mathsf{fut}(x, P')$ and $F \nvdash \mathsf{req}(x, P')$ for any x and P'. (In other words, the forked thread can only wait for `ci` and grant on `cj`.)

Lemma 6.14 (construction of signal-ordered traces)**.** Let $\kappa = \langle \delta, \eta, \mathcal{L} \rangle$ be a state and t a thread identifier such that $\delta(t) = \langle \mathbb{P}_{\text{par}}, \sigma, (\emptyset, \emptyset), \emptyset \rangle$ and $\mathbb{P}_{\text{par}}$ matches the standard form for parallelised programs, described above. Let $\mathcal{K}$ be a trace with initial state κ. Then $\mathcal{K}$ is signal-ordered with respect to t.

Proof. We establish this property by defining an invariant $L(\kappa)$:

- Let $\kappa = \langle \delta, \eta, \mathcal{L} \rangle$ and $\delta(t) = \langle C, \sigma, \omega, \gamma \rangle$.
- There exists a channel c_1 such that $\sigma_s(\mathtt{ci}) = c_1$, and $\text{waits}(\kappa, t) = \{c_1\}$.
- Either $\eta(c_1) \notin \{\bot, \triangledown, \vartriangle\}$, or $\exists t' \in \text{child}(\kappa, t).\, c_1 \in \text{grants}(\kappa, t')$.

We now show that signal-ordering holds for every step of the semantics, and that the invariant holds for any state immediately preceding the execution of an iteration of the main `while`-loop; that is, at any point when $C = \mathtt{while(B)}\{\ldots\}\ \mathtt{wait(ci)};$.

- Suppose we execute from the beginning of the program to the beginning of the first iteration **while**. It is easy to see by inspection of the semantics that the resulting state κ will satisfy the invariant L. We create the channel c_1, and the call to **grant** ensures $\eta(c_1)$ is associated with some state, as required by the invariant.

- Now suppose we execute a single iteration of the loop: that is, we assume the boolean condition `B` holds, and apply the appropriate rule in the operational semantics. The invariant cannot be disturbed by other running threads, because by definition they can only call **grant** on the channel c_1.

  We call **newchan**, which creates a channel c_2, associated with the variable `cj`. Now we observe that, by the structure of P call to **fork** creates a thread t' with $\text{waits}(\kappa, t') = \{c_1\}$ and $\text{grants}(\kappa, t') = \{c_2\}$. As a result, in the resulting state $\text{waits}(\kappa, t) = \{c_2\}$. The final assignment associates c_2 to `ci`, which reestablishes the invariant.

Each step in this execution satisfies signal-ordering. By induction this completes the proof. □

In Lemma 6.14 we assumed a specific program form constructed by the parallelising transformation from §4.2. However, the soundness proof could be adapted to hold for other parallelisation backends. As long as the analysis generates a parallelised program with traces that are signal-ordered we could establish an analogue of Lemma 6.14 and proceed with other steps of the soundness proof in the same way.

Lemma 6.15 (synchronisation erasure)**.** Let $\mathbb{P}$ be a sequential program, and $\mathbb{P}_{\text{par}}$ be a program resulting from applying our parallelisation algorithm. Then $\mathbb{P} = [\![[\![\mathbb{P}_{\text{par}}]\!]^{\downarrow}_{\text{fk}}]\!]^{\downarrow}_{\text{ch}}$, where the (syntactic) equality of programs is defined up to insertion/erasure of **skip** statements—that is syntactic manipulation using the identity $C = C; \mathbf{skip}$.

Proof. The result follows immediately as our analysis only inserts channels and calls to fork. □

Lemma 6.16 (materialised variables)**.** Materialising local variables, as described in §2.4.2, has no effect on the operational behaviour of a program.

Proof. Let $\mathbb{P}$ be a program, and let $\mathbb{P}'$ be a corresponding program in which the local variables are materialised. Let $\mathcal{K}$ be a trace in which the program $\mathbb{P}$ is executed. When inserted, the materialised variables are written but not read (they are *subsequently* used in conditionals after loop-splitting). Therefore, there must be a trace $\mathcal{K}'$ in which $\mathbb{P}'$ is substituted for $\mathbb{P}$, which is identical to $\mathcal{K}$ aside from the materialised variables. □

Lemma 6.17 (loop-splitting)**.** Loop-splitting, as described in §4.2.4, has no effect on the operational behaviour of the program.

Proof. By the fact that either a conditional or its negation must hold, and a straightforward appeal to the semantics of loops. □

Finally, we prove the main theorem stating the soundness of our parallelisation algorithm.

Theorem 6.18 (parallelisation soundness)**.** Let $\mathbb{P}$ be a sequential program, and let $\mathbb{P}_{\text{par}}$ be the program resulting from applying our parallelisation algorithm to $\mathbb{P}$ (including variable materialisation and loop-splitting). Let $\mathcal{K}$ be a non-faulting terminating trace with initial state $\kappa = \langle \delta, \eta, \mathcal{L} \rangle$, and let t be a thread identifier such that $\delta(t) = \langle C_{\text{par}}, \sigma, (\emptyset, \emptyset), \emptyset \rangle$. Then there exists a trace $\mathcal{K}''$ with initial state $\kappa'' = \langle \delta[t \mapsto \langle C, \sigma, (\emptyset, \emptyset), \emptyset \rangle], \eta, \mathcal{L} \rangle$ such that:

1. For all thread identifiers t' defined in $\mathcal{K}''$ such that $t \neq t'$, their thread-local behaviour in $\mathcal{K}$ is identical to $\mathcal{K}''$.

2. Let κ_Ω be the final state in $\mathcal{K}$, and κ''_Ω the final state of $\mathcal{K}''$. Let σ be the local state associated with thread t in κ_Ω and σ'' the corresponding state associated with t in κ''. There exists a state σ' such that $\sigma \oplus \sigma' = \sigma''$.

Proof. By Lemma 6.14 the trace $\mathcal{K}$ must be signal-ordered with respect to t. Therefore by Lemma 6.13 there exists a behaviourally-equivalent trace $\mathcal{K}_{\text{seq}}$ that is sequentialised with respect to t. Because t has no child threads in κ, we can choose $\mathcal{K}_{\text{seq}}$ such that κ is also its initial state.

We now show that there exist traces $\mathcal{K}'$ and $\mathcal{K}''$, such that transitions in $\mathcal{K}$ are related to $\mathcal{K}'$ by the invariant $J(t)$ (Lemma 6.8) and transitions in $\mathcal{K}'$ are related to $\mathcal{K}''$ by the invariant $I(t)$ (Lemma 6.7).

We proceed by induction on the length of a prefix of $\mathcal{K}_{\text{seq}}$. By Lemma 6.15, $\mathbb{P} = [\![[\![\mathbb{P}_{\text{par}}]\!]^{\downarrow}_{\text{fk}}]\!]^{\downarrow}_{\text{ch}}$. From this, it is straightforward to see that there exists a state $\kappa' = \langle \delta[t \mapsto \langle [\![\mathbb{P}_{\text{par}}]\!]^{\downarrow}_{\text{fk}}, \sigma, (\emptyset, \emptyset), \emptyset \rangle], \eta, \mathcal{L} \rangle$ such that $J(t, \kappa, \kappa')$ and $I(t, \kappa', \kappa'')$.

Assume traces $\mathcal{K}'$ and $\mathcal{K}''$ exist for the first n transitions of $\mathcal{K}_{\text{seq}}$. Let κ_n be the final state in this prefix. By assumption there exist states κ'_n and κ''_n such that $J(t, \kappa_n, \kappa'_n)$ and $I(t, \kappa'_n, \kappa''_n)$. Now show that for the next transition $\kappa_n \Longmapsto_{\mathsf{s}(t)} \kappa_{n+1}$ there exist transitions $\kappa'_n \Longmapsto^* \kappa'_{n+1}$ and $\kappa''_n \Longmapsto^* \kappa''_{n+1}$ such that $J(t, \kappa_{n+1}, \kappa'_{n+1})$ and $I(t, \kappa'_{n+1}, \kappa''_{n+1})$. This can be visualised by the following diagram:

$$\begin{tikzcd}
\kappa_n \arrow[r, "J(t)"] \arrow[d, Rightarrow, "\mathsf{s}(t)"'] & \kappa'_n \arrow[r, "I(t)"] \arrow[d, Rightarrow, "*"] & \kappa''_n \arrow[d, Rightarrow, "*"] \\
\kappa_{n+1} \arrow[r, "J(t)"] & \kappa'_{n+1} \arrow[r, "I(t)"] & \kappa''_{n+1}
\end{tikzcd}$$

This is an immediate consequence of Lemmas 6.7 and 6.8.

By induction, this suffices to establish the existence of the trace $\mathcal{K}''$. Corresponding thread-local behaviour for threads $t' \neq t$, and corresponding final state for thread t follow immediately from the definitions of $I(t)$ and $J(t)$.

This result also holds if our analysis performs variable materialisation and loop-splitting—to show this, we need only appeal to Lemma 6.16 and Lemma 6.17. □

Corollary 6.19. If the chosen post-condition in the proof of the program is precise, then the result of parallelisation is an equality, not a sub-state.

Proof. Consequence of the fact that at most one sub-state of a given state can satisfy a precise assertion. □

6.3 Termination

In this section we prove that our parallelisation algorithm cannot introduce non-termination to the parallelised program.

Lemma 6.20 (ensuring grant)**.** Let C_{par} be a program resulting from applying our parallelisation algorithm to some sequential program C. Let $\mathcal{K}$ be a trace with initial state $\kappa = \langle \delta, \eta, \mathcal{L} \rangle$, such that $\delta(t) = \langle C_{\text{par}}, \sigma, (\emptyset, \emptyset), \emptyset \rangle$. For all states in $\kappa' = \langle \delta, \eta, \mathcal{L} \rangle \in \mathcal{K}$ and all children $c \in \text{child}(\kappa', t)$ if c has terminated in κ' (i.e. reduced to **skip**) then $\text{grants}(\kappa', c) = \emptyset$.

Proof. Consequence of the fact that in our analysis, each **grant** is injected along every control-flow path. Consequently, a terminating thread must call **grant** for every channel to which it has access. □

Lemma 6.21 (trace extension)**.** Let $\mathcal{K}$ be a trace that is signal-ordered with respect to t, and let $\kappa = \langle \delta, \eta, \mathcal{L} \rangle$ be the final state of $\mathcal{K}$. Let t' be the minimum child of t such that $\delta(t') = \langle C, \sigma, \omega, \gamma, \rangle$ and $C \neq$ **skip**. Then there exists a non-faulting transition $\kappa \overset{t'}{\longmapsto} \kappa'$.

Proof. By examination of the thread-local semantics, we can see that only calls to **wait** can block without faulting. For all other commands, either the thread can take a step, or it faults immediately. Consequently, the lemma reduces to asking whether every call to **wait** in the thread t' can take a step.

A call to **wait** can only block if some prior thread has not called **grant** on some channel. By the definition of signal-ordered (Def. 6.11), the channels for which t' can wait must either contain state, or must be held by some child thread earlier in the order. By assumption, we know that all earlier threads have terminated, so by Lemma 6.20, we know that no thread holds a pending grant. Consequently, the channel must contain state, and **wait** can take a step. □

Lemma 6.22 (trace erasure)**.** Let $\mathcal{K}$ be a trace that is signal-ordered with respect to t, and let $\kappa = \langle \delta, \eta, \mathcal{L} \rangle$ be the final state of $\mathcal{K}$. Let t' be the maximum child of t in $\text{dom}(\delta)$. Then there exists a trace $\mathcal{K}'$ such that $\mathcal{K}'$ has no transitions over t', and all threads other than t' have identical behaviour in $\mathcal{K}$ and $\mathcal{K}'$.

Proof. By induction over the number of transitions over t in the trace. identify the final transition over t' in $\mathcal{K}$, $\kappa_n \overset{t'}{\longmapsto} \kappa_{n+1}$. We erase this transition, and then re-execute the remainder of the transitions from the trace. By the definition of signal-order (Def. 6.11) no other child thread can call **wait** on any channel held by t. Consequently, we must be able to run the same sequence of thread-local actions after erasing the transition. By applying this process, we can erase all transitions over t'. □

Theorem 6.23 (termination)**.** Let $\mathbb{P}$ be a sequential program, and let $\mathbb{P}_{\text{par}}$ be a program resulting from applying our parallelisation algorithm to $\mathbb{P}$. If $\mathbb{P}$ is guaranteed to terminate, then so is $\mathbb{P}_{\text{par}}$.

Proof. If $\mathbb{P}$ is terminating, then for any initial state, there must be a maximum number of steps that the thread t running $\mathbb{P}$ can take. To prove the theorem, we prove the contrapositive result: given that $\mathbb{P}_{\text{par}}$ does not terminate, we can construct a trace for $\mathbb{P}$ where t takes more than this number of steps.

Suppose we have an initial state $\kappa = \langle \delta, \eta, \mathcal{L} \rangle$ such that $\delta(t) = \langle \mathbb{P}_{\text{par}}, \sigma, (\emptyset, \emptyset), \emptyset \rangle$. Suppose that κ can result in a nonterminating trace over t—that is, given any arbitrary bound n, there exists a trace with initial state κ and more than n steps over t and its child threads.

Construct a sequential state $\kappa_{\mathrm{seq}} = \langle \delta[t \mapsto \langle \mathbb{P}_{\mathrm{par}}, \sigma, (\emptyset, \emptyset), \emptyset\rangle], \eta, \mathcal{L}\rangle$, and identify a bound q such that no trace starting with κ'' can take more than q steps over t. Choose a trace $\mathcal{K}$ with initial state κ such that t and its children take more than q steps, *excluding* synchronisation (that is, calls to **fork**, **newchan**, **signal** and **wait**). Note that by Lemma 6.14, $\mathcal{K}$ is signal-ordered.

Now we rearrange $\mathcal{K}$ to give a sequentialised trace of length greater than q.

- By applying Lemma 6.21, we extend the trace until the first n children of t together take more than q non-synchronisation steps, and all children but the nth have terminated.

- By applying Lemma 6.22, we erase the $(n+1)$th to maximum child threads of t, along with associated calls to fork from t.

The resulting trace $\mathcal{K}$ has more than q non-synchronisation transitions from t and its first n children, and in the final state κ', all but the nth child thread have terminated. By applying Lemma 6.13, we can construct a trace $\mathcal{K}''$ that has equivalent thread-local behaviour, and that is sequentialised with respect to t.

We observe that, by Lemma 6.15, $\mathbb{P} = [\![[\![\mathbb{P}_{\mathrm{par}}]\!]^{\downarrow}_{\mathrm{fk}}]\!]^{\downarrow}_{\mathrm{ch}}$. By applying Lemmas 6.7 and 6.8, construct a trace $\mathcal{K}_{\mathrm{seq}}$ with initial state κ_{seq}. The invariants in Lemmas 6.7 and 6.8 only erase synchronisation transitions. Consequently, the resulting sequential trace has more than q transitions over the thread t. This contradicts our assumption, and completes the proof. □

CHAPTER 7

RELATED WORK

Traditional approaches to automated parallelisation are generally based on checking the independence of program statements. Here, a compiler performs dependency analysis, generating constraints about program statements, and schedules statements that are independent of each other for parallel execution. Dependency constraints are typically derived from a program analysis that discovers how the result of one computation affects the result of another.

These techniques have been successful for programs with simple data types, statically allocated arrays, and structured control-flow operators, such as loops, but have been less effective when programs manipulate pointers and make extensive use of dynamic data structures [HPR89, HN90, GHZ98, GPPS99]. In this dissertation, we leverage separation logic because it provides a convenient means to express assertions about dynamic or recursively defined resources. For example, [RCG09] investigates how a separation-logic-based analysis can be adapted to express memory-separation properties, thereby detecting independence between statements via interference checking of proof assertions. In our approach, we do not check for such independence directly in the proof, but instead use the proof to discover what resources different parts of the program depend on, and inject appropriate synchronisation to ensure that the parallelised version of the program respects these dependencies.

Thus, there are three main differences between our proposed parallelisation method and prior work:

- In contrast to other approaches where the proof produced by a compiler is of special form, strictly targeted at checking independence, we do not make any assumptions about the program proof except that it be written in separation logic. By virtue of separation logic's semantics, any such proof describes all resources accessed by the program and keeps track of their independence via separating conjunction—a property that we crucially rely on to track fine-grained sequential dependencies.

- Our approach starts with an unconstrained parallelised version of the program, and then inserts just enough synchronisation to preserve sequential dependencies and guarantee preservation of observable behaviour. By releasing the resources that a thread does not need and acquiring the resources that it does, we implicitly establish (an under-approximated) independence of certain parts of the program as a byproduct of our analysis.

- While the main usage scenario that we have presented is do-across parallelisation of for-loops with *heap-carried* dependencies, our key concepts are in fact not dependent on a specific loop structure. We foresee our approach being equally applicable to less regular code patterns and other iteration structures.

Next we contrast our parallelisation method in more details with respect to approaches across related lines of work.

Resource-usage inference by abduction. We have defined an inter-procedural, context-sensitive and control-flow-sensitive analysis capable of determining the resource that will (and will not) be accessed between particular points in the program. At its core, our analysis uses abductive reasoning [CDOY11] to discover *redundancies*—that is, state used earlier in the program that will not be accessed subsequent to the current program point. Using abduction in this way was first proposed in [DF10], where it is used to discover memory leaks, albeit without conditionals, procedures, loops, or code specialisation.

In [CDV09], abduction is used to infer resource invariants for synchronisation, using a process of counterexample-driven refinement. Our approach similarly infers resource invariants, but using a very different technique: invariants are derived from a sequential proof (which also contains invariants for loops, etc), and we also infer synchronisation points and specialise the program to reveal synchronisation opportunities.

Behaviour-preserving parallelisation. We expect our resource-usage analysis can be used in other synchronisation-related optimisations, but in this dissertation, we have used it as the basis for a parallelising transformation. This transformation is in the style of *deterministic parallelism* [BYLN09, JAD+09, BAD+10, BS10]—although our approach does not, in fact, require determinacy of the source program. In this vein, our transformation ensures that every behaviour of the parallelised program is a behaviour of the source sequential program (modulo the caveats about allocation and termination discussed in §4.4).

Previous approaches to deterministic parallelism have not used specifications to represent dynamically allocated resources. This places a substantial burden on the analysis and runtime to safely extract information on resource usage and transfer—information that is readily available in a proof. As a result, these analyses tend to be much more conservative in their treatment of mutable data. Our proof-based technique gives us a general approach to splitting mutable resources; for example, by allowing the analysis to perform ad-hoc list splitting, as we do with `move()`.

Loop parallelisation. Our approach transforms a sequential **`for`**-loop by running all the iterations in parallel and signalling between them, so it can be seen as a variant of a do-across style of loop parallelisation [TZ94]. There has been a vast amount of work in last couple of decades on do-across dependence analysis and parallelisation for loop nests. However, the focus of interest of these approaches are loop nests over linear data structures (i.e., arrays). For instance, polyhedral models [BVB+09], the most powerful dependence abstraction to date, assume an affine iteration domain of loop nests. In contrast, our approach does not require a specific iteration domain or fixed data-structure access patterns.

Instead of parallel-**`for`**, we could have used a more irregular concurrency annotation, for example safe futures [NZJ08], or an unordered parallel-**`for`**, as in the Galois system [PNK+11]. In the former case, our resource-usage analysis would be mostly unchanged, but our parallelised program would construct a set of syntactically distinct threads, rather than a pipeline of identical threads. Removing ordering between iterations, as in the latter case, would require replacing ordered `grant-wait` pairs with, e.g., conventional locks and would then introduce an obligation to show that locks were always acquired together, as a set.

Proof-driven parallelisation and program analyses. A central insight of our approach is that a separation logic proof expresses data dependencies for *parts* of a program, as well as for the whole program. These internal dependencies can be used to inject safe parallelism. This insight is due to [RCG09, CMR+10] and [Hur09], which propose parallelisation analyses based on separation logic. The analyses proposed in these papers are much more conservative than ours, in that they discover independence which already exists between commands of the program. They do not insert synchronisation constructs, and consequently cannot enforce sequential dependencies amongst concurrent computations that share and modify state. Indeed, [RCG09] does not consider any program transformations, since the goal of that work is to identify memory separation of different commands, while [Hur09] expresses optimisations as reordering rewrites on proof trees.

Bell *et al.* [BAW10] construct a proof of an *already-transformed* multi-threaded program parallelised by the DSWP (Decoupled Software Pipelining) transformation [ORSA05]. This approach assumes a specific pattern of (linear) dependencies in the while-loop consistent with DSWP, a specific pattern of sequential proof, and a fixed number of threads. In our `move()` example, the outermost (parallelising) loop contains two successive inner loops, while the example in Fig. 2.1 illustrates how the technique can deal with inter-procedural and control-flow sensitive dependencies. In both cases, the resulting parallelisation is specialised to inject synchronisation primitives to enforce sequential dependencies. We believe examples like these do not fall within the scope of either DSWP or the proof techniques supported by [BAW10].

Outside separation logic, Deshmukh *et al.* [DRRV10] propose an analysis which augments a sequential library with synchronisation. This approach takes as input a sequential proof expressing the correctness criteria for the library, and generates synchronisation ensuring this proof is preserved if methods run concurrently. A basic assumption is that the sequential proof represents all the properties that must be preserved. In contrast, we also preserve sequential order on access to resources. Consequently, Deshmukh *et al.* permit parallelisations that we would prohibit, and can introduce new behaviours into the parallelised program. Another difference is that [DRRV10] derives a linearisable implementation given a sequential specification in the form of input-output assertions; because they do not consider *specialisation* of multiple instances of the library running concurrently, it is unclear how their approach would deal with transformations of the kind we use for `move()`.

Golan-Gueta *et al.* [GBA+11] also explore adding locks to sequential libraries; in particular, they focus on tree- and forest-based data structures. They instrument the library's code so that it counts stack and heap references to objects, and then use these reference counts to determine when to acquire and release locks to guarantee conflictserialisability (and consequently, linearisability). The use of proofs in [GBA+11] is, however, only indirect: they use a shape analyzer to check that the library's heap graph is tree-shaped and live variable analysis to eliminate redundant code. Another difference is that our analysis does not assume a specific heap structure at any single point in the program nor it requires additional instrumentation of heap objects.

Raychev *et al.* [RVY13] describe an alternative program analysis and synchronisation inference mechanism. They take as an input a non-deterministic program (obtained by e.g. "naively" parallelising the sequential program) and make it deterministic by inserting barriers that enforce a thread schedule that avoids races. The thread ordering is determined by eliminating conflicts in a thread-modular abstraction of the program, following a set of rules to decide between feasible schedules. As a consequence, Raychev *et al.* end up with a determinisation whose behaviour is implied by these rules rather than by the behaviour of the original sequential program. To abstract the unbounded state, they use a flow-insensitive pointer analysis and numerical abstract domains, which works well for the structured numerical computations over arrays analyzed in their examples, but which would be much less effective for more irregular computations over dynamic data structures, as we consider here. For convergence, they employ a simple widening strategy (they widen after a constant number of iterations around the loop), which does not allow splitting of loop invariants, as we do in the transformation of `move()`.

Separation logic and concurrency. Separation logic is essential to our approach. It allows us to localise the behaviour of a program fragment to precisely the resources it accesses. Our proofs are written using concurrent separation logic [O'H07]. CSL has been extended to deal with dynamically-allocated locks [GBC+07, HAN08, JP09], re-entrant locks [HHH08], and primitive channels [HO08, BAW10, VLC10, LMS10]. Sequential tools for separation logic have achieved impressive scalability—for example [CDOY11] has verified a large proportion of the Linux kernel. Our work can be seen as an attempt to leverage the success of such sequential tools. Our experiments are built on coreStar [BDD+11, DP08], a language-independent proof tool for separation logic.

The parallelisation phase of our analysis makes use of the specifications for parallelisation barriers proposed in [DJP11]. That paper defined high-level specifications representing the abstract behaviour of barriers, and verified those specifications against the barriers' low-level implementations. However, it assumed that barriers were already placed in the program, and made no attempt to infer barrier positions. In contrast, we assume the high-level specification, and define an analysis to insert barriers. The semantics of barriers used in that paper and here was initially proposed in [NZJ08].

Deterministic Parallelism. As questions regarding the programmability of multicore systems become increasingly vocal, deterministic parallelism offers an attractive simplified programming model that nonetheless yield performance gains. Safe futures [WJH05] provide deterministic guarantees for Java programs annotated with a `future` construct using software transactional memory infrastructure; Grace [BYLN09] is another runtime technique that uses memory protection hardware for the same purpose. In both these systems, the observable behaviour of the parallelised program is the same as a sequential version, a version in which future-creation operations are treated as no-ops.

Deterministic parallelism can also be profitably applied to explicitly parallel applications; these systems guarantee that the program produces the same output regardless of the order in which threads run, but this result need not necessarily match a sequential execution. CoreDet [BAD+10], is a compiler and runtime system that uses ownership information along with a versioned memory to commit changes deterministically for multi-threaded C++ programs. Dthreads [LCB11] works at a higher level as a direct replacement for the `pthreads` library, improving the scalability with an efficient commit protocol and amortisation of overheads. Deterministic Parallel Java [JAD+09] uses a type and region effect system to enforce deterministic guarantees for concurrent Java programs.

Part II

Specification-directed parallelisation synthesis

Part II is concerned with a method for specification-directed parallelisation synthesis. We propose an approach in which we extract a faithful input-output specification of the program behaviour, representing it in a more compact form than the original program, and then use this specification for parallelised code synthesis. (As we show, such specifications can also be used to solve certain program analysis and verification tasks more easily.)

But why would we want to consider an approach like this? After all, if the specification embeds all material information about the program behaviour, it is hard to expect that the specification would be much easier to deal with than the original program. However, as it turns out, for *certain* classes of programs we might be able to extract the specification in an easier, "algorithmically manageable" form. Using such specification in place of the original program we can then reduce verification and synthesis tasks on programs to potentially more manageable tasks on specifications.

For instance, although checking whether two arbitrary programs have the same behaviour is undecidable, if such programs have specifications with decidable equivalence checking, we may be able to decide the equivalence of programs by checking equivalence of the corresponding specifications. Next, by using specifications, we might be able to perform certain algebraic optimisations more easily—for instance, reordering two pieces of code. Finally, compactness of specifications might allow us to generate a more efficient, optimised code directly from the specification, such as aforementioned parallelised version of the program.

Extracting a faithful input-output specification of a program is, however, hugely challenging. Automated analysis of program code containing loops and recursion or infinite data domains easily enters the undecidable territory. We propose to combat these problems by simultaneously exploiting two kinds of methods for exploring program paths: one which over-approximates program behaviour (an example of which is predicate abstraction) and another one which under-approximates it (an example of which is symbolic testing). If the program under-approximation ever becomes equal to the over-approximation then we have obtained a faithful specification.

The problem is that for realistic programs under-approximations will never converge to a true specification, as they may involve infinite sets of potentially infinite paths. The crux of our approach for extracting program specifications is to use grammar induction to generate conjectures which systematically generalise the set of program paths discovered in under-approximations. As we show, by alternating between such conjectures and over-approximations, we can reach convergence for some non-trivial classes of programs.

The attractiveness of specifications obtained this way is in that even for programs with unbounded data and loops we may infer finite-state symbolic specifications. For program parallelisation, the number of states in such specifications is often small enough so that we can parallelise the program by speculatively running the specification from all its states in parallel. In contrast to our proof-directed method in Part I which explicitly enforces all relevant dependencies using the proof of a program, here we merely acknowledge them by considering all execution scenarios. The role of the specification is to account for all executions, ensuring that we do not miss some potential behaviour.

Structure of Part II. In §8 we give an informal overview of our method for specification-directed parallelisation synthesis, with a focus on inference of faithful input-output specifications. Technical background needed for the formal development of further concepts and algorithms is given in §9. In §10 we formalise a class of input-output specifications which we can faithfully extract from a certain class of programs. §11 presents a technique for synthesising such symbolic input-output specifications from over-approximations and under-approximations generalised using grammar induction. In §12 we present several practical applications of our inference technique. We finish by discussing related work (§13).

CHAPTER 8

OVERVIEW

In this chapter, we give an informal overview of our specification-directed parallelisation synthesis. Central to this method is solving the problem of inferring a faithful specification of program behaviour. We propose a technique called Σ^*, which by combining black- and white-box analyses infers input-output specifications for a certain class of programs known as *stream filters* (see §12 for practical examples of such programs). Σ^* works on the control-flow graph without assuming any specific syntactic structure of the program and relies on grammar induction to inductively generalise program traces and an abstraction to guide the convergence. Once we obtain a faithful specification of the program, we use the specification for program parallelisation.

We start the chapter by briefly introducing stream filters in §8.1. We then give an overview of Σ^* in §8.2. We illustrate how Σ^* works on the example in §8.3. We conclude the chapter by briefly describing how the specifications inferred by Σ^* can be used for parallelisation.

8.1 Stream filters

Modern software systems often process large amounts of data in a stream-like fashion. By streams, we mean sequences of data items, processed in some order. Such stream processing is inherent to many domains such as digital signal processing, embedded applications, network event processing, financial applications, web applications, and even to common desktop software. Likewise, big-data services running on a cloud are nowadays faced with a need to not only scalably analyse large batches of past data but also to continuously process streams of moving data.

In this part we focus on a simple notion of programs that applies to many of these systems, called a *filter*. Filters are typically small fragments of code, but they facilitate large scale stream processing [BFH+04, CLL+05, DHRA06, GAW+08, GTA06, GCTR08, TKA02]. A filter iteratively reads a bounded number of items from an input stream at once, performs some computation on those items, and writes the results to an output stream. Filters exchange data via stream(s) with other parts of the program, and can also be glued together into more complex structures. Unfortunately, reasoning about even these relatively simple programs is difficult as their low-level implementation details often obscure the high-level input-output behaviour that we want to reason about.

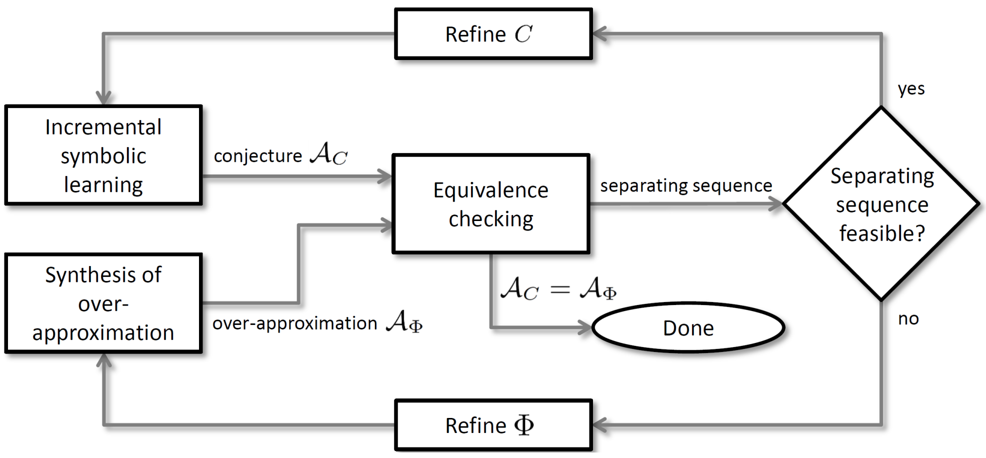

Refine C
Incremental symbolic learning
conjecture $\mathcal{A}_C$
Equivalence checking
separating sequence
Separating sequence feasible?
yes
no
Synthesis of over-approximation
over-approximation $\mathcal{A}_\Phi$
$\mathcal{A}_C = \mathcal{A}_\Phi$
Done
Refine Φ

Figure 8.1: The design of Σ^*. Arrows indicate the data flow.

8.2 Overview of Σ^*

The goal of Σ^* is to automatically synthesise input-output specifications of programs such as stream filters. In a nutshell, Σ^* can be seen as an extension of Angluin's L^* automata-learning algorithm [Ang87] to learning models of programs,[1] which exploits two key ideas:

1. Σ^* uses dynamic symbolic execution to discover path predicates (constraints on input values) and output terms (expressions which generate output values), and builds symbolic conjectures out of these using an L^*-like algorithm. By symbolically representing equivalence classes of program's input-output steps in the symbolic conjectures, Σ^* can discover input-output behaviour independently of the alphabet of concrete values. However, there are no guarantees that symbolic conjectures are either under- or over-approximations of the program's input-output behaviour.

2. Along with symbolic conjectures, Σ^* builds an abstraction that is a finite-state symbolic over-approximation of the program behaviour that we wish to learn. Checking equivalence of (deterministic) symbolic conjectures and such (possibly non-deterministic) over-approximations can be done algorithmically. Σ^* implements an algorithm that either generates a *separating sequence* showing how the conjecture and abstraction differ, or proves they are equivalent.

Starting with the trivial symbolic conjecture and the coarsest abstraction, Σ^* iteratively refines the conjecture or the abstraction until the two become equivalent (see Fig. 8.1). Otherwise, we compare the separating sequence returned by the equivalence

[1] L^*, designed to learn regular languages in a black-box manner, is not applicable to the general software setting for two reasons. Firstly, L^* requires a priori knowledge of the concrete alphabet of the language. While that fits certain scenarios (such as inference of component interfaces [ACMN05, SGP10, GRR12], where the alphabet corresponds to the set of component's methods), the exact alphabet of software internals (including, for instance, all values that arise during the execution) is hard to specify in advance. Even if known, such alphabets are typically large (or practically infinite), so treating them concretely as in L^* hinders scalability. Secondly, L^* merely conjectures the inferred model and to check the conjecture relies upon an oracle (a teacher) that can answer equivalence queries. An omniscient oracle for equivalence checking against software artefacts is, however, computationally intractable.

```
prev = 0; cur = 0; i = 0;
while (true) {
    cur = peek(0);
    if (i < 2)
        out(cur);
    else
        if (cur < prev)
            out(cur + prev);
        else
            out(cur - prev);
    prev = in();
    i++;
}
```

Figure 8.2: Code snippet for the running example in §8.3.

checking algorithm with the output produced by the program. If the program does not produce the same sequence of outputs as the abstraction then we call the separating sequence *spurious* (infeasible in the program) and refine the abstraction. Otherwise, the separating sequence represents a behaviour that the conjecture failed to capture, and we refine the conjecture. Upon termination (which in general is not guaranteed, but can be proven for a certain class of programs), the final conjecture faithfully captures the input-output behaviour of the program, and is in a certain sense minimal such specification.

8.3 Σ^* illustrated by example

We continue by showing how Σ^* works on the example given in Fig. 8.2. The program reads input in an infinite loop using commands **peek**() and **in**(); **peek**(0) reads the current input symbol and leaves it in the input stream, while **in**() reads the current input symbol and moves onto the next one. Both commands halt if there is no more input to read. In the first two iterations, the program prints the last read symbol (using command **out**), and in every subsequent iteration it outputs the sum or difference of the last two symbols, depending on their relative ordering.

We represent learnt conjectures and synthesised abstractions in the example using a variant of symbolic finite-state transducers [BGMV12] that we call symbolic *lookback* transducers (SLTs). To generate outputs, SLTs use a sliding window over a bounded number of past inputs (see §10 for a formal definition). Each transition is labelled by a symbolic predicate-term pair of the form p/t. The transition is taken when the predicate p evaluates to true for a given sequence of concrete input symbols. The term t describes the computed output. Predicates and terms are expressions over input symbols and constants. We use λ_0 to denote the current symbol and λ_{-i} to denote symbols read i transitions before. SLTs with lookback k are allowed to read only the current and the last k symbols. All SLTs in the example have lookback 1.

We use (dynamic) symbolic execution [GKS05, SMA05, CE05] to gather sequences of predicates and output terms occurring along the paths in the program execution.

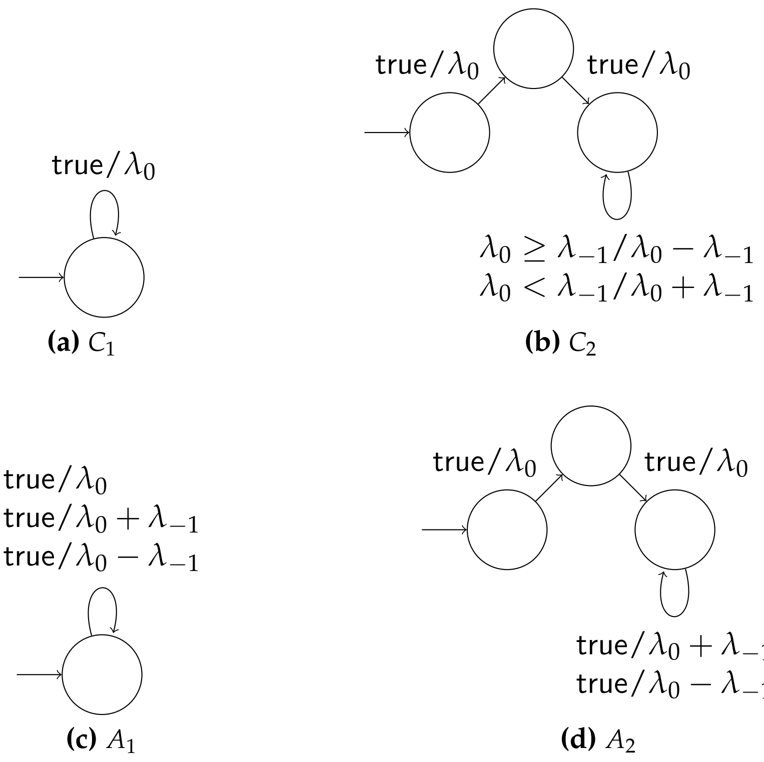

true/λ_0
(a) C_1
true/λ_0
true/λ_0
$\lambda_0 \geq \lambda_{-1}/\lambda_0 - \lambda_{-1}$
$\lambda_0 < \lambda_{-1}/\lambda_0 + \lambda_{-1}$
(b) C_2
true/λ_0
true/$\lambda_0 + \lambda_{-1}$
true/$\lambda_0 - \lambda_{-1}$
(c) A_1
true/λ_0
true/λ_0
true/$\lambda_0 + \lambda_{-1}$
true/$\lambda_0 - \lambda_{-1}$
(d) A_2

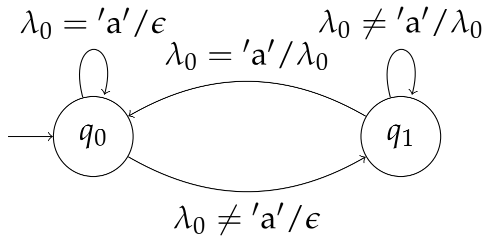
Figure 8.3: Conjectures and abstractions for the example in Fig. 8.2.

For instance, given an input of length four, a possible execution might take twice the then branch of the first if, then the else branch of the second if, and finally the then branch of the second if. For that execution we would obtain a predicate sequence (the path condition) $\vec{p} = \mathsf{true} \cdot \mathsf{true} \cdot (\mathsf{cur}_3 \geq \mathsf{prev}_2) \cdot (\mathsf{cur}_4 < \mathsf{prev}_3)$, and an output sequence $\vec{t} = \mathsf{cur}_1 \cdot \mathsf{cur}_2 \cdot (\mathsf{cur}_3 - \mathsf{prev}_2) \cdot (\mathsf{cur}_4 + \mathsf{prev}_3)$. In sequences such as $\vec{p}$ and $\vec{t}$, we use "$\cdot$" to denote the concatenation operator.

To obtain predicates and output terms formulated with regard to the input, we relativise all references of variables in $\vec{p}$ and $\vec{t}$ so that they become expressed with respect to the current position in the input stream. For instance, the relativised version of $\vec{p}$ would be the sequence $\mathsf{true} \cdot \mathsf{true} \cdot (\lambda_0 \geq \lambda_{-1}) \cdot (\lambda_0 < \lambda_{-1})$, and of $\vec{t}$, the sequence $\lambda_0 \cdot \lambda_0 \cdot (\lambda_0 - \lambda_{-1}) \cdot (\lambda_0 + \lambda_{-1})$. As for SLTs, the variable λ_{-j} in each element $\vec{p}|_i$ of the sequences above corresponds to the input symbol j positions before the current position (i) in the input stream. Thus, e.g., λ_0 in the third and fourth element of the sequences correspond to the last two symbols read by the program. To generate a concrete input from relativised path conditions, we need to derelativise them and pass the derelativised formula to an SMT solver. In the case above, the solver might return a concrete sequence $0 \cdot 1 \cdot 3 \cdot 2$ satisfying the relativised path condition.

Σ^* begins processing the example by learning the first conjecture C_1 (Fig. 8.3a), and compares it against predicate abstraction A_1 of the program with respect to an empty set of predicates (Fig. 8.3c). Note that, unlike in the classic predicate abstraction, we keep the input-output data flow intact. Equivalence checking of C_1 and A_1 produces the first counterexample to equivalence between C_1 and A_1, say a predicate $\vec{p}_1 = \mathsf{true}$ and output $\vec{t}_1 = (\lambda_0 + \lambda_{-1})$, which is a behaviour captured by the abstraction, but not the conjecture. Any concrete sequence satisfying those two formulae is a separating sequence. For

```
while (true) {
    x = in();
    while (x != 'a') {
        x = in();
        out(x);
    }
}
```



$\lambda_0 = 'a'/\epsilon$
$\lambda_0 = 'a'/\lambda_0$
$\lambda_0 \neq 'a'/\lambda_0$
q_0
q_1
$\lambda_0 \neq 'a'/\epsilon$

Figure 8.4: Left: a code example containing two nested loops. Right: the corresponding SLT, as inferred by Σ^*.

example, take input $\lambda_{-1} = 1, \lambda_0 = 1$. The program will produce output λ_0 for that input without reading λ_{-1}, showing that $\vec{p}_1/\vec{t}_1$ is infeasible in the program because symbolic outputs do not match. We refine the abstraction A_1 with respect to a new set of predicates, obtained by some predicate selection method (e.g., by taking atomic predicates in the weakest precondition computed along the path [Bal05]). Suppose such method returns $\{i = 0, i \geq 2\}$ at this instance. The refined abstraction A_2 is shown in Fig. 8.3d. Equivalence checking of A_2 and C_1 produces the second counterexample to equivalence, say $\vec{p}_2 = \mathsf{true} \cdot \mathsf{true} \cdot \mathsf{true}$ and $\vec{t}_2 = \lambda_0 \cdot \lambda_0 \cdot (\lambda_0 - \lambda_{-1})$. Any sequence of three symbols will satisfy $\vec{p}_2$ and it turns out that this counterexample to equivalence is not spurious. Thus, we now have to refine the conjecture C_1.

After processing the counterexample, Σ^* learns conjecture C_2 shown in Fig. 8.3b. C_2 is correct, but Σ^* does not know that yet, because equivalence checking of C_2 and A_2 returns the third counterexample to equivalence, say $\vec{p}_3 = \mathsf{true} \cdot \mathsf{true} \cdot (\lambda_0 < \lambda_{-1})$ and $\vec{t}_3 = \lambda_0 \cdot \lambda_0 \cdot (\lambda_0 - \lambda_1)$. The third counterexample is again spurious. Thus, one more time, we need to refine the abstraction. Using the same method for refinement as above, we extend the set of predicates with $\lambda_0 < \lambda_{-1}$. Predicate abstraction computes A_3, which is equivalent to C_2, and the process terminates, faithfully inferring the program's input-output relation, i.e., the specification of program's input-output behaviour.

For the example shown on the left of Fig. 8.4, which contains two nested loops, Σ^* terminates inferring a SLT with lookback 0 (shown on the right of Fig. 8.4).

8.4 Illustration of the parallelisation process

The objective of our specification-directed parallelisation synthesis is to take a sequential program and produce a parallelised version of the program by using its specification. Specifications in the form of SLTs can be easily executed but are inherently sequential. We now illustrate how such specifications can exploit available hardware concurrency using a simple speculative execution technique.

SLT specifications have an a-priori known number of internal states (which may be fairly small for stream filters typically encountered in some domains, such as those explored in §12). This allows an SLT to be speculatively executed on a portion of the input by concurrently running replicas of the SLT, each starting from a different initial state, and taking as the output what was generated in the run started from the "right" initial state.

For example, let us consider the stream filter from Fig. 8.4. Its corresponding SLT

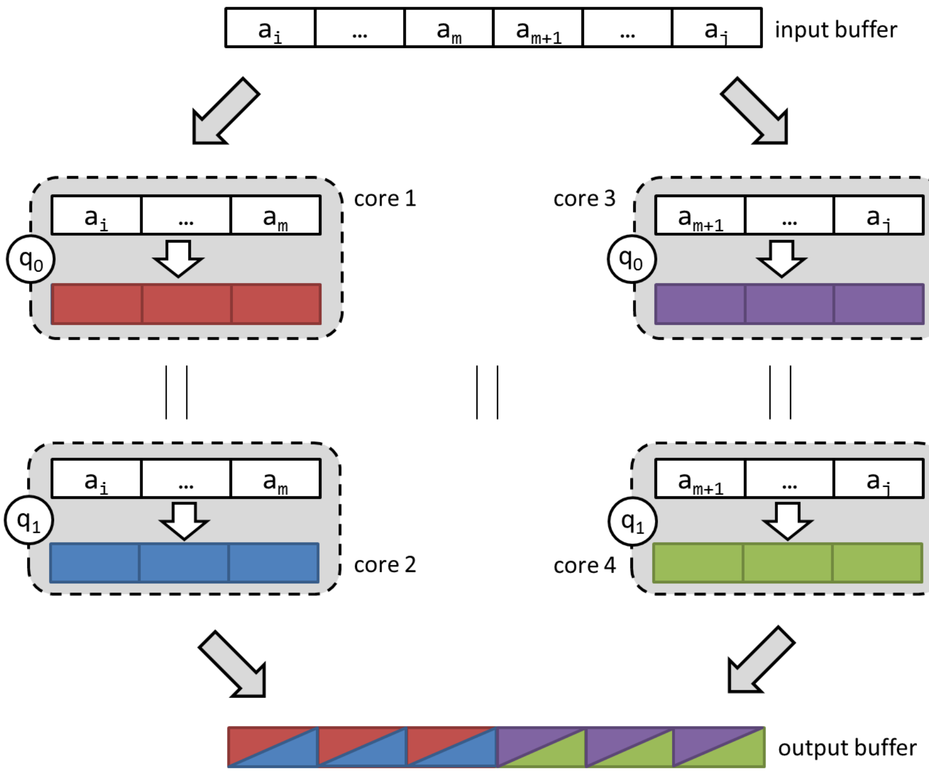


Figure 8.5: Parallelisation of the SLT from example in Fig. 8.4 using speculative execution.

has two states, say, q_0 and q_1. Let us assume that we wish to parallelise the execution of this filter on a system with 4 processor cores. Let us denote the portion of the input stream held in the input buffer by $a_i \cdots a_j$. Then we proceed as illustrated in Fig. 8.5. We split the input buffer into $a_i \cdots a_m$ and $a_{m+1} \cdots a_j$ (where $i < m < j$), and process each portion of the input by a separate set of SLT replicas. In this example, each portion is processed by two SLT replicas, started from states q_0 and q_1. After each SLT has finished processing, we merge suitable outputs so that if say, processing of $a_i \cdots a_m$ has finished in state q_1 then for $a_{m+1} \cdots a_j$ we take the output obtained by an SLT started from q_1 (the other case is analogous). Note that if a_i is the first element of the input stream or we know from which state to start (e.g. because the previous execution batch has already completed) then to process the first portion of the input buffer we need only a single copy of the SLT.

This naïve strategy for splitting the input and mapping the SLT states to cores may of course not be optimal, and more elaborate strategies as well as other parallelisation backends are certainly possible. Our focus in the remainder of this part is, however, not on such practical applications but on describing in detail the inference of SLTs by Σ^*.

Chapter 9

Technical background

In the previous chapter we gave an overview of Σ^* and our specification-directed parallelisation method. In this chapter we turn to the formal development of Σ^*. We begin, in §9.1 by formalising the kind of programs that facilitate stream processing, examples of which we saw in §8. We also formally define concepts associated with program analysis, which we use in the synthesis of over-approximations (§11.1) and learning conjectures (§11.2). At the end of the chapter (§9.2), we introduce the class of transductive and k-lookback programs. Our goal in subsequent chapters will be to discern a subclass of such programs whose behaviour can be represented by SLTs (§10).

9.1 Transductive programs

We start by introducing the notation used throughout this part. Next we formalise the syntax (§9.1.1) and semantics (§9.1.2) of programs that transform an input stream into an output stream. We finish by explaining the concepts associated with the dynamic symbolic execution (§9.1.3). Throughout the formal development we assume programs working on streams of integers.

Preliminaries. Let $\mathbb{Z}$ be the set of integers, $\mathbb{N} = \{i \in \mathbb{Z} \mid i > 0\}$ the set of naturals, and $\mathbb{B} = \{\mathsf{false}, \mathsf{true}\}$ the set of booleans. For a set D, we denote by D^* the free monoid generated by D with concatenation as the operation and the empty word ϵ as identity. We refer to words in D^* as sequences. Sequences of predicates we denote by $\vec{p} = p_1 \cdots p_m$. Individual predicates will typically be associated with sequences of terms, for which we use notation $\mathbf{t} = t_1 \cdots t_n$. We identify sequences of such $\mathbf{t}$-s as members of the free monoid over terms, and denote them by $\vec{t}$. We denote the length of $\vec{p}$ by $|\vec{p}|$, the i-th element of $\vec{p}$ by $\vec{p}|_i$ and the subsequence $p_i \cdots p_{i+k}$ by $\vec{p}|_{[i,i+k]}$. For $f \colon D \to C$ and $D' \subseteq D$, we write $f|_{D'}$ to denote the restriction of f on D'. For $\vec{a}$ and $\vec{b}$ such that $n \triangleq |\vec{a}| = |\vec{b}|$, we write $f[\vec{a} \leftarrow \vec{b}]$ to denote the function f' such that for all $i \in \{1, \dots, n\}$, $f'(\vec{a}|_i) = \vec{b}|_i$ and for all $x \notin \{\vec{a}|_1, \dots, \vec{a}|_n\}$, $f'(x) = f(x)$. We use $\mathbf{t}[x \mapsto y]$ to denote the sequence of terms obtained from $\mathbf{t}$ by simultaneously replacing all free occurrences of x with y. $\mathsf{Vars}(t)$ denotes the set of all free variables in t.

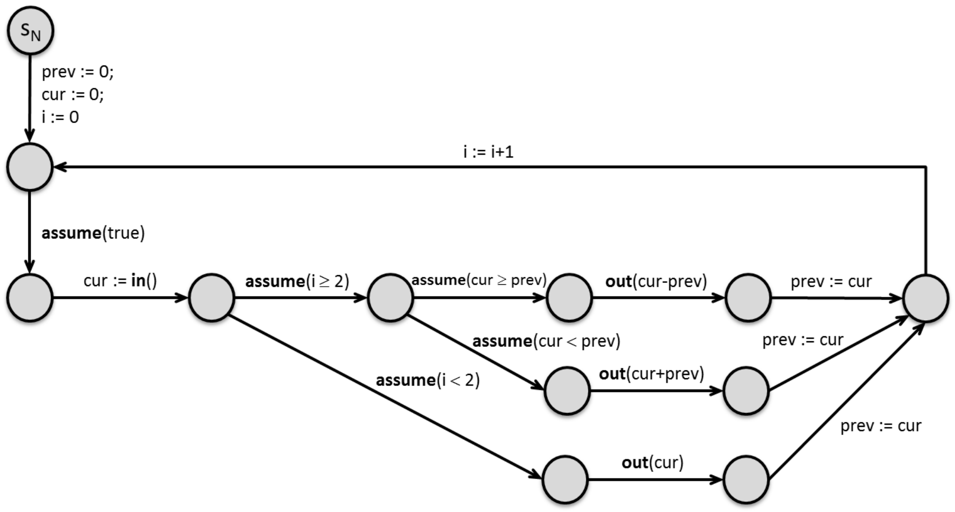


Figure 9.1: Control-flow automaton based representation of the example in Fig. 8.2 from §8.3. (The sequential composition of assignments on the outgoing edge of s_N is merely a "syntactic sugar" for a chain of individual assignments.)

9.1.1 Program representation

We first define syntactic representation of programs. Let V be the set of program variables. Expressions (terms) $Exp[V]$, predicates $BExp[V]$, and atomic commands $Cmd[V]$ over V are given by:

$$
\begin{array}{rcll}
e & ::= & \mathsf{k} \mid x \mid \mathsf{f}(\bar{e}) \mid \ldots & \in Exp[\mathsf{V}] \\
b & ::= & \mathsf{false} \mid \mathsf{true} \mid \neg b \mid b \wedge b \mid b \vee b \mid & \\
 & & e = e \mid e \neq e \mid \mathsf{R}(\bar{e}) \mid \ldots & \in BExp[\mathsf{V}] \\
c & ::= & \mathsf{assume}(b) \mid x := e \mid x := \mathsf{peek}(\mathsf{k}) \mid & \\
 & & x := \mathsf{in}() \mid \mathsf{out}(e) & \in Cmd[\mathsf{V}]
\end{array}
$$

where $\mathsf{k} \in \mathbb{Z}$, $x \in \mathsf{V}$, $\bar{e}$ denotes a tuple of terms, while f and R are constructors for terms and predicates. As long as the quantifier-free theory of $Exp[V]$ with equality and satisfiability of $BExp[V]$ are decidable, particular details of the grammar are not important. The command $x := \mathsf{peek}(\mathsf{k})$ reads a data item at offset k from the current symbol in the input stream and stores the item to x if the input is available, otherwise it causes the program to halt. The $\mathsf{in}()$ command works like $\mathsf{peek}(0)$ but in addition consumes the item from the input stream and moves onto the next input symbol. The command $\mathsf{out}(e)$ writes the value of e to the output stream.

We represent programs using control-flow automata [HJMS02] over the language of atomic commands. The control-flow automaton is determined by a set of control nodes N containing a distinguished node $s_N \in N$ representing the starting point of the program and a function $\mathsf{succ}\colon N \times Cmd[V] \rightharpoonup N$ representing labelled edges. All nodes either have a single successor or have all outgoing edges labelled with a label of the form $\mathsf{assume}(b)$. In the latter case, we assume that all corresponding predicates b are mutually exclusive and that their disjunction is a tautology.

Example 9.1. Fig. 9.1 shows the control-flow automaton for the example in Fig. 8.2 from §8.3.

$$\begin{array}{rcl}
[\![\mathsf{k}]\!]_\sigma & = & \mathsf{k} \\
[\![x]\!]_\sigma & \doteq & \begin{cases} \sigma(x) & \text{if } x \in \mathsf{V}, \\ x & \text{if } x \in \mathsf{V}_\mathsf{I}, \\ \text{undefined} & \text{otherwise} \end{cases} \\
[\![\mathsf{f}(\bar{e})]\!]_\sigma & \doteq & \mathsf{f}([\![e_1]\!]_\sigma, \ldots, [\![e_n]\!]_\sigma)
\end{array}$$

Figure 9.2: Symbolic evaluation of expressions. Symbol σ denotes a map of type Mem.

9.1.2 Symbolic semantics

Given a program $\mathcal{P}$ we now define its symbolic semantics. We represent the contents of the input and the output stream using special variables $\mathsf{V}_\mathsf{I} \triangleq \{in_1, in_2, \ldots\}$ (*in*-variables) and $\mathsf{V}_\mathsf{O} \triangleq \{out_1, out_2, \ldots\}$ (*out*-variables), respectively (with V_I, V_O and V mutually disjoint from each other). On a particular run of $\mathcal{P}$, all except finitely many of these variables are undefined, and for those which are defined, $in_\imath$ corresponds to the $\imath$-th element of the input stream, and $out_\jmath$ to the $\jmath$-th element of the output stream. By $\vec{in}$ we refer to the sequence of *in*-variables $in_1 in_2 \ldots$ and we use ℓ_{in} to denote the length of the data sequence in the input stream.

We evaluate expressions on a memory $\sigma \in \mathsf{Mem} \triangleq \mathsf{V} \rightharpoonup Exp[\mathsf{V}_\mathsf{I}]$ as indicated in Fig. 9.2. We execute commands on states of the form

$$\mathsf{State} \triangleq \mathsf{N} \times \mathsf{Mem} \times \mathsf{OutStr} \times \mathbb{N} \times \mathbb{N}.$$

The third component, $\mathsf{OutStr} \triangleq \mathsf{V}_\mathsf{O} \rightharpoonup Exp[\mathsf{V}_\mathsf{I}]$, represents the symbols in the output stream. The last two components of the state, the *in*-index and the *out*-index, store indices of the current element in the input stream and the output stream, respectively. We denote the components of a state $s \in \mathsf{State}$ by $s.\mathsf{n}$, $s.\sigma$, $s.\rho$, $s.\imath$, and $s.\jmath$, respectively. The initial state is $s_0 = (\mathsf{s_N}, \emptyset, \emptyset, 1, 1)$.

We represent the symbolic semantics of $\mathcal{P}$ by a labelled transition system (LTS) $\mathcal{T} = (\mathsf{State}, C, \rightsquigarrow)$ with State as the set of states, $C \subseteq Cmd[\mathsf{V}]$ as the finite set of labels, and $\rightsquigarrow$ as the labelled transition relation. The relation $\rightsquigarrow \subseteq \mathsf{State} \times C \times \mathsf{State}$ is defined by the rules given in Fig. 9.3, showing the effect of each command on a state. In each rule we have $n' = \mathsf{succ}(n, c)$. We write $s \overset{c}{\rightsquigarrow} s'$ to denote the transition $(s, c, s') \in \rightsquigarrow$.

9.1.3 Dynamic symbolic traces, path predicates and outputs

The LTS $\mathcal{T}$ encodes the symbolic semantics of $\mathcal{P}$ with respect to any input. Given a concrete input $\vec{a} \in \mathbb{Z}^*$, we can construct a sequence of states in $\mathcal{T}$ representing the (dynamic) symbolic trace of $\mathcal{P}$ driven by $\vec{a}$, as is standard in dynamic symbolic execution [GKS05, SMA05, CE05]. We start with the initial state s_0 and at each step we follow the transition $s_i \overset{c}{\rightsquigarrow} s_{i+1}$, such that if c is of the form $\mathsf{assume}(b)$ then b is true in $s_i.\sigma[\vec{in} \leftarrow \vec{a}]$. We stop if we ever reach a state s_f, which we call ending, such that the *in*-index $s_f.\imath$ equals $\ell_{\mathsf{in}} + 1$ or s_f has an outgoing $\overset{\_:=\mathsf{peek}(\mathsf{k})}{\rightsquigarrow}$-transition with k such that $s_f.\imath + \mathsf{k} > \ell_{\mathsf{in}}$. We define the symbolic trace of $\mathcal{P}$ on input $\vec{a}$, denoted $\tau_{\mathcal{P}}(\vec{a})$, as the

$$
\begin{array}{lcl}
(\mathsf{n},\sigma,\rho,\imath,\jmath) & \overset{\mathsf{assume}(b)}{\rightsquigarrow} & (\mathsf{n}',\sigma,\rho,\imath,\jmath) \\
(\mathsf{n},\sigma,\rho,\imath,\jmath) & \overset{x:=e}{\rightsquigarrow} & (\mathsf{n}',\sigma[x \leftarrow [\![e]\!]_\sigma],\rho,\imath,\jmath) \\
(\mathsf{n},\sigma,\rho,\imath,\jmath) & \overset{x:=\mathsf{peek}(\mathsf{k})}{\rightsquigarrow} & (\mathsf{n}',\sigma[x \leftarrow [\![in_{\imath+\mathsf{k}}]\!]_\sigma],\rho,\imath,\jmath) \\
(\mathsf{n},\sigma,\rho,\imath,\jmath) & \overset{x:=\mathsf{in}()}{\rightsquigarrow} & (\mathsf{n}',\sigma[x \leftarrow [\![in_\imath]\!]_\sigma],\rho,\imath+1,\jmath) \\
(\mathsf{n},\sigma,\rho,\imath,\jmath) & \overset{\mathsf{out}(e)}{\rightsquigarrow} & (\mathsf{n}',\sigma,\rho[out_\jmath \leftarrow [\![e]\!]_\sigma],\imath,\jmath+1)
\end{array}
$$

Figure 9.3: Symbolic semantics of commands. In each rule we have $n' = \mathsf{succ}(n,c)$, where c is the label of the $\overset{c}{\rightsquigarrow}$-transition.

finite sequence $s_0 s_1 \dots s_f$ if an ending state is reached, or as the infinite sequence $s_0 s_1 \dots$ otherwise.

We refer to a state s_i as *in-state* if $s_i = s_0$, $s_i = s_f$ or the incoming transition reads an input symbol, i.e., $s_{i-1} \overset{\_:=\mathsf{in}()}{\rightsquigarrow} s_i$. Let $\tau^{\mathsf{in}}_{\mathcal{P}}(\vec{a}) \triangleq \tilde{s}_0 \tilde{s}_1 \dots \tilde{s}_n$ be the sequence of *in*-states from $\tau_{\mathcal{P}}(\vec{a})$. We define a *big-step transition* to be a transition between two successive states in $\tau^{\mathsf{in}}_{\mathcal{P}}(\vec{a})$. For $1 \leq i \leq n$, let p_i be the conjunction of predicates b of the form $\mathsf{assume}(b)$ encountered on all transitions between $\tilde{s}_{i-1}$ and $\tilde{s}_i$. In case no $\overset{\mathsf{assume}(\_)}{\rightsquigarrow}$-transitions exist between $\tilde{s}_{i-1}$ and $\tilde{s}_i$ we set p_i to true. Furthermore, for $1 \leq i \leq n$, let $\mathbf{t}_i$ be the sequence of symbolic outputs (as written to *out*-variables by $\overset{\mathsf{out}(\_)}{\rightsquigarrow}$-transitions) between $\tilde{s}_{i-1}$ and $\tilde{s}_i$. If no output is generated we set $\mathbf{t}_i$ to ϵ.

We define *path predicates* of $\mathcal{P}$ on input $\vec{a}$ as the sequence $\pi_{\mathcal{P}}(\vec{a}) \triangleq p_1 \cdots p_n$. Analogously, we define the *symbolic output* of $\mathcal{P}$ on $\vec{a}$ by $\theta_{\mathcal{P}}(\vec{a}) \triangleq \mathbf{t}_1 \cdots \mathbf{t}_n$. The *concrete output* of $\mathcal{P}$ on $\vec{a}$ is defined by $\gamma_{\mathcal{P}}(\vec{a}) \triangleq \theta_{\mathcal{P}}(\vec{a})[\vec{in} \mapsto \vec{a}]$.

9.2 Transductive and k-lookback programs

In order to be able to learn program behaviour, we have to make certain assumptions about programs. Firstly, when considering the input-output behaviour of a program, no path in a program should end up in a loop without reading an input, i.e., we do not wish to allow infinite periods of silence during which the program would not read any input. This first assumption allows us to extend conjectures of program behaviour in the learning algorithm with new big-step transitions. Secondly, we need to assume that programs produce uniformly bounded number of output symbols per each big-step transition. This second assumption is inherent to symbolic transducers [BGMV12], on which we base our notion of SLTs (§10). In the rest of this section we formalise these assumptions.

Let $\mathcal{T}$ be the LTS of $\mathcal{P}$ and ξ any symbolic trace in $\mathcal{T}$ starting with s_0 and ending with an *in*-state. Let us denote by $\#in(\xi)$ the number of $\overset{\_:=\mathsf{in}()}{\rightsquigarrow}$-transitions in ξ and by $\#out(\xi)$ the number of $\overset{\mathsf{out}(\_)}{\rightsquigarrow}$-transitions in ξ.

Definition 9.1. We say that $\mathcal{P}$ is *transductive*[1] if there are no infinite symbolic traces in $\mathcal{T}$ with only finitely many *in*-states, and there exists $M \in \mathbb{N}$ such that for every ξ as above we have $\#out(\xi) < M \cdot \#in(\xi)$.

As a consequence, if $\mathcal{P}$ is transductive then for every input $\vec{a}$, $\tau_{\mathcal{P}}(\vec{a})$ is finite and $|\theta_{\mathcal{P}}(\vec{a})| = |\gamma_{\mathcal{P}}(\vec{a})| < M \cdot |\vec{a}|$. Also note that all transductive programs have finite *lookahead*, which is bounded by the largest k appearing in $\overset{\_:=\mathsf{peek}(\mathsf{k})}{\rightsquigarrow}$-transitions.

Example 9.2. Consider the following program:

```
n := peek(0);
i := 1;
while (i <= n) {
  out(i);
  i := i+1;
}
```

This program is not transductive because the number of output symbols produced in its single big-step transition is dependent on the value of the first input symbol, which cannot be a priori bounded.

Besides transductiveness, we impose an additional requirement that is specific to the type of transducers that we use to represent program behaviour in Σ^*, but that could potentially be relaxed if we would use more expressive models (such as, e.g., transducers with registers [AC11, BGMV12]). This additional requirement asks that transductive programs produce outputs from a bounded number of inputs in the past. More formally:

Definition 9.2. We say that $\mathcal{P}$ is *k-lookback* if $\mathcal{P}$ is transductive and for every ξ as above, each $\overset{\mathsf{out}(e)}{\rightsquigarrow}$-transition in ξ depends only on previous k inputs, i.e., if $s \overset{\mathsf{out}(e)}{\rightsquigarrow} s'$ then $\mathsf{Vars}([\![e]\!]_{s.\sigma}) \subseteq \{in_{s.\iota-k}, \ldots, in_{s.\iota}\}$.

Fig. 9.4 illustrates the relationship between the program classes of transductive and lookback programs. In §11.1 and §11.2 we focus on a subclass of k-lookback programs whose behaviour can be represented by SLTs (SLT-representable programs in Fig. 9.4), which, as we show, can be learned by Σ^*'s learning algorithm.

Example 9.3. Consider the following program which computes prefix sums of the sequence of numbers in the input stream:

```
n := 1;
while (true) {
  s := in();
  i := 1;
  while (i < n) {
    x := peek(-1*i - 1);
    s := s + x;
```

[1]We use the term "transductive" to indicate that such a program implements a *transduction*, i.e., a partial function $\Gamma_1^* \rightharpoonup \Gamma_2^*$, where Γ_1 and Γ_2 are alphabets. Note, however, that there are transductions that cannot be implemented by any transductive program as defined by Def. 9.1.

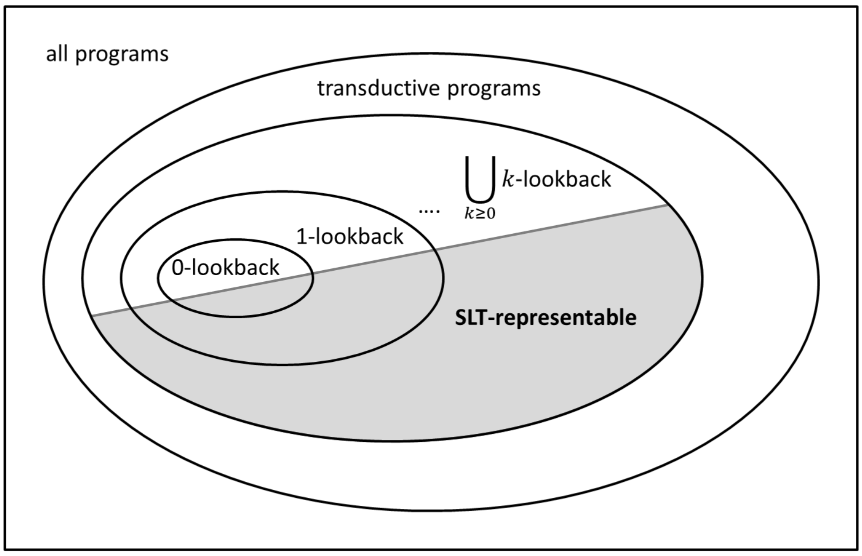


Figure 9.4: Relationship between program classes: transductive (Def. 9.1), lookback (Def. 9.2) and SLT-representable programs (Def. 11.2 in §11.1).

```
    i := i + 1;
  }
  out(s);
  n := n + 1;
}
```

This program is transductive, however, since the output computed in each big-step transition involves all previous inputs, the program is not *k-lookback* for any *k*.

9.2.1 Input relativisation

In the development of over-approximations (§11.1) and the learning algorithm (§11.2), we will need to rephrase symbolic expressions pertaining to program states so that they use the offset from the current position in the input stream rather than from the beginning. We next show how to relativise these expressions so that the offset becomes relative to the current *in*-index.

Let $\mathsf{V}_\lambda \triangleq \{\ldots, \lambda_{-1}, \lambda_0, \lambda_1, \ldots\}$ be an infinite set of λ-variables, indexed by a relative offset. We use λ-variables so that λ_0 stands for the current symbol in the input stream, λ_{-i} for the symbol i positions back and λ_i for the symbol i positions ahead. Let $\vec{in} \overset{\imath}{\mapsto} \overleftarrow{\lambda}$ denote a variable substitution that maps each *in*-variable $in_\jmath$ to λ-variable $\lambda_{\jmath-\imath}$. We define *relativisation* of a state $s = (\mathsf{n}, \sigma, \rho, \imath, \jmath)$ by $\Lambda(s) \triangleq (\mathsf{n}, \sigma[\vec{in} \overset{\imath}{\mapsto} \overleftarrow{\lambda}], \rho[\vec{in} \overset{\imath}{\mapsto} \overleftarrow{\lambda}], \jmath)$. Intuitively, $\Lambda(s)$ relativises symbolic expressions in s with respect to the current *in*-index. For general transductive programs, expressions in σ- and ρ-components of $\Lambda(s)$ may use unboundedly many λ-variables as the expressions can refer to inputs from arbitrary far in the past. However, k-lookback programs use only up to k λ-variables with negative index, namely, $\lambda_{-k}, \ldots, \lambda_{-1}$.

Chapter 10

Symbolic transducers with lookback

In the previous chapter we formalised the syntax and semantics of programs that transform an input stream of symbols into an output stream of symbols, and introduced the notion of transductive and lookback programs. In this chapter we turn to a formal description of symbolic finite-lookback transducers (SLTs), which serve as input-output specifications for a subclass of lookback programs (namely, SLT-representable programs, formally defined in §11.1). We carefully limited the expressivity of SLTs so as to be able to represent interesting classes of stream-manipulating programs but still be able to learn and check the corresponding SLTs for equivalence. Σ^* uses SLTs to represent both the synthesised over-approximations (§11.1) and symbolic conjectures (§11.2) in the learning algorithm.

We parameterise SLTs by a parameter k, which limits the set of input variables that can be used within expressions in SLT transitions. The k factor can be seen as a size of the sliding window, from which an SLT can read inputs. In our formalisation (§10.1), this window begins at the current input tape head position and covers k last symbols, because we observed that such setting is most suitable for the practical examples we considered (see §12). In §10.2, we present constructive algorithms for composition (§10.2.1) and equivalence checking (§10.2.1) of such SLTs. There is no fundamental reason why the sliding window could not also cover symbols ahead of the current head position, which we briefly address in §10.2.3. We conclude by highlighting characteristics of SLTs learned by Σ^* (§10.2.4), which among other things allows efficient equivalence checking of such SLTs.

10.1 Definitions

Symbolic lookback transducers (SLTs) can be seen as a generalisation of the classical finite-state transducers. Finite-state transducers extend the basic model of finite-state automata by augmenting state transitions with production of output (a single or a sequence of output symbols) after receiving an input symbol (see [LY96] for a detailed overview of various models of finite-state transducers). *Symbolic* finite-state transducers, introduced by Bjørner et al. [BGMV12], extend finite-state transducers by symbolic transitions, which allow the input-output step of the transducer to be defined using arbitrary predicates as triggers for accepting an input and symbolic terms for producing an output. Our notion of SLTs extends symbolic finite-state transducers of Bjørner et al. with a sliding window over the input, which allows an SLT to generate outputs based on a bounded number of inputs from the past.

We now formally define symbolic finite-state transducers with k-lookback — k-SLTs. The formal development is specialized to the case where the alphabet of input and output symbols is $\mathbb{Z}$, and the label theory for predicates and output terms is built using the same grammar rules as program predicates and terms in §9. This technical simplification aligns the assumptions in the exposition of SLTs with the assumptions about programs given in §9.[1]

Instead of reading a single symbol from the input tape at a time, the tape head of a k-SLT is effectively a window of size $k+1$, reading the current and the last k symbols. Equivalently, such a transducer can be seen as a transducer with k registers updated in a FIFO manner on each transition—the newly read symbol is inserted, while the oldest is removed from the queue. Rather than using registers, we use the set of λ-variables $\mathsf{V}_\lambda^k := \{\lambda_{-k}, \ldots, \lambda_{-1}, \lambda_0\}$ so that λ_0 is the current symbol and λ_{-i} is the symbol i positions back.

Definition 10.1. A symbolic finite transducer with lookback k (k-SLT) is a tuple $\mathcal{A} = (Q, q_0, F, \Delta)$ where Q is a finite set of states, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final states and $\Delta \subseteq Q \times BExp[\mathsf{V}_\lambda^k] \times Q \times Exp[\mathsf{V}_\lambda^k]^*$ is a finite transition relation.

Each $(q, \varphi, q', \mathbf{t}) \in \Delta$ from the the Def. 10.1 defines a symbolic transition between states q and q', with the predicate φ being an input guard and the sequence of terms $\mathbf{t}$ a symbolic output. We use a more intuitive notation $q \xrightarrow{\varphi/\mathbf{t}} q'$ to denote such symbolic transitions.

Definition 10.2. We say that SLT $\mathcal{A}$ is *deterministic* if for every two transitions $q \xrightarrow{\varphi/\mathbf{t}} r$ and $q \xrightarrow{\varphi'/\mathbf{t}'} r'$ if $\varphi \wedge \varphi'$ is satisfiable then $r = r'$ and $(\varphi \wedge \varphi') \Rightarrow \mathbf{t} = \mathbf{t}'$ is valid.

We next define how SLTs produce output. Before defining a run of an SLT, we introduce some convenient notation. For brevity, we refer to the sequence $\lambda_{-k} \ldots \lambda_0$ as $\overleftarrow{\lambda}$. To process the input, k-SLT prepends it with k dummy symbols $\perp \notin \mathbb{Z}$. Any operation with $\perp$ yields $\perp$ and every comparison with $\perp$ (except $\perp = \perp$) is false. For a sequence $\vec{a}$, let us denote $\vec{a}^\perp \triangleq \perp^k \cdot \vec{a}$.

Definition 10.3. A *run of* a k-SLT $\mathcal{A} = (Q, q_0, F, \Delta)$ on $\vec{a} \in \mathbb{Z}^n$ is a finite sequence of states $q_0 \ldots q_n$ such that there exists a sequence of transitions

$$q_0 \xrightarrow{\varphi_1/\mathbf{t}_1} q_1 \xrightarrow{\varphi_2/\mathbf{t}_2} q_2 \cdots q_{n-1} \xrightarrow{\varphi_n/\mathbf{t}_n} q_n,$$

where $\varphi_1, \ldots, \varphi_n \in BExp[\mathsf{V}_\lambda^k]$ and $\mathbf{t}_1, \ldots, \mathbf{t}_n \in Exp[\mathsf{V}_\lambda^k]^*$ such that for all $1 \le i \le n$, $\vec{a}^\perp|_{[i,i+k]}$ satisfies φ_i. We say that $\mathcal{A}$ on the input $\vec{a}$ *produces the output* $\vec{o} \in \mathbb{Z}^*$ and write $\vec{a} \twoheadrightarrow_{\mathcal{A}} \vec{o}$ if $q_n \in F$ and for all $1 \le i \le n$, $\mathbf{o}_i = \mathbf{t}_i \left[\overleftarrow{\lambda} \mapsto \vec{a}^\perp|_{[i,i+k]}\right]$, where $\vec{o}|_i = \mathbf{o}_i$.

[1]While the choice of the label theory is not important as long as the satisfiability of predicates and the quantifier-free theory of terms is decidable, generalisation to background structures (sorts) other than $\mathbb{Z}$, while straightforward, may render the equivalence checking undecidable.

If $\mathcal{A}$ is deterministic, the run is uniquely determined by the input sequence. For a deterministic $\mathcal{A}$ and $\vec{a} \in \mathbb{Z}^*$, let us denote by $\pi_{\mathcal{A}}(\vec{a})$, $\theta_{\mathcal{A}}(\vec{a})$ and $\gamma_{\mathcal{A}}(\vec{a})$ the corresponding sequences $\varphi_1 \cdots \varphi_n$, $\mathbf{t_1} \cdots \mathbf{t_n}$, and $\mathbf{o}_1 \cdots \mathbf{o}_n$, respectively.

We define the *transduction* of (a possibly non-deterministic) $\mathcal{A}$ as a function $\mathcal{T}_{\mathcal{A}} \colon \mathbb{Z}^* \to 2^{\mathbb{Z}^*}$ defined by $\mathcal{T}_{\mathcal{A}}(\vec{a}) \triangleq \{\vec{o} \mid \vec{a} \twoheadrightarrow_{\mathcal{A}} \vec{o}\}$. The domain of $\mathcal{A}$ is $\mathrm{dom}(\mathcal{A}) \triangleq \{\vec{a} \in \mathbb{Z}^* \mid \mathcal{T}_{\mathcal{A}} \neq \emptyset\}$.

Definition 10.4. We say that $\mathcal{A}$ is *single-valued* if for all $\vec{a}$, $|\mathcal{T}_{\mathcal{A}}(\vec{a})| \leq 1$.

Note that single-valued SLTs capture transductions that behave as partial functions $\mathbb{Z}^* \rightharpoonup \mathbb{Z}^*$. Also note that deterministic SLTs are necessary single-valued (however, the converse is generally not true [BGMV12]).

We conclude this section by noting that in the terminology of finite-state transducers [LY96], an SLT can be seen as a variant of a symbolic sequential ϵ-input-free (i.e., real-time) transducer which in general can be non-deterministic, and the number of different outputs it may produce on an arbitrary input need not be bounded.

10.2 Algorithms for SLTs

In this section we describe algorithms for composition and equivalence checking of SLTs.

10.2.1 Composition of SLTs

Given two transductions $\mathcal{T}_1, \mathcal{T}_2 \colon \mathbb{Z}^* \to 2^{\mathbb{Z}^*}$ we define the composition $\mathcal{T}_1 \circ \mathcal{T}_2$ as follows:[2]

$$(\mathcal{T}_1 \circ \mathcal{T}_2)(\vec{a}) \triangleq \bigcup_{\vec{o} \in \mathcal{T}_1(\vec{a})} \mathcal{T}_2(\vec{o})$$

SLTs are in general not closed under composition, as we now show:

Proposition 10.5. There exist deterministic SLTs $\mathcal{A}$ and $\mathcal{B}$ such that $\mathcal{T}_{\mathcal{A}} \circ \mathcal{T}_{\mathcal{B}}$ is not a transduction of an SLT.

Proof. [3] We take $\mathcal{A}$ to be an SLT which outputs all input symbols satisfying some predicate φ, i.e., we let $\mathcal{A} \triangleq (\{p\}, p, \{p\}, \Delta_{\mathcal{A}})$, where $\Delta_{\mathcal{A}} = \left\{ p \xrightarrow{\varphi/\lambda_0} p,\ p \xrightarrow{\neg\varphi/\epsilon} p \right\}$. Note that $\mathcal{A}$ is a deterministic 0-SLT.

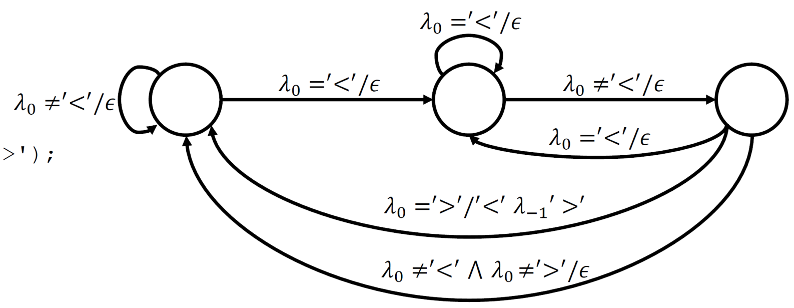
We take $\mathcal{B}$ to be an SLT which swaps input symbols in pairs, i.e., we let $\mathcal{B} \triangleq (\{q_0, q_1\}, q_0, \{q_0, q_1\}, \Delta_{\mathcal{B}})$, where $\Delta_{\mathcal{B}} = \left\{ q_0 \xrightarrow{\mathsf{true}/\epsilon} q_1, q_1 \xrightarrow{\mathsf{true}/\lambda_0 \cdot \lambda_{-1}} q_0 \right\}$. Note that $\mathcal{B}$ is a deterministic 1-SLT.

Then in $(\mathcal{T}_{\mathcal{A}} \circ \mathcal{T}_{\mathcal{B}})(\vec{x})$ we obtain elements of $\vec{x}$ that match φ, swapped in pairs. However, the distance between two consecutive elements that match φ is unbounded, thus there is no possible lookback bound for a transducer that would realise $\mathcal{T}_{\mathcal{A}} \circ \mathcal{T}_{\mathcal{B}}$. $\square$

[2] Contrary to the standard function composition, here we first apply $\mathcal{T}_1$ and then $\mathcal{T}_2$.

[3] This counterexample was suggested by Margus Veanes and Loris D'Antoni in a discussion following POPL 2013.

```
int q = 0;
char d = '\0';
char c = '\0';
while (true) {
  c = in();
  if (q == 0)
    q = (c == '<' ? 1 : 0);
  else if (q == 1)
    q = (c == '<' ? 1 : 2);
  else {
    if (c == '>') {
      out('<'); out(d); out('>');
    }
    q = (c == '<' ? 1 : 0);
  }
  d = c;
}
```



$\lambda_0 \neq '<' / \epsilon$
$\lambda_0 = '<' / \epsilon$
$\lambda_0 = '<' / \epsilon$
$\lambda_0 \neq '<' / \epsilon$
$\lambda_0 = '<' / \epsilon$
$\lambda_0 = '>' / '<' \lambda_{-1} ' >'$
$\lambda_0 \neq '<' \wedge \lambda_0 \neq '>' / \epsilon$

Figure 10.1: An example program (`GetTags` from [BGMV12]) and the associated 1-SLT. The program extracts from an input stream all sequences of characters of the form $' <' \cdot c \cdot ' >'$, where $c \neq ' <'$.

Intuitively, if an SLT can output unboundedly long sequences of empty words (i.e., if there exist a sequence of transitions forming a cycle with satisfiable predicates and ϵ outputs) then such SLT cannot be composed with another SLT since ϵ-output-cycles can introduce arbitrary "gaps" in the output passed over for consumption to the SLT being composed with. However, in practice the composition of SLTs can be effectively computed in some common cases by the following recipe:

1. If SLTs $\mathcal{A}$ and $\mathcal{B}$ are both deterministic and $\mathcal{A}$ does not contain ϵ-output-cycles then their composition can be incrementally computed by a depth-first search similarly as the composition of symbolic finite-state transducers (SFTs) [VML11]. Specifically, given deterministic k-SLT $\mathcal{A}$ with no ϵ-output transitions and l-SLT $\mathcal{B}$, their composition will be a $(k+l)$-SLT $\mathcal{A} \circ \mathcal{B}$ such that $\mathcal{T}_{\mathcal{A} \circ \mathcal{B}}(\vec{a}) = \mathcal{T}_{\mathcal{A}} \circ \mathcal{T}_{\mathcal{B}}$.

2. If both SLTs can be translated to (possibly non-deterministic) SFTs then we can directly employ the composition algorithm from [BGMV12]. For instance, any 0-SLT is obviously an SFT, however, in some cases a k-SLT (with $k > 0$) can also be translated to an SFT (e.g., 1-SLT shown in Fig. 10.1 is equivalent to a 4-state non-deterministic SFT from Example 3 in [BGMV12]).

3. As the final option, we translate SLTs to symbolic transducers with registers [BGMV12], which can be trivially composed.

10.2.2 Equivalence checking

Equivalence checking of arbitrary non-deterministic SLTs is undecidable because it is already undecidable to check equivalence of concrete non-deterministic ϵ-free finite-state transducers [Iba77]. The principal cause for the undecidability in this general case is the possibility of unboundedly many different outputs in runs of a transducer on a given input. To combat the undecidability of equivalence checking we have to restrict the allowed class of transducers. For instance, Demers et al. [DKR82] and Bjørner et al. [BGMV12] have shown how to decide the equivalence of transducers (symbolic transducers, resp.) when they are single-valued.

While the technique of Bjørner et al. [BGMV12] could be adapted to SLTs to decide the single-valued case, that would not suffice for Σ^* as over-approximations of programs (§11.1) need not to be bounded-valued. Fortunately, Σ^*'s learning conjectures are always deterministic, so in fact we only need to check equivalence between a deterministic and a (possibly non-single-valued) non-deterministic SLT.

The key insight to our algorithm uses the fact that to check whether a deterministic and non-deterministic SLT are equivalent, it suffices to check whether their domains are equal and their union (which is a non-deterministic SLT) is single-valued.

Lemma 10.6. A deterministic SLT $\mathcal{A}$ and non-deterministic SLT $\mathcal{B}$ are equivalent iff $\mathrm{dom}(\mathcal{A}) = \mathrm{dom}(\mathcal{B})$ and $\mathcal{A} \cup \mathcal{B}$[4] is single-valued.

Checking domain equivalence of SLTs can be done similarly as for symbolic transducers ([BGMV12]), by checking equivalence of the underlying symbolic automata defined by predicate labels. We do not discuss the details of this construction as SLTs learned by Σ^* are total (see §10.2.4) in which case the domain check becomes trivial.

We next present a constructive algorithm for deciding single-valuedness of SLTs, which is a symbolic adaptation of the algorithm of Demers et al. [DKR82] for deciding single-valuedness of finite-state transducers.

Given a k-SLT $\mathcal{A} = (Q, q_0, F, \Delta)$, we define a linear context-free grammar[5] $\mathcal{G}_\mathcal{A} = \big(Q \times Q, (BExp[\mathsf{V}^k_\lambda] \times Exp[\mathsf{V}^k_\lambda]^*) \cup \{\bullet\}, P, [q_0, q_0]\big)$, where $\bullet \notin BExp[\mathsf{V}^k_\lambda] \times Exp[\mathsf{V}^k_\lambda]^*$ and the set of production rules P is given by

$$\begin{array}{rcl} [q_1, q_1'] & \to & (\varphi/\mathbf{t})[q_2, q_2'](\varphi'/\mathbf{t}') \quad \text{for all } q_1 \xrightarrow{\varphi/\mathbf{t}} q_2, q_1' \xrightarrow{\varphi'/\mathbf{t}'} q_2' \in \Delta \\ & & \text{such that } \varphi \wedge \varphi' \text{ is satisfiable, } \mathbf{t} \neq \bot \text{ and } \mathbf{t}' \neq \bot; \\ [q, q'] & \to & \bullet \quad \text{for all } q, q' \in F. \end{array}$$

By $\bot$, our learning algorithm denotes outputs on transitions that are either infeasible because of the constraints on the path condition, or subsumed by other predicates.

Theorem 10.7. Assuming decidability of the satisfiability of $BExp[\mathsf{V}^k_\lambda]$ and the quantifier-free theory of $Exp[\mathsf{V}^k_\lambda]$ with equality, it is decidable whether a non-deterministic SLT is single-valued.

[4]The union $\mathcal{A} \cup \mathcal{B}$ of $\mathcal{A} = (Q^A, q_0^A, F^A, \Delta^A)$ and $\mathcal{B} = (Q^B, q_0^B, F^B, \Delta^B)$ is defined by a product construction, as the least fixed point (which can be effectively computed by a depth-first search procedure) of pairs of states $Q \subseteq Q^A \times Q^B$ and transition relation under the following rules:

- $(q_0^A, q_0^B) \in Q$ is the initial state;
- if $(q_A, q_B) \in Q$, $q_A \xrightarrow{\varphi_A/\mathbf{t}_A} q_A'$, $q_B \xrightarrow{\varphi_B/\mathbf{t}_B} q_B'$, and $\varphi_A \vee \varphi_B$ is satisfiable then:
  - $(q_A', q_B') \in Q$
  - $(q_A, q_B) \xrightarrow{\varphi_A/\mathbf{t}_A} (q_A', q_B')$ and $(q_A, q_B) \xrightarrow{\varphi_B/\mathbf{t}_B} (q_A', q_B')$

[5]A grammar (N, T, P, s) with a finite set of non-terminals (resp. terminals) N (resp. T), a finite set of productions P, and a start symbol $s \in N$ is linear context-free if its set of productions is of the form $N ::= TNT \mid T$.

Proof. From Def. 10.4 a non-deterministic SLT $\mathcal{A} = (Q, q_0, F, \Delta)$ is single-valued iff for each $\vec{a} \in \mathbb{Z}^*$, $\vec{o}, \vec{o}' \in \mathcal{T}_\mathcal{A}(\vec{a})$ implies $\vec{o} = \vec{o}'$. However, $\vec{o}, \vec{o}' \in \mathcal{T}_\mathcal{A}(\vec{a})$ iff there are sequences of transitions $q_0 \xrightarrow{\varphi_1/\mathbf{t}_1} q_1 \xrightarrow{\varphi_2/\mathbf{t}_2} q_2 \cdots q_{n-1} \xrightarrow{\varphi_n/\mathbf{t}_n} q_n$ and $q_0 \xrightarrow{\varphi_1'/\mathbf{t}_1'} q_1' \xrightarrow{\varphi_2'/\mathbf{t}_2'} q_2' \cdots q_{n-1}' \xrightarrow{\varphi_n'/\mathbf{t}_n'} q_n'$ generating $\vec{o}$ (resp. $\vec{o}'$) on $\vec{a}$ such that $(\varphi_1/\mathbf{t}_1)\ldots(\varphi_n/\mathbf{t}_n) \bullet (\varphi_n'/\mathbf{t}_n')\ldots(\varphi_1/\mathbf{t}_1)'$ is in the language of $\mathcal{G}_\mathcal{A}$.

Thus, $\mathcal{A}$ is single-valued iff $(\varphi_1/\mathbf{t}_1)\ldots(\varphi_n/\mathbf{t}_n) \bullet (\varphi_n'/\mathbf{t}_n')\ldots(\varphi_1/\mathbf{t}_1)' \in L(\mathcal{G}_\mathcal{A})$ implies equality of the outputs, which is the case iff the grammar $\mathcal{G}_\mathcal{A}$ generates a set of palindromes. Checking whether the outputs match under the guards reduces to checking the validity of formula $(\bigwedge_i \varphi_i) \Rightarrow \mathbf{t}_1 = \mathbf{t}_2$, which can be done with $\mathcal{O}(|Q|^2)$ calls to the SMT solver implementing decision procedures for the satisfiability of $BExp[\mathsf{V}_\lambda^k]$ and the quantifier-free theory of $Exp[\mathsf{V}_\lambda^k]$. Checking whether a linear grammar generates a language of palindromes can be done in linear time by the result of Hopcroft [Hop69].[6] □

If $\mathcal{A}$ is not single valued then we can construct a witness for disequality between the deterministic and the non-deterministic SLT by finding the shortest path from the start symbol to the first reachable rule that does not generate a palindrome. This witness for disequality is called a *separating sequence*.

10.2.3 Extension with lookahead

SLTs can be straightforwardly extended to incorporate lookahead. In our model of SLTs, the sliding window begins at the current input tape head position and covers k last symbols. Such window can also be made to cover symbols ahead of the current head position by using additional λ-variables $\lambda_1, \ldots, \lambda_l$ such that λ_i stands for the symbol i positions ahead. A k-SLT with a lookahead l can be reduced to a $(k+l)$-SLT, so equivalence checking of SLTs with lookahead can be reduced to equivalence checking of SLTs.

10.2.4 SLTs learned by Σ^*

The definition of an SLT (Def. 10.1) uses a distinguished set of final states, as is customary in the definitions of other kinds of transducers. However, SLTs that are synthesised by Σ^*, either as over-approximations (§11.1) or symbolic conjectures (§11.2), have an additional property that all their states are final. This property stems from the specific way how the Angluin's model of learning is applied in Σ^*, where it would be impossible to learn SLTs with non-final states, as such transducers would allow inherent ambiguity in where the output is produced.[7]

SLTs synthesised during the Σ^* abstraction-refinement loop have an additional property that the disjunction of guards from every state of such SLTs is valid. Together with all states being final, this guarantees totality of the SLTs synthesised by Σ^*. Furthermore, SLTs conjectured by the Σ^* learning algorithm have mutually disjoint guards, which

[6]Hopcroft's original algorithm [Hop69] decides whether an arbitrary context-free grammar generates a set of palindromes and may require exponential time. However, linear context-free grammars are of special form for which the algorithm can be straightforwardly adjusted to run in linear time.

[7]Existing algorithms for learning concrete transducers (e.g., [SG09, Vil96]) also require all states to be final.

ensures that they will always be deterministic. All these properties together allow us to reduce equivalence checking between conjectures and over-approximations to deciding single-valuedness as shown in §10.2.2.

CHAPTER 11

SYNTHESIS OF SLTS WITH Σ^*

This chapter formally describes Σ^*—the technique central to Part II—that synthesises symbolic input-output specifications of program behaviour. Σ^* uses dynamic symbolic execution [GKS05, SMA05, CE05] to discover symbolic input-output steps of the program, and counterexample-guided abstraction refinement [CGJ$^+$00, BMMR01] to over-approximate program behaviour, and combines the two techniques into a sound and complete (relative to abstraction) symbolic learning algorithm. The algorithm represents specifications using symbolic lookahead transducers (SLTs), which we presented in §10.

The core algorithmic idea behind Σ^* is to extend Angluin's L^* learning algorithm so to use SLTs as symbolic conjectures. However, answering equivalence queries between SLTs and programs is in general impossible. We therefore introduce a class of programs for which we can compute over-approximations in the form of (possibly non-deterministic) SLTs. Then we can use the equivalence checking algorithm from §10.2.2 to decide the equivalence between symbolic conjectures and over-approximations.

The rest of this chapter is structured as follows. In §11.1 we present our approach for synthesis of over-approximations in form of SLTs. Then in §11.2 we describe the learning components of Σ^* algorithm. We finish the chapter by a concluding discussion in §11.3.

11.1 Synthesis of over-approximations

We now present how to synthesise sound program over-approximations for a class of programs with finite lookback. Overapproximations are computed in two steps. In the first step, we construct an over-approximation of the program that is based on predicate abstraction [GS97, BPR01]. In the second step, we transform the obtained abstraction to an equivalent non-deterministic SLT. We refine such an SLT by augmenting the set of predicates in predicate abstraction.

11.1.1 Abstraction of transductive programs

The goal of the abstraction is to abstract away the internal computation of the program but in doing so to keep all of its original input-output behaviour. Let us consider the LTS, $\mathcal{T} = (\mathsf{State}, C, \rightsquigarrow)$, of a transductive program $\mathcal{P}$ as defined in §9.1.2. The abstraction ove-approximates the valuation of program variables V with predicates, but maintains the original valuation of program variables that participate in the computation of output terms. We parameterise our abstraction by a set of predicates Φ over program variables, interpreted over $\mathsf{V} \rightharpoonup \mathbb{Z}$. Let us denote by $\mathsf{Pred}(\Phi)$ the set of boolean combinations over predicates from Φ. We define the abstraction of $\mathcal{T}$ as the LTS $\mathcal{T}^\sharp = (\mathsf{State}^\sharp, C, \rightsquigarrow^\sharp)$ constructed as follows. The set of abstract states $\mathsf{State}^\sharp$ is given by

$$\mathsf{State}^\sharp \triangleq \mathsf{N} \times \mathsf{Pred}(\Phi) \times (\mathsf{V_D} \rightharpoonup \mathit{Exp}[\mathsf{V_I}]) \times \mathsf{OutStr} \times \mathbb{N} \times \mathbb{N}$$

where the valuations of program variables are mapped to predicates in $\mathsf{Pred}(\Phi)$ satisfied by the valuation, while the data variables $\mathsf{V_D} \subseteq \mathsf{V}$ that are strongly live at any one of the $\mathsf{out}(\_)$ commands, are kept intact along with other components of the state. Using the approximate post operator on $\mathsf{Pred}(\Phi)$ computed with predicate abstraction we obtain the transition relation $\rightsquigarrow^\sharp$ on abstract states. We denote the components of an abstract state $s \in \mathsf{State}^\sharp$ by $s.\mathsf{n}$, $s.\sigma$, $s.\varphi$, $s.\rho$, $s.\imath$, and $s.\jmath$, respectively. We rely on the soundness of the predicate abstraction to obtain the following.

Proposition 11.1. For every input $\vec{a}$, if $\tau_{\mathcal{P}}(\vec{a}) = s_0 \ldots s_n$ is a trace in $\mathcal{T}$, then there exists a trace $\tau^\sharp_{\mathcal{P}}(\vec{a}) = s'_0 \ldots s'_n$ in $\mathcal{T}^\sharp$ such that for every $0 \leq i \leq n$, $s_i.\mathsf{n} = s'_i.\mathsf{n}$, $s_i.\rho = s'_i.\rho$ $s_i.\imath = s'_i.\imath$, $s_i.\jmath = s'_i.\jmath$, $s_i.\sigma|_{\mathsf{V_D}} = s'_i.\sigma$ and $s_i.\sigma$ satisfies $s'_i.\varphi$. Consequently, the output $\gamma^\sharp_{\mathcal{P}}(\vec{a})$ corresponding to the trace $\tau^\sharp_{\mathcal{P}}(\vec{a})$ is equal to $\gamma_{\mathcal{P}}(\vec{a})$.

11.1.2 Translation to SLTs

We now translate the abstract LTS $\mathcal{T}^\sharp = (\mathsf{State}^\sharp, C, \rightsquigarrow^\sharp)$ into an equivalent (possibly non-deterministic) SLT. As in §9.2.1, let us also denote by $\Lambda(s)$ the input-relativisation of an abstract state $s = (\mathsf{n}, \varphi, \sigma, \rho, \imath, \jmath) \in \mathsf{State}^\sharp$, defined analogously by $\Lambda(s) \triangleq (\mathsf{n}, \varphi[\vec{in} \overset{\imath}{\mapsto} \overleftarrow{\lambda}], \sigma[\vec{in} \overset{\imath}{\mapsto} \overleftarrow{\lambda}], \rho[\vec{in} \overset{\imath}{\mapsto} \overleftarrow{\lambda}], \jmath)$. We define an equivalence relation $\sim$ on $\mathsf{State}^\sharp$ as follows. For $s, s' \in \mathsf{State}^\sharp$, we let $s \sim s'$ iff for $\Lambda(s) = (n, \varphi_\lambda, \sigma_\lambda, \rho_\lambda, \jmath)$ and $\Lambda(s') = (n', \varphi'_\lambda, \sigma'_\lambda, \rho'_\lambda, \jmath')$ we have $n = n'$, $\varphi_\lambda \Leftrightarrow \varphi'_\lambda$ and $\sigma_\lambda = \sigma'_\lambda$. (Note that $\sim$ does not impose equality of *out*-indices since the intent is to identify pairs of states which produce equivalent output suffixes but with possibly different output prefixes.)

Relation $\sim$ can have infinitely many distinct equivalence classes in general, however, we focus on programs having finite-index[1] $\sim$, as we can represent such programs with SLTs and learn with Σ^*.

Definition 11.2. We say that $\mathcal{P}$ is SLT-*representable* if for any choice of Φ, the corresponding relation $\sim$ is of finite index.

SLT-representable programs necessarily have a bounded lookback.

Let us define $\mathsf{paths}^\sharp_{\mathsf{in}}(s, t)$ as the set of all $\rightsquigarrow^\sharp$-sequences of states between s and t such that there is a single input transition between states on the path from s to t. For $\xi \in \mathsf{paths}^\sharp_{\mathsf{in}}(s, t)$, let us denote by $\pi^\sharp(\xi)$ the conjunction of assumed predicates on transitions in ξ and let $\theta^\sharp(\xi)$ be the produced symbolic output. We need the following lemma for our translation to an SLT to be well-defined.

[1]An equivalence relation is said to be of *finite index* if the number of distinct equivalence classes is finite.

Lemma 11.3. If $s \sim s'$ and $t \sim t'$ then for every path $\xi \in \mathsf{paths}^{\sharp}_{\mathsf{in}}(s,t)$, there exists an equivalent path $\xi' \in \mathsf{paths}^{\sharp}_{\mathsf{in}}(s',t')$ such that $\pi^{\sharp}(\xi) = \pi^{\sharp}(\xi')$ and $\theta^{\sharp}(\xi) = \theta^{\sharp}(\xi')$.

We define $\mathcal{A}_{\Phi}$ to be the SLT (Q, q_0, Δ) with $Q \triangleq \{[s]_{\sim} \mid s \text{ is } \textit{in}\text{-state}\}$ as the set of states, $q_0 \triangleq [s_0]_{\sim}$ as the initial state and Δ as the transition relation such that $[s] \xrightarrow{\varphi/\mathbf{t}} [s'] \in \Delta$ iff there exist $\xi \in \mathsf{paths}^{\sharp}_{\mathsf{in}}(s,s')$, such that $\pi^{\sharp}(\xi) = \varphi$ and $\theta^{\sharp}(\xi) = \mathbf{t}$. Intuitively, $\mathcal{A}_{\Phi}$ represents all isomorphism classes of big-step transitions between the abstracted *in*-states.

Lemma 11.4. If $\mathcal{P}$ is SLT-representable then $\mathcal{A}_{\Phi}$ is an SLT.

We can now show that $\mathcal{A}_{\Phi}$ captures exactly the behaviour of $\mathcal{T}^{\sharp}$ thus $\mathcal{A}_{\Phi}$ soundly over-approximates the behaviour of $\mathcal{P}$.

Proposition 11.5. For all $\vec{a}$, $\gamma^{\sharp}_{\mathcal{P}}(\vec{a}) = \vec{o}$ iff $\vec{a}^{\perp} \twoheadrightarrow_{\mathcal{A}_{\Phi}} \vec{o}$.

Corollary 11.6. For all $\vec{a}$, if $\gamma_{\mathcal{P}}(\vec{a}) = \vec{o}$ then $\vec{a}^{\perp} \twoheadrightarrow_{\mathcal{A}_{\Phi}} \vec{o}$.

11.1.3 Refinement

Suppose that $\mathcal{A}_{\Phi}$ strictly over-approximates the behaviour of an SLT-representable $\mathcal{P}$ on some input $\vec{a}$, i.e., there exist $\vec{o}$ and $\vec{o}'$, $\vec{o} \neq \vec{o}'$, such that $\gamma_{\mathcal{P}}(\vec{a}) = \vec{o}$ and $\vec{a} \twoheadrightarrow_{\mathcal{A}_{\Phi}} \vec{o}'$. Then we need to refine the abstraction $\mathcal{A}_{\Phi}$. We do so by adding enough predicates to Φ to evidence infeasibility of the spurious run in $\mathcal{A}_{\Phi}$ that generates $\vec{o}'$.

Our approach to finding a new set of predicates for refining $\mathcal{A}_{\Phi}$ is based on standard counterexample feasibility analysis with weakest preconditions [Bal05]. The basic idea is to build a predicate α from the spurious trace in $\mathcal{A}_{\Phi}$ generating $\vec{o}'$ so that α is unsatisfiable if and only if the given trace is infeasible in $\mathcal{P}$. Starting with false, α is obtained by applying the weakest precondition of commands in $\mathcal{P}$ along the trace, until the beginning of the trace. To maintain the syntactic form of assume-predicates collected along the trace, we keep all substitutions in α explicitly. The set of all atomic predicates in α is then used to augment Φ.[2] Although more involved methods (e.g., based on Craig interpolation) are possible, our experiments showed that this heuristic works well in practice for our target classes of programs.

Completeness of the predicate selection method. Since our abstraction is based on predicate abstraction and fully precise isomorphic representation of other components of the state, $\mathcal{A}_{\Phi}$ in fact defines the strongest inductive invariant containing the reachable set of states of $\mathcal{P}$ that is expressible as a Boolean combination of the given set of predicates, while faithfully preserving the input-output relation in the relativised form. If there exists a quantifier-free inductive invariant ψ which can be built using some set of predicates Φ such that ψ uniquely identifies the reachable states of $\mathcal{P}$, then the construction of abstraction would converge. Assuming a complete decision procedure for the underlying theory and a predicate selection method that would eventually build such Φ, by the relative completeness of predicate abstraction [BPR02, JM06], we could generate an invariant as strong as ψ.

[2]In some examples we employ additional heuristic that uses templates built from syntactic predicates in the code.

Although it is not clear how to construct such ψ directly from $\mathcal{P}$, we can construct it if the number of states n of an SLT $\mathcal{A}_\mathcal{P}$ that is behaviourally equivalent to $\mathcal{P}$ is known a priori—by explicitly encoding a checking sequence [LY96] that distinguishes $\mathcal{A}_\mathcal{P}$ from all other transducers up to n states. To abstract away the complexity of such construction, we assume existence of a predicate selection method that eventually yields a set Φ resulting in an abstraction $\mathcal{A}_\Phi$ equivalent to $\mathcal{P}$. We will say that a predicate selection method is *complete* if it is guaranteed to eventually generate a sufficient set of predicates to construct $\mathcal{A}_\Phi$ equivalent to $\mathcal{P}$. Our main result expresses completeness of our learning algorithm relative to existence of such a complete predicate selection method. However, we emphasise that the predicate selection mechanism does not affect the soundness of Σ^*, nor the quality of inferred specifications, only the eventual convergence.

11.2 The Σ^* learning algorithm

In this section, we describe the Σ^* learning algorithm and prove its main properties. Our algorithm iteratively builds conjectures similarly as L^*. To make the presentation reasonably self-contained, we first give a brief overview of L^* (§11.2.1). We then define the representation of symbolic conjectures (§11.2.2), followed by the detailed presentation of the learning algorithm (§11.2.3) and its properties (§11.2.4).

11.2.1 Background: Overview of L^*

Here we give an informal overview of the classic L^*. See the paper of Angluin [Ang87] for a more detailed description (or Shahbaz and Groz [SG09] for the Mealy-machine version of L^*).

The goal of L^* is to learn an unknown regular language D by generating a deterministic finite automaton (DFA) that accepts D. L^* starts by asking a teacher, who knows D, membership queries over a known concrete alphabet to check whether certain words are in D. The results of these queries are recorded in a so-called *observation table*. Membership queries are iteratively asked until certain technical conditions are met, upon which L^* conjectures an automaton. L^* asks the teacher an equivalence query to check if the conjectured language is equivalent to D. The teacher either confirms the equivalence or returns a counterexample, which is a word that distinguishes D from the conjecture. L^* uses the counterexample to devise new membership queries, refining the conjecture. This procedure is repeated possibly forever until the conjecture becomes equivalent to D. If D is indeed a regular language then L^* is guaranteed to find a (minimal) DFA for D after at most a polynomial number of membership and equivalence queries.

An efficient approach to learning Mealy machines [SG09] can be achieved by modifying the concept of the membership and equivalence queries to include the output and the structure of the observation table to record output symbols instead of recording just boolean answers. In Σ^*, we answer the membership queries by using a combination of concrete and symbolic execution [GKS05, SMA05, CE05], and the equivalence queries by equivalence-checking learnt conjectures and increasingly refined abstractions, using the equivalence-checking algorithm for SLTs (§10.2.2).

11.2.2 Symbolic conjectures

We proceed by giving notational conveniences used in the rest of the section and defining the representation of symbolic conjectures used in Σ^*.

We first define a relativisation function for path predicates and symbolic outputs by $\Lambda(s_1 \dots s_n) \triangleq s_1[\vec{in} \overset{1}{\mapsto} \overleftarrow{\lambda}] \dots s_n[\vec{in} \overset{n}{\mapsto} \overleftarrow{\lambda}]$, where $\vec{in} \overset{i}{\mapsto} \overleftarrow{\lambda}$ is the variable substitution defined in §9.2.1 and $\vec{s}$ is either a path predicate or symbolic output. If $\vec{s}$ is a symbolic output $\vec{t}$, the same relativisation function is applied to each individual term in the subsequence, i.e., if $\mathbf{t} = t_1 \dots t_m$ then $\mathbf{t}[\vec{in} \overset{i}{\mapsto} \overleftarrow{\lambda}] = t_1[\vec{in} \overset{i}{\mapsto} \overleftarrow{\lambda}] \dots t_m[\vec{in} \overset{i}{\mapsto} \overleftarrow{\lambda}]$. We next define $\mathsf{witness}$ as a function that takes a sequence of relativised predicates, derelativises them by applying the inverse of the Λ substitution, computes a conjunction of derelativised predicates $\bigwedge_{1 \leq i \leq |\vec{p}|} \left(\Lambda^{-1}(\vec{p})\right)|_i$, passes the conjunction to an SMT solver, and returns a concrete sequence of input symbols of length $|\vec{p}|$ satisfying the conjunction, or $\bot$ if the conjunction is infeasible. Finally, we point out that all equality (resp. inequality) checks $=$ (resp. $\neq$) over predicates and terms in this section are syntactic equality (resp. inequality) checks.[3]

Σ^* constructs symbolic observation table similarly to L^*, but table entries are path predicates and symbolic outputs (§9.1.3), rather than concrete words. The finished table can be easily translated into an SLT, representing a conjecture. As in L^*, such conjecture will always be deterministic.

Definition 11.7. A *symbolic observation table* is a quadruple $(R, S, E, T) \subseteq (BExp[\mathsf{V}_\lambda]^*, BExp[\mathsf{V}_\lambda]^*, BExp[\mathsf{V}_\lambda]^*, BExp[\mathsf{V}_\lambda]^* \times BExp[\mathsf{V}_\lambda]^* \rightharpoonup (Exp[\mathsf{V}_\lambda] \cup \{\bot, \epsilon\})^*)$, where

- $R \subseteq S$ is a set of relativised path predicates representing the identified states,
- S (resp. E) is a prefix- (resp. suffix-) closed set of relativised path predicates, and
- T is a map indexed by $\vec{p}_\mathsf{p} \in S, \vec{p}_\mathsf{s} \in E$, containing the suffix of the relativised symbolic output generated when processing $\vec{p}_\mathsf{s}$ immediately after $\vec{p}_\mathsf{p}$, i.e., if $\vec{a} = \mathsf{witness}(\vec{p}_\mathsf{p} \cdot \vec{p}_\mathsf{s})$ and $\vec{t} = \Lambda(\theta_{\mathcal{P}}(\vec{a}))$ then $T[\vec{p}_\mathsf{p}, \vec{p}_\mathsf{s}] = \vec{t}_\mathsf{s}$, where $\vec{t}_\mathsf{s}$ is a suffix of $\vec{t}$ such that $|\vec{t}_\mathsf{s}| = |\vec{p}_\mathsf{s}|$. Outputs generated by infeasible transitions are denoted by $\bot$.

Intuitively, rows of the symbolic observation table are indexed by the elements of S and columns by the elements of E. The set S contains exactly the path predicates from R—the set of potential states of the conjecture—and additionally all the sequences that extend sequences from R by exactly one big-step transition. The role of S is to exercise all the transitions in the conjectured SLT. Finally, E is the set of suffixes that distinguish different states in R from each other.

For $\vec{p} \in S$, we denote by *$\vec{p}$-row* the set $\{T[\vec{p}, \vec{r}] \mid \vec{r} \in E\}$, i.e., the set of all outputs from the observation table associated with path predicates obtained by splicing $\vec{p}$ with all distinguishing suffixes from E.

The classic L^* makes a conjecture when the table is *closed*, which means that every sequence in S has a representative in R, i.e., $\forall \vec{p} \in S \,.\, \exists \vec{r} \in R \,.\, \vec{p}\text{-}row = \vec{r}\text{-}row$. We define closedness in the same way as L^*. From a closed table, we can easily construct a

[3]At the cost of more complex exposition, we could use semantic equality and check that output terms are equal under the guard restrictions. Such an approach might allow us to learn more compact SLTs.

complete (for all states and input symbols, all transitions are defined) SLT as follows (e.g., see [Ang87, SG09]):[4]

Definition 11.8. Let (R, S, E, T) be a closed observation table. Then the conjecture SLT $\mathcal{A}_C = (Q, q_0, F, \Delta)$ is constructed by letting:

- $Q = R$;
- $q_0 = \{\epsilon\}$;
- $F = R$;
- $\Delta = \{(\vec{p}, \vec{r}, \vec{p} \cdot \vec{r}, T[\vec{p}, \vec{r}]) \mid \vec{p} \in R, \vec{r} \in E, |\vec{r}| = 1\}$.

11.2.3 Symbolic learning

We begin by describing the FILLROWS algorithm that computes the missing entries of the observation table, continue with the EXTENDTABLE algorithm that explores the successor states of all states discovered at certain step, and end with the Σ^* learning algorithm.

Algorithm 7 computes the missing entries in the observation table. If the entry is missing for some prefix $\vec{p}_\mathsf{p} \in S$ and suffix $\vec{p}_\mathsf{s} \in E$, we first try to compute a concrete witness $\vec{a}$ by splicing together the prefix and the suffix (Line 2). While the sets S (and R) contain path predicates that are collected along prefixes of some feasible paths in $\mathcal{P}$ starting from the initial state, the set E contains suffixes of feasible paths. Naturally, when we arbitrarily splice prefixes and suffixes of different paths, the resulting formula might be infeasible. If feasible, we execute $\vec{a}$ on $\mathcal{P}$ using dynamic symbolic execution (Line 4) and collect the predicates ($\vec{r}$) and output terms ($\vec{t}$) from big-step transitions. Note that the collected predicates $\vec{r}$ might differ from $\vec{p}_\mathsf{p} \cdot \vec{p}_\mathsf{s}$, but at least the prefix (corresponding to $\vec{p}_\mathsf{p}$) will always match. The suffix will not match if at any position a path predicate exercised by $\vec{r}$ following $\vec{p}_\mathsf{p}$ is more specific than the corresponding predicate in $\vec{p}_\mathsf{s}$ (observation table is always extended so that added predicates cover guards of all big-step transition from a state and are mutually disjoint). The outputs corresponding to subsumed predicates and infeasible path conditions are marked $\perp$. Lines 6–10 replace the output terms at positions where the $\vec{p}_\mathsf{p} \cdot \vec{p}_\mathsf{s}$ and $\vec{r}$ sequences differ syntactically. Finally, lines 20–21 close the table.

Algorithm 8 takes a state representative $\vec{r}$, i.e., a path predicate that holds on the shortest path to the identified state, and finds all the outgoing big-step transitions from that state, adding the predicates from those transitions to E and the entire sequence ($\vec{r}$ extended by one transition) to S. The only interesting part of the algorithm is the discovery of new transitions and the corresponding predicates. In the first iteration, Line 6 extends the representative sequence $\vec{r}$ with predicate true, effectively allowing the solver to produce an arbitrary value for the last element of the concrete input sequence. Executing the obtained concrete sequence and collecting predicates along the path, we

[4]In L^*, one also defines the *consistency* property, which would in our setting say that if two sequences $\vec{p}_1, \vec{p}_2$ from R are equivalent, then both states reached by $\gamma_\mathcal{P}(\mathsf{witness}(\vec{p}_1))$ and $\gamma_\mathcal{P}(\mathsf{witness}(\vec{p}_2))$ must produce the same output in the next big-step transition given the same input symbol. We maintain the consistency of our symbolic observation table by always assuring that each state has only one representative in the R set.

Algorithm 7: The FILLROWS Algorithm.

```
input and output : Observation table $OT$
forall $\vec{p}_\mathsf{p} \in S, \vec{p}_\mathsf{s} \in E$ such that $T[\vec{p}_\mathsf{p}, \vec{p}_\mathsf{s}]$ is undefined do
    $\vec{a} := \mathsf{witness}(\vec{p}_\mathsf{p} \cdot \vec{p}_\mathsf{s})$
    if $\vec{a} \neq \bot$ then
        $(\vec{r}, \vec{t}) := (\Lambda(\pi_{\mathcal{P}}(\vec{a})), \Lambda(\theta_{\mathcal{P}}(\vec{a})))$
        // assert($|\vec{r}| = |\vec{t}|$)
        $\vec{t}_\mathsf{s} := \epsilon$
        forall $1 \leq i \leq |\vec{t}|$ do
            if $i > |\vec{p}_\mathsf{p}|$ then
                if $(\vec{p}_\mathsf{p} \cdot \vec{p}_\mathsf{s})|_i = \vec{r}|_i$ then $\vec{t}_\mathsf{s} := \vec{t}_\mathsf{s} \cdot \vec{t}|_i$

                else $\vec{t}_\mathsf{s} := \vec{t}_\mathsf{s} \cdot \bot$

            else
                // assert($\vec{p}_\mathsf{p}|_i = \vec{r}|_i$)
            end
        end
        $T[\vec{p}_\mathsf{p}, \vec{p}_\mathsf{s}] := \vec{t}_\mathsf{s}$
    else
        $T[\vec{p}_\mathsf{p}, \vec{p}_\mathsf{s}] := \bot$
    end
end
// Close the table
forall $\vec{p} \in S$ s.t. $\neg\exists \vec{r} \in R$ . $\vec{p}$-row $= \vec{r}$-row do
    $R := R \cup \vec{p}$ // New state
end
Return $OT$
```

identify the first big-step transition guard predicate (r_s). In every following iteration, we negate a disjunction of the predicates discovered so far, until the disjunction becomes valid (test at Line 4). All infeasible traces lead to a ghost state that has one self-loop transition labelled $\mathsf{true}/\bot$. The algorithm creates such a state automatically, if needed, by filling the corresponding row with $\bot$, as even the prefix to the ghost state is infeasible.

Finally, Algorithm 9 infers a symbolic transducer. Lines 1–10 discover all the big-step transitions from the initial state and the corresponding predicates. All the discovered predicates are added to the set E. In the next three lines, we extend and close the table, producing the first $\mathcal{A}_C$ conjecture and the first abstraction $\mathcal{A}_\Phi$ of $\mathcal{P}$. The loop beginning on Line 16 checks the equivalence between the conjecture and abstraction. If they are equivalent, the algorithm terminates returning the exact transducer implemented by $\mathcal{P}$. Otherwise, the counterexample is checked against $\mathcal{P}$. If it is spurious, we refine the abstraction, otherwise, we refine the conjecture. We use Shahbaz and Groz's [SG09] technique for processing the counterexamples adapted for our symbolic setting. First, we collect the predicates from $\mathcal{P}$ along the path determined by the counterexample and

Algorithm 8: The EXTENDTABLE Algorithm.

```
input and output : Observation table $OT$
forall $\vec{r} \in R$ do // Extend all sequences from $R$
    if $\neg\exists \vec{p}_s \in E \,.\, |\vec{p}_s| = 1 \wedge \vec{r} \cdot \vec{p}_s \in S$ then
        $s :=$ false
        while $s \not\Leftrightarrow$ true do
            $r_s := \epsilon$
            $\vec{a} :=$ witness$(\vec{r} \cdot \neg s)$
            if $a = \bot$ then
                $r_s := \neg s$
                $s :=$ true
            else
                $\vec{p} := \Lambda(\pi_{\mathcal{P}}(\vec{a}))$
                // assert$(|\vec{p}| = |\vec{r}| + 1)$
                $r_s := \vec{p}|_{|\vec{r}|+1}$
                $s := s \vee r_s$
            end
            $E := E \cup \{r_s\}$
            $S := S \cup \{\vec{r} \cdot r_s\}$
            $OT :=$FILLROWS$(OT)$
        end
    end
end
Return $OT$
```

discard the longest prefix that is already in S. We denote the remaining suffix by $\vec{p}_s$. We add all suffixes of $\vec{p}_s$ to E (Line 26), to assure that E remains suffix closed.

11.2.4 Correctness and relative completeness

First, we state the main properties of Σ^*, and then proceed with the proof of relative completeness and a discussion of computational complexity.

Lemma 11.9. Let $T = (R, S, E, T)$ be a symbolic observation table. Then Σ^* preserves the following invariants:

1. R and S (resp. E) are always prefix- (resp. suffix-) closed.
2. For every $\vec{p} \in S$ there is a unique $\vec{r} \in R$ such that $\vec{p}$-*row* $= \vec{r}$-*row*.
3. For every $\vec{r} \in R$, there are $r_s^1, \ldots, r_s^n$ such that $\bigvee_{i=1}^{n} r_s^i \Leftrightarrow$ true and for all i, it holds that $\vec{r} \cdot r_s^i \in S$.
4. The conjecture $\mathcal{A}_C$ is closed at the end of each step.

Proof. (1) and (2) follow from Lines 15–16 in EXTENDTABLE, and from the initial state and Lines 8 and 26 of Σ^*. Closedness follows from Lines 20–21 of FILLROWS. □

Correctness. R represents the part of the symbolic execution tree of $\mathcal{P}$ on which the conjecture faithfully represents the behaviour of $\mathcal{P}$, which is stated with the following proposition.

Proposition 11.10 (Bounded correctness). After each step, for all $\vec{r} \in R$ and $\vec{a}$ such that $\vec{a} = \mathsf{witness}(\vec{r})$, $\gamma_{\mathcal{P}}(\vec{a}) = \gamma_{\mathcal{A}_C}(\vec{a})$ holds.

One can rephrase bounded correctness as a guarantee given by Σ^* in case Σ^* is interrupted before reaching a convergence. In other words, we can consider Σ^* as performing bounded model checking of the program with a state-space exploration strategy guided by the learnt specification.

Completeness. The following lemma ensures that a progress is made after each conjecture refinement.

Lemma 11.11. If at some step of Σ^*, $(\vec{a}, \vec{o})$ is a separating sequence such that $\gamma_{\mathcal{P}}(\vec{a}) = \vec{o}$, then at the end of the step, for all $\vec{a}'$ such that $\vec{a}' = \mathsf{witness}(\pi_{\mathcal{P}}(\vec{a}))$, $\gamma_{\mathcal{A}_C}(\vec{a}') = \gamma_{\mathcal{P}}(\vec{a}')$ holds.

We state the completeness of Σ^* relative to the completeness of the predicate selection method.

Theorem 11.12 (Relative completeness). If $\mathcal{P}$ is SLT-representable and the predicate selection method for refinement is complete, then Σ^* terminates with $\mathcal{A}_C$ being behaviourally equivalent to $\mathcal{P}$.

Proof. First note that when $\mathcal{A}_\Phi$ is single-valued, then by the soundness of abstraction, $\mathcal{A}_\Phi$ is in fact behaviourally equivalent to $\mathcal{P}$. As the predicate selection method is assumed to be complete, we will eventually obtain a single-valued $\mathcal{A}_\Phi$.

The equivalence check of $\mathcal{A}_\Phi$ and $\mathcal{A}_C$ always returns the shortest separating sequence (if one exists). There can be only finitely many shortest separating sequences of a given length, and at each step such a sequence is used either to refine the conjecture or to refine the abstraction. Therefore, the number of conjecture refinements must also be finite. □

11.2.5 Computational complexity

Next, we analyse the complexity of Σ^*. Let n be the number of states of the inferred SLT, k the maximal number of outgoing big-step transitions from any state, m the maximal length of any counterexample, and c the number of counterexamples. There can be at most $n - 1$ counterexamples, as each counterexample distinguishes at least one state. Since we initialise E with k predicates, $|E|$ can grow to at most $k + m(n-1)$. The size of S is at most $n \cdot k$. Thus, the table can contain at most $n \cdot k \cdot (k + m \cdot n - m) = \mathcal{O}(n \cdot k^2 + m \cdot n^2 \cdot k)$ entries. The k factor is likely to be small in practice, and our equivalence checking algorithm finds the minimal counterexample. The SMT solver is called once per each state (i.e., representative in R) and for each outgoing transition. For each equivalence check, we might need to call the solver n^2 times. Thus, the total worst-case number of calls to the solver is $\mathcal{O}(n \cdot k + n^2)$, not including the number of calls required for abstraction refinement, which depends on the abstraction technique used.

```
void foo(int a)
{
    int i, c;
    i = 0;
    c = 0;
    while (i < 1000) {
        c = c + i;
        i = i + 1;
    }
    assume(a <= 0);
    error();
}
```

```
while (true)
{
    i := 0;
    c := 0;
    while (i < 1000)
    {
        c := c + i;
        i := i + 1;
        x := in();
        out(x);
    }
}
```

Figure 11.1: Left: the original example from Figure 1 in [GHK+06], on which predicate abstraction does not work well. Right: the example adapted to a transductive program by adding reading an input and writing some output in the loop, and enclosing it with an outer `while(true)`-loop.

11.3 Discussion

We conclude the chapter by briefly discussing use of predicate abstraction, SLT limitations and inference of inexact specifications.

11.3.1 Predicate abstraction

To obtain an exact SLT specification of a program, Σ^* relies on convergence of the abstraction. One could rightfully ask why we need the whole learning machinery of Σ^* as we presume the abstraction eventually becoming behaviourally equivalent (although, in a possibly non-deterministic form) to the input-output specification. Our program is deterministic, so as long as the abstraction is not single-valued it must contain infeasible behaviour. Therefore, we could consider a procedure that just iteratively refines the abstraction until the abstraction becomes single-valued.

Setting aside how to efficiently refine abstractions alone, an abstraction-only approach suffers from a fundamental limitation: SLTs generated by the abstraction can be unboundedly larger than the ones learnt by Σ^*. For instance, consider the program from Figure 11.1, which is considered as an example on which predicate abstraction does not work well [GHK+06]. Then Σ^* infers a single-state SLT, while by using predicate abstraction alone we would obtain an SLT with 1000 states.

11.3.2 SLT limitations

Along with the dependency on the abstraction, ability of Σ^* to infer complete input-output specifications of program behaviour is limited by the expressive power of symbolic conjectures. We have judiciously restricted SLTs so as to be able to represent interesting real-world examples while still being able to learn and equivalence-check them efficiently. However, there are still other interesting classes of programs that are not SLT-representable. At the cost of additional complexity, we believe it would be possible to extend Σ^* to work with symbolic versions of more expressive transducers such as subsequential [Vil96] and streaming transducers [AC11] and in this way enable learning of larger classes of programs.

11.3.3 Inexact specifications

While the main concern of this work is learning of exact input-output specifications, inexact specifications inferred by Σ^* before reaching a convergence could also be useful, e.g., for automated testing based on dynamic symbolic execution [GKS05, SMA05, CE05]. The learning component of Σ^* can be seen as systematically exploring all paths in the execution tree of a program, but in addition conjecturing the behaviour of not yet explored branches. Conjectures could be used to guide the exploration strategy, similarly as in MACE [CBP+11], which showed how concrete transducer conjectures enable more effective testing of protocol implementations.

Algorithm 9: The Σ^*'s Learning Algorithm.

```
init   : $R = \{\epsilon\}, S = \{\epsilon\}, E = \emptyset, T = \emptyset, i = 0, s = \mathsf{false}$
result : $k$-SLT $\mathcal{A}_C$
repeat // Fill 1st row
    if $s \Leftrightarrow \mathsf{false}$ then
        $a :=$ randomly generated array of type $\mathbb{Z}[1]$
    else
        $a := \mathsf{witness}(\neg s)$
    end
    $p := \Lambda\left(\pi_{\mathcal{P}}(a)|_1\right)$ // 1st pred. from path predicate
    $E := E \cup p$
    $T[\epsilon, p] := \Lambda\left(\theta_{\mathcal{P}}(a)|_1\right)$
    $s := s \vee p$
until $s \Leftrightarrow \mathsf{true}$
$(R, S, E, T) :=$ ExtendTable(FillRows($R, S, E, T$))
Generate $\mathcal{A}_C$ from $(R, S, E, T)$ as per Def. 11.8
Compute initial $\mathcal{A}_\Phi$ of $\mathcal{P}$
while true do
    Let $(\vec{a}, \vec{o})$ be a separating sequence between $\mathcal{A}_C$ and $\mathcal{A}_\Phi$
    if $\vec{a} = \epsilon$ then
        Return $\mathcal{A}_C$
    end
    if $\gamma_{\mathcal{P}}(\vec{a}) \neq \vec{o}$ then // Spurious counterexample?
        Refine $\mathcal{A}_\Phi$ of $\mathcal{P}$ on $(\vec{a}, \vec{o})$
    else
        $\vec{p} := \Lambda(\pi_{\mathcal{P}}(\vec{a}))$
        Let $\vec{p}_{\mathsf{p}}$ be the longest prefix of $\vec{p}$ s.t. $\vec{p}_{\mathsf{p}} \in S$
        Let $\vec{p}_{\mathsf{s}}$ be s.t. $\vec{p} = \vec{p}_{\mathsf{p}} \cdot \vec{p}_{\mathsf{s}}$
        $E := E \cup \mathit{Suffix}\,(\vec{p}_{\mathsf{s}})$
        $(R, S, E, T) :=$ ExtendTable(FillRows($R, S, E, T$))
        Generate $\mathcal{A}_C$ from $(R, S, E, T)$ as per Def. 11.8
    end
end
```

Chapter 12

Applications of Σ^*

The main application of Σ^* advocated in Part II of this dissertation is automated synthesis of program parallelisation, directed by input-output specifications. We described in §8.4 a simple idea for speculative parallel execution of such specifications which relies on the fact that SLT specifications inferred by Σ^* have a finite, a-priori-known number of states. In this short chapter we demonstrate that Σ^* could be useful not only for program parallelisation but also for other practical applications. In all, we consider three application domains: web-sanitiser verification (§12.2), stream filter optimisation (§12.3) and parallelisation (§12.4). For each application, we briefly explain the problem setting and report empirical results obtained using our prototype implementation of Σ^*.

12.1 Implementation

We built our implementation on top of KLEE [CDE08] (a symbolic virtual machine for the LLVM intermediate language) with STP [GD07] (an SMT solver for the theory of bit-vectors and arrays) and CPAchecker [BK11] (a framework for program analysis and verification, implementing lazy predicate abstraction). We patched KLEE to generate relativised symbolic traces and answer membership queries for the learning algorithm (§11.2), and used STP in the equivalence checking algorithm (§10.2.2). We used CPAchecker for counterexample-guided predicate abstraction refinement (§11.1). All experiments were run on a machine with an Intel Core2 Quad 2.8 GHz CPU and NVIDIA GeForce GTX 460 GPU (with seven 1.42 GHz multiprocessors, 48 cores each).

12.2 Web sanitisers verification

To make web applications more secure, sanitisation is used to remove possibly malicious elements from user input. Although small in numbers of lines of code, sanitisers in real-world applications are surprisingly difficult to implement correctly [BCF+08, HLM+11], often because low-level implementation tricks are obscuring the intended input-output behaviour. By automatically inferring input-output specifications of sanitisers, we can automatically check their properties such as commutativity, reversibility, and idempotence using tools such as Bek [HLM+11].

We evaluated Σ^* on the sanitisers from Google AutoEscape (GA) web sanitisation framework, the same benchmarks used by Hooimeijer et al. [HLM+11]. We dropped the

Benchmark	Learned	Int. bits	#States	#Trans.	Lookback	$\#\mathcal{A}_C$ refs.	$\#\mathcal{A}_\Phi$ refs.	$\lvert\Phi\rvert$
`EncodeHtml` [HLM+11]	✓	8	1	2	0	0	1	10
`GetTags` [BGMV12]	✓	48	3	7	1	4	5	6
`CleanseAttribute`	✓	40	2	4	1	0	3	14
`CleanseCss`	✓	40	1	2	0	0	2	16
`CssUrlEscape`	✓	40	1	11	0	0	1	12
`HtmlEscape`	✓	8	$L+1$	$7 \cdot L$	L	L	$L+3$	$L+11$
`JavascriptEscape`	✓	16	3	16	2	2	6	20
`JavascriptNumber`	✓	41	17	19	16	17	23	41
`JsonEscape`	✓	8	$L+1$	$11 \cdot L$	L	L	$L+3$	$L+12$
`PreEscape`	✓	8	$L+1$	$5 \cdot L$	L	L	$L+3$	$L+6$
`UrlQueryEscape`	✓	8	2	11	0	2	5	19
`XMLEscape`	✓	8	$L+1$	$7 \cdot L$	L	$L-1$	$L+2$	$L+11$
`PrefixLine`	–	8						
`SnippetEscape`	–	8						

Table 12.1: Experimental results. The benchmarks that Σ^* successfully learned are denoted by ✓. The last benchmark could be learned if Σ^* were extended to handle subsequential transducers. *Int. bits* is the size of the internal control and data state in bits, *#States* the number of states in the final SLT, *#Trans.* the number of transitions in the final SLT, *$\#\mathcal{A}_C$ refs.* the number of refinements of the conjecture, *$\#\mathcal{A}_\Phi$ refs.* the number of refinements of the abstraction, and $|\Phi|$ the size of the set of predicates in the final abstraction. In each of the four benchmarks whose rows contain symbolic expressions dependent on L, the particular L denotes the bound on the size of the buffer internally used by the benchmark. The value of L depends on the characteristics of the input language of the benchmark and is a priori fixed in our four benchmarks (e.g., for `HtmlEscape`, it is bounded by the maximal length of tag names, attribute names and special symbols).

`ValidateUrl` sanitiser, as it is effectively just a wrapper for calling other sanitisers. In addition to GA sanitisers, we added to our benchmark suite the sanitisers `EncodeHtml` [HLM+11] and `GetTags` [BGMV12].

We found out that 86% of GA sanitisers (12 out of 14) along with `EncodeHtml` and `GetTags` are SLT-representable and were successfully learned by Σ^* (Table 12.1). In comparison, the authors of [HLM+11] had to invest several hours of a human expert's time into manual synthesis of each transducer from the source code. Only two sanitisers could not be inferred by Σ^*. `PrefixLine` is effectively a 0-SLT, but it is implemented by calling the `memchr` function in guards, which could only be represented by nondeterminism, as `memchr`'s return value could be arbitrarily far ahead from the start of the input string. `SnippetEscape` generates output from a predefined set of symbols only at the end of the input, but it could be learned if Σ^* were extended to subsequential transducers by generalizing Vilar's algorithm [Vil96] to the symbolic setting.

12.3 Stream filter optimisation

Stream programs often consist of many interacting filters that have to be optimised against various goals (e.g., speed, latency, throughput, ...). Two vital filter optimisations are reordering and fusion. For instance, reordering might be profitable when it would place a filter that is selective (i.e., drops some data) before a costly filter that does some processing. However, we must ensure that the two filters are commutative in order for

Benchmark	Linear	Stateless
Saxpy kernel [UGT09]	+	+
Radix-2 complex FFT filter [KDR+02]	+	+
FIR filter [LTA03]	+	+
Zig-zag descrambling filter in MPEG-2 [DHRA06]	+	+
Clsfr in L3-Switch bridge [CLL+05]	−	+
FM demodulator in StreamIt [TKA02]	−	+
MPEG-2 internal parser in StreamIt [TKA02]	−	−
GSM encoder/decoder in MiBench [GRE+01]	−	−

Table 12.2: Properties of eight filters inferred by Σ^*. The first four filters are linear and can be optimised with existing techniques. Σ^* enables fusion and reordering optimisations for all eight filters. By stateless, we mean single-state transducers.

reordering to be safe. Filter fusion is used when it would be profitable to trade pipeline parallelism for lower communication cost.

We conducted a set of experiments to explore the ability of Σ^* to enable stream filter optimisations such as reordering and fusion. Table 12.2 shows eight filters inferred by Σ^* indicating their two key properties: whether they encode a linear or non-linear output function, and whether they are stateless (i.e., have a single state) or stateful (i.e., have more then one state). First four filters can be optimised with existing techniques for optimizing linear filters [LTA03, ATA05]. To our knowledge, no optimisation techniques used in existing stream program compilers can optimise the remaining filters in Table 12.2. On the other hand, Σ^* enables fusion and reordering optimisations for all eight filters.

12.4 Parallelisation

Finally, we report a brief experiment on how SLT specifications of filters may enable automated parallelisation. We considered two approaches to improving utilisation of hardware concurrency by translating filters (1) to a vectorised implementation (available in the CPU) and (2) to execute on multiple cores (available in the CPU and GPU). Stateless filters are inherently data-parallel and lend themselves naturally to multi-core implementations by simple replication and to vectorised implementation using vector instructions (similarly as proposed in the approach of Veanes et al. [VMLL11]). Stateful filters normally cannot be parallelised this way as their next invocation depends on the previous invocation. However, SLT filters have usually small, a priori known, number of states, so we can run each replica of an SLT filter from each state in parallel and merge suitable outputs, as we described in §8.4.

To provide some insight into possible practical speedups, we generated SIMD implementations using intrinsics for Intel SSE4 vector instructions and GPU implementations using NVIDIA CUDA 4.2 of 11 single-state SLTs (3 from Table 12.1, 6 from Table 12.2, and 2 hand-crafted) and one three-state SLT inferred by Σ^*. Original source code representations of all SLTs are stateful and some of them contain non-affine loop nests. For the stateless SLTs, we obtained speedups in the range of 1.72 to 3.89 for the SSE versions, and 1.5 to 2.7 for the CUDA versions. For the 3-state SLT, the speedup of the CUDA version was 2.3. CUDA versions were run using 1024 threads (resp. 1024 threads in 3 blocks) for the single-state (resp. three-state) SLTs. We established the baseline by running the sequential version of SLTs compiled with `gcc` with all optimisations (`-O3`) and SSE flags turned on.

We expect that it would be possible to build other parallelisation backends for SLTs, such as for instance based on field-programmable gate arrays (see e.g. [HKM+08, HWBR09]) or based on using more elaborate speculative techniques (e.g. [PRV10]).

CHAPTER 13

RELATED WORK

In this chapter, we position our work with respect to the most relevant existing literature. After discussing symbolic transducers and their learning algorithms, we proceed with a discussion of connections with predicate abstraction, and conclude by briefly contrasting Σ^* approach to stream compiler optimisation and automated parallelisation to other related approaches.

Symbolic transducers. Symbolic finite-state transducers have become a popular topic lately [HLM$^+$11, BV11, BGMV12, AC11]. Van Noord and Gerdemann [vNG01] introduced the first simple symbolic transducers. The expressivity of those transducers is severely limited as they are not able to establish functional dependencies from input to output. In a series of papers [HLM$^+$11, BV11, BGMV12], a group of authors proposed symbolic finite-state transducers (SFTs) with registers (SFTRs), where input (resp. output) symbols range over predicates (resp. terms). Bjørner and Veanes [BV11] show an effective composition construction and prove that equivalence is decidable for single-valued SFTRs with simple updates, as long as the underlying theory is decidable. Unfortunately, their proof is not constructive and does not reveal how to efficiently compute a separating sequence. Also, k-SLTs are less expressive than SFTRs. We carefully limited the expressivity of SFTRs (to obtain k-SLTs) so as to be able to represent interesting real-world examples and constructively equivalence check learned k-SLTs.

One feature of SFTRs which makes them difficult to learn is that the set of final states of an SFTR can be a strict subset of all states. Transducers with some non-final states are partial functions that accumulate the output and yield it only when the last reached state is final. Furthermore, the existence of non-final states allows ambiguity of where the output is created. Thus, it is not surprising that the existing algorithms for learning concrete transducers [SG09, OGV93, Vil96] require all states to be final, as we do.

Alur et al. proposed streaming transducers [AC11], which have two types of registers—one for symbols and the other for strings. At every step, a streaming transducer can make decisions based on the current state, the tag (ranging over a finite set of values) of the next input symbol, and the ordering between next input symbol and symbols stored in data registers. Each value stored in string registers can be either copied or concatenated with a symbol, but has to be used only once. Alur et al. showed an intricate proof of decidability of equivalence of streaming transducers. Inference of both SFT(R)s [BGMV12, BV11] and streaming transducers [AC11] has been left as an open question.

Learning of symbolic transducers. Previous work on learning some variants of symbolic transducers [HSJC12, BJR08, AJU10] focused on limited variations that have simple guards which a priori fix the shape of predicates, and can either update registers or output a sequence of concrete values in every transition. Learning algorithms for these transducers postulate existence of an equivalence-checking oracle and work over concrete alphabets, relying on the shape restrictions to translate the L^*-inferred concrete transducer to a symbolic one. These algorithms are exponential in *both* the length of counterexamples and the number of parameters the guard predicates can have, as well as in the number of registers. As software artefacts feature predicates that are unknown prior to the analysis, if such algorithms were to be applied in the software setting, they would need to exhaustively enumerate predicates, whose set is unbounded. In contrast, by allowing Σ^* to inspect the internal symbolic state of the program, we can handle arbitrarily complex guard and output expressions, as long as (1) the program's input-output relation can be described using finitely many such expressions, and (2) there exists a solver capable of checking satisfiability of guards and (in)equality of output expressions.

There is another important point of difference. Instead of just assuming existence of a suitable equivalence-checking oracle, like the prior work [HSJC12, BJR08, AJU10], we propose an efficient algorithm for checking the learned conjectures against a synthesized abstraction. Most importantly, the alternation between learning and abstraction allows us to detect convergence, assuming an abstraction that can infer the strongest invariant of program's transition relation.

Learning in verification. Techniques for learning various types of automata have been applied in a wide variety of verification settings: inference of interface invariants [ACMN05, SGP10, GRR12], learning-guided concolic execution [CBP$^+$11], compositional verification [CGP03], and regular model checking [HV05]. In those settings, except for [GRR12], learning is performed using concrete alphabets in a black-box manner, even though source code is usually available. Independently of us, Giannakopoulou et al. [GRR12] have developed a technique for learning component interfaces that combines L^* with symbolic execution to discover method input guards. However, their approach treats method invocations as a single input step with no output, and, unlike Σ^*, assumes that each method has a finite number of paths. Under this later constraint, equivalence checking can be done by merely executing a sufficiently large number of membership queries. Alur and Černý's approach to synthesis of interface specifications with JIST [ACMN05] also uses predicate abstraction. However, predicate abstraction in JIST is used to check whether the synthesised interfaces are safe for a given safety property. In general, such interfaces are approximate and may incorporate infeasible behaviour. In addition, the paper [ACMN05] does not demonstrate that the technique can be fully automated. In contrast, Σ^* is fully automated and infers exact models upon termination.

Predicate abstraction and property checking. Σ^* uses predicate abstraction [GS97, BPR01] to generate a finite-state model of the unbounded internal state space of the program. Predicate abstraction has been successfully applied in counterexample-guided abstraction refinement schemes for checking safety properties. Tools like SLAM [BMMR01], BLAST [HJMS02], IMPACT [McM06], and SATABS [CKSY05] have demonstrated that predicate abstraction scales well to real-world programs with intricate control-flow patterns. Compared to classical predicate abstraction, our goal is different—we infer the exact input-output specification of a program so that we can generate optimised code.

Although the goal of Σ^* is different, it resembles some ideas used in property checking. Most closely related is the idea of collecting predicates on conditional statements as in automated testing [GKS05, SMA05, CE05], which has been previously combined with predicate abstraction in the SYNERGY algorithm [GHK+06] and its descendants [BNRS08, GNRT10]. While SYNERGY uses a combination of *must* and *may* analyses to check safety properties faster, Σ^* uses *must* (learning) and *may* (abstraction) to infer a more compact input-output specification. Similarly as safety properties could be checked with *may* analysis only, input-output specifications could be learned with our predicate abstraction (Section 11.1.2) only, by refining abstractions until the relation becomes single-valued. However, such relations can be unboundedly larger than the ones learned by Σ^*, as we showed in §11.3.1.

The completeness of predicate abstraction depends on the choice of predicates for abstracting the state space. Completeness of predicate abstraction is an active research area [JM06, BPR02] and a number of heuristics exist whose incompleteness does not manifest itself in practice [AKD+08, FQ02]. Assuming a predicate selection oracle that eventually yields enough predicates for constructing the strongest invariant of the program's transition relation, one can show relative completeness of predicate abstraction [BPR02, JM06]. We found that for the examples used in our study, a simple oracle which mines predicates from sequences of predicates and output terms obtained from the dynamic symbolic execution of the program was sufficient.

Optimizations of filters. The closest work to our application of Σ^* for stream compiler optimisations is on optimisations of linear filters [LTA03] and (slightly more general) linear state space systems [ATA05] in StreamIt. In another line of work [LDWL06], fusion of filters for Brook has been developed using an affine model. Soulé et al. [SHG+10] developed a proof calculus that allows analysis of non-linear filters, but they can perform reordering optimisations only if filters are stateless. In contrast, Σ^* allows fusion and reordering of stateful filters with non-linear input-output relations. Approaches for numerical static analysis of linear filters (e.g., [Fer04, Mon05]) do not relate directly to ours as their goal is to obtain deep numerical properties of filters (such as numerical bounds) for safety-critical applications.

Automated parallelisation. Thies et al. [TCA07] extract pipeline partitions from legacy streaming applications written in C by relying on programmer-provided annotations indicating pipeline boundaries. In contrast, our approach does not require any annotations. For synchronous dataflow streaming applications, parallelism has been exploited either by replicating stateless filters [BFH+04, GTA06, KM08], or by using affine partitioning of loop nests [LDWL06]. In contrast, Σ^* enables parallelisation of non-linear stateful filters with non-affine loop bounds. There is a massive amount of work on general cross-loop iteration dependence scheduling (e.g., see [BVB+09]). Σ^* is aimed at complementing such advanced techniques by enabling naïve parallelisation of behaviourally simple loop nests that are not necessary polyhedral, disjoint or with a clear pipeline structure.

CHAPTER 14

CONCLUSION

14.1 Summary

This dissertation has presented two formal verification-based methods for automated synthesis of program parallelisation:

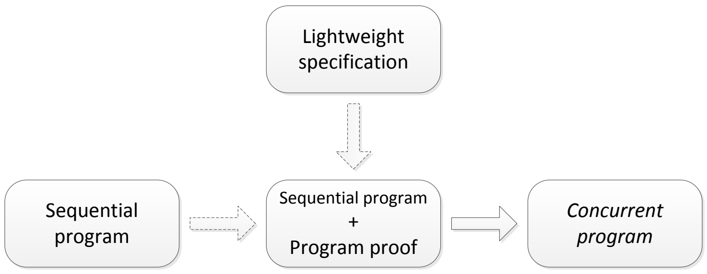
- The method in Part I follows the proof-directed synthesis pattern illustrated in Fig. 14.1a: a programmer supplies a sequential program and (optionally) a lightweight specification, which are given to a formal verification tool to generate a program proof; the method then transforms the sequential program into a concurrent program and uses the program proof to inject the appropriate synchronisation.

- The method in Part II follows the specification-directed synthesis pattern illustrated in Fig. 14.1b: formal verification techniques are used to extract a rich specification of the sequential program; the specification, compactly summarising the exact program behaviour, is then used to generate a concurrent version of the program.

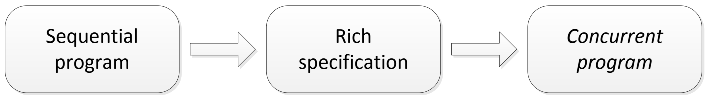
14.2 Future research directions

Exploiting formal verification methods as advocated in this dissertation is by no means limited only to automated synthesis of program parallelisation. Programs often have to be optimised with respect to other goals such as memory usage or energy consumption, which are, like speed, not intrinsic to the functional behaviour of the program. Thus it may make sense to consider using formal verification techniques to develop ways for automated program optimisation. With new, more automated and scalable formal verification techniques being actively developed, there lies a potential to discover novel program optimisation paradigms, and methods for improving existing program optimisations, with a guaranteed reliability.

We conclude this chapter by laying out three potential research directions in *formal verification-driven program optimisation*.



(a) Proof-directed parallelisation synthesis.



(b) Specification-directed parallelisation synthesis.

Figure 14.1: Two patterns of formal verification-based parallelisation synthesis presented in this dissertation. (Dashed arrows indicate black-box use of existing verification technology.)

14.2.1 Proof-directed compiler optimisation

The idea of using an (automatically constructed) program proof to guide the program optimisation may have many undiscovered applications. Particularly interesting is the separation logic proof setting: if a sequence of assertions forms a valid proof then no command can ever access a memory location (or other resource) not described in the precondition. In other words, a separation logic proof, regardless of the property being proven, will always describe all the resources accessed by the program. This insight opens up several follow-up topics. For instance, knowing memory footprints at each program point may allow us to take better advantage of cache locality in programs with recursive datatypes. Also, when checking the dependence of load and store instructions, having deep alias information may enable better instruction scheduling. Accurately tracking the heap-carried data dependencies and the heap layout may also enable synthesis of more efficient FPGA circuits realising heap-manipulating programs. Finally, using abductive reasoning, a valid separation logic proof can be constructed automatically even for incomplete program fragments, which may enable new, more holistic ways for utilising the instruction-level parallelism and performing code vectorisation for processors with SIMD extensions.

14.2.2 Optimisation for energy-constrained devices

As smartphones are becoming ubiquitous computing devices, optimisation of energy consumption is gaining more and more importance over the traditional optimisations for merely speed or memory. Energy consumption of a mobile application is a result of a complex interplay of many mobile phone components such as CPU, GPU, sensors, network, etc. The role of the network is particulary interesting since modern mobile applications often come partitioned into a client component running on the phone and a server component running across the network, enabling offloading of on-device computation and other tasks to a remote server (cloud). By representing energy resources with numerical tokens, separation logic-based approaches could enable new ways for reasoning about energy consumption and its distribution, and consequently, discover novel energy-oriented program optimisations. A more straightforward goal in optimising specifically for energy would be to develop practical methods for automatically discovering faithful models of power usage. Power models are of paramount importance for optimising mobile applications in practice, however, device manufacturers and operating system providers regularly do not provide them. Machine learning techniques like grammar induction may be able to systematically infer such models from power data.[1]

14.2.3 Optimisation for the cloud

Cloud computing is another emerging domain with under-developed program optimisation techniques. In addition to optimisation for speed and memory, in this context one often has to work towards satisfying various distributed systems' goals such as workload balance, fault tolerance, network topology, etc. While separation logic could provide ways for capturing the underlying concurrency patterns (e.g. the barrier signalling used in Part I can be extended to capture the MapReduce synchronisation), dealing with workload balance, fault tolerance or cloud-specific concepts such as elasticity is difficult and would require prior theoretical research to discover adequate formal models. Nevertheless, some specific problems can be immediately approached by existing methods. For instance, an important practical problem is how to reduce network congestion of MapReduce programs on clouds with separate storage and compute nodes. The network has much lower throughput than compute nodes and is often the bottleneck. Mappers typically receive a stream of data items, each with one or more components, while the computation may require only a subset of data items, or for a data item, only a subset of components. Such programs could be optimised by synthesising filters that would filter the data at storage nodes before transferring it to compute nodes. With some additional static analysis, the method developed in Part II could synthesise such filters automatically.[2]

[1]In fact, I have succeeded in representing several smoothed power traces of various mobile phone activity cycles by a variant of timed automata. I expect it to be possible to develop a learning algorithm that combined with a guided program execution (in a spirit similar to Σ^*) could automatically infer fine-grained power models of mobile phone components, and energy consumption specifications of operating system API.

[2]For instance, I was able to infer several such symbolic transducer-based filters for example programs provided with Hadoop.

Bibliography

[AC11] Rajeev Alur and Pavol Cerný. Streaming transducers for algorithmic verification of single-pass list-processing programs. In Ball and Sagiv [BS11], pages 599–610.

[ACMN05] Rajeev Alur, Pavol Cerný, P. Madhusudan, and Wonhong Nam. Synthesis of interface specifications for Java classes. In Palsberg and Abadi [PA05], pages 98–109.

[AJU10] Fides Aarts, Bengt Jonsson, and Johan Uijen. Generating models of infinite-state communication protocols using regular inference with abstraction. In Alexandre Petrenko, Adenilso da Silva Simão, and José Carlos Maldonado, editors, *Testing Software and Systems - 22nd IFIP WG 6.1 International Conference, ICTSS 2010, Natal, Brazil, November 8-10, 2010. Proceedings*, volume 6435 of *Lecture Notes in Computer Science*, pages 188–204. Springer, 2010.

[AKD⁺08] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Danny Dig, Amit M. Paradkar, and Michael D. Ernst. Finding bugs in dynamic web applications. In Ryder and Zeller [RZ08], pages 261–272.

[AL09] Shail Arora and Gary T. Leavens, editors. *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*. ACM, 2009.

[Ang87] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.

[ASKM14] Rahul Asati, Amitabha Sanyal, Amey Karkare, and Alan Mycroft. Liveness-based garbage collection. In Albert Cohen, editor, *Compiler Construction - 23rd International Conference, CC 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, volume 8409 of *Lecture Notes in Computer Science*, pages 85–106. Springer, 2014.

[ATA05] Sitij Agrawal, William Thies, and Saman P. Amarasinghe. Optimizing stream programs using linear state space analysis. In Thomas M. Conte, Paolo Faraboschi, William H. Mangione-Smith, and Walid A. Najjar, editors, *Proceedings of the 2005 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, CASES 2005, San Francisco, California, USA, September 24-27, 2005*, pages 126–136. ACM, 2005.

[BAD$^+$10] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. CoreDet: a compiler and runtime system for deterministic multithreaded execution. In James C. Hoe and Vikram S. Adve, editors, *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2010, Pittsburgh, Pennsylvania, USA, March 13-17, 2010*, pages 53–64. ACM, 2010.

[Bal05] Thomas Ball. Formalizing counterexample-driven refinement with weakest preconditions. In Manfred Broy, Johannes Grünbauer, David Harel, and Tony Hoare, editors, *Engineering Theories of Software Intensive Systems: Proceedings of the NATO Advanced Study Institute on Engineering Theories of Software Intensive Systems Marktoberdorf, Germany 3–15 August 2004*, pages 121–139, Dordrecht, 2005. Springer Netherlands.

[BAW10] Christian J. Bell, Andrew W. Appel, and David Walker. Concurrent separation logic for pipelined parallelization. In Radhia Cousot and Matthieu Martel, editors, *Static Analysis - 17th International Symposium, SAS 2010, Perpignan, France, September 14-16, 2010. Proceedings*, volume 6337 of *Lecture Notes in Computer Science*, pages 151–166. Springer, 2010.

[BB13] Matko Botinčan and Domagoj Babić. Sigma*: symbolic learning of input-output specifications. In Roberto Giacobazzi and Radhia Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 443–456. ACM, 2013.

[BCF$^+$08] Davide Balzarotti, Marco Cova, Viktoria Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *2008 IEEE Symposium on Security and Privacy (S&P 2008), 18-21 May 2008, Oakland, California, USA*, pages 387–401. IEEE Computer Society, 2008.

[BCO05a] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, volume 4111 of *Lecture Notes in Computer Science*, pages 115–137. Springer, 2005.

[BCO05b] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. In Kwangkeun Yi, editor, *Programming Languages and Systems, Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005, Proceedings*, volume 3780 of *Lecture Notes in Computer Science*, pages 52–68. Springer, 2005.

[BCOP05] Richard Bornat, Cristiano Calcagno, Peter W. O'Hearn, and Matthew J. Parkinson. Permission accounting in separation logic. In Palsberg and Abadi [PA05], pages 259–270.

[BDD+11] Matko Botinčan, Dino Distefano, Mike Dodds, Radu Griore, Daiva Naudžiūnienė, and Matthew Parkinson. coreStar: The Core of jStar. In K. R. M. Leino and M. Moskal, editors, *Boogie 2011*, pages 65–77, 2011.

[BDJ12] Matko Botinčan, Mike Dodds, and Suresh Jagannathan. Resource-sensitive synchronization inference by abduction. In John Field and Michael Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 309–322. ACM, 2012.

[BDJ13] Matko Botinčan, Mike Dodds, and Suresh Jagannathan. Proof-directed parallelization synthesis by separation logic. *ACM Trans. Program. Lang. Syst.*, 35(2):8:1–8:60, 2013.

[BDM15] Matko Botinčan, Mike Dodds, and Stephen Magill. Refining existential properties in separation logic analyses. *CoRR*, abs/1504.08309, 2015.

[BFH+04] Ian Buck, Tim Foley, Daniel Reiter Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, 2004.

[BGMV12] Nikolaj Bjørner, Vijay Ganesh, Raphaël Michel, and Margus Veanes. SMT-LIB sequences and regular expressions. In Pascal Fontaine and Amit Goel, editors, *10th International Workshop on Satisfiability Modulo Theories, SMT 2012, Manchester, UK, June 30 - July 1, 2012*, volume 20 of *EPiC Series in Computing*, pages 77–87. EasyChair, 2012.

[BJR08] Therese Berg, Bengt Jonsson, and Harald Raffelt. Regular inference for state machines using domains with equality tests. In José Luiz Fiadeiro and Paola Inverardi, editors, *Fundamental Approaches to Software Engineering, 11th International Conference, FASE 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4961 of *Lecture Notes in Computer Science*, pages 317–331. Springer, 2008.

[BK11] Dirk Beyer and M. Erkan Keremoglu. CPAchecker: A tool for configurable software verification. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 184–190. Springer, 2011.

[BMMR01] Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In Michael Burke and Mary Lou Soffa, editors, *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001*, pages 203–213. ACM, 2001.

[BNRS08] Nels E. Beckman, Aditya V. Nori, Sriram K. Rajamani, and Robert J. Simmons. Proofs from tests. In Ryder and Zeller [RZ08], pages 3–14.

[BPR01] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Boolean and cartesian abstraction for model checking C programs. In Tiziana Margaria and Wang Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*, volume 2031 of *Lecture Notes in Computer Science*, pages 268–283. Springer, 2001.

[BPR02] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Relative completeness of abstraction refinement for software model checking. In Joost-Pieter Katoen and Perdita Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 8th International Conference, TACAS 2002, Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, volume 2280 of *Lecture Notes in Computer Science*, pages 158–172. Springer, 2002.

[BS10] Jacob Burnim and Koushik Sen. Asserting and Checking Determinism for Multithreaded Programs. *Commun. ACM*, 53:97–105, June 2010.

[BS11] Thomas Ball and Mooly Sagiv, editors. *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*. ACM, 2011.

[BV11] Nikolaj Bjørner and Margus Veanes. Symbolic transducers. Technical Report MSR-TR-2011–3, Microsoft Research, 2011.

[BVB$^+$09] Muthu Manikandan Baskaran, Nagavijayalakshmi Vydyanathan, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Compiler-assisted dynamic scheduling for effective parallelization of loop nests on multicore processors. In Daniel A. Reed and Vivek Sarkar, editors, *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2009, Raleigh, NC, USA, February 14-18, 2009*, pages 219–228. ACM, 2009.

[BYLN09] Emery D. Berger, Ting Yang, Tongping Liu, and Gene Novark. Grace: safe multithreaded programming for C/C++. In Arora and Leavens [AL09], pages 81–96.

[CBP$^+$11] Chia Yuan Cho, Domagoj Babic, Pongsin Poosankam, Kevin Zhijie Chen, Edward XueJun Wu, and Dawn Song. MACE: model-inference-assisted concolic exploration for protocol and vulnerability discovery. In Wagner [Wag11].

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In Richard Draves and Robbert van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 209–224. USENIX Association, 2008.

[CDOY11] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. *J. ACM*, 58(6), 2011.

[CDV09] Cristiano Calcagno, Dino Distefano, and Viktor Vafeiadis. Bi-abductive resource invariant synthesis. In Zhenjiang Hu, editor, *Programming Languages and Systems, 7th Asian Symposium, APLAS 2009, Seoul, Korea, December 14-16, 2009. Proceedings*, volume 5904 of *Lecture Notes in Computer Science*, pages 259–274. Springer, 2009.

[CE05] Cristian Cadar and Dawson R. Engler. Execution generated test cases: How to make systems code crash itself. In Patrice Godefroid, editor, *Model Checking Software, 12th International SPIN Workshop, San Francisco, CA, USA, August 22-24, 2005, Proceedings*, volume 3639 of *Lecture Notes in Computer Science*, pages 2–23. Springer, 2005.

[CGJ$^+$00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson and A. Prasad Sistla, editors, *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.

[CGP03] Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Pasareanu. Learning assumptions for compositional verification. In Hubert Garavel and John Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, volume 2619 of *Lecture Notes in Computer Science*, pages 331–346. Springer, 2003.

[CHO$^+$11] Byron Cook, Christoph Haase, Joël Ouaknine, Matthew J. Parkinson, and James Worrell. Tractable reasoning in a fragment of separation logic. In Joost-Pieter Katoen and Barbara König, editors, *CONCUR 2011 - Concurrency Theory - 22nd International Conference, CONCUR 2011, Aachen, Germany, September 6-9, 2011. Proceedings*, volume 6901 of *Lecture Notes in Computer Science*, pages 235–249. Springer, 2011.

[CKSY05] Edmund M. Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. SATABS: sat-based predicate abstraction for ANSI-C. In Nicolas Halbwachs and Lenore D. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3440 of *Lecture Notes in Computer Science*, pages 570–574. Springer, 2005.

[CLL$^+$05] Michael K. Chen, Xiao-Feng Li, Ruiqi Lian, Jason H. Lin, Lixia Liu, Tao Liu, and Roy Ju. Shangri-La: achieving high performance from compiled network applications while enabling ease of programming. In Sarkar and Hall [SH05], pages 224–236.

[CMR+10] Byron Cook, Stephen Magill, Mohammad Raza, Jiri Simsa, and Satnam Singh. Making Fast Hardware with Separation Logic, 2010. Unpublished, http://cs.cmu.edu/~smagill/papers/fast-hardware.pdf.

[CZ06] Nadia Creignou and Bruno Zanuttini. A complete classification of the complexity of propositional abduction. *SIAM J. Comput.*, 36(1):207–229, 2006.

[DF10] Dino Distefano and Ivana Filipovic. Memory leaks detection in java by bi-abductive inference. In David S. Rosenblum and Gabriele Taentzer, editors, *Fundamental Approaches to Software Engineering, 13th International Conference, FASE 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6013 of *Lecture Notes in Computer Science*, pages 278–292. Springer, 2010.

[DHRA06] M. Drake, Henry Hoffmann, Rodric M. Rabbah, and Saman P. Amarasinghe. MPEG-2 decoding in a stream programming language. In *20th International Parallel and Distributed Processing Symposium (IPDPS 2006), Proceedings, 25-29 April 2006, Rhodes Island, Greece*. IEEE, 2006.

[DJP11] Mike Dodds, Suresh Jagannathan, and Matthew J. Parkinson. Modular reasoning for deterministic parallelism. In Ball and Sagiv [BS11], pages 259–270.

[DKR82] Alan J. Demers, C. Keleman, and Bernd Reusch. On some decidable properties of finite state translations. *Acta Informatica*, 17:349–364, 1982.

[DP08] Dino Distefano and Matthew J. Parkinson. jStar: towards practical verification for java. In Gail E. Harris, editor, *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-23, 2008, Nashville, TN, USA*, pages 213–226. ACM, 2008.

[DRRV10] Jyotirmoy V. Deshmukh, G. Ramalingam, Venkatesh Prasad Ranganath, and Kapil Vaswani. Logical concurrency control from sequential proofs. In Gordon [Gor10], pages 226–245.

[EG95] Thomas Eiter and Georg Gottlob. The complexity of logic-based abduction. *J. ACM*, 42(1):3–42, 1995.

[Fer04] Jérôme Feret. Static analysis of digital filters. In David A. Schmidt, editor, *Programming Languages and Systems, 13th European Symposium on Programming, ESOP 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, volume 2986 of *Lecture Notes in Computer Science*, pages 33–48. Springer, 2004.

[FQ02] Cormac Flanagan and Shaz Qadeer. Predicate abstraction for software verification. In Launchbury and Mitchell [LM02], pages 191–202.

[GAW$^+$08] Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S. Yu, and Myungcheol Doo. SPADE: the system s declarative stream processing engine. In Jason Tsong-Li Wang, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 1123–1134. ACM, 2008.

[GBA$^+$11] Guy Golan-Gueta, Nathan Grasso Bronson, Alex Aiken, G. Ramalingam, Mooly Sagiv, and Eran Yahav. Automatic fine-grain locking using shape properties. In Cristina Videira Lopes and Kathleen Fisher, editors, *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, pages 225–242. ACM, 2011.

[GBC$^+$07] Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. Local reasoning for storable locks and threads. In Zhong Shao, editor, *Programming Languages and Systems, 5th Asian Symposium, APLAS 2007, Singapore, November 29-December 1, 2007, Proceedings*, volume 4807 of *Lecture Notes in Computer Science*, pages 19–37. Springer, 2007.

[GCTR08] Jayanth Gummaraju, Joel Coburn, Yoshio Turner, and Mendel Rosenblum. Streamware: programming general-purpose multicore processors using streams. In Susan J. Eggers and James R. Larus, editors, *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2008, Seattle, WA, USA, March 1-5, 2008*, pages 297–307. ACM, 2008.

[GD07] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, volume 4590 of *Lecture Notes in Computer Science*, pages 519–531. Springer, 2007.

[GHK$^+$06] Bhargav S. Gulavani, Thomas A. Henzinger, Yamini Kannan, Aditya V. Nori, and Sriram K. Rajamani. SYNERGY: a new algorithm for property checking. In Michal Young and Premkumar T. Devanbu, editors, *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2006, Portland, Oregon, USA, November 5-11, 2006*, pages 117–127. ACM, 2006.

[GHZ98] Rakesh Ghiya, Laurie J. Hendren, and Yingchun Zhu. Detecting parallelism in C programs with recursive data structures. In Kai Koskimies, editor, *Compiler Construction, 7th International Conference, CC'98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*, volume 1383 of *Lecture Notes in Computer Science*, pages 159–173. Springer, 1998.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In Sarkar and Hall [SH05], pages 213–223.

[GNRT10] Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and SaiDeep Tetali. Compositional may-must program analysis: unleashing the power of alternation. In Manuel V. Hermenegildo and Jens Palsberg, editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 43–56. ACM, 2010.

[Gor10] Andrew D. Gordon, editor. *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6012 of *Lecture Notes in Computer Science*. Springer, 2010.

[GPPS99] Rajiv Gupta, Santosh Pande, Kleanthis Psarris, and Vivek Sarkar. Compilation techniques for parallel systems. *Parallel Computing*, 25(13-14):1741–1783, 1999.

[GRE$^+$01] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proc. of the Workload Characterization. WWC-4 IEEE Int. Workshop*, pages 3–14, 2001.

[GRR12] Dimitra Giannakopoulou, Zvonimir Rakamaric, and Vishwanath Raman. Symbolic learning of component interfaces. In Antoine Miné and David Schmidt, editors, *Static Analysis - 19th International Symposium, SAS 2012, Deauville, France, September 11-13, 2012. Proceedings*, volume 7460 of *Lecture Notes in Computer Science*, pages 248–264. Springer, 2012.

[GS97] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In Orna Grumberg, editor, *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 1997.

[GTA06] Michael I. Gordon, William Thies, and Saman P. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In John Paul Shen and Margaret Martonosi, editors, *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*, pages 151–162. ACM, 2006.

[HAN08] Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. Oracle semantics for concurrent separation logic. In Sophia Drossopoulou, editor, *Programming Languages and Systems, 17th European Symposium on Programming, ESOP 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4960 of *Lecture Notes in Computer Science*, pages 353–367. Springer, 2008.

[HHH08] Christian Haack, Marieke Huisman, and Clément Hurlin. Reasoning about java's reentrant locks. In G. Ramalingam, editor, *Programming Languages* *and Systems, 6th Asian Symposium, APLAS 2008, Bangalore, India, December 9-11, 2008. Proceedings*, volume 5356 of *Lecture Notes in Computer Science*, pages 171–187. Springer, 2008.

[HJMS02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In Launchbury and Mitchell [LM02], pages 58–70.

[HKM$^+$08] Amir Hormati, Manjunath Kudlur, Scott A. Mahlke, David F. Bacon, and Rodric M. Rabbah. Optimus: efficient realization of streaming applications on fpgas. In Erik R. Altman, editor, *Proceedings of the 2008 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, CASES 2008, Atlanta, GA, USA, October 19-24, 2008*, pages 41–50. ACM, 2008.

[HLM$^+$11] Pieter Hooimeijer, Benjamin Livshits, David Molnar, Prateek Saxena, and Margus Veanes. Fast and precise sanitizer analysis with BEK. In Wagner [Wag11].

[HN90] Laurie J. Hendren and Alexandru Nicolau. Parallelizing programs with recursive data structures. *IEEE Trans. Parallel Distrib. Syst.*, 1(1):35–47, 1990.

[HO08] Tony Hoare and Peter W. O'Hearn. Separation logic semantics for communicating processes. *Electr. Notes Theor. Comput. Sci.*, 212:3–25, 2008.

[Hop69] J. E. Hopcroft. On the equivalence and containment problems for context-free languages. *Theory of Computing Systems*, 3:119–124, 1969.

[HPR89] Susan Horwitz, Phil Pfeiffer, and Thomas W. Reps. Dependence analysis for pointer variables. In Richard L. Wexelblat, editor, *Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation (PLDI), Portland, Oregon, USA, June 21-23, 1989*, pages 28–40. ACM, 1989.

[HSJC12] Falk Howar, Bernhard Steffen, Bengt Jonsson, and Sofia Cassel. Inferring canonical register automata. In Viktor Kuncak and Andrey Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation - 13th International Conference, VMCAI 2012, Philadelphia, PA, USA, January 22-24, 2012. Proceedings*, volume 7148 of *Lecture Notes in Computer Science*, pages 251–266. Springer, 2012.

[Hur09] Clément Hurlin. Automatic parallelization and optimization of programs by proof rewriting. In Jens Palsberg and Zhendong Su, editors, *Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings*, volume 5673 of *Lecture Notes in Computer Science*, pages 52–68. Springer, 2009.

[HV05] Peter Habermehl and Tomás Vojnar. Regular model checking using inference of regular languages. *Electr. Notes Theor. Comput. Sci.*, 138(3):21–36, 2005.

[HWBR09] Andrei Hagiescu, Weng-Fai Wong, David F. Bacon, and Rodric M. Rabbah. A computing origami: folding streams in FPGAs. In *Proceedings of the 46th*

Design Automation Conference, DAC 2009, San Francisco, CA, USA, July 26-31, 2009, pages 282–287. ACM, 2009.

[Hyd09] Randall Hyde. The fallacy of premature optimization. *Ubiquity*, 2009(February), February 2009.

[Iba77] Oscar H. Ibarra. The unsolvability of the equivalence problem for epsilon-free ngsm's with unary input (output) alphabet and applications. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 74–81. IEEE Computer Society, 1977.

[JAD+09] Robert L. Bocchino Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A type and effect system for deterministic parallel Java. In Arora and Leavens [AL09], pages 97–116.

[JM06] Ranjit Jhala and Kenneth L. McMillan. A practical and complete approach to predicate refinement. In Holger Hermanns and Jens Palsberg, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference, TACAS 2006 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25 - April 2, 2006, Proceedings*, volume 3920 of *Lecture Notes in Computer Science*, pages 459–473. Springer, 2006.

[JP09] Bart Jacobs and Frank Piessens. Modular full functional specification and verification of lock-free data structures. Technical Report CW 551, Katholieke Universiteit Leuven, Dept. of Computer Science, 2009.

[KDR+02] Ujval J. Kapasi, William J. Dally, Scott Rixner, John D. Owens, and Brucek Khailany. The imagine stream processor. In *20th International Conference on Computer Design (ICCD 2002), VLSI in Computers and Processors, 16-18 September 2002, Freiburg, Germany, Proceedings*, pages 282–288. IEEE Computer Society, 2002.

[KM08] Manjunath Kudlur and Scott A. Mahlke. Orchestrating the execution of stream programs on multicore platforms. In Rajiv Gupta and Saman P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 114–124. ACM, 2008.

[LCB11] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. Dthreads: efficient deterministic multithreading. In Ted Wobber and Peter Druschel, editors, *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, pages 327–336. ACM, 2011.

[LDWL06] Shih-Wei Liao, Zhaohui Du, Gansha Wu, and Guei-Yuan Lueh. Data and computation transformations for brook streaming applications on multiprocessors. In *Fourth IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2006), 26-29 March 2006, New York, New York, USA*, pages 196–207. IEEE Computer Society, 2006.

[LM02] John Launchbury and John C. Mitchell, editors. *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*. ACM, 2002.

[LMS10] K. Rustan M. Leino, Peter Müller, and Jan Smans. Deadlock-free channels and locks. In Gordon [Gor10], pages 407–426.

[LTA03] Andrew A. Lamb, William Thies, and Saman P. Amarasinghe. Linear analysis and optimization of stream programs. In Ron Cytron and Rajiv Gupta, editors, *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003*, pages 12–25. ACM, 2003.

[LY96] David Lee and Mihalis Yannakakis. Principles and methods of testing finite state machines: A survey. In *Proc. of the IEEE*, volume 84, pages 1090–1123, 1996.

[McM06] Kenneth L. McMillan. Lazy abstraction with interpolants. In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4144 of *Lecture Notes in Computer Science*, pages 123–136. Springer, 2006.

[Mon05] David Monniaux. Compositional analysis of floating-point linear numerical filters. In Kousha Etessami and Sriram K. Rajamani, editors, *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, volume 3576 of *Lecture Notes in Computer Science*, pages 199–212. Springer, 2005.

[NZJ08] Armand Navabi, Xiangyu Zhang, and Suresh Jagannathan. Quasi-static scheduling for safe futures. In Siddhartha Chatterjee and Michael L. Scott, editors, *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2008, Salt Lake City, UT, USA, February 20-23, 2008*, pages 23–32. ACM, 2008.

[OGV93] J. Oncina, P. García, and E. Vidal. Learning subsequential transducers for pattern recognition interpretation tasks. *IEEE Transactions Pattern Analysis and Machine Intelligence*, 15:448–458, May 1993.

[O'H07] Peter W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.

[ORSA05] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. Automatic thread extraction with decoupled software pipelining. In *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-38 2005), 12-16 November 2005, Barcelona, Spain*, pages 105–118. IEEE Computer Society, 2005.

[PA05] Jens Palsberg and Martín Abadi, editors. *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*. ACM, 2005.

[Par05] Matthew J. Parkinson. Local reasoning for Java. Technical Report UCAM-CL-TR-654, University of Cambridge, Computer Laboratory, November 2005.

[PNK$^+$11] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, Muhammad Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. The tao of parallelism in algorithms. In Mary W. Hall and David A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 12–25. ACM, 2011.

[PRV10] Prakash Prabhu, Ganesan Ramalingam, and Kapil Vaswani. Safe programmable speculative parallelism. In *Proc. of the 2010 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 50–61, 2010.

[RCG09] Mohammad Raza, Cristiano Calcagno, and Philippa Gardner. Automatic parallelization with separation logic. In Giuseppe Castagna, editor, *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5502 of *Lecture Notes in Computer Science*, pages 348–362. Springer, 2009.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 55–74. IEEE Computer Society, 2002.

[RVY13] Veselin Raychev, Martin T. Vechev, and Eran Yahav. Automatic synthesis of deterministic concurrency. In Francesco Logozzo and Manuel Fähndrich, editors, *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*, volume 7935 of *Lecture Notes in Computer Science*, pages 283–303. Springer, 2013.

[RZ08] Barbara G. Ryder and Andreas Zeller, editors. *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008*. ACM, 2008.

[SG09] Muzammil Shahbaz and Roland Groz. Inferring mealy machines. In Ana Cavalcanti and Dennis Dams, editors, *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings*, volume 5850 of *Lecture Notes in Computer Science*, pages 207–222. Springer, 2009.

[SGP10] Rishabh Singh, Dimitra Giannakopoulou, and Corina S. Pasareanu. Learning component interfaces with may and must abstractions. In Tayssir Touili, Byron Cook, and Paul B. Jackson, editors, *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, volume 6174 of *Lecture Notes in Computer Science*, pages 527–542. Springer, 2010.

[SH05] Vivek Sarkar and Mary W. Hall, editors. *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*. ACM, 2005.

[SHG$^+$10] Robert Soulé, Martin Hirzel, Robert Grimm, Bugra Gedik, Henrique Andrade, Vibhore Kumar, and Kun-Lung Wu. A universal calculus for stream processing languages. In Gordon [Gor10], pages 507–528.

[SMA05] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In Michel Wermelinger and Harald C. Gall, editors, *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, pages 263–272. ACM, 2005.

[SRW96] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. In Hans-Juergen Boehm and Guy L. Steele Jr., editors, *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, pages 16–31. ACM Press, 1996.

[SRW98] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Trans. Program. Lang. Syst.*, 20(1):1–50, 1998.

[SRW99] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In Andrew W. Appel and Alex Aiken, editors, *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*, pages 105–118. ACM, 1999.

[SRW02] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.

[SV-13] 2nd International Competition on Software Verification (SV-COMP) held at Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013., 2013. http://sv-comp.sosy-lab.org/2013/.

[TCA07] William Thies, Vikram Chandrasekhar, and Saman P. Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in C programs. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-40 2007), 1-5 December 2007, Chicago, Illinois, USA*, pages 356–369. IEEE Computer Society, 2007.

[TKA02] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. Streamit: A language for streaming applications. In R. Nigel Horspool, editor, *Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint*

European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings, volume 2304 of *Lecture Notes in Computer Science*, pages 179–196. Springer, 2002.

[TZ94] Peiyi Tang and John N. Zigman. Reducing data communication overhead for DOACROSS loop nests. In John R. Gurd and William Jalby, editors, *Proceedings of the 8th international conference on Supercomputing, ICS 1994, Manchester, UK, July 11-15, 1994*, pages 44–53. ACM, 1994.

[UGT09] Abhishek Udupa, R. Govindarajan, and Matthew J. Thazhuthaveetil. Software pipelined execution of stream programs on gpus. In *Proceedings of the CGO 2009, The Seventh International Symposium on Code Generation and Optimization, Seattle, Washington, USA, March 22-25, 2009*, pages 200–209. IEEE Computer Society, 2009.

[Vil96] Juan Miguel Vilar. Query learning of subsequential transducers. In Laurent Miclet and Colin de la Higuera, editors, *Grammatical Inference: Learning Syntax from Sentences, 3rd International Colloquium, ICGI-96, Montpellier, France, September 25-27, 1996, Proceedings*, volume 1147 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 1996.

[VLC10] Jules Villard, Étienne Lozes, and Cristiano Calcagno. Tracking heaps that hop with heap-hop. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6015 of *Lecture Notes in Computer Science*, pages 275–279. Springer, 2010.

[VML11] Margus Veanes, David Molnar, and Benjamin Livshits. Decision procedures for composition and equivalence of symbolic finite state transducers. Technical Report MSR-TR-2011-32, Microsoft Research, 2011.

[VMLL11] Margus Veanes, David Molnar, Benjamin Livshits, and Lubomir Litchev. Generating fast string manipulating code through transducer exploration and SIMD integration. Technical Report MSR-TR-2011–124, Microsoft Research, 2011.

[vNG01] Gertjan van Noord and Dale Gerdemann. Finite state transducers with predicates and identities. *Grammars*, 4(3):263–286, 2001.

[Wag11] David Wagner, editor. *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings*. USENIX Association, 2011.

[WJH05] Adam Welc, Suresh Jagannathan, and Antony L. Hosking. Safe futures for java. In Ralph E. Johnson and Richard P. Gabriel, editors, *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, pages 439–453. ACM, 2005.

[YLB$^+$08] Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter W. O'Hearn. Scalable shape analysis for systems code. In Aarti Gupta and Sharad Malik, editors, *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*, volume 5123 of *Lecture Notes in Computer Science*, pages 385–398. Springer, 2008.