Efficiently Verifying Programs with Rich Control Flows

Cristian Andrei Gherghina
School of Computing
National University of Singapore

A thesis submitted for the degree of
Doctor of Philosophy
November 23, 2012

DECLARATION

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

_____

Gherghina Cristian Andrei

21 August 2012

To my always steady compass: my dear parents, sister and my better half.

Multumesc dragii mei.

Acknowledgements

I am grateful to my advisor and mentor, Professor Chin Wei-Ngan, for his constant guidance and encouragement both professionally and personally during these five years. I hope I can eventually internalize his example on patience, kindness, focus, immense passion and drive and constant search for both elegance and relevance.

I thank my Thesis Committee Members: Professors Khoo Siau Cheng, Joxan Jaffar and Naoki Kobayashi for their feedbacks on my work, which greatly helped shape my thesis. I also thank my colleagues and seniors Razvan Voicu, Aquinas Hobor, Cristina David, Florin Craciun, Loc Le, Chanh Le for the fruitful research collaborations and the many lessons they have taught me.

For interesting discussions, entertaining moments and mostly for making Singapore a home away from home, I thank Andrei Hagiescu, Bogdan Tudor, Cristina Carbunaru, Dan Tudose, Cristina David, Corneliu Popeea, Florin Craciun, Yamilet Serrano and Andreea Costea. I thank Mihail Popa, Tudor Barbu and Mihai Mihailescu for their companionship and close friendship through my PH.D. years, for many more years before that and hopefully for many more to come.

I thank my parents, sister and my better half for their unconditional love and support, their trust, patience, and understanding.

Abstract

In the era of multicore processing, formal verification is more important than ever. This thesis narrows the gap between the increased complexity of control flow patterns, often spanning across multiple threads, and the stringent need for accuracy. We introduce a sound verification logic designed to efficiently handle programs with complex control flow patterns.

We advocate for an extension of separation logic that can uniformly handle exceptions, program errors and other kinds of control flows. This approach is supported through a uniform mechanism that captures static control flows (such as normal execution) and dynamic control flows (such as exceptions) within a single formalism. Following Stroustrup's definition [94, 70], our verification technique could ensure exception safety in terms of four guarantees of increasing quality, namely no-leak guarantee, basic guarantee, strong guarantee and no-throw guarantee.

A second component of our verification logic handles Pthreads barriers. Unlike locks and critical sections, Pthreads barriers generate complex control flow and resource ownership exchange patterns. They enable simultaneous resource redistribution between multiple threads and are inherently stateful, leading to significant complications in the design of the logic and its soundness proof. We equip our logic with a novel mechanism for explicitly capturing and reasoning about barrier behaviour.

The last essential component of our proposal consists of a novel predicate pruning technique targeting user-defined disjunctive predicates. Although we will introduce a Hoare logic that successfully verifies programs with exceptions and barriers, in order for our proposal to gain acceptance, it is not sufficient to work we must also ensure that verification can be done quickly and precisely. Our proposed predicate specialization and pruning mechanism is designed with this goal in mind.

Our barrier extension can be viewed as an instance of a highly specialized verification logic that relies on user-defined disjunctive predicates. We address the performance penalty

associated with the proof search induced by disjunctive predicates in general and by our barrier handling in particular, by proposing a predicate specialization technique that allows efficient symbolic pruning of infeasible disjuncts inside each predicate instance. Our technique is presented as a specialization operation whose derivations preserve the satisfiability of formulas, while reducing the subsequent cost of their manipulation. Initial experimental results have confirmed significant speed gains from the deployment of predicate specialization. It yields up to a 10 fold increase in discharging proof obligations generated by the verification of general sequential programs and up to 37 fold increase in the speed of barrier reasoning.

As support for our proposal, we showcase a program verification toolset, that uses our logic to automatically prove the correctness of programs with exceptions and barriers.

Contents

List of Figures

Chapter 1

Introduction

The explosive growth in the software industry has led to a vast assortment of software being created to control computer systems involving almost all aspects of our lives. While some of these software components have been successfully built, there are also many software components whose quality has continued to plague the users who are stuck with them. This in turn pushed the need for better software engineering guidelines that would help in the creation and quality control of software systems. Traditionally, software quality control relied on simplistic methodologies e.g. peer inspection and repeated regression testing. Though such methods can often discover the presence of problems, they cannot guarantee bug-free and/or low-defects software, and unfortunately these are exactly the guarantees that we would expect for the complex software tasked with controlling safety critical systems.

Two high-profile examples of safety critical software failures are the Mars Climate Orbiter which crashed due to an incorrect conversion to metric units, and the Ariane 5 failure due to a floating-point conversion which raised an exception that was not properly handled [30]. The latter example is particularly significant for us since it highlighted the need for considering all manner of *control flows*, particularly those that are related to exceptional scenarios. While it might be challenging to investigate and consider all corner cases it is precisely these cases that could haunt us if our software is not adequately prepared for them.

Formal methods aim to address this issue with surgical precision: they aim for proving software correctness and thus for guaranteeing that systems never fail [74, 46]. Program verification is one successful approach to proving software correctness by focusing on proving specific, user provided, correctness statements [49, 48]. Such statements are typically expressed in rich specification logics that allow for expressive yet concise specifications and more importantly are designed with the goal of lending themselves to systematic checking by verification systems [35].

An example of a logic that is suitable for specification and verification is Hoare logic [47]. It was designed to help describe, modularly, the effect of sequential programs. The Hoare logic approach consists of constructing triples of the form $\{P\}\ c\ \{Q\}$ where c is a code sequence while P and Q denote abstractions of concrete program states. Each such abstraction represents a set of reachable concrete states which can be viewed as one abstract program state. A triple is said to hold if and only if whenever c is executed from a concrete state that is captured by an abstraction in P then, if c terminates, it will do so in a state whose abstraction can be described by Q.

The core of Hoare logic is comprised of a set of axioms which define triples corresponding to the basic statements in the programming language. However, depending on the programming language of choice, on their specific features and the logical framework chosen for expressing the program state, different Hoare logic variants can be constructed.

For example, many commonly used programming languages have complex memory models in which resources can be allocated on the stack or on the heap. Heap allocation introduces an extra hurdle, as it facilitates sharing and aliasing of resources which requires the abstraction mechanism to be able to capture aliasing in a preferably concise manner. One particularly elegant and concise framework of expressing and reasoning about such sharing and aliasing is separation logic [57, 86].

As the common usage of separation logic revolves around abstracting and reasoning about program states, typical models for separation logic are centered on abstractions of machine

states with both stack and heap stores. Thus, the basic separation logic assertions capture allocatedness facts of the form $x \mapsto a$ read as x points-to a heap location containing value a. In order to concisely capture non-aliasing information, on top of the common logic connectives of first-order logic, separation logic also exhibits two new connectives: separating conjunction $*$ and separating implication, $-\!\!*$. The separating conjunction assertion $p_1 * p_2$ holds if and only if there exists a division of the current heap such that one sub-heap satisfies p_1, the other satisfies p_2 and the two sub-heaps are disjoint. Embedding heap disjointedness in the definition of the connective ensures a clutter free mechanism for capturing non-aliasing information. The advantage of a concise and precise logics has led to several automatic or semi-automatic verification tools being developed: [9, 42, 75, 58].

Concurrency is another language feature with a big impact on the abstraction formalism. In the last decade, languages that take advantage of the multicore/multiprocessor hardware platforms have become mainstream. Thus verification tools are expected to cater for multithreaded programs. O'Hearn, in [80], introduces an extension of Hoare logic, Concurrent Separation logic (CSL), with the goal of allowing a form of parallelism in the verified program. One big advantage of CSL is that it maintains the Hoare Logics modularity even at the concurrent computation level: a correctness proof for the entire program can be constructed by verifying each of the parallel computations individually. Since then, a considerable body of work has focused on allowing in the verified program various forms of communication and synchronization while still maintaining the guarantee that no races occur.

Recently, concurrent separation logic has been used to formally reason about shared-memory programs that use critical sections and (first-class) locks [80, 51, 43, 50]. Programs verified with concurrent separation logic are provably data-race free. However more sophisticated synchronization mechanisms are inherently trickier to reason about. The general assumption is that other mechanisms can be implemented with locks, and that reasonable Hoare rules can be derived by verifying their implementation. Indeed, the first published example of concurrent separation logic was implementing semaphores using critical sections [80]. Unfortunately, not

all synchronization mechanisms can be easily reduced to locks in a way that allows for a reasonable Hoare rule to be derived. Therefore, despite fundamental theoretical advances such as Hoare logic, separation logic, CSL, plus a frenzy of further developments that tackle complex verification problems, support for specifying and verifying several important language features of modern programming languages is still lacking. As a consequence, languages are often too complex to fully analyse, causing verification tools to omit some of the fancy features. For instance, an essential feature of modern programming languages often overlooked in verification is the ability to generate *complex non-local control flows*, e.g. exception handling .

Exception handling is an important mechanism for dynamically altering control flows. It is instrumental for building robust software with good error handling capability. However, exceptions are often omitted during the initial formulation of program analysis and optimization.

Furthermore, with respect to the shared memory paradigm, there is a considerable challenge in verifying programs in the presence of the complex control flow patterns generated by the use of sophisticated synchronization mechanisms. The Pthreads-style barriers are a prime example of such a synchronization mechanism that is surprisingly often used in practice [1] and yet has been overlooked by current verification systems.

When a thread issues a barrier call it waits until a specified number (typically all) of other threads have also issued a barrier call; at that point, all of the threads continue. Even the common barrier usage exhibits complex control flow patterns: usually programs with barriers use multiple threads advancing in lockstep through a complex computation such that they will not "step on each others toes" when accessing shared data, the usual access pattern is concurrent read exclusive write. Thus in common usage, barriers are implicitly associated with a complex resource ownership redistribution. The hardship of verifying multithreaded programs with barriers lies in designing a mechanism for encoding the complex control and fractional resource ownership change patterns associated with barriers.

[1] 38% of the total workloads in PARSEC, a standard benchmarking suite for multicore architectures, use barriers [10]

1.1 Thesis Objectives

The principal goal of this thesis is to introduce a sound verification logic designed to efficiently handle programs with complex control flow patterns. The logic will explicitly and precisely track control flows essential to the verification of the program like various exception related flows however it will also incorporate abstractions required to maintain modularity and avoid the exponential blowup when reasoning in a multithreaded setting. We will also describe several contributions related to the integration of our verification logic in order to broaden the applicability of an existing verification tool chain.

The first problem we tackle is the lack of a high level programming language which can be middle ground between programmers and verification tools. When considering the traditional approach of converting programs from high level languages to machine code, the target code often turns out to be too cryptic (or low level) for program analysis. Our goal is to design an intermediate, minimal but expressive, core language which can be easily analysed and manipulated, and to show that this language can handle major language features by translating a significant imperative source language into it. The translation to the core language enables us to easily analyse and optimize the code, while not sacrificing the flexibility and rich characteristic of the source language.

Therefore we first design an intermediate, expressive, syntactically simple, core language focused on simplifying the task of program verification in the presence of complex control flows. The insight underlying the core language is that it is possible to simplify the verification effort by recasting in one syntactically simple form most of the control flow generating mechanisms.

Secondly, we advocate for an extension of concurrent separation logic that can uniformly handle exceptions, program errors and other kinds of control flows. This is elegantly achieved by designing our extension for the syntactically simple structures of the core language. Our logic treats exceptions as possible outcomes that could be later remedied, while errors are

conditions that should be avoided by user programs. This distinction is supported through a uniform mechanism that captures static control flows (such as normal execution) and dynamic control flows (such as exceptions) within a single formalism. Following Stroustrup's definition [94, 70], our verification technique could ensure exception safety in terms of four guarantees of increasing quality, namely no-leak guarantee, basic guarantee, strong guarantee and no-throw guarantee.

A third component of our verification logic handles Pthreads barriers. Unlike locks and critical sections, Pthreads barriers generate complex control flow and resource ownership exchange patterns. They enable simultaneous resource redistribution between multiple threads and are inherently stateful, leading to significant complications in the design of the logic and its soundness proof. We equip our logic with a novel mechanism for explicitly capturing and reasoning about barrier behaviour.

As support for our proposal, we showcase a program verification toolset, based on the HIP verifier [78, 20], that uses our logic to automatically prove the correctness of programs with the features discussed so far. Unfortunately, the inherently complex ownership exchange patterns require HIP to support a shared resource ownership accounting scheme. Therefore we introduce in HIP a fractional ownership control mechanism based on the binary tree model described by Dockings *et al.* in [28].

The last essential component of our proposal consists of a novel general predicate pruning technique targeting disjunctive predicates. Although we will introduce a Hoare logic that successfully verifies programs with exceptions and barriers, in order for our proposal to gain acceptance, it is not sufficient that it works but it needs to work fast too. Our barrier extension can be viewed as an instance of a highly specialized verification logic that relies on user-defined disjunctive predicates. Therefore we will incorporate our new predicate pruning technique into our verification logic in order to greatly speedup the verification process.

In general, separation logic-based abstraction mechanisms, enhanced with user-defined disjunctive predicates, represent a powerful, expressive means of specifying heap-based data

structures with strong invariant properties. However, expressive power comes at a cost: the manipulation of such logics typically requires the unfolding of disjunctive predicates which may lead to expensive proof search.

We address the performance penalty induced by disjunctive predicates in general and by our barrier handling in particular, by proposing a general predicate specialization technique that allows efficient symbolic pruning of infeasible disjuncts inside each predicate instance. While specialization is a familiar technique for code optimization, its use in program verification is new. Our technique is presented as a specialization operation whose derivations preserve the satisfiability of formulas, while reducing the subsequent cost of their manipulation. Initial experimental results have confirmed significant speed gains from the deployment of predicate specialization. It yields up to a 10 fold increase in discharging proof obligations generated by the verification of general sequential programs and up to 37 fold increase in the speed of barrier reasoning.

1.2 Contributions of the Thesis

The contributions of these thesis can be organized by four main vectors:

A core language with unified control flows

(Chapter 2, first presented in [25])

- We propose a core language, `Core−U` , with a novel view of the control flows unifying both normal and exceptional executions. This new design is supported by a pair of unified constructs that are considerably more general than previous approaches. Due to this unification of the control flows, the core language is easier to analyse and optimize.

- We define a translation from an expressive Java-like imperative language into our core language. The translation is based on rewrite rules and illustrates how advanced language features, such as `try−finally` and multi-return functions, can be easily captured by our

core language. Moreover, we prove two important properties of the translation, namely completeness and termination.

- We provide a set of optimization rules for our language, designed to reduce the implementation overhead. These rules are specified at a high-level which facilitates both human understanding and the construction of correctness proofs. While the set of optimization rules is by no means exhaustive, these rules can help support better practical prospects for our core language. For all the rules we supply correctness proofs, which are also meant to illustrate the ease of designing optimizations and proving them correct.

A Specification Logic for Exceptions

(Chapter 3, first presented in [37])

- We introduce a specification logic that captures the states for both normal and exceptional executions. Our design is guided by the novel unification of both static control flows (such as break and return), and dynamic control flows (such as exceptions and errors).

- We revisit exception safety guarantees as introduced in [94], and extended in [70]. Additionally, we improve the strong guarantee for exception safety. To support a tradeoff between precision and cost of verification, our verification system is flexible in enforcing different levels of exception safety.

- We introduce a set of very simple Hoare rules for `Core−U` .

- We have included the above features in the HIP verifier and validated it with a suite of exception-handling examples.

Pthreads Barriers in Concurrent Separation Logic

(Chapter 4 , first presented in [52] and extended in [53])

- We give a formal characterization for sound Pthreads barrier definitions.

- We extend the `Core−U` verification logic with a natural Hoare rule for verifying barrier calls and include it in the HIP verifier.
- We give a formal resource-aware *unerased* concurrent operational semantics for barriers and prove our Hoare rules sound with respect to our semantics. Our soundness results are machine-checked in Coq
- We add support for a fractional resource ownership accounting scheme to the entailment procedure in the SLEEK separation logic entailment checker [20] which is the core of the HIP verifier.
- We describe a solver for the binary tree domain proposed by Dockings *et al.* in [28] as a model for fractional resource ownership accounting.

Specialization for Pruning Disjunctive Predicates to Support Verification

(Chapter 5, first presented in [21])

- We propose a *new specialization calculus* that leads to more effective program verification. Our calculus specializes proof obligations produced in the program verification process, and can be used as a preprocessing step before the obligations are fed into third party theorem provers or decision procedures.
- We adapt *memoization* and *incremental pruning* techniques to obtain an optimized version of the specialization calculus.
- We included our specialization calculus in the HIP/SLEEK together with the previous extensions.
- We apply the specializer to barrier definitions. The use of our specializer yields dramatic reductions in verification times, both for large sequential programs and programs employing barrier synchronization. Even for simple examples with barrier usage we show a specialization induced speedup of up to 37 .

1.3 Outline

In Chapter 2 we describe the formal preliminaries. We introduce `SrcLang`, the target language of our verification solution together with a syntactically simple core language to which the input language can be translated and for which we have designed our verification logic. We will also describe a specification language with support for capturing various control flow types and barrier related assertions.

In Chapter 3 we will elaborate on common expectations for exception safety guarantees and introduce a set of elegantly simple rules for verifying programs with exceptions.

In Chapter 4 we further extend the exception logic by adding support for barrier reasoning through a novel mechanism for describing barrier behaviours. Furthermore, we will introduce a Hoare rule for verifying barrier calls which is surprisingly simple when compared to the complex synchronization pattern the barriers introduce. We will outline the soundness proof for our logic and also describe the work required to integrate our verification logic into an existing verification toolset.

In Chapter 5 we describe the last essential component of our proposal which consists of a predicate specialization and pruning technique targeting disjunctive predicates and showcase how it can be applied to our verification logic with impressive verification time improvements.

Chapters 6 and 7 conclude the thesis with discussions on related work and possible directions for future research.

Chapter 2

Preliminaries

In this chapter we will describe `SrcLang`, a Java-like programming language and the specification language we will use as the target of our verification solution. The programming language is endowed with rich syntactic constructs which induce complex control flows: it supports exception handling and also multithreaded programming through the presence of fork/join statements and barriers as synchronization mechanisms.

We will also introduce a syntactically simpler core language to which the input language can be reduced and for which we have designed a verification logic targeted in proving correctness of programs with exceptions and barriers. We will define a small-step semantics for the core language. We will also describe a separation logic based specification language with support for capturing various control flow types and barrier related assertions.

We conclude the chapter with a translation from `SrcLang` to the core language followed by a set of optimization rules for the core language, designed to reduce the implementation overhead. These rules are specified at a high-level which facilitate both human understanding and the construction of correctness proofs.

2.1 Source Language

As input language for our system, we consider a Java-like language which we call `SrcLang`. Although we make use of the class hierarchy to define a subtyping relation for exception objects, the treatment of the other object-oriented features, such as instance methods and method overriding, is outside the scope of the current work. We have opted for a first-order imperative language that permits only static methods and single inheritance. These simplifications are orthogonal to our verification goals. We have originally intended for our language to support only exception handling and barrier related features. However, we were pleasantly surprised that other complex features with respect to how control flow is transferred such as the break/continue statements, the try with multiple catch handlers construct, the finally construct, and other more fancy features, such as the multi-return function call ([91]), can be unambiguously expressed in terms of simpler constructs of our core language and thus are easily handled by our verification logic.

We outline the full syntax for the input source language in Figure 2.1. Notice that most of `SrcLang`'s syntactic constructs are straightforward therefore in the rest of the section we elaborate on the slightly less common features like the multi-return function calls and clarify the allowed interactions between concurrency and exception handling in `SrcLang`.

We represent the multi-return function call in our language as $(m\ \overrightarrow{v})$ `with` $\overrightarrow{\lambda v.e}$. Evaluating such a form involves evaluating the inner application, $(m\ \overrightarrow{v})$, in a context with n return points. The first return point is for the context of the call itself. The other $n-1$ return points are captured by return points of the form $\lambda v_2.e_2, \ldots, \lambda v_n.e_n$. If the application eventually returns a value val to a return point of the form $\lambda v_k.e_k$, then v_k is bound to the value val and expression e_k is evaluated in the caller's context. The return construct, `ret`$-i\ e$, specifies that the result of evaluating expression e is to be returned to the i-th return point of the caller.

The second peculiar `SrcLang` language feature are the concurrency related statements: fork/join/barrier. A fork operation, `fork` $(m(\overrightarrow{v}))$ creates a new thread which executes the

P	$::= \overrightarrow{D}\ ;\ \overrightarrow{V}\ ;\ \overrightarrow{B}\ ;\ \overrightarrow{M}$	*program*
V	$::=$ `pred self::pname`$\langle\overrightarrow{v}\rangle \equiv \Phi$ `inv` π	*pred declaration*
B	$::=$ `barrier self::bname`$\langle\overrightarrow{v}\rangle \equiv$ $\bigvee$(`requires` Φ_{pr} `ensures` Φ_{po})	*barrier declaration*
D	$::= class\ c_1\ extends\ c_2\ \{\overrightarrow{t\ f}\}$	*data declaration*
t	$::= c \mid p$	*user or prim. type*
M	$::= t\ m\ [$`with` $n]\ (\overrightarrow{t\ v})[$`throws` $t^+]$ `requires` Φ_{pr} `ensures` $\Phi_{po}\ \{e\}$	*method declaration*
w	$::= v \mid v.f$	*variable or field*
e	$::= v$	*variable*
	$\mid k$	*primitive constant*
	$\mid$ `new` c	*new object*
	$\mid v.f$	*field access*
	$\mid (t)\ e$	*casting*
	$\mid$ `throw` e	*throw exception*
	$\mid$ `break` $[L] \mid$ `ret`$-i\ e$	*break and return*
	$\mid$ `continue` $[L]$	*loop continue*
	$\mid w{:=}e$	*assignment*
	$\mid e_1;\ e_2$	*sequence*
	$\mid (m\ \overrightarrow{v})$ `with` $\overrightarrow{\lambda v.e}$	*multi−return call*
	$\mid \{t\ v;\ e\}$	*local var block*
	$\mid$ `if` e `then` e_1 `else` e_2	*conditional*
	$\mid$ `let` $v{=}e_1$ `in` e_2	*local binding*
	$\mid L : e$	*labelled expression*
	$\mid$ `fork` $(m(\overrightarrow{v}))$	*thread creation*
	$\mid$ `join` (tid)	*thread join*
	$\mid$ `barrier` v	*barrier call*
	$\mid e_1$ `finally` e_2	*finally*
	$\mid$ `try` e `catch` $c_1\ v_1\ e_1$ $[$`catch`$(c_i\ v_i)\ e_i]^{\mathtt{n}}_{\mathtt{i=2}}$	*multiple catch handlers*
	$\mid$ `while` e_1 `requires` Φ_{pr} `ensures` $\Phi_{po}\ \{e_2\}$	*loop*

Figure 2.1: Source Language : `SrcLang`

method m with arguments v. The fork returns the thread identifier of the child thread. Conversely, the `join` (tid) waits until the child thread finishes. Finally `barrier` v blocks the calling thread until all other threads have issued a similar barrier call for barrier v.

With regards to the interaction of exception handling with multithreaded computations the crux of the problem lies in how to minimize the disturbance an exception in one thread has on the other threads. There are two possible approaches: either disallow exceptions to escalate beyond the thread boundary or allow them but in a deterministic fashion, only at specific program points. Java for example, supports both approaches. In Java there are two mechanisms for thread execution, each with its own approach to exception handling: i)providing a run method when extending the $Thread$ class or when implementing the $Runnable$ interface ii) implementing the $Callable$ interface by providing a $call$ method.

In the first case, the run method does not allow a result to be returned, either normal or exceptional. The return type is $void$ and the method header does not allow any checked exceptions to be declared while any unchecked exceptions occurring at runtime are routed automatically to a thread specific $UncaughtExceptionHandler$ handler. The common usage of this method has the threads store their results in a shared resource.

The behaviour in the second case is more refined. However, exceptions still can not be automatically escalated beyond the thread boundary. If a thread has encountered an unhandled exception, its execution finishes without interfering with any of the other threads. However, the result of the computation including the exception can be retrieved by the get method in the $Future$ class. The get method is allowed to throw $ExecutionException$ exceptions which encapsulate the actual exception thrown during the threads execution.

We postulate that both behaviours can be easily handled by our verification logic. However in the rest of the presentation, for simplicity we will adopt the $Runnable$ approach with no exceptions allowed to escape the threads.

`SrcLang` allows for functions (and loops) to be decorated with pre and post conditions which are pairs of formulas expressed in a separation logic specification language described in §2.4. `SrcLang` is also equipped with mechanisms for describing inductive predicates (predicate definitions) and specifying barrier behaviour (user-defined barrier definitions). Barrier defini-

tions are given as sets of pre and post conditions[1]. Unlike SPEC# or ESC/Java, where even specifications for exceptions are captured by a special syntax for exceptional postconditions, we aim for a unified logic that is capable of capturing all kinds of control flow jumps through specialized constraints included in the specification language. The specification constraints allow explicit capturing of control flow jump information, in particular the class of language constructs that generated a given control flow jump. Furthermore we introduce the concept of control flow type to denote classes of such language constructs. Thus the specification constraints in effect capture control flow types. These specifications are verified automatically by our tool.

2.2 Control Flow Hierarchy

Our proposal is based on a novel view of non-local purely *sequential control flow types* [2], in which both normal and abnormal control flows are being handled in a uniform way. We will organise these control flow types into a tree hierarchy, as illustrated in Figure 2.2. The control flow type hierarchy incorporates all the possible control flow types: both the ones pertaining to user-defined exceptions and the predefined flow types. Thus it incorporates all language constructs generating control flow jumps.

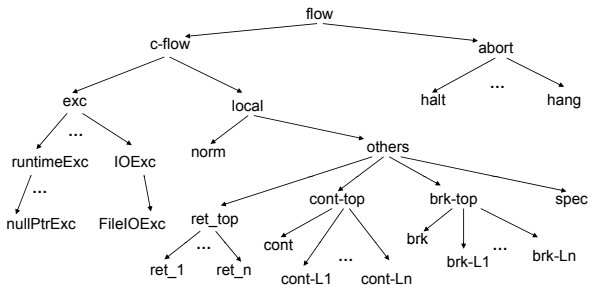


Figure 2.2: A Subtype Hierarchy on Control Flows

[1] barrier definitions will be discussed in detail in Chapter 4

[2] We use the term *sequential control flows* to denote control flows occurring within a thread

Each arrow $c_2 \rightarrow c_1$ denotes a subtyping relation $c_1 <: c_2$. In this tree hierarchy, `exc` captures dynamic control flows due to exceptions, while `local` captures static control flows, such as: `brk` to denote the break out of a loop, `cont` to denote a jump to the beginning of a loop and `ret` to signal a method return (covering also methods with multi-return options [91]). The control flow `norm` for normal execution is a special instance of this static control flow that will be transferred to the default next instruction for execution. A key feature of static control flows is that they can be efficiently implemented as local control transfers through either direct or indirect jumps. On the other hand, dynamic control flows from exceptions would involve non-local transfer of control via catch handlers present in the function calling hierarchy at runtime. All control flow types are subtypes of $\top$. All control flows that can be 'caught' by our language are placed under the `c-flow` category, while the `abort` category denotes control flows that cannot be caught. `abort` includes program errors, program termination by `halt`, and non-termination by `hang`. The latter could, in principle, be used by our language to reason about non-terminating behaviors but this aspect is not addressed in this work.

The use of a tree hierarchy, rather than a lattice, for our control flow is important for finite abstraction. A useful property of the tree hierarchy is that every two nodes of the tree, say $c1$ and $c2$, are either *mutually-exclusive*, as denoted by $\forall c \cdot (c <: c1 \implies \neg(c <: c2))$, or they *overlap*, as denoted by $c1 <: c2 \vee c2 <: c1$. This property is helpful for formal reasoning since we can statically determine disjointedness of two flow types with the help of only their subtyping relation. This decision allows us to build finite set abstractions required to model multiple flows. While exceptions in Java are implicitly organised as a hierarchical tree, the previous use of effects-based type system does not require this finitary abstraction property.

Although other systems enforce the restriction that the try-catch construct applies only to exceptional flows, our unified view on control flows allows us to generalize the try-catch construct across the entire domain of control flow types. This domain extension permits a much more streamlined verification mechanism.

2.3 Core Language

In this section we introduce a concise language, `Core−U` , to which `SrcLang` can be translated. `Core−U` has the benefit of allowing a much simpler formulation of the verification rules. In designing `Core−U` we aimed for:

- *Unified Constructs :* To minimise on language features, our language should unify together constructs that have similar functionality, where possible.
- *Syntactically Minimal :* To keep our language small, we shall aim for fewer and simpler constructs, where possible. This can make our language easier to formalise and analyse.
- *Expressively Maximal :* We strive to provide language constructs that are as general as possible, to allow them to be used in more scenarios. The acid test is whether the language can succinctly encode more advanced language features.
- *Computationally Positive :* The language should not hinder efficient compilation. Firstly, it supports a set of optimization rules. Secondly, intermediate steps used to make the language easier to analyse can be directly removed later by efficient compilation.

The unified view of control flows presented in §2.2 lies at the heart of our core language. An unexpected benefit is that our core language with exceptions is *as small as* the corresponding core language without exceptions. Designing analyses and optimizations for the core language is therefore much simpler than it would be for the source language.

2.3.1 Syntax

We will detail the key `Core−U` constructs meant to allow us to take full advantage of the complex control flow hierarchy introduced above. Also, a list of the syntactic constructs of `Core−U` is given in Figure 2.3.

In previous core languages with exceptions for Java, such as [64] and [60], a variable v would return a value with normal flow, while `throw` v would invoke an exceptional flow based

P	$::= \overrightarrow{D}\ ;\ \overrightarrow{V}\ ;\ \overrightarrow{B}; \overrightarrow{M}$	*program*
D	$::= \texttt{class}\ c_1\ \texttt{extends}\ c_2\ \{\overrightarrow{t\ v}\}$	*data declaration*
V	$::= \texttt{pred self::pname}\langle\overrightarrow{v}\rangle \equiv \Phi\ \texttt{inv}\ \pi$	*pred declaration*
B	$::= \texttt{barrier self::bname}\langle\overrightarrow{v}\rangle \equiv \bigvee(\texttt{requires}\ \Phi_{pr}\ \texttt{ensures}\ \Phi_{po})$	*barrier declaration*
ft	$::= c \mid predef\_flows$	*flow types*
M	$::= t\ m(\overrightarrow{[\texttt{ref}]\ t\ v})\ \texttt{requires}\ \Phi_{pr}\ \texttt{ensures}\ \Phi_{po}\ \{e\}$	*method decl*
e	$::= fn\#x$	*output* $(flow\&value)$
	$\mid v.f$	*field access*
	$\mid w{:=}v$	*assignment*
	$\mid m(\overrightarrow{v})$	*method call*
	$\mid \{t\ v;\ e\}$	*local var block*
	$\mid \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2$	*conditional*
	$\mid \texttt{fork}\ (m(\overrightarrow{v}))$	*thread creation*
	$\mid \texttt{join}\ (tid)$	*thread join*
	$\mid \texttt{barrier}\ v$	*barrier call*
	$\mid \texttt{try}\ e_1\ \texttt{catch}\ (ft[@v_1]\ v_2)\ e_2$	*catch handler*
fn	$::= \texttt{Ex}(ft) \mid fv \mid \texttt{ty}(v) \mid v.1$	*flow*
t	$::= c \mid p$	*user or prim. type*
w	$::= v \mid v.f$	$var/field$
x	$::= v \mid k \mid \texttt{new}\ c \mid (fv, v) \mid v.2$	*basic values* $(vars, consts, \ldots)$

Figure 2.3: Core Language : Core−U

on the exception object in v. In our approach, we unify both these constructs with the $ft\#v$ statement which has the effect of explicitly generating a control flow of type ft with return value v. With this construct normal flow is realised by $\texttt{norm}\#v$ while exceptions of the same type as the object indicated by v may be thrown using $\texttt{ty}(v)\#v$ which also ensures that the returned value is the exception object. The type of a raised exception object v is captured as its control flow. The function $\texttt{ty}(v)$ returns the runtime type of an exception object pointed by v. In case $v{=}\texttt{null}$, it returns a special nullPtrExc flow type. This unified construct is a generalization of the exception mechanism used in Java since we allow each flow type to be

unrelated to the type of the value being thrown. For example, we may use `exc`$\#$`13` to raise an exception with integer value `13`. This is not *directly* expressible in Java, though it could be mimicked by a user-defined exception that embeds an integer value.

The core language allows the embedding of control flows directly as values, by allowing a pair of control flow and its value (fv, v) to be specified. With this notation, we make explicit the distinction between the control flow corresponding to an exception and its return value. Furthermore, we can save each exception and its output value as an embedded pair that could be later re-thrown. Operations `v.1` and `v.2` are used to access the control flow and the value, respectively, from an embedded pair in `v`.

Another major construct of our language is a try-catch mechanism of the form:
`try` e_1 `catch` $((c@fv)\#v)$ e_2 which specifies a control flow c and two bound variables to capture a control flow type fv and its thrown value v, provided that $fv{<:}c$. This try-catch construct is more general than that used in Java since it can capture not only exceptional flow, but also normal flow and other abnormal control flows due to `break`, `continue` and `return` statements that can be translated to the corresponding control flows. As a pleasant surprise, the usual sequential composition $e_1; e_2$ is now a syntactic sugar for `try` e_1 `catch` $((\texttt{norm}@\_)\#\_)$ e_2 whereby each $\_$ denotes a distinct anonymous bound variable. Although this desugaring simplifies the verification process by making explicit the control flow paths, in this presentation we will still use the $e_1; e_2$ for conciseness.

With the $ft\#v$ and `try` e_1 `catch` $((c@fv)\#v)$ e_2 statements it is easy to reduce various traditional statements to one of the two expressions:

x	$\Rightarrow$	`norm`$\#x$	`return` x	$\Rightarrow$	`ret`$\#x$
`continue`	$\Rightarrow$	`cont`$\#()$	`break` L	$\Rightarrow$	`brk`$-L\#()$
`throw` v	$\Rightarrow$	`ty`$(v)\#v$	`continue` L	$\Rightarrow$	`cont`$-L\#()$

While the previous rewritings are intuitive, rewriting a labeled while loop to use our structures is a bit more involved as `Core`$-$`U` makes explicit several control flow jumps tipically hidden

in common languages. For example the destination points of break and continue control flow jumps become try/catch statements in Core−U .

$$L : \texttt{while}\ e_1\ \{e_2\} \quad \Rightarrow \quad \left\{ \begin{array}{l} \texttt{try}\ \{\texttt{bool}\ v;\ v{:=}e_1; \\ \qquad\qquad \texttt{while}\ v\ \{ \\ \qquad\qquad\quad \texttt{try}\ e_2\ \texttt{catch cont norm\#}() \\ \qquad\qquad\qquad\quad \texttt{catch}\ (\texttt{cont}{-}L)\ \texttt{norm\#}(); \\ \qquad\qquad\quad v{:=}e_1\} \\ \qquad \texttt{catch}\ (\texttt{brk})\ \texttt{norm\#}() \\ \qquad \texttt{catch}\ (\texttt{brk}{-}L)\ \texttt{norm\#}() \end{array} \right.$$

Note that while Core−U may appear inefficient due to an apparent need to unwind through a nested series of handlers, we emphasize that our primary goal is to make program codes easier to analyse. For actual execution, we could use compilation techniques to ensure that every static control flow is efficiently implemented by either a direct or indirect jump into its corresponding handler code. Moreover, we could also use a similar optimization to efficiently implement some of the dynamic control flows. Under suitable conditions, we can use an optimization rule, called *throw-catch linking*, that could directly link a throw operation for an exception with its intended handler through a parameterized jump (see § 2.5.2 later).

We point out that the fork/join/barrier statements carry forward from SrcLang: a fork operation, fork $(m(\overrightarrow{v}))$ creates a new thread which executes the method m with arguments v returning the thread identifier of the child thread, the join (tid) waits until the child thread finishes while a barrier v call blocks the calling thread until all other threads have issued a similar barrier call for barrier v.

2.3.2 Semantic Model

In this section we will introduce an erased operational semantics for Core−U . We use erased semantics to denote language semantics which use a machine model with few or no virtual

components, one that is close to the *on-chip implementation*.

Note that we will use Γ to denote the code memory, basically Γ is a function from function names to function definitions. For simplicity of the presentation we will elide adding Γ to the program state.

We use σ to model a thread state as a triple of stack s, heap h, barrier map b. Local variables and other meta variables live in the stack s, which is a function from variable names to values (either a constant, an address or a pair of control flow type and value). In contrast, a heap h contains the locations shared between threads; heaps are partial functions from addresses to objects. We use the notation with $c[f_1 \mapsto \nu_1, .., f_n \mapsto \nu_n]$ for an object value of data type c where $\nu_1, .., \nu_n$ are current values of the corresponding fields $f_1, .., f_n$. We also equip heaps with a distinguished location, called the *break*, that tracks the boundary between allocated and unallocated locations. The break lets us provide semantics for the $x := \texttt{new}\ e$ instruction in a natural way by setting x equal to the current break and then incrementing the break. Since threads share a common break, there is a covert communication channel (one thread can observe when another thread is allocating memory); however the existence of this channel is a small price to pay for avoiding the necessity of a concurrent garbage collector.

Finally, in modeling barriers, we associate to each barrier a pair of integers: the number of threads that are synchronized by the barrier and the number of threads that are currently waiting. The barrier map b is a partial function from addresses of barriers to pairs of positive integers.

In the style of Hobor *et al.* [51], our operational semantics is divided into three parts: purely sequential, which executes all of the instructions, except for barrier, fork and join, in a thread-local manner; concurrent, which manages thread scheduling and handles the barrier, fork, join instructions; and oracle, which provides a pseudosequential view of the concurrent machine to enable simple proofs of the sequential Hoare rules.

For the purely sequential semantics, the form of the step judgment is $(\sigma, c) \mapsto (\sigma', c')$, where σ is the thread state and c is a command of our language with the observation that if the

step relation reaches a barrier or fork or join call then it simply gets stuck. Later in this section we will elaborate on the purely sequential semantics, however we will start with the concurrent and oracle semantics.

Concurrent Semantics

We define the notion of a *concurrent state* as a four-tuple $(\Omega, thds, h, b)$ of scheduler Ω (modeled as a list of natural numbers representing the thread identifiers), a list of *thread*s $thds$, heap h, and the barrier map b. The scheduler encodes the order in which threads get execution rights: a scheduler $\Omega = 5, 3, \ldots$ would have thread 5 execute until it blocks on a join or barrier call, then pass control to thread 3 which will execute until it blocks and so on. A thread contains its stack (the state store s) and a *concurrent control*, which is either $\mathsf{Running}(c)$, meaning the thread is available to run command c, or $\mathsf{Waiting}(bn, c)$, meaning that the thread is currently waiting on barrier bn ; after the barrier call the thread will resume running with command c. Before we run a thread we transfer the heap and barrier map into the thread. When we suspend the thread we remove the heap and barrier map and transfer it to the next thread. The concurrent step relation has the form $(\Omega, thds, h, b) \rightsquigarrow (\Omega', thds', h', b')$. It has only six cases; it relies on the CStep-Seq case to run all of the sequential commands:

$$\frac{thds[i] = (s, \mathsf{Running}(c)) \qquad \big((s, h, b), c\big) \mapsto \big((s', h', b), c'\big) \qquad thds' = [i \mapsto (s', \mathsf{Running}(c'))]thds}{(i :: \Omega, thds, h, b) \rightsquigarrow (\Omega, thds', h', b)}\ \text{CStep-Seq}$$

That is, the thread whose thread id is at the head of the scheduler is selected to run. Before the sequential step relation is applied to the chosen thread, the heap and barrier map are transferred into the thread. If the command c is a barrier, fork or join call then the sequential relation will not be able to run and so the CStep-Seq relation will not hold; otherwise the sequential step relation will be able to handle any command. After a sequential step is taken, the heap and

barrier map are taken out of the thread state and reinsert the modified sequential state into the thread list. Since we quantify over all schedulers and our language does not have input/output, it is sufficient to utilize a non-preemptive scheduler.

For example, given a machine with one thread with some stack s and running the expression $v := 1; e_1$ with heap h and barriers b, one CStep-Seq step is:

$$\big(1 :: \Omega, \big[1 \mapsto (s, \mathit{Running}(v := 1; e_1))\big], h, b\big) \rightsquigarrow \big(\Omega, \big[1 \mapsto ([v \to 1]s, \mathit{Running}(e_1))\big], h, b\big)$$

The second case of the concurrent step relation handles the case when a thread has reached the last instruction, which must be a `skip`:

$$\frac{\mathit{thds}[i] = (s, \mathsf{Running}(\texttt{skip}))}{(i :: \Omega, \mathit{thds}, h, b) \rightsquigarrow (\Omega, \mathit{thds}, h, b)}\ \text{CStep-Exit}$$

When the end on a thread is reached a context switch to the next thread occurs.

The interesting cases occur when the instruction for the running thread is a barrier or fork or join call ; here the CStep-Seq rule does not apply. The concurrent semantics handles the barrier call directly via the next two cases of the step relation. First, if a thread executes a barrier but is not the last thread to do so:

$$\frac{\begin{array}{c}\mathit{thds}[i] = (s,\ (\mathsf{Running}\ (\texttt{barrier}\ bn;\ c))) \\ \mathit{thds}' = [i \to (s,\ (\mathsf{Waiting}(bn, c)))]\ \mathit{thds} \\ b[bn] = (\mathit{waiting}_{cnt}, \mathit{total}_{cnt}) \\ b' = [bn \to (\mathit{waiting}_{cnt} + 1, \mathit{total}_{cnt})]\ b \\ \mathit{waiting}_{cnt} + 1 < \mathit{total}_{cnt}\end{array}}{((i :: \Omega), \mathit{thds}, h, b) \rightsquigarrow (\Omega, \mathit{thds}', h, b')}\ \text{CStep-Suspend}$$

After it increments the waiting threads counter of the bn barrier, CStep-Suspend checks to see if the barrier is full by comparing the waiting threads with the total expected number. Because

the barrier is not full, the thread is suspended and the context is switched.

In the case of two threads, a barrier call on one would trigger the following application of the CStep-Suspend step:

$$\big((1 :: \Omega),\, \big[1{\mapsto}(s_1, \mathsf{Running}(\texttt{barrier}\ \ 1; c_1))\ ;\ 2{\mapsto}(s_2, \mathsf{Running}(c_2))\big],\, h,\, \big[1{\mapsto}(0,2)\big]\big)$$
$$\rightsquigarrow$$
$$\big(\Omega,\, \big[1{\mapsto}(s_1, \mathsf{Waiting}(1, c_1))\ ;\ 2{\mapsto}(s_2, \mathsf{Running}(c_2))\big],\, h,\, \big[1{\mapsto}(1,2)\big]\big)$$

If the barrier is ready, then instead of using the CStep-Suspend case of the concurrent step relation, the CStep-Release case is applied:

$$\frac{\begin{array}{c} thds[i] = (s,\ (\mathsf{Running}\ (\texttt{barrier}\ bn;\ c))) \\ thds' = [i \rightarrow (s,\ (\mathsf{Waiting}(bn, c)))]\ thds \\ b[bn] = (waiting_{cnt}, total_{cnt}) \qquad b' = [bn \rightarrow (0, total_{cnt})]\ b \\ waiting_{cnt} + 1 = total_{cnt} \\ \mathsf{transition\_threads}\ \ (bn,\ thds') = thds'' \end{array}}{((i :: \Omega), thds, h, b) \rightsquigarrow (\Omega, thds'', h, b')}\ \text{CStep-Release}$$

The first requirement of CStep-Release is exactly the same as CStep-Suspend: the thread must be suspended. However, now all of the threads have arrived at the barrier and so it is ready. The waiting threads counter is reset. Finally, the suspended threads are simultaneously resumed by the transition_threads predicate which changes to the $Running$ state all threads that are currently in the $Waiting$ state for barrier bn. If in the previous example the second thread also executes a barrier call for barrier number 1 then:

$$\big((2 :: \Omega),\, \big[1{\mapsto}(s_1, \mathsf{Waiting}(1, c_1))\ ;\ 2{\mapsto}(s_2, \mathsf{Running}(\texttt{barrier}\ 1; c_2))\big],\, h,\, \big[1{\mapsto}(1,2)\big]\big)$$
$$\rightsquigarrow$$
$$\big(\Omega,\, \big[1{\mapsto}(s_1, \mathsf{Running}\ c_1)\ ;\ 2{\mapsto}(s_2, \mathsf{Running}\ c_2)\big],\, h,\, \big[1{\mapsto}(0,2)\big]\big)$$

The last two cases describe the fork and join. When encountering a fork operation, `fork` $(m(\overrightarrow{v}))$, a completely new thread is generated, the stack is initialized by copying the values corresponding to the method arguments from the stack of the calling thread, the body of method m is retrieved from program memory Γ, and the thread is set to execute the method body.

$$\frac{\begin{array}{c} thds[i] = (s,\ (\mathsf{Running}\ (\texttt{fork}\ (m(\overrightarrow{v}));\ c))) \\ \Gamma[m] = void\ m(\overrightarrow{t\ w})\ \{e\} \qquad tid = fresh\_tid() \\ thds' = [i \rightarrow ([\texttt{res} \rightarrow tid]s,\ (\mathsf{Running}\ c))]\ thds \\ ns[\overrightarrow{w}] = s[\overrightarrow{v}] \qquad thds'' = [tid \rightarrow (ns,\ (\mathsf{Running}\ e))]\ thds' \end{array}}{((i :: \Omega), h, thds, b) \rightsquigarrow (\Omega, h, thds'', b)}\ \text{CStep-Fork}$$

Note that we enforce the Java $Runnable$ approach by allowing as fork arguments only functions without a return value. Also note that we rely on a special variable `res` to convey the operation result, the id of the newly created thread. Furthermore, the assumption that the remaining scheduler Ω will contain the newly created tid is acceptable as the domain of the scheduler list is not restricted to some statically defined thread ids, it is the set of natural numbers.

Lastly, in order to execute a join on a given thread id the semantics require that the thread be finished, in the (Running `skip`) state.

$$\frac{\begin{array}{c} thds[i] = (s,\ (\mathsf{Running}\ (\texttt{join}\ (tid);\ c))) \\ thds[tid] = (s',\ (\mathsf{Running}\ \texttt{skip})) \\ thds' = [i \rightarrow (s,\ (\mathsf{Running}\ c)))]thds \\ thds'' = free(tid, thds') \end{array}}{((i :: \Omega), h, thds, b) \rightsquigarrow (\Omega, h, thds'', b)}\ \text{CStep-Join}$$

Given two threads, the first having finished its execution when the second executes a join 1

call then the CStep-Join steps results in the following transition:

$$\big((2 :: \Omega), \big[1{\mapsto}(s_1, \mathsf{Running}\ \texttt{skip})\ ;\ 2{\mapsto}(s_2, \mathsf{Running}\ (\texttt{join}\ (1); c_2))\big], h, b\big)$$
$$\rightsquigarrow$$
$$\big(\Omega, \big[2{\mapsto}(s_2, \mathsf{Running}\ c_2)\big], h, b\big)$$

Oracle Semantics

We denote the oracle semantic step by $(\sigma, o, c) \mapsto (\sigma', o', c')$. Here the sequential state σ and command c are exactly the same as in the purely sequential step. The new parameter o is an oracle, a kind of box containing "the rest" of the concurrent machine—that is, o contains a scheduler and a list of other threads. The oracle semantics encapsulates the concurrent and purely sequential semantics in a "sequential looking" semantics thus providing a clean abstraction of the concurrent machine.

The oracle semantics behaves exactly the same way as the purely sequential semantics on all of the instructions except for the barrier, fork or join calls, with the oracle o being passed through unchanged. That is to say:

$$\frac{\big(\sigma, c\big)\ \mapsto\ \big(\sigma', c'\big)}{\big(\sigma, o, c\big)\ \mapsto\ \big(\sigma', o, c'\big)}\ \text{os-seq}$$

When the oracle semantics reaches a barrier, fork or join instruction, it relies on the consult operation to consult the oracle o to determine the state of the machine after the instruction:

$$\frac{\begin{array}{c} d = \texttt{barrier}\ \mathsf{bn}\ \ \vee\ \ d = \texttt{fork}\ (m(\overrightarrow{v}))\ \ \vee\ \ d = \texttt{join}\ (tid) \\ \mathsf{consult}(h, b, o) = (h', b', o') \end{array}}{\big((s, h, b), o, d; c\big)\ \mapsto\ \big((s, h', b'), o', c\big)}\ \text{os-consult}$$

The consult operation would either execute the operation according to the concurrent semantics or if other threads need to be waited for, the consult operation unpacks the concurrent machine

stored in o and runs all of the other threads until control returns to the original thread. Consult then returns the current h' and b' (that resulted from the barrier call, fork or join) and repackages the concurrent machine into the new oracle o'.

Purely Sequential Semantics

Here we provide a description of Core−U 's purely sequential semantic. Note that the language semantic reduces expressions to final values of the form $(fn\#a)$ where constant fn denotes the current flow type while constant a denotes the result of the computation, either a value (constant or address) or a pair (fn_1, a_1) to embed a control flow fn_1 with another value a_1. The type of each final value can be obtained by a semantic function *type*(a). Syntactically $a ::= k \mid l \mid (fn, a)$.

For the dynamic semantics to follow through, we have introduced an intermediate construct: $\texttt{BLK}(\{\overrightarrow{v}\}, e_1)$ where e_1 denotes a residual code of the current block. This new construct is used for handling try-catch constructs, method calls and local blocks. Its main purpose is to provide a lexical scope for local variables that are removed once the expression has been completely evaluated.

The full set of transitions is given in Figure 2.4. Take note that, following the translation to Core−U , $v.f$ is exception-free, meaning that if v was null, a nullPtrExc exception would have been previously raised. Consequently, our rules for $v.f$ and $v_1.f{:=}v_2$ do not test for nullness. In the rules, $s(v)$ retrieves the value of variable v present on the stack using $\texttt{lookup}(s, v)$. We also provide an overloaded function $s(ft)$ for $ft ::= c \mid \texttt{ty}(v) \mid fv$ which is defined as $s(c){=}c$ and $s(\texttt{ty}(v)){=}$*type*$(\texttt{lookup}(s, v))$ and $s(fv){=}\texttt{lookup}(s, fv)$. A reminder that the sequence operation $e_1; e_2$ used in the semantic steps for the do-while construct is just syntactic sugar for try e_1 catch norm e_2. Take note that the symbol $\perp$, appearing in the rules, stands for uninitialized.

The semantic steps for local variable declaration, method call and try catch, use the $newid()$ function to return a fresh identifier, and the $[u'/u]$ notation to represent the substitution of u by

$$\big((s,h,b), fn\#v\big)\mapsto\big((s,h,b), s(fn)\#s(v)\big)$$

$$\big((s,h,b), fn\#(fv,v)\big)\mapsto\big((s,h,b), s(fn)\#(s(fv),s(v))\big)$$

$$\big((s,h,b), v.f\big)\mapsto\big((s,h,b), \texttt{norm}\#(h(s(v)).f)\big)$$

$$\big((s,h,b), fn\#a\big)\mapsto\big((s,h,b), s(fn)\#a\big)$$

$$\big((s,h,b), v_1{:=}v_2\big)\mapsto\big((s[v_1\mapsto s(v_2)],h,b),\ \texttt{norm}\#()\big)$$

$$\big((s,h,b), v_1.f{:=}v_2\big)\mapsto\big((s,h[s(v_1).f\mapsto s(v_2)],b),\ \texttt{norm}\#()\big)$$

$$\frac{l = inc\_break(h)}{\big((s,h,b), fn\#\texttt{new}\ c\big)\mapsto\big((s,[l\mapsto c(\overrightarrow{\perp})]h,b),\ s(fn)\#l\big)}$$

$$\frac{\begin{array}{c}u{=}newid()\\ s' = s{+}[u\mapsto\perp]\\ e' = \texttt{BLK}(\{u\}, e[u/v])\end{array}}{\big((s,h,b),\ \{t\ v; e\}\big)\mapsto\big((s',h,b),\ e'\big)} \qquad \frac{\begin{array}{c}c_1{<:}c\\ u{=}newid()\quad fu{=}newid()\\ s' = s{+}[fu\mapsto c_1, u\mapsto a_1]\\ e' = \texttt{BLK}(\{fu,u\}, e_2[u/v, fu/fv])\end{array}}{\begin{array}{r}\big((s,h,b), \texttt{try}\ c_1\#a_1\ \texttt{catch}\ c@fv\#v\ e_2\big)\\ \mapsto\big((s',h,b),\ e'\big)\end{array}}$$

$$\frac{\neg(c_1{<:}c)}{\big((s,h,b), \texttt{try}\ c_1\#a_1\ \texttt{catch}\ c@fv\#v\ e_2\big)\mapsto\big((s,h,b), c_1\#a_1\big)}$$

$$\frac{\big((s,h,b),\ e\big)\mapsto\big((s_1,h_1,b_1),\ e_1\big)}{\big((s,h,b), \texttt{try}\ e\ \texttt{catch}\ c@fv\#v\ e_2\big)\mapsto\big((s_1,h_1,b_1), \texttt{try}\ e_1\ \texttt{catch}\ c@fv\#v\ e_2\big)}$$

$$\frac{\texttt{if}\ s(v)\ \texttt{then}\ e' = e_1\ \texttt{else}\ e' = e_2}{\big((s,h,b), \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2\big)\mapsto\big((s,h,b), e'\big)}$$

$$\frac{\begin{array}{c}t_0\ mn\ (\overrightarrow{t\ u})\ \{e\}\quad \overrightarrow{u'{=}newid()}\\ s' = s{+}[\overrightarrow{u'\mapsto s(v)}]\\ e' = \texttt{BLK}(\{\overrightarrow{u}'\}, e[\overrightarrow{u'/u}])\end{array}}{\big((s,h,b), mn(\overrightarrow{v})\big)\mapsto\big((s',h,b),\ e'\big)} \qquad \frac{\begin{array}{c}\big((s,h,b),\ e\big)\mapsto\big((s_1,h_1,b_1), e_1\big)\\ e' = \texttt{BLK}(\{\overrightarrow{v}\}, e_1)\end{array}}{\big((s,h,b), \texttt{BLK}(\{\overrightarrow{v}\}, e)\big)\mapsto\big((s_1,h_1,b_1),\ e'\big)}$$

$$\big((s,h,b), \texttt{BLK}(\{\overrightarrow{v}\}, c\#a)\big)\mapsto\big((s{-}\{\overrightarrow{v}\},h,b), c\#a\big)$$

Figure 2.4: Small-Step Semantics

u'. In order to avoid dynamic binding, every variable whose binding is added to the stack is substituted by a fresh identifier. For instance, in the case of the method call, after performing the renaming of the callee formal parameters, the mappings for the fresh identifiers are tem-

porarily added to the stack, s. These bindings will be removed after the evaluation of the callee body.

2.4 Specification Language

Predicate	spred	::=	$[\mathsf{self}::]c\langle\overrightarrow{v}\rangle \equiv \Phi\ [\mathbf{inv}\ (\pi, \overrightarrow{v})]$
Barrier	bar	::=	$[\mathsf{self}::]\mathtt{bname}\langle\overrightarrow{v}\rangle \equiv \bigvee(\mathtt{requires}\ \Phi_{pre}\ \mathtt{ensures}\ \Phi_{post})$
Formula	Φ	::=	$\bigvee_i(\exists\overrightarrow{v_i}\cdot(\kappa_i \wedge \pi_i))$ $\quad\Delta ::= \bigvee_i(\kappa_i \wedge \pi_i \wedge \beta_i)$
Heap form.	κ	::=	$\mathtt{emp} \mid v::c^{v_f}\langle\overrightarrow{v}\rangle \mid \kappa * \kappa$
Pure form.	π	::=	$\gamma \wedge \phi \wedge \mu \wedge \tau$
Flow types	ft	::=	$c \mid predef\_flow$
Flow form.	β	::=	$fv{=}ft \mid fv_1{=}fv_2$
Current flow	μ	::=	$\mathtt{flow} = fset \quad fset = \mathtt{Ex}(ft) \mid ft - \{ft_1, .., ft_n\}$
Frac form.	τ	::=	$v_f \oplus v_f = v_f \mid v = \chi \mid \tau \wedge \tau$
Pointer form.	γ	::=	$v = v \mid v = null \mid v \neq v \mid v \neq null \mid \gamma \wedge \gamma$
Presburger arith.	ϕ	::=	$\mathsf{arith} \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi \mid \exists v \cdot \phi \mid \forall v \cdot \phi$
	arith	::=	$\mathsf{a} = \mathsf{a} \mid \mathsf{a} \neq \mathsf{a} \mid \mathsf{a} < \mathsf{a} \mid \mathsf{a} \leq \mathsf{a}$
	a	::=	$z \mid v \mid z \times \mathsf{a} \mid \mathsf{a} + \mathsf{a} \mid -\mathsf{a} \mid max(\mathsf{a}, \mathsf{a}) \mid min(\mathsf{a}, \mathsf{a})$

where v, w *are variable names*;
c *is a data type name or a predicate name or barrier name*;
z *is an integer constant*;
τ *represents the fractional permission constraints*
χ *represents constant fractional shares*

Figure 2.5: Specification Language

Figure 2.5 outlines the syntax of our specification language, a rich separation logic based language that supports user-defined disjunctive predicates together with constraints over control flow types(for explicitly capturing flow types) and other various domains like linear and non linear arithmetics. It also includes support for tracking access permissions to shared resources in the form of partial ownership annotations, plus a novel specification feature for capturing barrier behaviour and for homogeneous reasoning about both heap and barrier resources.

The specification language is equipped with an expressive means for encoding complex abstract data structure properties in the form of user-defined inductive predicates. Predicates are defined as separation formulae that describe the shape of data structures and associated properties (*e.g.*, list length, tree height, and bag of values contained in a list). Predicate defini-

tions make use of the self special variable to denote the pointer to the root of the data structures they abstract. Predicate invariants can increase the precision of the verification (*e.g.*, length $\geq$ 0). An invariant for a predicate instance has two parts: a pure formula describing arithmetic constraints on the arguments and the set of non null pointer arguments (*e.g.* the outward pointer for a list segment). For example, given a node data structure defined as:

$$\texttt{data node} \{ \texttt{ int } val \texttt{ ; node } next \}$$

It is possible to express a predicate abstraction for a list segment constructed of $node$ instances. The following predicate captures both the shape and the size, n, of the segment and the outward pointer, p:

$$\begin{array}{lcll} \texttt{pred self} :: \texttt{lseg}\langle n, p\rangle & \equiv & \texttt{self} = p \wedge n = 0 & \\ & & \vee\ \texttt{self} :: \texttt{node}\langle \_, r\rangle * r :: \texttt{lseg}\langle n-1, p\rangle & \texttt{inv } n \geq 0,\ p. \end{array}$$

Furthermore, the user defined predicates can be incorporated in any formula. For example, given an append method that takes a null terminated list segment and appends another list segment, the pre/post conditions can be expressed using the lseg predicate as follows:

$$\begin{array}{l} \texttt{void } append\ (\texttt{node } x,\ \texttt{node } y) \\ \quad \texttt{requires } x :: \texttt{lseg}\langle n, null\rangle * y :: \texttt{lseg}\langle m, p\rangle \qquad \texttt{ensures } x :: \texttt{lseg}\langle n+m, p\rangle \end{array}$$

Note that the formulae describing the predicate body and the method specifications contain both heap and arithmetic constraints. In the following paragraphs we will expand further on formulae structure and give more details on each component.

In order to be able to reason about barrier behaviour we equip the specification language with barrier definitions. Akin to predicate definitions which specify properties of abstract data structures associated with concrete memory resources, barrier definitions are a novel mechanism for encoding the complex resource invariants associated with barrier usage. Barrier

definitions consist of sets of formula pairs, acting as pre/post conditions. Barrier definitions and predicate definitions are similar in that they describe properties associated with resources, heaps and barriers, however, they differ in that predicate definitions describe a property of the current state while barrier definitions describe the behaviour of a barrier call in that program state. For a clearer presentation we defer the in-depth presentation of barrier definitions and their usage until Chapter 4.

Secondly, as both programming languages presented earlier handle two classes of resources: heap and barriers, we propose to extend the syntax typically used for heap specification to cover barrier resources as well. In our language, each disjunct within a disjunctive formula Φ consists of a resource describing subformula κ, and π, the pure subformula that represents the resource-independent part of the formula with constraints over several domains: arithmetic, bag/list, etc.

κ describes the resource footprint which is composed of $*$-separated resource nodes written as $p{::}a^{v_f}\langle\overrightarrow{v}\rangle$ and interpreted as the current thread owns fraction v_f of a resource of type a pointed to by variable p with arguments $\overrightarrow{v}$. The v_f argument is optional. If none is specified, it is implied that the resource is fully owned. Depending on a, the node can be either an instance of a user-defined data structure, in which case the $\overrightarrow{v}$ arguments correspond to the structure fields, or it can be an instance of a user-defined predicate, in which case $\overrightarrow{v}$ correspond to the predicate arguments, or it could be a barrier, in which case the first argument, v_1, gives an indication of the barrier state while the rest $v_{2...}$ denote the shared variables guarded by the barrier.

Bornat *et al*. introduced the concept of *fractional share* to handle the necessary accounting of fractional ownership [14]. *Shares* form a disjoint multi-unit separation algebra [1]; a *full share* (complete ownership of a resource) can be broken into various *partial shares*; these shares can then be rejoined into the full share. The *empty share* is the identity for shares. With respect

[1] Dockins *et al*. described in [28] a disjoint multi-unit separation algebra (hereafter just "DSA") . Briefly, a DSA is a set S and an associated three-place partial *join relation* $\oplus$, written $x \oplus y = z$, such that $\oplus$ is commutative, associative, a function ($x \oplus y = z_1 \wedge x \oplus y = z_2 \Rightarrow z_1 = z_2$), cancelative ($x_1 \oplus y = z \wedge x_2 \oplus y = z \Rightarrow x_1 = x_2$), has multiple units ($\forall x.\ \exists u_x.\ x \oplus u_x = x$) and exhibits disjointedness ($x \oplus x = y \Rightarrow x = y$).

to resource ownership, we often need non-empty (strictly *positive*) shares. We require the full share to modify a resource however we only require a positive share to read from one. There is no danger of a data race even though we do not require the full share to read: if a thread has a positive share of some location, no other thread can have a full share for the same location.

Here we use the following notations : the full share ■; two **distinct** nonempty partial shares, ◩ and ◪, and the empty share □. The key point is that ◩ $\oplus$ ◪ = ■. Also we will use χ to denote generic constants of this domain. Also note that if a denotes a predicate, then the notation $p{::}a^{v_f}\langle\overrightarrow{v}\rangle$ indicates that p points to a memory region whose shape is described by the definition of a and that all heap nodes abstracted by this predicate instance are owned with permission v_f.

We allow the pure part π to capture pointer equalities/inequalities (γ), linear arithmetic (ϕ), a share subformula (τ) and the current flow constraint (μ). The share subformula contains constraints expressing share facts of the form $v_{f1} \oplus v_{f2} = v_{f3}$, $v_1 = v_2$ or $v = \chi$. The current flow constraint μ expresses the control flow type associated with the respective program state. It is captured by a special variable `flow` whose domain is the set of subtrees in the control flow hierarchy. With this hierarchy, we can be as precise as required for verifying different exception safety guarantees: the possible values of control flow are either $Ex(ft)$ to denote an exact control flow type (not including its subclasses), or $ft - \{ft_1, .., ft_n\}$ to denote a control flow from ft but not from subclasses $ft_1, .., ft_n$.

Note that for simplicity we also introduce Δ, to denote a composite formula that is enhanced with an extra pure component β to capture the bindings for flow type variables other than `flow`. In Chapter 3 we will show how such bindings are generated and how are they handled.

Lastly, each variable in our specification logic may be expressed in either primed form (e.g. v') or unprimed form (e.g. v). The former denotes the latest value of the corresponding variable, while the latter denotes the original value of the same variable. When used in the postcondition, they denote state changes that occur for parameters that are being passed by

reference.

2.4.1 Semantic Model

We interpret the assertions in the specification language with respect to the state model proposed for `Core−U` : states σ are triples of a store, heap, and barrier map ($\sigma = (s, h, b)$) with the addition that heaps are partial functions to pairs of shares and values and barrier maps are partial functions to pairs of shares and barriers. Furthermore, barriers expose an integer variable denoting the barriers state. The semantic model for our formulae is defined below. It follows the models given in [86, 75].

Definition 2.4.1 (State join)**.** *Given two states* $\sigma_1 = (s_1, h_1, b_1)$ *and* $\sigma_2 = (s_2, h_2, b_2)$ *it is possible to define an extension of the* $\oplus$ *operator from the share domain to the state domain:* $\sigma_1 \oplus \sigma_2 = (s_1 \oplus s_2, h_1 \oplus h_2, b_1 \oplus b_2,)$ *where:*

- $s_1 \oplus s_2$ *is defined only if* $s_1 = s_2$ *and* $s_1 \oplus s_2 = s_1$*;*
- $h_1 \oplus h_2 = h$ *such that* $dom(h) = dom(h_1) \cup dom(h_2)$ *and:*
  - $x \in dom(h_1)/dom(h_2)$ *then* $h(x) = h_1(x)$
  - $x \in dom(h_2)/dom(h_1)$ *then* $h(x) = h_2(x)$
  - *else if* $h_1(x) = (\pi_1, v_1)$ *and* $h_2(x) = (\pi_2, v_2)$ *then if* $v_1 = v_2$ *and* $\exists \pi \cdot \pi_1 \oplus \pi_2 = \pi$ *then* $h(x) = (\pi_1 \oplus \pi_2, v_1)$*. We observe that if* $v_1 \neq v_2$ *then the join fails*
- *The definition of* $b_1 \oplus b_2 = b$*, follows identically the definition of* $h_1 \oplus h_2 = h$

Here function $dom(f)$ returns the domain of function f.

Definition 2.4.2 (Model for Separation Constraint)**.**

$$
\begin{array}{lll}
s,h,b \models \Phi_1 \vee \Phi_2 & \texttt{iff} & s,h,b \models \Phi_1 \textit{ or } s,h,b \models \Phi_2 \\
s,h,b \models \exists \overrightarrow{v} \cdot \kappa \wedge \pi & \texttt{iff} & \exists \overrightarrow{\nu} \cdot s[\overrightarrow{v} \mapsto \overrightarrow{\nu}], h, b \models \kappa \textit{ and } s[\overrightarrow{v} \mapsto \overrightarrow{\nu}] \models \pi \\
s,h,b \models \texttt{emp} & \texttt{iff} & \textit{dom}(h) = \emptyset \\
s,h,b \models v_0 :: c^{v_f} \langle \overrightarrow{v} \rangle & \texttt{iff} & c \texttt{ is a data structure } c\ \{\texttt{t}_1\ \texttt{f}_1, .., \texttt{t}_\texttt{n}\ \texttt{f}_\texttt{n}\} \\
 & & \textit{and } \pi = s(v_f) \textit{ and } r = c[\texttt{f}_1 \mapsto s(v_1), .., \texttt{f}_\texttt{n} \mapsto s(v_n)] \\
 & & \textit{and } h = [s(v_0) \mapsto (\pi, r)] \\
s,h,b \models v_0 :: c^{v_f} \langle \overrightarrow{v} \rangle & \texttt{iff} & c \texttt{ is a predicate } (c(\overrightarrow{w}) \equiv \Phi) \\
 & & \Phi' = \textit{set\_perm}(\Phi, v_f) \textit{ and } s,h,b \models \Phi'[\overrightarrow{v}/\overrightarrow{w}] \\
s,h,b \models v_0 :: c^{v_f} \langle \overrightarrow{v} \rangle & \texttt{iff} & c \texttt{ is a barrier} \\
 & & b(s(v_0)) = (s(v_f), r) \textit{ and } r.state = s(v_1) \\
s,h,b \models \kappa_1 * \kappa_2 & \texttt{iff} & \exists h_1, h_2, b_1, b_2 \cdot (s,h_1,b_1) \oplus (s,h_2,b_2) = (s,h,b) \\
 & & \textit{and } s,h_1,b_1 \models \kappa_1 \textit{ and } s,h_2,b_2 \models \kappa_2
\end{array}
$$

$$
\begin{array}{lll}
s \models \pi_1 \wedge \pi_2 & \texttt{iff} & s \models \pi_1 \textit{ and } s \models \pi_2 \\
s \models v_1 \oslash v_2 & \texttt{iff} & s(v_1) \oslash s(v_2),\ \textit{where } \oslash \in \{=, \neq\} \\
s \models v \oslash \textit{null} & \texttt{iff} & s(v) \oslash 0,\ \textit{where } \oslash \in \{=, \neq\} \\
s \models a \oslash 0 & \texttt{iff} & [\![a]\!] s \oslash 0,\ \textit{where } \oslash \in \{=, \leq, \neq\} \\
\ldots & &
\end{array}
$$

The semantic entailment relation $\Phi_1 \models \Phi_2$ holds, if and only if, forall stacks s, heaps h and barrier maps b, we have $s,h,b \models \Phi_1 \implies s,h,b \models \Phi_2$.

2.5 Translation to the Core Language

In this section we present the translation from `SrcLang` to `Core`$-$`U` . The translation is based on rewrite rules and illustrates how advanced language features, such as `try`$-$`finally` and multi-return functions, can be easily captured by the core language. Moreover, we prove two

important properties of the translation, namely completeness and termination.

We also provide a set of optimization rules for `Core−U` designed to reduce the implementation overhead. These rules are specified at a high-level thus facilitating both human understanding and the construction of correctness proofs. For all the rules we supply correctness proofs.

2.5.1 Translation Steps

The translation from SrcLang to our core language consists of several steps. These steps are preceded by a preprocessing phase dedicated to checking the validity of local control flows. More specifically, we check that the `continue` $[L]$ construct appears inside loops, `break` $[L]$ and `continue` $[L]$ mention only accessible labels, and, in the case of the multi-return function declaration with n return points, each of its return instruction must be of the form $(\texttt{ret}{-}i\ e)$ such that $i{\leq}n$. Furthermore, method call must also be invoked with $n-1$ return points.

Following the Stratego approach [98], we base the translation from `SrcLang` to `Core−U` on rewrite rules. The translation process is split into four independent steps such that, in each of the steps, there is no interference between the rules to be applied. In other words, each rule will neither trigger nor block the application of any other rule from the same rewrite set. Additionally, the rules from each phase will not trigger the application of any rule from a previous phase. As a consequence of this non-interference property, the order in which the rules are applied in each of the steps is irrelevant. The non-interference property is also important for guaranteeing the termination of the rewriting process.

Phase I: Preprocessing

In the first step, we transform expressions that appear in a more complex form than allowed by the core language. For example, we may encounter `if` e `then` e_1 `else` e_2, while our core language can only accept a simpler `if` v `then` e_1 `else` e_2. This mismatch can be handled in a standard way with the help of the local `let` binding construct that always generates a fresh

new variable.

$(c)\ e$	$\Rightarrow_T$	$\texttt{let}\ v{=}e\ \texttt{in}\ (c)\ v$	$\neg Var(e)$
$\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2$	$\Rightarrow_T$	$\texttt{let}\ v{=}e\ \texttt{in}\ \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2$	$\neg Var(e)$
$\texttt{throw}\ e$	$\Rightarrow_T$	$\texttt{let}\ v{=}e\ \texttt{in}\ \texttt{throw}\ v$	$\neg Var(e) \wedge \neg New(e)$
$\texttt{ret}{-}i\ e$	$\Rightarrow_T$	$\texttt{let}\ v{=}e\ \texttt{in}\ \texttt{ret}{-}i\ v$	$\neg Var(e) \wedge \neg New(e)$ $\wedge \neg Const(e)$

These rules contain guards, which are meant to restrict their applicability. For instance, the rule corresponding to the cast construct, $(c)\ e$, contains the guard $\neg Var(e)$, meaning that it can only be applied if the expression e is not a variable. Otherwise, the translation is not needed as the construct is already in the expected `Core−U` form. Similarly, $\neg New(e)$ and $\neg Const(e)$ check that the expression e is different from a new construct or a primitive constant, respectively. Take note that the guards were used in order to restrict the introduction of new variables, wherever possible.

The above rewrite rules would require local type inference to determine a suitable type declaration for each of the intermediate variables. Intermediate variables allow us to support a *syntactically minimal* core language that should be easier for formal reasoning. For computational purpose, their overheads can be eliminated (after analysis) by applying either the converse transformation or through better re-use of variables.

Phase II: Main Translation

In the main translation step, we aim to use `Core−U` 's expressivity in order to uniformly manipulate several of the `SrcLang` constructs. For this purpose, we will exploit a basic construct of our core language, $ft\#x$, where x denotes a simple term that would directly produce a value, e.g. v, k, `new` c or (fv, v), while ft denotes the desired control flow. Using it, we can easily

translate a number of constructs from the source language, as shown below.

$$
\begin{array}{ll}
x & \Rightarrow_T \texttt{norm}\#x \\
\texttt{break} & \Rightarrow_T \texttt{brk}\#() \\
\texttt{break}\ L & \Rightarrow_T \texttt{brk}{-}L\#() \\
\texttt{continue} & \Rightarrow_T \texttt{cont}\#() \\
\texttt{continue}\ L & \Rightarrow_T \texttt{cont}{-}L\#() \\
\texttt{ret}{-}i\ x & \Rightarrow_T \texttt{ret}{-}i\#x \\
\texttt{throw}\ v & \Rightarrow_T \texttt{ty}(v)\#v \\
\texttt{throw new}\ c & \Rightarrow_T c\#\texttt{new}\ c \\
(c)\ v & \Rightarrow_T \texttt{if ty}(v){<:}c\ \texttt{then norm}\#v \\
 & \qquad \texttt{else ClassCastExc}\#v
\end{array}
$$

For the `throw` command, we have to decide which flow type to raise. In case of `new` c, we can determine that it is an exact type of class c. In case of v, we have to use $\texttt{ty}(v)$ to retrieve the runtime type of the object. Assuming the well-typedness of source programs, we expect these flow types from `throw` commands to be subtypes of `exc`.

The other unified construct is for supporting a generalised `try`$-$`catch` mechanism. Its format is $\texttt{try}\ e_1\ \texttt{catch}\ (c@fv)\#v\ e_2$ where c is the type of flow captured, must be a subtype of `c`$-$`flow`, fv is a variable that binds to the concrete control flow type captured and v is a variable capturing the thrown value. However, we may sometimes ignore the bound variables fv and v. For convenience, we allow the following shorthands:

$$
\begin{array}{ll}
\texttt{try}\ e_1\ \texttt{catch}\ c\#v\ e_2 & \equiv \texttt{try}\ e_1\ \texttt{catch}\ c@\_\#v\ e_2 \\
\texttt{try}\ e_1\ \texttt{catch}\ c\ e_2 & \equiv \texttt{try}\ e_1\ \texttt{catch}\ c@\_\#\_\ e_2
\end{array}
$$

The unified try-catch construct can be used to:

- **translate the try-catch construct from the source language:**

An important feature of Java for exception handling is the try construct with multiple catch handlers. An example of its usage is the following construct with two handlers:

$\texttt{try}\ \{e\}$

$\texttt{catch}\ (c_1\ v_1)\ \{e_1\}$

$\texttt{catch}\ (c_2\ v_2)\ \{e_2\}$

It is tempting to think that the above feature can be directly implemented in our language, as a nested pair of $\texttt{try}-\texttt{catch}$ constructs, as follows:

$$\texttt{try}\ (\texttt{try}\ e\ \texttt{catch}\ (c_1@\_\#v_1)\ e_1)\ \ \texttt{catch}\ (c_2@\_\#v_2)\ e_2$$

However, this translation is not correct, since an exception raised inside $\texttt{e}_1$ may be caught by the second handler for the target (nested) code but not the source (unnested) code. To avoid this problem, we must attach a side-condition which ensures that $\forall\, c\ \in\ \mathit{throwsAll}(e_1) \cdot \neg(c{<:}c_2 \vee c_2{<:}c)$, where $\mathit{throwsAll}(e_1)$ statically computes all the flow types that may escape from e_1, before we allow the above translation.

Alternatively, we will use the following more general translation:

$$\begin{array}{l}
\texttt{try}\ e\ \texttt{catch}\ c_1\ v_1\ e_1 \\
\qquad [\texttt{catch}(c_i\ v_i)\ e_i]_{i=2}^{n} \\
\Rightarrow_T\ \texttt{try}\ (\\
\qquad\qquad [\ \texttt{try}\ ]_{i=2}^{n} \\
\qquad\qquad \texttt{try}\ e\ \texttt{catch}\ (c_1@fv_1\#v_1)\ \texttt{embed}(\texttt{spec}, e_1) \\
\qquad\qquad [\ \texttt{catch}\ (c_i@fv_i\#v_i)\ \texttt{embed}(\texttt{spec}, e_i)\ ]_{i=2}^{n} \\
\qquad)\ \texttt{catch}\ (\texttt{spec}@fv\#v)\ v.1\#v.2
\end{array}$$

We provide the notation $\texttt{embed}(ft, e)$ as a shorthand for :

$$\texttt{try}\ e\ \texttt{catch}\ ((\texttt{c}-\texttt{flow}@fv)\#v)\ ft\#(fv, v)$$

For each flow that can be caught, it embeds the flow and its value into a pair (fv, v)

and attaches them to the flow ft taken as parameter. In the context of the previous translation, all the flows that can be caught from $e_i, 1{\leq}i{\leq}n$, will be embedded with their corresponding values, and attached to a flow type named `spec`. Consequently, every outcome from evaluating e_i will indeed skip the outer handlers $e_j, i{<}j{\leq}n$, and will only be caught by the outermost handler. Operations $v.1$ and $v.2$ are used to extract the components of the pair denoted by v. Take note that the translation of a try catch with n handlers will result in $n{+}1$ try catch constructs.

- **translate the finally construct:**

  Let us consider the `finally` construct from Java, used in the following two code fragments below:

```
try { return 2 } catch (Exception v) {print "catch"}
finally { print "tidy up"}
```

```
try { System.exit(0)}catch (Exception v){print "catch"}
finally { print "tidy up" }
```

  A frequent doubt is whether the finally block would be executed after the occurrence of non-exceptional control flows, such as `return` or `System.exit`. We can unambiguously and succinctly specify the intention of the `finally` construct by the following translation to our core language:

$$
\begin{array}{l}
e_1\ \texttt{finally}\ e_2 \\
\Rightarrow_T\ \texttt{try}\ e_1\ \texttt{catch}\ ((\texttt{c-flow}@fv)\#v)\ (e_2;\ fv\#v)
\end{array}
$$

  Any control flow (including `norm`) that escapes e_1 and can be caught (under the `c-flow` subtype) will lead to the execution of the finally block. For the above mentioned Java example, we would model `return` by flow type `ret-1`, while `System.exit`, to terminate the Java Virtual Machine, is modelled by flow type `halt` in our language. Our translation

therefore indicates that the finally $\mathtt{e}_2$ block is executed for the first code fragment but not the second code fragment, since $\mathtt{ret{-}1{<:}c{-}flow}$ and $\neg(\mathtt{halt{<:}c{-}flow})$.

- **translate the multi-return function call and declaration:**

$$\begin{aligned}&(m\ \overrightarrow{v})\ \mathtt{with}\ \lambda v_2.e_2 \ldots \lambda v_n.e_n\\ \Rightarrow_T\ &\mathtt{try}\ (m\ \overrightarrow{v})\ \mathtt{catch}\ \mathtt{ret}{-}2\#v_2\ e_2\\ &\qquad\qquad\ldots\\ &\qquad\qquad\mathtt{catch}(\mathtt{ret}{-}n\#v_n)\ e_n\end{aligned}$$

The translation to our core language explicitly captures the choice of the return point, based on the control flow that is caught after the evaluation of the inner application, $(m\ \overrightarrow{v})$.

Note that the first return of each method declaration is considered to be the normal value that is to be returned directly to its callers' sites. This is achieved by changing $\mathtt{ret}{-}1$ to the $\mathtt{norm}$ *control* flow, for each method declaration, as follows:

$$\begin{aligned}&m\ [\mathtt{with}\ n]\ (\overrightarrow{t\ v})\ \{e\}\\ \Rightarrow_T\ &m\ [\mathtt{with}\ n]\ (\overrightarrow{t\ v})\{\mathtt{try}\ e\ \mathtt{catch}\ \mathtt{ret}{-}1\#v\ \mathtt{norm}\#v\}\end{aligned}$$

- **translate the local binding construct:**

$$\mathtt{let}\ v{=}e_1\ \mathtt{in}\ e_2\quad\Rightarrow_T \mathtt{try}\ e_1\ \mathtt{catch}\ \mathtt{norm}\#v\ e_2$$

- **handle abnormal controls due to breaks and continues for loops:**

$$
\begin{array}{l}
L : \texttt{while}\ e_1\ \{e_2\} \\
\Rightarrow_T\ \texttt{try}\ \{\texttt{bool}\ v;\ v{:=}e_1; \\
\qquad\qquad \texttt{while}\ v\ \{ \\
\qquad\qquad\quad \texttt{try}\ e_2\ \texttt{catch cont norm}\#() \\
\qquad\qquad\qquad\quad \texttt{catch}\ (\texttt{cont}{-}L)\ \texttt{norm}\#(); \\
\qquad\qquad\quad v{:=}e_1\} \\
\qquad \texttt{catch}\ (\texttt{brk})\ \texttt{norm}\#() \\
\qquad \texttt{catch}\ (\texttt{brk}{-}L)\ \texttt{norm}\#()
\end{array}
$$

  The try-catch mechanism provides a uniform way to deal with non-local control flows, such as `break` and `continue` via exception handling. If these mechanisms had not been used, our language would resort to lower-level constructs, such as `goto` statements, for formal reasoning. The unstructured `goto` construct may be useful for efficient implementation, but has a lower abstraction level for formal reasoning.

- **handle abnormal controls due to breaks for labelled expressions:**

$$L : e \Rightarrow_T \texttt{try}\ e\ \texttt{catch brk}{-}L\ \texttt{norm}\#()$$

Phase III: Wrapping-up the Translation

For this phase, the unified `try`$-$`catch` construct is used to replace the sequence operator $e_1;\ e_2$, and to simplify the RHS of the assignment construct (to a variable), as shown below.

$$
\begin{array}{ll}
e_1;\ e_2 & \Rightarrow_T \texttt{try}\ e_1\ \texttt{catch norm}\ e_2 \\
w := e & \Rightarrow_T \texttt{try}\ e\ \texttt{catch norm}\#v\ w := v
\end{array}
$$

The placement of these two rules in a separate phase is meant to maintain the non-interference property of the translation process. This is due to the fact that some of the translations rules from the previous phase trigger the application of the assignment and sequence translation rules.

Phase IV: Handling Implicitly Raised Exceptions

To complete our translation, in the last rewriting step, we provide a set of rules to deal with constructs that may implicitly raise some exceptions, such as null dereferencing or memory overflow. These rules can help make all raised exceptions explicit.

$$
\begin{array}{lll}
v.f & \Rightarrow_T & \texttt{if } v\texttt{=null then nullPtrExc}\#v \\
 & & \texttt{else } v.f \\
v.f := v_2 & \Rightarrow_T & \texttt{if } v\texttt{=null then nullPtrExc}\#v \\
 & & \texttt{else } v.f := v_2 \\
ft\#\texttt{new } c & \Rightarrow_T & \texttt{if } \textit{enoughMem}(c) \texttt{ then } ft\#\texttt{new } c \\
 & & \texttt{else OutofMemoryExc}\#\texttt{null} \\
\texttt{ty}(v)\#v & \Rightarrow_T & \texttt{if } v\texttt{=null then nullPtrExc}\#v \\
 & & \texttt{else ty}(v)\#v
\end{array}
$$

As a wrap-up step unrelated to control flow handling, loops are transformed to tail-recursive methods where the parameters are passed by reference.

Note that the translated codes can be more efficiently implemented than the original code, since they can be specialised as exception-free code. Furthermore, there is potential for a systematic optimization to eliminate some of the checks that have been explicitly inserted. For example, we could use a nullness analysis to help statically determine those nullness tests that are known to be redundant. Similarly, it is possible to aggregate a series of tests on memory sufficiency to be replaced by a bigger test at the beginning. These steps can lead to simpler (and more efficient) core programs.

It is possible to provide a semantics for Core−U and another semantics for SrcLang, before proving that the translation rules are *fully abstract* [89] by preserving observational equivalence between the source and target programs under their respective semantics. This would have proven the correctness of the translation rules between the two languages. However, since the core language is at a relatively high level and can be viewed as a subset of the source language, we could also define the semantics of the SrcLang directly in terms of the Core−U . With this simplified approach, the translation rules would be correct by construction. This approach is not at all new. For example, the Haskell language [54] is largely defined in this way, by translating more complex language features into a simpler core.

Nevertheless, we will prove two important properties of our translation rules. Firstly, we shall show that given any arbitrary SrcLang expression, we can always translate it into a Core−U counterpart. Secondly, we shall prove that the application of the translation rules always terminates.

Lemma 2.5.1 (Completeness of Translation). *Consider any term* $e \in$ SrcLang. *Repeated applications by our transformation rules via* $e \Rightarrow_T^* e'$ *would eventually result in* $e' \in$ Core−U *, when the transformation terminates.*

Proof [(sketch)] There is at least one transformation rule for each syntactic construct of SrcLang, except for the local block $\{t\ v;\ e\}$ which remains unchanged in Core−U . Furthermore, each target form in the RHS of the translation rules belongs to Core−U or can be transformed as so in a subsequent step. Hence, by induction on the syntactic structure of SrcLang, we can prove that the final expression belongs to Core−U , should the transformation terminate. □

Lemma 2.5.2 (Termination of Translation). *The transformation* $e \Rightarrow_T^* e'$ *always terminates.*

Proof [(sketch)] Let us first note that all the rewrite rules used in the translation process follow a common design, namely the RHS does not contain the pattern from the LHS that triggers the application of the rule. This property ensures that each application of a rewrite

rule will eliminate one occurrence of a specific trigger. Moreover, none of the rules trigger the application of any other rule from the same set, nor the application of a rule from any of the previous steps. Consequently, the rewriting process, for each of the four steps, must terminate. □

2.5.2 Optimization Rules

We shall now provide a set of equivalence rules for our language that can be used for formal reasoning and optimization. One feature of our rules is that they are formulated at a relatively high-level based on our core language. We expect such high-level rules to be easier to understand and prove correct.

Our first rule is inspired by the associativity property of sequential composition, namely $(e_1;\ e_2);\ e_3 \Leftrightarrow e_1;\ (e_2;\ e_3)$. A corresponding rule, generalised to nested try-catch commands, is shown below:

$$[\mathbf{Re{-}Ordering\ Rule}]$$

$$\frac{c_2 <: c_1}{\begin{array}{l}\texttt{try}\ (\texttt{try}\ e\ \texttt{catch}\ c_1\ e_1)\ \texttt{catch}\ c_2\ e_2 \\ \Leftrightarrow \texttt{try}\ e\ \texttt{catch}\ c_1\ (\texttt{try}\ e_1\ \texttt{catch}\ c_2\ e_2)\end{array}}$$

This re-ordering may be used to help group a nested series of expressions with a similar property together, so that we may have a larger expression with the same property. For example, if expressions e_1 and e_2 are free of dynamic control flows (namely exceptions), we may group them closer via the above rule. Grouping together expressions without exceptions can give us larger code blocks that can be better optimized.

The second optimization rule ensures the elimination of redundant catch handlers. This can occur if the try block never raises any of the control flows that are caught by a given handler. Take note that, *throws*(e) is intended to capture the entire set of exceptions (or control flows) that may escape from the expression e. The auxiliary function *overlap*(c_1, c_2) signifies the

$$\frac{val = (s(ft), s(x))}{\big((s,h,b), \texttt{jump}\ L(ft,x)\big) \mapsto \big((s,h,b), \texttt{jump}{-}L\#val\big)}$$

$$\frac{\neg(c_1{<:}c)}{\big((s,h,b), \texttt{try}\ c_1\#a_1\ \texttt{catch}\ c\ L(fv,v){:}e_2\big) \mapsto \big((s,h,b), c_1\#a_1\big)}$$

$$\frac{\begin{array}{c} e = \texttt{jump}{-}L\#(c_1,a_1) \qquad u = newid() \qquad fu = newid() \\ e' = \texttt{BLK}(\{u, fu\}, e_2[u/v, fu/fv]) \end{array}}{\big((s,h,b), \texttt{try}\ e\ \texttt{catch}\ c\ L(fv,v){:}e_2\big) \mapsto \big((s{+}[fu{\mapsto}c_1, u{\mapsto}a_1], h, b), e'\big)}$$

$$\frac{\begin{array}{c} c_1{<:}c \\ u = newid() \qquad fu = newid() \\ e' = \texttt{BLK}(\{u, fu\}, e_2[u/v, fu/fv]) \end{array}}{\big((s,h,b), \texttt{try}\ c_1\#a_1\ \texttt{catch}\ c\ L(fv,v){:}e_2\big) \mapsto \big((s{+}[fu{\mapsto}c_1, u{\mapsto}a_1], h, b), e'\big)}$$

Figure 2.6: Semantics for the Jump Construct

condition that c_1 shares some common subtypes with c_2, *overlap*(c_1, c_2)=$c_1{<:}c_2 \vee c_2{<:}c_1$.

$$[\textbf{Catch Elimination Rule}]$$

$$\frac{\forall c \in \mathit{throws}(e) \cdot \neg \mathit{overlap}(c, c_1)}{\texttt{try}\ e\ \texttt{catch}\ c_1@fv_1\#v_1\ e_1 \Rightarrow\ e}$$

Our next rule is intended to optimize the stack unwinding mechanism that is typically used to propagate a raised exception to its handler. Occasionally, it is possible to determine that the invocation of a particular exception (or control flow) will always be caught by a given handler. If this scenario is detected, we can transform a raised exception into a direct jump to its handler's code. Such a jump feature shall be added directly to the target compiled language. As our core language also supports the capture of both a control flow and its thrown value, we will have to pass these items to the handler during such a jump. Argument passing can be implemented with the help of a parameterized jump. A $\texttt{jump}(L(\overrightarrow{v}))$ command is said to be parameterized by $\overrightarrow{v}$ if it carries a list of arguments for its labelled location. Correspondingly,

a labelled location $\texttt{jump}(L(\overrightarrow{v}) : e)$ is said to be parameterized with $\overrightarrow{v}$, if variables $\overrightarrow{v}$ in code e can be initialized by its corresponding jump. Our parameterized label shall always be used in the context of a catch handler of the form $\texttt{try}\ e\ \texttt{catch}\ c@fv\#v\ L(fv, v){:}e_2$, which shall be abbreviated as $\texttt{try}\ e\ \texttt{catch}\ c\ L(fv, v){:}e_2$.

In real languages, the jump instruction would be a machine primitive that changes the current program counter to the address of the labelled instruction. However, for convenience, we shall model the jump construct (in the same way as `break` construct) with an abnormal control flow that would be caught by its labelled handler. Using this interpretation, the jump construct, $\texttt{jump}\ L(ft, x)$, is essentially modelled using $\texttt{jump}{-}L\#(ft, x)$, where each flow type $\texttt{jump}{-}L$ will only be captured by its handler at the labelled location L. In Figure 2.6 we provide the semantics for the newly introduced jump construct. Take note that the formalised semantic mirrors the state transition relation for the actual machine.

We shall now consider the optimization itself. As a simple example, consider the code fragment:

$$\begin{array}{l}\texttt{try try}\\ \qquad\qquad (\texttt{try exc}\#3\ \texttt{catch}\ (\texttt{nullPtrExc})\ e_1)\\ \qquad \texttt{catch}\ (\texttt{exc}\#v)\ e_2\\ \texttt{catch}\ (\texttt{norm})\ e_3\end{array}$$

An exception exc#3 is being thrown that will be caught by the second catch handler. We may directly link this throw with its corresponding catch handler by the following transformed code:

$$\begin{array}{l}\texttt{try try}\\ \qquad\qquad (\texttt{try jump}\ L1(\texttt{exc}, 3)\ \texttt{catch}\ (\texttt{nullPtrExc})\ e_1)\\ \qquad \texttt{catch}\ (\texttt{exc})\ L1(\_, v) : e_2\\ \texttt{catch}\ (\texttt{norm})\ e_3\end{array}$$

This is a desirable optimization since a jump command can usually be implemented much more efficiently. The rule for linking a throw with its corresponding catch handler can be

formally expressed, as follows:

$$\frac{[\textbf{Throw}-\textbf{Catch Linking Rule}] \quad \textit{escape}(C[], fc) \quad \Gamma(ft) = fc \quad fc{<:}c}{\begin{array}{l}\texttt{try}\ C[ft\#x]\ \texttt{catch}\ c@fv\#v\ e\ \Rightarrow \\ \texttt{try}\ C[\texttt{jump}\ L(ft, x)]\ \texttt{catch}\ c\ L(fv, v) : e\end{array}}$$

Note that we assume a prior type inference algorithm. Consequently, Γ denotes the type environment, with its corresponding runtime stack capturing both values and control flows. For each flow, ft, $\Gamma(ft)$ will capture the flow type. We use a context notation $C[]$ to denote an expression with a single hole $[]$, and $C[e]$ to denote the replacement of the hole by e for the given context. Formally, the context notation is defined as:

$$\begin{array}{l} C[] ::= [] \mid \{t\ v;\ C[]\} \mid \texttt{if}\ v\ \texttt{then}\ C[]\ \texttt{else}\ e \\ \quad \mid \texttt{if}\ v\ \texttt{then}\ e\ \texttt{else}\ C[] \mid \texttt{try}\ C[]\ \texttt{catch}\ @fv\#v\ e \\ \quad \mid \texttt{try}\ e\ \texttt{catch}\ c@fv\#v\ C[] \mid \texttt{do}\ C[]\ \texttt{while}\ v \end{array}$$

We also provide an operator *escape*$(C[], fc)$ that can determine if a control flow fc will escape its given context $C[]$. This operation is defined, as follows:

$$\begin{array}{l} \textit{escape}([], fc) = \texttt{true} \\ \textit{escape}(\texttt{try}\ C[]\ \texttt{catch}\ c\ e, fc) \mid \textit{overlap}(fc, c)\ = \texttt{false} \\ \textit{escape}(\texttt{try}\ C[]\ \texttt{catch}\ c\ e, fc) \mid \neg\textit{overlap}(fc, c) \\ \qquad\qquad = \textit{escape}(C[], fc) \end{array}$$

$$\begin{array}{l} \textit{escape}(E, fc) = \textit{escape}(C[], fc) \\ \text{where } E ::= \texttt{if}\ v\ \texttt{then}\ C[]\ \texttt{else}\ e \mid \texttt{if}\ v\ \texttt{then}\ e\ \texttt{else}\ C[] \\ \quad \mid \texttt{do}\ C[]\ \texttt{while}\ v \mid \{t\ v;\ C[]\} \mid \texttt{try}\ e\ \texttt{catch}\ c\ C[] \end{array}$$

After some throw-catch linkings, it may be possible to obtain code involving a series of con-

secutive jumps. In order to shortcut such a series of jumps, we consider the scenario under which a parameterized jump can be inlined. This can occur if control flows from e are never caught by the context $(\texttt{try}\ C[]\ \texttt{catch}\ c\ L(fv, v) : e)$, as defined in the rule below:

$$[\textbf{Jump Inlining Rule}]$$

$$\frac{\forall c_1 \in \mathit{throws}(e) \cdot (\mathit{escape}(C[], c_1) \wedge \neg \mathit{overlap}(c, c_1))}{\begin{array}{l}\texttt{try}\ C[\texttt{jump}\ L(ft, x)]\ \texttt{catch}\ c\ L(fv, v) : e \\ \Rightarrow\ \texttt{try}\ C[[fv \mapsto ft, v \mapsto x]e]\ \texttt{catch}\ c\ L(fv, v) : e\end{array}}$$

To avoid name capture during this non-local inlining, we have to ensure that the variables from $(\mathit{free}(e) - \{fv, v\})$ do not clash with the bound variables from the context $C[]$. This can be ensured by uniquely renaming the bound variables in $C[]$.

Soundness of Optimization Rules

Definition 2.5.1 (**Semantics Preserving Transformation).** *An expression e' is said to be a semantics preserving transformation of e if, whenever the evaluation of e terminates and*

$$\big(e, h, s\big) \mapsto^* \big(c\#a, h_1, s_1\big),$$

then the evaluation of e' terminates and

$$\big(e', h, s\big) \mapsto^* \big(c\#a, h_1, s_1\big).$$

Definition 2.5.2 (**Sound Optimization Rule).** *An optimization rule rewriting an expression exp into an expression exp' is said to be sound if exp' is a semantics preserving transformation of exp.*

Def. 2.5.1 and Def. 2.5.2 are used to state the soundness of the optimization rules. When sketching the proofs of the soundness lemmas, we will make use of some notational conven-

tions. The initial expression, before optimization, will be denoted by exp, while the optimized one, obtained after rewriting, will be denoted by exp'. Additionally, h will stand for the current heap, and s for the current stack.

Lemma 2.5.3 (**Soundness of Re-Ordering Rule ($\Rightarrow$ direction)).** *If $c_2 <: c_1$ then the following expression:*

$$\texttt{try}\ e\ \texttt{catch}\ c_1 @ f v_1 \# v_1\ (\texttt{try}\ e_1\ \texttt{catch}\ c_2 @ f v_2 \# v_2\ e_2)$$

is a semantics preserving transformation of:

$$\texttt{try}\ (\texttt{try}\ e\ \texttt{catch}\ c_1 @ f v_1 \# v_1\ e_1)\ \texttt{catch}\ c_2 @ f v_2 \# v_2\ e_2$$

Proof: According to the hypothesis that the evaluation of exp terminates and to the semantics of the try catch construct, we can assume that e gets evaluated to $c\#a$. Consequently, there are two possibilities:

- $c <: c_1$. In this case, the evaluation of exp is reduced to the evaluation of the handler e_1 in a $\texttt{try}\ e_1\ \texttt{catch}\ c_2\ e_2$ construct. On the other hand, when considering the expression exp' obtained after optimization, e gets evaluated in a similar way and, assuming that $c <: c_1$, everything is reduced again to $\texttt{try}\ e_1\ \texttt{catch}\ c_2\ e_2$. □

- $\neg(c <: c_1)$. Conform to the semantics, the handler e_1 is not reachable in exp. Moreover, due to the assumption that $c_2 <: c_1$, the handler e_2 is also unreachable. Therefore, exp is evaluated to $c\#a$. On the other hand, in exp', the handler $\texttt{try}\ e_1\ \texttt{catch}\ c_2 @ f v_2 \# v_2\ e_2$ is also unreachable. Hence, exp' also evaluates to $c\#a$. □

Take note that the **[Re-Ordering Rule]** rule claims the equivalence of the two expressions, exp and exp'. Therefore, we need another lemma for proving soundness when the rule is applied from right to left.

Lemma 2.5.4 (**Soundness of Re-Ordering Rule ($\Leftarrow$ direction)).** *If $c_2 <: c_1$ then the following*

expression:

$$\texttt{try}\ (\texttt{try}\ e\ \texttt{catch}\ c_1@fv_1\#v_1\ e_1)\ \texttt{catch}\ c_2@fv_2\#v_2\ e_2$$

is a semantics preserving transformation of:

$$\texttt{try}\ e\ \texttt{catch}\ c_1@fv_1\#v_1\ (\texttt{try}\ e_1\ \texttt{catch}\ c_2@fv_2\#v_2\ e_2)$$

Proof: Similar to the proof for Lemma 2.5.3. □

Lemma 2.5.5 (Soundness of Catch Elimination Rule). *Assuming that*

$$\forall c \in \mathit{throws}(e) \cdot \neg \mathit{overlap}(c, c_1)$$

then expression e *is a semantics preserving transformation of*

$$\texttt{try}\ e\ \texttt{catch}\ c_1@fv_1\#v_1\ e_1$$

Proof: We show that exp' is a semantics preserving transformation of exp by proving that the handler e_1 of exp is unreachable. Let us assume that e evaluates to $c\#a$. According to the assumption that all the exceptions are checked (any exception escaping from a method's body must be declared in its throws list), to the semantics for try catch, and the hypothesis that all the control flows of e escape from exp, the handler e_1 is unreachable. □

Lemma 2.5.6. Soundness of Throw-Catch Linking Rule: *If the following conditions hold:*

$$\mathit{escape}(C[], fc) \quad \mathit{and} \quad \Gamma(ft) = fc \quad \mathit{and} \quad fc{<:}c$$

then the expression: $\texttt{try}\ C[\texttt{jump}\ L(ft, x)]\ \texttt{catch}\ c\ L(fv, v)\ :\ e$ *is a semantics preserving transformation of the expression*

$$\texttt{try}\ C[ft\#x]\ \texttt{catch}\ c@fv\#v\ e.$$

Proof: By structural induction on the context of the try block $C[]$.

- Case []. Straightforward.

- Case `try` $C[]$ `catch` $c\ e$. Conform to the semantics, the evaluation of both exp and exp' implies first the evaluation of the inner try block. According to the hypothesis that $\Gamma(ft) = fc$ and that the control flow fc escapes the context $C[]$ being caught by the external handler, both expressions will be reduced to the following machine configuration: $\langle(s{+}[fv{\mapsto}ft, v{\mapsto}x], h, b), e\rangle$, where s , h and b denote the stack, heap and barrier map after the evaluation of the try block. The conclusion is immediate.

- Case `do` $C[]$ `while` v. The initial expression exp is reduced to:

$$\texttt{try}\ C[ft\#x]; \texttt{if}\ v\ \texttt{then}\ (\texttt{do}\ C[ft\#x]\ \texttt{while}\ v)\ \texttt{else}\ ()$$
$$\texttt{catch}\ c@fv\#v\ e$$

  which is syntactic sugar for:

$$\texttt{try}\ (\texttt{try}\ C[ft\#x]\ \texttt{catch}\ norm$$
$$\texttt{if}\ v\ \texttt{then}\ (\texttt{do}\ C[ft\#x]\ \texttt{while}\ v)$$
$$\texttt{else}\ ())\ \texttt{catch}\ c@fv\#v\ e$$

  On the other hand, exp' is reduced to:

$$\texttt{try}\ (\texttt{try}\ C[\texttt{jump}\ L(ft, x)]\ \texttt{catch}\ norm$$
$$\texttt{if}\ v\ \texttt{then}\ (\texttt{do}\ C[\texttt{jump}\ L(ft, x)]\ \texttt{while}\ v)$$
$$\texttt{else}\ ())\ \texttt{catch}\ c\ L(fv, v) : e$$

  The conclusion follows from the hypothesis that *escape*$(C[], fc)$ and $\Gamma(ft) = fc$ and $fc{<:}c$ and from the dynamic semantics.

- Case $t\ v;\ C[]$. We conclude immediately from the semantics and the hypothesis that

$\Gamma(ft) = fc$ and the control flow fc escapes the context $C[]$ and it is caught by the external handler. Both exp and exp' are reduced to

$$\langle (s{+}[fv{\mapsto}ft, v{\mapsto}x], h, b), e \rangle$$

where s,h and b denote the stack, heap and barrier map after the evaluation of the try block, respectively.

- Case `if` v `then` $C[]$ `else` e_1. According to the semantics for the if construct, there are two cases:
  - $s(v)$=`true`. The two expressions, exp and exp', are reduced respectively to:

    $$\texttt{try}\ C[ft\#x]\ \texttt{catch}\ c@fv\#v\ e$$

    and $\texttt{try}\ C[\texttt{jump}\ L(ft, x)]\ \texttt{catch}\ c\ L(fv, v) : e$

    Conclusion follows immediately.
  - $s(v)$=`false` According to the semantics:
    - exp is reduced to: `try` e_1 `catch` $c@fv\#v\ e$
    - exp' is reduced to: `try` e_1 `catch` $c\ L(fv, v) : e$

  The conclusion follows from the semantics, as e_1 does not contain any jump construct.
- Case `if` v `then` e_1 `else` $C[]$. Similar to the previous case.
- Case `try` e_1 `catch` $(c_1@fv_1)\#v_1\ C[]$. According to the semantics, there are two cases:
  - the evaluation of e_1 generates a control flow c' and $c'{<:}c_1$. From the dynamic semantics, the handlers $C[ft\#x]$ and $C[\texttt{jump}\ L(ft, x)]$ are to be evaluated, respectively. The conclusion follows from the induction hypothesis.
  - the evaluation of e_1 generates a control flow c' and $\neg(c'{<:}c_1)$. The conclusion follows from the fact that e_1 does not contain any jump construct.

□

Lemma 2.5.7 (Soundness of Jump Inlining Rule:). *Assuming that the following conditions hold:*

$$\forall c_1 \in \mathit{throws}(e) \cdot (\mathit{escape}(C[], c_1) \wedge \neg \mathit{overlap}(c, c_1))$$

then expression

$$\texttt{try}\ C[[fv \mapsto ft, v \mapsto x]e]\ \texttt{catch}\ c\ L(fv, v) : e$$

is a semantics preserving transformation of the expression

$$\texttt{try}\ C[\texttt{jump}\ L(ft, x)]\ \texttt{catch}\ c\ L(fv, v) : e.$$

Proof: By structural induction on the context of the try block $C[]$.

- Case $[]$. Straightforward.
- Case $\texttt{try}\ C[]\ \texttt{catch}\ c\ e$. Conform to the semantics, the evaluation of both exp and exp' implies first the evaluation of the inner try block. The conclusion comes from the assumption that all the control flows from e escape the context $C[]$ and from the induction hypothesis.
- Case $\texttt{do}\ C[]\ \texttt{while}\ v$. The initial expression exp is reduced to:

$$\begin{gathered}\texttt{try}\ (C[\texttt{jump}\ L(ft, x)];\\ \texttt{if}\ v\ \texttt{then}\ (\texttt{do}\ C[\texttt{jump}\ L(ft, x)]\ \texttt{while}\ v)\ \texttt{else}\ ())\\ \texttt{catch}\ c\ L(fv, v) : e\end{gathered}$$

  which is syntactic sugar for:

$$\begin{gathered}\texttt{try}\ (\texttt{try}\ C[\texttt{jump}\ L(ft, x)]\ \texttt{catch}\ norm\\ \texttt{if}\ v\ \texttt{then}\ (\texttt{do}\ C[\texttt{jump}\ L(ft, x)]\ \texttt{while}\ v)\\ \texttt{else}\ ())\ \texttt{catch}\ c\ L(fv, v) : e\end{gathered}$$

On the other hand, exp' is reduced to:

$$\texttt{try}\ (\texttt{try}\ C[[fv{\mapsto}ft, v{\mapsto}x]e]\ \texttt{catch}\ norm$$
$$\texttt{if}\ v\ \texttt{then}\ (\texttt{do}\ C[[fv{\mapsto}ft, v{\mapsto}x]e]\ \texttt{while}\ v)$$
$$\texttt{else}\ ())\ \texttt{catch}\ c\ L(fv, v) : e$$

From the semantics and the assumption that all the control flows from e escape both the context $C[]$ and the external try catch, the two expressions will be reduced to: $[fv{\mapsto}ft, v{\mapsto}x]e$.

- Case $t\ v_1;\ C[]$. The two expressions, exp and exp', are reduced to

$$\texttt{try}\ (t\ v_1;\ C[\texttt{jump}\ L(ft, x)])\ \texttt{catch}\ c\ L(fv, v) : e$$

and $\texttt{try}\ (t\ v_1;\ C[[fv{\mapsto}ft, v{\mapsto}x]e])\ \texttt{catch}\ c\ L(fv, v) : e$, respectively. According to the semantics and the assumption that all the control flows from e escape both the context $C[]$ and the external try catch construct, the two expressions will evaluate e. Note that, for the latter case, the stack will also contain the local variable v_1. In order to avoid name clashing, we uniquely rename the free variables of e. Consequently, the result of the evaluation is the same regardless of whether or not v_1 is on the stack.

- Case $\texttt{if}\ v\ \texttt{then}\ C[]\ \texttt{else}\ e_1$. According to the semantics for the if construct, there are two possibilities:

  - $s(v)$=$\texttt{true}$. The conclusion follows immediately from the induction hypothesis as the two expressions, exp and exp', are reduced respectively to:

$$\texttt{try}\ C[\texttt{jump}\ L(ft, x)]\ \texttt{catch}\ c\ L(fv, v) : e$$

    and $\texttt{try}\ C[[fv{\mapsto}ft, v{\mapsto}x]e]\ \texttt{catch}\ c\ L(fv, v) : e$.

  - $s(v)$=$\texttt{false}$. Both expressions reduce to:

$\texttt{try}\ e_1\ \texttt{catch}\ c\ L(fv, v) : e$.

- Case $\texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ C[]$. Similar to the previous case.
- Case $\texttt{try}\ e_1\ \texttt{catch}\ (c_1@fv_1)\#v_1\ C[]$. According to the semantics, there are two possibilities:
  - the evaluation of e_1 generates a control flow c' with the property $c' <: c_1$. From the dynamic semantics, the handlers $C[\texttt{jump}\ L(ft, x)]$ and $C[[fv \mapsto ft, v \mapsto x]e]$ are to be evaluated, respectively. The conclusion follows from the induction hypothesis.
  - the evaluation of e_1 generates a control flow c' and $\neg(c' <: c_1)$. Both expressions are reduced to $\texttt{try}\ e_1\ \texttt{catch}\ c\ L(fv, v) : e$.

□

Chapter 3

Exception Verification

So far we have introduced the input language, we defined a novel core language which unifies various control flow types under one formalism and we showed how the input language can be translated to the core language.

In this chapter we will elaborate on common expectations for exception safety guarantees, and show how easily they can be captured in our specification language. We then introduce a set of elegantly simple rules for verifying programs with exceptions. We conclude by reporting on experimental results of introducing our verification logic into the HIP verifier.

3.1 Motivation

Exception handling is considered to be an important but yet controversial feature for many modern programming languages. It is important since software robustness is highly dependent on the presence of good exception handling codes. Exception failures can account for up to 2/3 of system crashes and 50% of system security vulnerabilities [72]. It is controversial since its dynamic semantics is often considered too complex to follow by both programmers and software tools.

Goodenough provides in [41] a possible classification of exceptions according to their us-

age. More specifically, they can be used:

- to permit dealing with an operation's failure as either domain or range failure. Domain failure occurs when an operation finds that some input assertion is not satisfied, while range failure occurs when the operation finds that its output assertion cannot be satisfied.

- to monitor an operation, e.g. to measure computational progress or to provide additional information and guidance should certain conditions arise.

In the context of Spec#, the authors of [67] use the terms client failures and provider failures for the domain and range failures, respectively. Client failures correspond to parameter validation, whereas provider failures occur when a procedure cannot perform the task it is supposed to. Moreover, provider failures are divided into admissible failures and observed program errors. To support admissible failures, Spec# provides checked exceptions and throws sets. In contrast, an observed program error occurs if the failure is due to an intrinsic error in the program (for e.g. an array bounds error) or a global failure that is not tied to a particular procedure (for e.g. an out-of-memory error). Such failures are signaled by Spec# with the use of unchecked exceptions, which do not need to be listed in the procedure's throws set. Likewise for Java, its type system has been used to track a class of admissible failures through checked exceptions.

However, omitting the tracking of unchecked exceptions is a serious shortcoming, since it provides a backdoor for some programmers to deal exclusively with unchecked exceptions. This shortcut allows programmers to write code without specifying or catching any exceptions. Although it may seem convenient to the programmer, it sidesteps the intent of the exception handling mechanisms and makes it more difficult for others to use the code. A fundamental contradiction is that, while unchecked exceptions are regarded as program errors that are not meant to be caught by handlers, the runtime system continues to support the handling of both checked and unchecked exceptions.

Here we show how it is possible to ensure a higher level of exception safety, as defined by

Stroustrup [94] and extended by Li and co-authors [70], which takes into consideration both checked and unchecked exceptions. According to Stroustrup, an operation on an object is said to be exception safe if it leaves the object in a valid state when it is terminated by throwing an exception. Based on this definition, exception safety can be classified into four guarantees of increasing quality:

- **No-leak guarantee**: For achieving this level of exception safety, an operation that throws an exception must leave its operands in well-defined states, and must ensure that every resource that has been acquired is released. For example, all memory allocated must be either deallocated or owned by some object whenever an exception is thrown.

- **Basic guarantee**: In addition to the no-leak guarantee, the basic invariants of classes are maintained, regardless of the presence of exceptions. To illustrate, consider the following Java example: a wrapper class `List` with two members, (i) `ll`, an instance of a linked list class `Ll` and (ii) `size`, an integer capturing the size of the linked list. Assuming that the append operation in the `Ll` class might throw an error, then we can define an append operation in the `List` that does not satisfy the **basic guarantee**:

```
void append(List lst_wrap; Object o)
{ lst_wrap.size++;
  lst_wrap.ll.append(o);}
```

  By incrementing `size` irrespective of the outcome of `ll.append(o)` then the invariant that `size` captures the length of the `ll` list might be broken.

- **Strong guarantee**: In addition to providing the basic guarantee, the operation either succeeds, or has no effects when an exception occurs. That is, for the exceptional case, all intermediary steps taken need to be unrolled. In other words, if an exception occurs, the only observable effects of the operation should be the raising of the exception.

- **No-throw guarantee**: In addition to providing the basic guarantee, the operation is guaranteed not to throw an exception.

We propose a methodology for program verification that can guarantee higher levels of exception safety. Our approach can deal with all kinds of control flows, including both checked and unchecked exceptions, thus avoiding the unsafe practice of using the latter to circumvent exception handling. Another aspect worth mentioning is that some of the aforementioned exception safety guarantees might be expensive. For instance, strong guarantee might incur the high cost of roll-back operations as it pushes all the recovery mechanism into the callee. However, such recovery may also be performed by the caller, or there might be cases when it is not even required. Consequently, in §3.2 we improve on the definition of strong guarantee for exception safety.

Moreover, verifying a strong guarantee is generally more expensive than the verification of a weaker guarantee, as it is expected to generate more complex proof obligations (details can be found in §3.4). Hence, according to the user's intention, our system can be tuned to enforce different levels of exception safety guarantees.

3.2 Examples with Higher Exception Safety Guarantees

We next illustrate our new specification mechanism through a few examples. Let us first consider the following class and predicate definitions.

$$\texttt{class node}\ \{\ \texttt{int val};\ \texttt{node next}\}$$

$$\begin{array}{l}\texttt{pred self::ll}\langle\texttt{n}\rangle \equiv \texttt{self}{=}\texttt{null} \wedge \texttt{n}{=}0 \vee \\ \qquad \exists \texttt{r}\cdot\texttt{self::node}\langle\_,\texttt{r}\rangle * \texttt{r::ll}\langle\texttt{n}{-}1\rangle\ \textbf{inv}\ \texttt{n}{\geq}0;\end{array}$$

Predicate `ll` defines a linear-linked list of length `n`. As elaborated earlier, each predicate describes a data structure, which is a collection of objects reachable from a base pointer denoted by `self` in the predicate definition. The expression after the **inv** keyword captures a pure formula that always holds for the given predicate.

Let us now consider the method list_alloc allocating memory for a linear-linked list to be pointed by x. The precondition requires x to be null, while the postcondition asserts that either no error was raised, i.e. the flow is norm, and the updated x points to a list with n elements, or an out of memory exception was raised, i.e. the flow is out_of_mem_exc, and x remains null.

Method list_alloc calls an auxiliary method list_alloc_helper which recursively allocates nodes in the list. If an out of memory exception is raised, the latter performs a rollback operation, inside a try-catch block, during which it frees all the memory that was acquired up to that point. Take note that, in the following examples, for illustration purposes, we provide an alternative set of library methods with explicit memory deallocation (via method list_dealloc) that coexists with our Java like language.

```
void list_alloc(int n, ref node x)
   requires x=null ∧ n≥0
   ensures (x′::ll⟨n⟩ ∧ flow=norm) ∨ (x′=null ∧ flow=out_of_mem_exc);
 {list_alloc_helper(n, 0, x);}
```

```
void list_alloc_helper(int n, int i, ref node x)
   requires x::ll⟨i⟩ ∧ n≥i ∧ i≥0
   ensures (x′::ll⟨n⟩ ∧ flow=norm) ∨ (x′=null ∧ flow=out_of_mem_exc);
{
 if(n>i)
   { try
     { x = new node(0, x); }
     catch(out_of_mem_exc exc)
       { list_dealloc(i, x);
         raise (new out_of_mem_exc());
       }
     list_alloc_helper(n, i+1, x);
     }
}
```

According to [94], method `list_alloc` is said to ensure a strong guarantee on exception safety, as it either succeeds in allocating all the required memory cells, $x'::ll\langle n\rangle \wedge flow=norm$, or it has no effect, $x'=null \wedge flow=out\_of\_mem\_exc$. However, this rollback operation can be expensive if we have been building a long list. Moreover, there can be cases when such rollbacks are not needed. For instance, in the context of the aforementioned method, a caller might actually accept a smaller amount of the allocated memory.

We propose to improve Stroustrup's definition on strong guarantee as follows: An operation is considered to provide a strong guarantee for exception safety if, in addition to providing basic guarantees, it either succeeds, or its effect is precisely known to the caller. Given this new definition, the method `list_alloc_helper_prime` defined below is also said to satisfy the strong guarantee. Compared to method `list_alloc_helper`, the newly revised method has an extra pass-by-reference parameter, `no_cells`, to denote the number of memory cells

that were already allocated. Through this output parameter, the caller is duly informed on the length of list that was actually allocated. This method has two possible outcomes :

- it succeeds and all the required memory cells were allocated:
  $\texttt{x}'::\texttt{ll}\langle\texttt{n}\rangle \wedge \texttt{no\_cells}'=\texttt{n}$;
- an out of memory exception is thrown and no_cells captures the number of successfully allocated memory cells. The caller is duly informed on the amount of acquired memory through the no_cells parameter, and could either use it, run a recovery code to deallocate it, or mention its size and location to its own caller.

```
void list_alloc_helper_prime(int n, int i, ref node x, ref int no_cells)
   requires x::ll⟨i⟩ ∧ n≥i ∧ i≥0
   ensures (x'::ll⟨n⟩ ∧ no_cells'=n ∧ flow=norm)∨
           (x'::ll⟨no_cells'⟩ ∧ flow=out_of_mem_exc);
{
  if(n>i)
    { try
       { x = new node(0, x); }
     catch(out_of_mem_exc exc)
       { no_cells=i;
         raise (new out_of_mem_exc());
       }
     list_alloc_helper(n, i+1, x);
    }
  else no_cells=n;
}
```

Next, we will illustrate how to make use of our verification mechanism for enforcing the

no-throw guarantee from [94]. For this purpose, let us consider the following `swap` method which exchanges the data fields stored in two disjoint nodes.

```
void swap(node x, node y)
  requires x::node⟨v1, q1⟩∗y::node⟨v2, q2⟩
  ensures x::node⟨v2, q1⟩∗y::node⟨v1, q2⟩ ∧ flow=norm;
{ int tmp = x.val;
   x.val = y.val;
   y.val = tmp;
}
```

The precondition requires the nodes pointed by `x` and `y`, respectively, to be disjoint, while the postcondition captures the fact that the values stored inside the two nodes are swapped. Additionally, the postcondition asserts that the only possible flow at the end of the method is the normal flow, `flow=norm`, i.e. no exception was raised. This specification meets the definition of no-throw guarantee given in [94]. Any presence of exception, including an exception caused by null pointer dereferencing, would require the postcondition to explicitly capture each such exceptional flow. Conversely, any absence of exceptional flow in our logic is an affirmation for the no-throw guarantee. We have shown how our specification language can capture various kinds of control flow types and how this can be used to specify different levels of exception safety guarantees. In the next section we will outline a Hoare logic that can be used to verify such specifications.

3.3 Verification for Unified Control Flows

As we mentioned in the previous chapters, we construct our Hoare logic with the `Core−U` as the target language. The beauty of having a properly designed high level core language as a stepping stone is that it allows really clean and simple forward verification rules.

The rules describe the construction of Hoare-style triples $\vdash \{\Delta_1\}\, e\, \{\Delta_2\}$, where Δ_1 is a captured program state whereby $\texttt{flow} = \top$, while Δ_2 is a disjunctive heap state capturing the entire set of control flows that may escape during the execution of e. For example, we may have:

$$\Delta_2 = \quad (\texttt{x}'{=}\texttt{x}{+}\texttt{1} \wedge \texttt{flow}{=}\texttt{norm}) \vee (\texttt{x}'{=}\texttt{x} \wedge \texttt{flow}{=}\texttt{exc})$$

to denote two possible final states, a state with normal control flow describing an increment in variable x and a second state in which an exception occurs and the state of x is unchanged.

In the remainder of the current section we will focus mainly on the verification of the unified throw construct, #, and the try-catch construct. We give the formulations for the rules for these constructs in Figure 3.1. We delay the presentation of the barrier rule until Chapter 4. Also since the other rules are largely conventional we omit them.

$$\mathit{split}(\Delta, c, fv, v) = \left(\begin{array}{c} [\textbf{FV}{-}[\textbf{SPLIT}]] \\ \exists \texttt{flow}\,.[\texttt{res}{\mapsto}v]\Delta \wedge \texttt{flow} <: c \wedge \texttt{flow} = fv, \\ \Delta \wedge \neg(\texttt{flow} <: c) \end{array}\right)$$

$$\frac{[\textbf{FV}{-}[\textbf{TRY}{-}\textbf{CATCH}]] \quad \vdash \{\Delta\}\, e_1\, \{\Delta_1\} \quad (\Delta_2, \Delta_3) = \mathit{split}(\Delta_1, c, fv, v) \quad \vdash \{\Delta_2\}\, e_2\, \{\Delta_4\}}{\vdash \{\Delta\}\, \texttt{try}\ e_1\ \texttt{catch}\ (c@fv\ v)\ e_2\, \{\Delta_3 \vee \exists v, fv \cdot \Delta_4\}}$$

$$\frac{[\textbf{FV}{-}[\textbf{OUTPUT}]] \quad \mathit{resolve}(\Delta, ft) = fset \quad \Delta_1{=}\exists \texttt{res}.(\Delta \wedge \texttt{flow} = fset) \wedge \texttt{res} = a}{\vdash \{\Delta\}\, ft\#a\, \{\Delta_1\}}$$

$$\mathit{resolve}(\Delta, Ex(ft)) = Ex(ft) \qquad \mathit{resolve}(\Delta \wedge (fv{=}ft), fv) = ft$$

$$\frac{\Delta \vdash \exists fv.v{=}(fv, \_) \quad \mathit{resolve}(\Delta, fv) = ft}{\mathit{resolve}(\Delta, v.1) = ft} \qquad \frac{v\ \mathit{was\ declared\ of\ type}\ ft}{\mathit{resolve}(,\texttt{ty}(v)) = ft}$$

Figure 3.1: Some Verification Rules

Notice that the # statement sets on one hand the control flow to a value denoting one of the subtrees from the control flow hierarchy and on the other hand it also captures the result of the operation. To capture this behaviour, we require each control flow variable to be resolved to an

appropriate flow set ($fset$) value, before we can set it as the current control flow by assigning it to the `flow` variable. We rely on an auxiliary operation, $resolve$ to obtain the corresponding control flow set from a given program state.

The verification system makes use of the `res` variable in order to store the result of the current operation. Note that for the (`fv#v`) statement, the result value is indicated by v, which is bound to the `res` variable as seen in the [**OUTPUT**] rules.

Regarding the [**FV–TRY–CATCH**] rule, we first compute the post-state of expression $\texttt{e}_1$ as Δ_1. Since Δ_1 may capture a range of control flows, it has to be split into two components, Δ_2 and Δ_3, with the help of the [**FV–SPLIT**] rule. The Δ_2 component will model the program states with control flows that can be captured by the catch handler, whereas Δ_3 will model those states with control flows that escape from the catch handler. Moreover, for the case when the control flow is being caught by the handler, the control flow type is bound to fv, and its thrown value is bound to v. These bindings are kept in Δ_2 which is made available as the pre-state for $\texttt{e}_2$. Keeping the bindings ensures that although `flow` has been quantified away in Δ_2 the information can still be available to the catch handler through fv. As these local variables are only valid in the catch handler, we quantify them away in the resulting postcondition.

Before further extending our verification logic to handle barriers as well, we first report on integration of exception handling constructs into HIP/SLEEK, an existing verification toolset.

3.4 Experiments

In order to prove the viability of our verification method we tried our prototype implementation against a few examples from SPECjvm2008[1], a widely used Java benchmark created by SPEC. Due to the focus of our work, we only considered those tests from SPECjvm2008 that are related to exception handling. After annotating the tested methods with pre and post conditions, we were able to successfully verify all the tests. Due to the fact that the KeY[6] approach is semi-automated in the sense that it occasionally prompts the user for choice of

rewriting rules, while our approach is fully-automated, we cannot perform a direct comparison of the verification timings.

Among these examples, MyClass is a Java program emphasizing the use of exception handling in the presence of user defined exceptions. Its aim is to detect mishandling of the exception class hierarchy. The main objective of While and ContLabel examples is testing abrupt termination in the presence of loops. PayCard is a Java class from a real life Java application that makes heavy use of exceptions while modelling the behaviour of a credit card.

Figure 3.2 contains the timings obtained when using our system to verify the aforementioned examples.

Programs	CODE LOC	ANN LOC	Time (seconds)	Focus
Break (KeY)	17	3	0.11	`break` handling
MyClass (KeY)	31	2	0.10	exception hierarchy
While (KeY)	91	49	2.47	while loops and `break`
ContLabel (KeY)	82	18	0.95	imbricated while loops and `continue`
PayCard (KeY)	54	16	0.91	general exception handling
SPECjvm2008	175	15	1.20	general exception handling

Figure 3.2: Verification Times

We also verified the examples presented throughout chapter. Method `list_alloc` was verified in 0.41 seconds, `list_alloc_helper_prime` in 0.26 seconds and method `swap` in 0.09 seconds. Take note that the verification of `list_alloc_helper_prime`, which ensures an improved strong exception safety guarantee as according to our approach, is faster by 36% than the verification of `list_alloc` which enforces the original strong guarantee defined in [94].

3.5 Summary

We have presented a new approach to the verification of exception-handling programs based on a specification logic that can uniformly handle exceptions, program errors and other kinds

of control flows. The specification logic is currently built on top of the formalism of separation logic, as the latter can give precise descriptions to heap-based data structures. Our main motivation for proposing this new specification logic is to adapt the verification method to help ensure exception safety in terms of the four guarantees of increasing quality introduced in [94] and extended in [70], namely no-leak guarantee, basic guarantee, strong guarantee and no-throw guarantee. During the evaluation process, we found the strong guarantee to be restrictive for some scenarios, as it always forces a recovery mechanism on the callee, should exceptions occur. Hence, we propose to generalise the definition of strong guarantee for exception safety. Our approach has been formalised and implemented in a prototype system, and tested on a suite of exception-handling examples. We hope it would eventually become a useful tool to help programmers build more robust software.

Chapter 4

Barrier Verification

In this chapter we further extend the verification logic introduced in the previous chapter by adding support for barrier synchronization. To this aim we introduce barrier definitions, a novel mechanism for describing barrier behaviours. We were able to introduce a Hoare rule for verifying barrier calls which is surprisingly simple when compared to the complex synchronization pattern the barriers introduce.

4.1 Motivation

In a shared-memory concurrent program, threads communicate via a common memory. Programmers use synchronization mechanisms, such as critical sections and locks, to avoid data races. In a data race, threads "step on each others' toes" by using the shared memory in an unsafe manner. Recently, concurrent separation logic has been used to formally reason about shared-memory programs that use critical sections and (first-class) locks [80, 51, 43, 50]. Programs verified with concurrent separation logic are provably data-race free.

What about shared-memory programs that use other kinds of synchronization mechanisms, such as semaphores? The usual assumption is that other mechanisms can be implemented with locks, and that reasonable Hoare rules can be designed based on the particular implemen-

tation. Indeed, the first published example of concurrent separation logic was implementing semaphores using critical sections [80]. Unfortunately, not all synchronization mechanisms can be easily reduced to locks in a way that allows for a reasonable Hoare rule to be derived. In this chapter we introduce a Hoare rule that *natively* handles one such synchronization mechanism, the Pthreads-style barrier.

Pthreads (POSIX Threads) is a widely-used API for concurrent programming, and includes various procedures for thread creation/destruction and synchronization [17]. When a thread issues a barrier call it waits until a specified number (typically all) of other threads have also issued a barrier call; at that point, all of the threads continue. Although barriers do not get much attention in theory-oriented literature, they are very common in numerical applications code. PARSEC is the standard benchmarking suite for multicore architectures, and has thirteen workloads selected to provide a realistic cross-section for how concurrency is used in practice today; a total of five (38%) of PARSEC's workloads use barriers, covering the application domains of financial analysis (blackscholes), computer vision (bodytrack), engineering (canneal), animation (fluidanimate), and data mining (streamcluster) [10].

A common use for barriers is managing large numbers of threads in a pipeline setting. For example, in a video-processing algorithm, each thread might read from some shared common area containing the most recently completed frame while writing to some private area that will contain some fraction of the next frame. (A thread might need to know what is happening in other areas of the previous frame to properly handle objects entering or exiting its part of the current frame.) In the next iteration, the old private areas become the new shared common area as the algorithm continues.

Our key insight is that a barrier is used to simultaneously redistribute ownership of resources (typically, permission to read/write memory cells) between multiple threads. In the video-processing example, each thread starts out with read-only access to the previous frame and write access to a portion of the current frame. At the barrier call, each thread gives up its write access to its portion of the (just-finished) frame, and receives back read-only access to

the entire frame.

Separation logic (when combined with fractional permissions [14, 28]) can elegantly model this kind of resource redistribution. Let Pre_i be the precondition satisfied by thread i when it enters the barrier, and $Post_i$ be the postcondition that will hold after thread i is released; then the following equation, stating that all the resources in the preconditions are carried unchanged in the postcondition, is *almost* true:

$$\circledast_i Pre_i \quad = \quad \circledast_i Post_i \tag{4.1}$$

Pipelined algorithms often operate in stages. Since barriers are used to ensure that one computation has finished before the next can start, the barriers need to have stages as well—a piece of ghost state associated with the barrier. We model this by building a finite automaton into the barrier definition. We then use the assertion, written $b{::}bname^{\pi}\langle cs, \overrightarrow{v}\rangle$, which says that the barrier pointed to by b, of type $bname$ corresponding to a barrier definition[1], owned with fractional permission π, is currently in state cs. Furthermore, arguments $\overrightarrow{v}$ are used to denote the memory locations guarded by the barrier. The state of a barrier changes exactly as the threads are released from the barrier. We can correct equation (4.1) by noting that barrier b is transitioning from state cs (current state) to state ns (next state), and that the other resources (frame F) are not modified:

$$\begin{aligned} \circledast_i Pre_i \quad &= \quad F * b{::}bname^{\blacksquare}\langle cs, \overrightarrow{v}\rangle \\ \circledast_i Post_i \quad &= \quad F * b{::}bname^{\blacksquare}\langle cs, \overrightarrow{v}\rangle \end{aligned} \tag{4.2}$$

We use the symbol $\blacksquare$ to denote the full ($\sim$100%) permission, which we require so that no thread has a "stale" view of the barrier state. Although the on-chip (or *erased*) operational behavior of a barrier, as described in Chapter 2, is conceptually simple[2], it may be already apparent that

[1] the barrier name, $bname$, is akin to the predicate names for user-defined predicates. In §4.2 we will give a concrete example of a barrier definition

[2] Suspend each thread as it arrives; keep a counter of the number of arrived threads; and when all of the threads have arrived, resume the suspended threads.

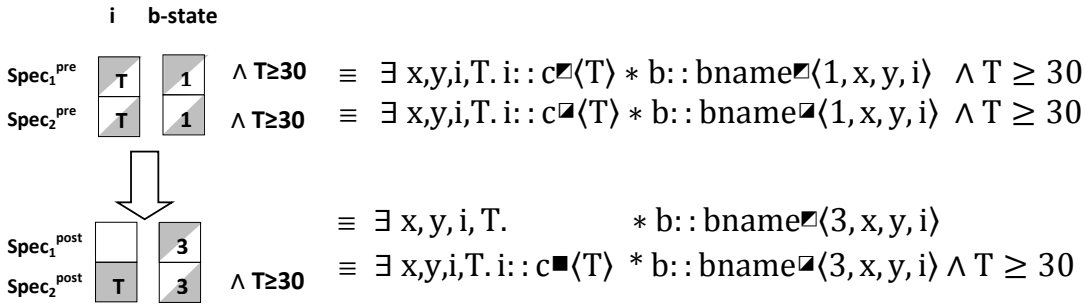
the verification can rapidly become quite complicated.

4.2 Example

In this section we use an example to outline the key observations and give the intuitions behind our approach to barrier verification. Our example consists of two threads implementing a simple map-reduce operation: at each iteration through a loop, the two threads work in parallel to first produce some data and store it in shared variables x and y (one for each thread) while using data from variable i; secondly one thread centralizes the data produced and generates a new value for i. The x, y and i variables have type c which is a data structure with one integer field named val.

Note that in this two thread example, other synchronization primitives could be employed with similar overhead as barriers. However, also note that if the example is extended to a larger number of threads, barriers would handle the thread count increase without any change however the other synchronization schemes would need significant changes, implicitly increasing the complexity of the synchronization. We give the code outline for the two threads in Figure 4.1.

In Figure 4.1 we give a pictorial representation of the state machine associated with the barrier used in the code snippets. Our pictorial representation relies on the following specialized notation:



This pictorial representation is used to express the pre- and postconditions for a given

0: $\{x::c^{\blacksquare}\langle 0\rangle * y::c^{\blacksquare}\langle 0\rangle * i::c^{\blacksquare}\langle 0\rangle * b::bname^{\blacksquare}\langle 0, x, y, i\rangle\}$

0':	$\{x::c^{\blacksquare}\langle 0\rangle * y::c^{\blacksquare}\langle 0\rangle * i::c^{\blacksquare}\langle 0\rangle * b::bname^{\blacksquare}\langle 0, x, y, i\rangle\}$	$\{x::c^{\blacksquare}\langle 0\rangle * y::c^{\blacksquare}\langle 0\rangle * i::c^{\blacksquare}\langle 0\rangle * b::bname^{\blacksquare}\langle 0, x, y, i\rangle\}$
1:	`barrier` b;	`barrier` b; // b transitions 0→1
2:	`while` $i.val < 30$ {	`while` $i.val < 30$ {
3:	$x.val := i.val + 1$;	$y.val := i.val + 2$;
4:	`barrier` b;	`barrier` b; // b transitions 1→2
5:	$i.val := (x.val + 2 * y.val)$;	
6:	`barrier` b;	`barrier` b; // b transitions 2→1
	}	}
7:	`barrier` b;	`barrier` b; // b transitions 1→3
8:	$i.val := 0$;	
	...	...

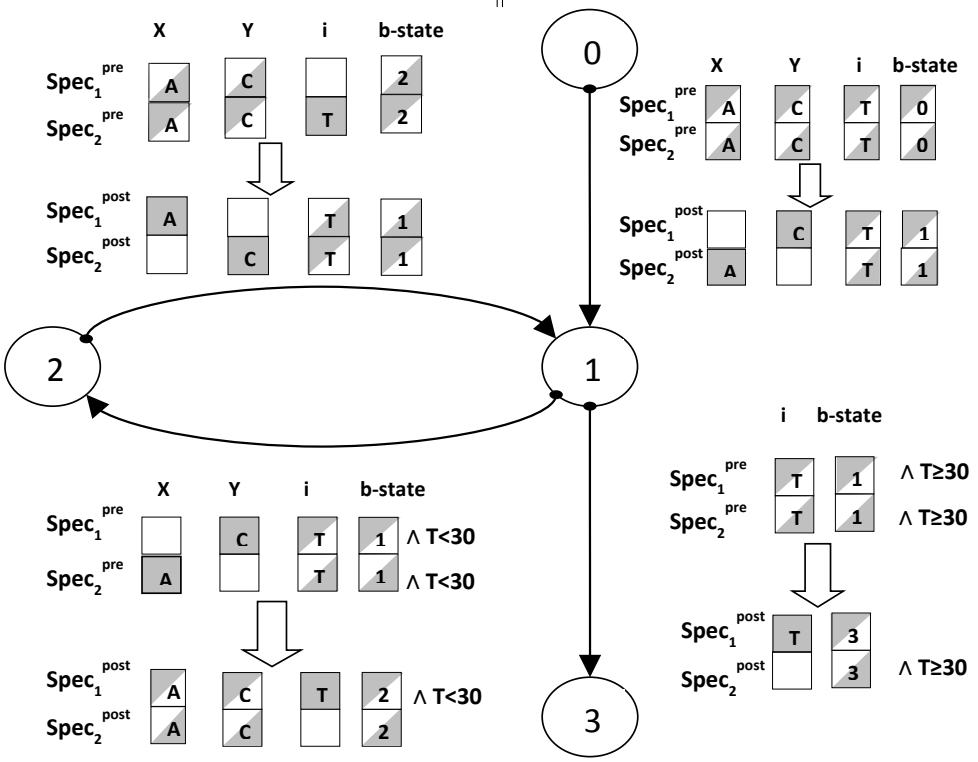


Figure 4.1: Example: Code and Barrier Diagram

barrier transition. Each row is a pictorial representation (values, barrier states, and shares) of a formula in separation logic as indicated above. In the above pictorial representation, the preconditions are given in the upper block (one per row) and the postconditions in the lower block.

In our pictorial representation, each row is associated with a *specification*; specification $Spec_1$ is a pair of the precondition $Spec_1^{pre}$ and the postcondition $Spec_1^{post}$, etc. A barrier that is waiting for n threads will have n specs; n can be less than the total number of threads. We do not require that a given thread always satisfies the same specification each time it reaches a given barrier transition.

Note that only the permissions on the memory cells change during a transition; the contents (values) do not.[1] The exception to this is the special column on the right side, which denotes the assertion associated with the barrier itself. As the barrier transitions, this value changes from the previous state to the next; we require that the sum of the preconditions includes the full share of the barrier assertion to guarantee that no thread has an out-of-date view of the barrier's state. Observe that all of the preconditions join together, and, except for the state of the barrier itself, are exactly equal to the join of the postconditions.

Each thread owns part of the entire state, described on line 0. The partitioning is given on line 0', with the left thread (A) and right thread (B) owning the shares ◩ and ◪, respectively of each resource.

The real action starts with the barrier call on line 1, which ensures that both threads start at the same time. Thread A satisfies spec 1 and thread B satisfies spec 2. Afterwards, thread A has full ownership over x and thread B has full ownership over y; the ownership of i remains split between A and B. While the ownership of the barrier is unchanged, it is now in state 1.

We then enter the main loop on line 2. On line 3, both threads read from the shared cell i and both threads update their fully-owned cell. The barrier call on line 4 ensures that these

[1] We use the same quantified variable names before and after the transition because an outside observer can tell that the values are the same. A local verification can use ghost state to prove the equality; alternatively we could add the ability to move the quantifier to other parts of the diagram, *e.g.*, over an entire pre-post pair.

updates have been completed before the threads continue. Since the value T at memory location i is less than 30, only the 1–2 transition is possible; the 1–3 transition requires T$\geq$ 30. Thread A satisfies spec 1 and thread B satisfies spec 2; afterwards, both threads have partial shares of x and y, thread A has the full share of i; the barrier is in state 2.

On line 5 , thread A updates cell i. The barrier on line 6 ensures that the update on line 5 has completed before the threads continue; thread A satisfies spec 2 while thread B satisfies spec 1[1]. Afterwards, the threads have the same permissions they had on entering the loop: A has full ownership of x, B has full ownership of y, and they share ownership of i; the barrier is again in state 1.

After 30 iterations, the loop exits and control moves to the barrier on line 7. Observe that since the (shared) value T at memory location i is greater than or equal to 30, only the 1–3 transition is possible; the 1–2 transition requires T$<$ 30. Thread A satisfies spec 1 while thread B satisfies spec 2; afterwards, both threads are sharing ownership of x, y (since the transition from 1 to 3 does not mention x and y, they are unchanged). Thread A has full permission over the condition variable i; the barrier is in state 3. Finally, on line 8, thread A updates i; the barrier on line 8 ensures that thread B's read of i associated with the while condition test on line 2 has already occurred.

4.3 Barrier Definitions and Consistency Requirements

Conceptually, a barrier definition is as a set of barrier states, each barrier state containing a set of possible transitions described by specifications. In this section we use two data structure types, one for barrier transitions and one for barrier definitions in order to clearly describe and group the consistency requirements imposed on barrier definitions. The two structures are outlined in Figure 4.2.

[1]Note that the threads are not constrained by the rules of our logic to always satisfy the same spec for a given transition.

BarDef	$\equiv$	$\{$ bd_bname : *String*	barrier id
(barrier definition)		bd_limit : *Nat*	# of threads
		bd_transitions : *list* BarSpecList$\}$	transition list
BarSpecList	$\equiv$	$\{$bt_ns : *Nat*	next state
(transition)		bt_cs : *Nat*	current state
		bt_specs : *list* (assert $\times$ assert)$\}$	pre/post pairs

Figure 4.2: Barrier Definitions

Definition 4.3.1 (Wellformed barrier definitions). *A barrier definition (*BarDef*) consists of a barrier name, the number of threads synchronized, and a transition list; such that all transitions are well-formed and for any given state, outgoing transitions are* **mutually exclusive**.

Definition 4.3.2 (Wellformed barrier transition). *A transition (*BarSpecList*) contains a barrier name (*bname*), the current state identifier (*cs*), a next state identifier (*ns*), and the list of precondition/postcondition pairs (the spec list bt_specs). We require that:*

1. *the length of list of specs (the bt_specs list) matches the limit in the containing* BarDef. [1]

2. *all of the pre/postconditions in the spec list ignore the store (stack), focusing only on the memory and barrier map. Since stores are private to each thread (on a processor these would be registers), it does not make sense for them to be mentioned in the "public" pre/post conditions.*

3. *all of the preconditions in the spec list are* precise. *Precision is a technical property involving the identifiability of states satisfying an assertion. An assertion P is precise when:*

$$\frac{\sigma_1 \oplus \sigma_2 = \sigma_3 \qquad \sigma_1 \models P \qquad \sigma_1' \oplus \sigma_2' = \sigma_3 \qquad \sigma_1' \models P}{\sigma_1 = \sigma_1'}$$

[1] A command to dynamically alter the number of threads managed by a barrier might allow different states/transitions to wait for different numbers of threads. However, in this proposal we do not consider such a command.

That is, P can hold on at most one substate of an arbitrary state σ_3. [1]

4. *each precondition P includes some positive share of the* barrier *assertion with* b, cs *and* $\overrightarrow{v}$, i.e., $\exists\pi.\ P \Rightarrow \top * \mathsf{b}::bname^{\pi}\langle \mathsf{cs}, \overrightarrow{v}\rangle$.

5. *the sum of the preconditions must equal the sum of the postconditions, except for the state of the barrier; moreover, the sum of the preconditions must include the full share of the barrier, that is the individual shares of barrier* b *owned by the threads synchronizing on* b *must add-up to the full share* ■ *(equation (4.2), repeated here):*

$$\begin{aligned} \mathop{\ast}_i Pre_i \quad &= \quad F * \mathsf{b}::bname^{\blacksquare}\langle \mathsf{cs}, \overrightarrow{v}\rangle \\ \mathop{\ast}_i Post_i \quad &= \quad F * \mathsf{b}::bname^{\blacksquare}\langle \mathsf{cs}, \overrightarrow{v}\rangle \end{aligned}$$

Item 1 is simple bookkeeping; items 2–4 are similar to technical requirements present in other variants of concurrent separation logic [80, 50, 43]. As previously mentioned, property 5 of Definition 4.3.2 is the fundamental insight of this approach .

Although the requirement that assertions are ”tokens” might seem a too restrictive, in practice it did not prove to be a hindrance on barrier reasoning. In particular during our experiments it has not affected the relative completeness of our reasoning. Furthermore, we postulate that ”token” assertions are expressive enough to maintain relative completeness.

We define the function lookup_spec in order to simplify the lookup of a spec in a BarDef: lookup_spec$(bd, cs, ns, spec)$ returns the pre/post pair on position $spec$ in the spec list corresponding to the transition from state cs to state ns in barrier definition bd.

Using this notation, we can express the important requirement that all transitions from the

[1] Precision may not be required; another property (tentatively christened “token”) that might serve would be if, for any precondition P, $P * P \equiv false$. Note that precision in conjunction with item (3) implies P is a token.

barrier state cs of the barrier definition bd are mutually exclusive:

$$\begin{array}{l}\forall tr_1, tr_2, spec_1, spec_2, pre_1, pre_2. \\ \quad tr_1 \neq tr_2 \ \wedge \ \mathsf{lookup\_spec}(bd, cs, tr_1, spec_1) = (pre_1, \_) \wedge \\ \quad\ \mathsf{lookup\_spec}(bd, cs, tr_2, spec_2) = (pre_2, \_) \ \Rightarrow \\ \quad\quad (\top * pre_1) \ \wedge \ (\top * pre_2) \ \equiv \ false\end{array}$$

In other words, it is *impossible* for any of the preconditions of more than one transition (of a given state) to be true at a time. The simplest way to understand this is to consider the 1–2 and 1–3 transitions in the example program. The 1–2 transition requires that the value in memory cell i be strictly less than 30; in contrast, the 1–3 transition requires that *the same cell* contains a value greater than or equal to 30. Plainly these are incompatible; but in fact the above property is stronger: *both* of the specs on the 1–2 transition, and *both* of the specs on the 1–3 transition include the incompatibility. Thus, if thread A takes transition 1–2, it knows for certain that thread B *cannot* take transition 1–3. This way we ensure that both threads always agree on the barrier's current state.

4.4 Hoare Logic

In this section we extend the verification rules presented in §3.3 to include barrier support. Our Hoare judgment has the form $\Gamma \vdash \{P\}\ c\ \{Q\}$, where Γ is a list of barrier definitions as given in §4.3, P and Q are assertions in separation logic, and c is a command. Our Hoare rules come in three groups: standard Hoare logic (Skip, If, Assignment, Consequence); standard separation logic (Frame, Store, Load, New, Free); and the barrier rule. While the first two groups are rather standard the highlight of this proposal is the barrier rule, as given below:

$$\frac{\Gamma[b] = bd \qquad \mathsf{lookup\_spec}(bd, cs, tr, spec) = (P, Q)}{\Gamma \vdash \{P\}\ \texttt{barrier}\ b\ \{Q\}}\ \text{Barrier}$$

The Hoare rule for barriers is so simple that at first glance it may be hard to understand. The ghost variables for the current state cs, transition tr, and specification $spec$ appear to be free in the lookup_spec! However, things are not quite as unconstrained as they initially appear. Recall from §4.3 that one of the consistency requirements for the precondition P is that P implies an assertion about the barrier itself: $P \Rightarrow Q * b{::}bname^{\pi}\langle cs, \overrightarrow{v}\rangle$; thus at a given program point we can only use transitions and specifications from the current state. Similarly, recall from §4.3 that since the transitions are mutually exclusive, tr is uniquely determined.

This leaves the question of the uniqueness of $spec$. If a thread only satisfies a single precondition, then the spec $spec$ is uniquely determined. Unfortunately, it is simple to construct programs in which a thread enters a barrier while satisfying the preconditions of multiple specs. What saves us is that we are developing a logic of partial correctness. Since preconditions to specs must be precise and nonempty (*i.e.*, token), only one thread is able to satisfy a given precondition at a time. The pigeonhole principle guarantees that if a thread holds multiple preconditions then some other thread will not be able to enter the barrier; in this case, the barrier call will never return and we can guarantee any postcondition.

We now apply the Barrier rule to the barrier calls in line 6 from our example program; the

lookup_specs are direct from the barrier state diagram:

$$\text{Thread A}\ \begin{cases} \mathsf{lookup\_spec}(b,2,1,2) = (P,Q) \\ P = x{::}c^{\blacktriangleleft}\langle A\rangle * y{::}c^{\blacktriangleleft}\langle C\rangle * i{::}c^{\blacksquare}\langle T\rangle * b{::}bname^{\blacktriangleleft}\langle 2,x,y,i\rangle \\ Q = y{::}c^{\blacksquare}\langle C\rangle * i{::}c^{\blacktriangleleft}\langle T\rangle * b{::}bname^{\blacktriangleleft}\langle 1,x,y,i\rangle \\ \hline \Gamma \vdash \{P\}\ \texttt{barrier}\ b\ \{Q\} \end{cases}$$

$$\text{Thread B}\ \begin{cases} \mathsf{lookup\_spec}(b,2,1,1) = (P,Q) \\ P = x{::}c^{\blacktriangleright}\langle A\rangle * y{::}c^{\blacktriangleright}\langle C\rangle * b{::}bname^{\blacktriangleright}\langle 2,x,y,i\rangle \\ Q = x{::}c^{\blacksquare}\langle A\rangle * i{::}c^{\blacktriangleright}\langle T\rangle * b{::}bname^{\blacktriangleright}\langle 1,x,y,i\rangle \\ \hline \Gamma \vdash \{P\}\ \texttt{barrier}\ b\ \{Q\} \end{cases}$$

Note that in this line of the example program, the frame is emp in both threads and that in the retrieved pre/post conditions (P, Q) variables x, y, i, A, C, T are existentially quantified over P and Q respectively. However, through the common view of the barrier predicate for b it is ensured that the bindings for x, y and i are consistent in the precondition P and postcondition Q.

4.5 Soundness Results

This section outlines the machinery developed for proving the soundness of our verification logic. We start by adding more comprehensive , albeit virtual, resource tracking features to the operational semantic presented earlier[1]. We outline the Coq proof of soundness for our verification logic with respect to the *unerased* semantics and finally we observe that the unerased semantic is a conservative approximation of the erased one, thus showing that the soundness result holds for both semantics.

[1] We use the term *unerased* for the resulting resource-aware concurrent operational semantics.

Various complex yet unrelated language features have already been properly formalized and their handling proved sound [4, 51] therefore including them in our proofs would not generate any ground breaking new insights with respect to the problem at hand but would serve only to clog-up the proof. In order to simplify the proof machinery, we discard several of the `Core−U` features orthogonal to the core issue of exception and barrier handling. Therefore we maintain in the programming language the following commands: `skip` (do nothing, syntactic sugar for `norm#TRUE`), $x := e$ (local variable assignment), $x := [e]$ (load from memory), $[e_1] := e_2$ (store to memory), x:= `new` e (memory allocation), `free` e (memory deallocation), `if` e `then` c_1 `else` c_2 (if-then-else), $e_1 \# e_2$ (sets current control flow type and return value), `try` e_1 `catch` $(ft[@v_1]\ v_2)\ e_2$ and `barrier` b (wait for barrier b).

We define five kinds of (tagged) values v: `TRUE`, `FALSE`, `ADDR`($\mathbb{N}$), `DATA`($\mathbb{N}$) and `FLOWS`($\mathbb{N}\ list$). `FLOWS` capture a flow type value, represented as a positive integer or an embedding of such values as lists of naturals. We have two (tagged) expressions e: `C`(v) and `V`(x), where x are local variable names. For simplicity, we model barrier names as positive integers. numbers, bname $\in \mathbb{N}$.

We also restrict the assertion language to: points-to assertions ($e_1 \overset{\pi}{\mapsto} e_2$), barrier assertions and flow assertions.

We follow the model presented in §2.4 and interpret the assertions with regard to thread states σ, triples of a store, heap, and barrier map ($\sigma = (s, h, b)$). The fractional points-to assertion, $e_1 \overset{\pi}{\mapsto} e_2$, means that the expression e_1 is pointing to an address a in memory; a is owned with positive share π, and contains the evaluated value v of e_2. The fractional points-to assertion does not include any ownership of the break. The barrier assertion, $b{::}bname^{\pi}\langle s \rangle$, means that the barrier b, owned with positive share π, is in state s. For simplicity but without a great loss of generality, we will assume that the set of shared variables is globally fixed for each barrier instance, therefore the barrier assertions have only one argument, the barrier state. Tracking these extra arguments would just serve to burden the proof with extra variable renaming tracking. Flow assertions describe the current flow type. By convention, we use one

fixed location in the stack as a special register recording the current flow indicated by the `flow` variable.

Our expressions e are evaluated only in the context of the store; we write $s \vdash e \Downarrow v$ to mean that e evaluates to v in the context of the store s. We denote the empty heap (which lacks ownership for both all memory locations and the distinguished break location) by h_0. Finally, the barrier map b is a partial function from barrier numbers to pairs of barrier states (represented as natural numbers) and positive shares; we denote the empty barrier map by b_0.

An *assertion* is a function from states to truth values. As is common, we define the usual logical connectives via a straightforward embedding into the metalogic; for example, the object-level conjunction $P \wedge Q$ is defined as $\lambda\sigma.\ (P\sigma) \wedge (Q\sigma)$. We will adopt the convention of using the same symbol for both the object-level operators and the meta-level operators to avoid symbol bloat; it should be clear from the context which operator applies in a given situation. We provide all of the standard connectives ($\top, \bot, \wedge, \vee, \Rightarrow, \neg, \forall, \exists$).

We model the connectives of separation logic in the standard way:

$$
\begin{array}{lcl}
\mathsf{emp} & = & \lambda(s,h,b).\ h = h_0 \ \wedge\ b = b_0 \\
P * Q & = & \lambda\sigma.\ \exists\sigma_1,\sigma_2.\ \sigma_1 \oplus \sigma_2 = \sigma \ \wedge\ P(\sigma_1) \ \wedge\ Q(\sigma_2) \\
e_1 \overset{\pi}{\mapsto} e_2 & = & \lambda(s,h,b).\ \exists a, v.\ (s \vdash e_1 \Downarrow \mathtt{ADDR}(a)) \ \wedge\ (s \vdash e_2 \Downarrow v) \wedge \\
 & & \quad b = b_0 \ \wedge\ h(a) = (v,\pi) \ \wedge\ \mathsf{dom}(h) = \{a\} \wedge \mathsf{break}(h) = \square \\
\mathsf{barrier}(bn,\pi,s) & = & \lambda(s,h,b).\ h = h_0 \ \wedge\ b(bn) = (s,\pi) \ \wedge\ \mathsf{dom}(b) = \{bn\} \\
\mathtt{flow}\ (e) & = & \lambda(s,h,b).\ h = h_0 \ \wedge\ b = b_0 \ \wedge\ \exists fl.\ (s \vdash e \Downarrow \mathtt{FLOWS}(fl) \\
 & & \quad \wedge\ s(0) = \mathtt{FLOWS}(fl))
\end{array}
$$

We also lift program expressions into the logic: $e \Downarrow v$, which evaluates e with σ's store (*i.e.*, $\lambda(s,h,b).\ h = h_0 \wedge b = b_0 \wedge s \vdash e \Downarrow v$); $[e]$, equivalent to $e \Downarrow \mathtt{TRUE}$; and $x = v$, equivalent to $\mathtt{V}(x) \Downarrow v$. These assertions have a "built-in" emp.

4.5.1 Unerased Semantics

In this section we introduce the unerased operational semantics which we use to prove the soundness of the verification logic. As with the erased version presented in Chapter 2 our unerased semantics is divided into three parts: purely sequential, concurrent and oracle. The main changes are related to the purely sequential and the concurrent semantics.

Purely sequential semantics. The bulk of the unerased purely sequential semantics is identical to the one presented in §2.3.2. The only "tricky" part is that the machine gets stuck if one tries to write to a location for which one does not have full permission or read from a location for which one has no permission; *e.g.*, here is the store rule:

$$\frac{\begin{array}{ccc} s \vdash e_1 \Downarrow \mathtt{C}(\mathtt{ADDR}(n)) & & s \vdash e_2 \Downarrow v \\ n < \mathsf{break}(h) & h(n) = (\blacksquare, v') & h' = [n \mapsto (\blacksquare, v)]h \end{array}}{\big((s,h,b),\ [e_1] := e_2; c\big) \ \mapsto\ \big((s,h',b),c\big)}\ \mathsf{sstep-store}$$

The test that $n < \mathsf{break}(h)$ ensures that the address for the store is "in bounds"—that is, less than the current value of the break between allocated and unallocated memory; since we are updating the memory we require that the permission associated with the location n be full ($\blacksquare$). We say that this step relation is *unerased* since these bounds and permission checks are virtual rather than on-chip.

Concurrent semantics. We define the notion of a *concurrent state* in Figure 4.3. A concurrent state contains a scheduler Ω (modeled as a list of natural numbers), a distinguished heap called the *allocation pool*, a list of *thread*s, and a *barrier pool*[1]. The allocation pool "owns" all of the unallocated memory cells and the "break" that indicates the division between allocated and unallocated cells. Before a thread runs, the allocation pool is transferred into the local heap

[1]There is also a series of consistency requirements such as the fact that all of the heaps in the threads and barrier pool join together with the allocation pool into one consistent heap

Cstate	$\equiv$	$\{$ cs_sched : list $\mathbb{N}$	schedule
		cs_allocpool : heap	alloc pool
		cs_thds : list Thread	thread pool
		cs_barpool : Barpool$\}$	barrier pool
Thread	$\equiv$	$\{$ th_stk : store	
		th_hp : heap	
		th_bs : BarrierMap	local view of barrier states
		th_ctl : conc_ctl$\}$	running or waiting
conc_ctl	$\equiv$	$\mid$ Running(c)	executing code c
		$\mid$ Waiting$(bn, tr, spec, c)$	waiting on bn
Barpool	$\equiv$	$\{$ bp_bars : list DyBarStatus	dynamic barrier status
		bp_st : store $\times$ heap $\times$ BarrierMap$\}$	current state
DyBarStatus	$\equiv$	$\{$ dbs_bn : $\mathbb{N}$	barrier id
		dbs_wp : Waitpool	waiting thread pool
		dbs_bd : BarDef$\}$	
Waitpool	$\equiv$	$\{$ wp_tr : $option\ \mathbb{N}$	transition id
		wp_slots : $option$ (list slot)	taken slots
		wp_limit : $\mathbb{N}$	
		wp_st : store $\times$ heap $\times$ BarrierMap$\}$	current state
slot	$\equiv$	(thread_id $\times$ heap $\times$ BarrierMap)	waiting slot

Figure 4.3: Concurrent state

owned by the thread so that `new` can transfer a cell from this pool into the local heap of a thread when required. When a thread is suspended, (what is left of) the allocation pool is transferred from the threads heap so that it can be passed to the next thread.

A thread contains a (sequential) state (store, heap, and barrier map) and a *concurrent control*, which is either Running(c), meaning the thread is available to run command c, or Waiting$(bn, tr, spec, c)$, meaning that the thread is currently waiting on barrier bn to take specification $spec$ while making transition tr; after the barrier call completes the thread will resume running with command c.

The barrier pool (Barpool) contains a list of *dynamic barrier statuses* (DBSes) as well as a state which is the join of all of the states inside the DBSes. Each DBS consists of a barrier number (which must be its index into the array of its containing Barpool), a barrier definition

(from §4.3), and a *waitpool* (WP). A waitpool consists of a transition option (`None` before the first barrier call in a given state; thereafter the unique transition to the next state), a limit (the number of threads synchronized by the barrier, and comes from the barrier definition in the enclosing DBS), a *slot* list, and a state (which is the join of all of the states in the slot list). A slot is a heap and barrier map (the store is unneeded since barrier pre/postconditions ignore it) as well as a thread id (the thread from which the heap and barrier map came as a precondition, and to which the postcondition will return).

The concurrent step relation has the form $(\Omega, ap, thds, bp) \rightsquigarrow (\Omega', ap', thds', bp')$, where Ω, ap, $thds$, and bp are the scheduler, allocation pool, thread list, and barrier pool respectively. The concurrent step relation has only four cases; the following case CStep-Seq is used to run all of the sequential commands:

$$\frac{thds[i] = (s, h, b, \mathsf{Running}(c)) \qquad h \oplus ap = h' \qquad \big((s, h', b), c\big) \mapsto \big((s', h'', b), c'\big) \qquad h''' \oplus ap' = h'' \qquad \mathsf{isAllocPool}(ap') \qquad thds' = [i \mapsto (s', h''', b, \mathsf{Running}(c'))]thds}{(i :: \Omega, thds, ap, bp) \rightsquigarrow (i :: \Omega, thds', ap', bp)} \text{ CStep-Seq}$$

That is, we look up the thread whose thread id is at the head of the scheduler, join in the allocation pool, and run the sequential step relation. If the command c is a barrier call then the sequential relation will not be able to run and so the CStep-Seq relation will not hold; otherwise the sequential step relation will be able to handle any command. After we have taken a sequential step, we subtract out the (possibly diminished) allocation pool, and reinsert the modified sequential state into the thread list. We observe that the resulting semantics does not allow interleavings on the non-synchronization related code sequences. Since we quantify over all schedulers and our language does not have input/output, it is sufficient to utilize a non-preemptive scheduler; for further justification on the use of such schedulers see [50].

The second case of the concurrent step relation handles the case when a thread has reached

the last instruction, which must be a `skip`:

$$\frac{thds[i] = (\_, \_, \_, \mathsf{Running}(\texttt{skip}))}{(i :: \Omega, thds, ap, bp) \rightsquigarrow (\Omega, thds, ap, bp)}\ \text{CStep-Exit}$$

When we reach the end of a thread we simply context switch to the next thread.

The interesting cases occur when the instruction for the running thread is a barrier call; here the CStep-Seq rule does not apply. The concurrent semantics handles the barrier call directly via the last two cases of the step relation; before presenting these cases we will first give a technical definition called fill_barrier_slot:

$$\frac{\begin{array}{c} thds[i] = (stk,\ hp,\ bs,\ (\mathsf{Running}\ (\texttt{barrier}\ bn;\ c))) \\ \mathsf{lookup\_spec}(bp.bp\_bars[bn],\ tr,\ spec) = (pre, post) \\ hp' \oplus hp'' = hp \qquad bs' \oplus bs'' = bs \qquad pre(stk, hp', bs') \\ \mathsf{bp\_inc\_waitpool}\ (bp,\ bn,\ tr,\ spec,\ (i,\ (hp',\ bs'))) = bp' \\ thds' = [i \rightarrow (\ stk,\ hp'',\ bs'',\ (\mathsf{Waiting}\ (bn,\ tr,\ spec,\ c)))]\ thds \end{array}}{\mathsf{fill\_barrier\_slot}\ (thds,\ bp,\ bn,\ i) = (thds',\ bp')}$$

The predicate fill_barrier_slot gives the details of removing the (sub)state satisfying the precondition of the barrier from the thread's state, inserting it into the barrier pool, and suspending the calling thread. The predicate bp_inc_waitpool does the insertion into the barrier pool; the details of manipulating the data structure are straightforward but lengthy to formalize.

We are now ready to give the first case for the barrier, used when a thread executes a barrier but is not the last thread to do so:

$$\frac{\begin{array}{c} \mathsf{fill\_barrier\_slot}\ (thds,\ bp,\ bn,\ i) = (thds',\ bp') \\ \neg\ \mathsf{bp\_ready}\ (bp',\ bn) \end{array}}{((i :: \Omega), ap, thds, bp) \rightsquigarrow (\Omega, ap, thds', bp')}\ \text{CStep-Suspend}$$

After using fill_barrier_slot, CStep-Suspend checks to see if the barrier is full by counting the number of slots that have been filled in the appropriate wait pool by using the bp_ready predicate, and then context switches.

If the barrier is ready then instead of using the CStep-Suspend case of the concurrent step relation, we must use the CStep-Release case:

$$\frac{\begin{array}{c}\mathsf{fill\_barrier\_slot}\ (thds,\ bp,\ bn,\ i) = (thds',\ bp') \\ \mathsf{bp\_ready}\ (bp',\ bn) \\ \mathsf{bp\_transition}\ (bp',\ bn,\ out) = bp'' \\ \mathsf{transition\_threads}\ (out,\ thds') = thds''\end{array}}{((i :: \Omega), ap, thds, bp) \rightsquigarrow (\Omega, ap, thds'', bp'')}\ \text{CStep-Release}$$

The first requirement of CStep-Release is exactly the same as CStep-Suspend: we suspend the thread and transfer the appropriate resources to the barrier pool. However, now all of the threads have arrived at the barrier and so it is ready. We use the bp_transition predicate to go through the barrier's slots in the waitpool, combine the associated heaps and barrier maps, redivide these resources according to the barrier postconditions, and remove the associated resources from the barrier pool into a list of slots called out. Finally, the states in out are combined with the suspended threads, which are simultaneously resumed by the transition_threads predicate. The formal definitions of the bp_transition and transition_threads predicates are extremely complex and very tedious and we refer interested readers to the mechanization.

4.5.2 Soundness Proof Outline

Our soundness argument falls into several parts. We define our Hoare tuple in terms of our oracle semantics using a definition by Appel and Blazy [4]; this definition was designed for a sequential language and we believe that other standard sequential definitions for Hoare tu-

ples would work as well[1]. We then prove (in Coq) all of the Hoare rules for the sequential instructions; since the os-seq case of the oracle semantics provides a straight lift into the purely sequential semantics this is straightforward.

Next, we prove (in Coq) the soundness for the barrier rule. This turns out to be much more complicated than a proof of the soundness of (non-first-class) locks and took the bulk of the effort. There are two points of particular difficulty: first, the excruciatingly painful accounting associated with tracking resources during the barrier call as they move from a source thread (as a precondition), into the barrier pool, and redistribution to the target thread(s) as postcondition(s). The second difficulty is proving that a thread that enters a barrier while holding more than one precondition will never wake up; the analogy is a door with n keys distributed among n owners; if an owner has a second key in his pocket when he enters then one of the remaining owners will not be able to get in.

After proving the Hoare rules for the sequential language statements sound with respect to the oracle semantics, the remaining task is to prove *oracle soundness*. That is, to connect the oracle semantics to the concurrent semantics. Oracle soundness says that if each of the threads on a machine is safe with respect to the oracle semantics, then the entire concurrent machine combining the threads together is safe. The (very rough) analogy to this result in Brookes' semantics is the parallel decomposition lemma. Here we use a progress/preservation style proof closely following that given in [50, *pp*.242–255]; the proof was straightforward and quite short to mechanize.

A direct consequence of oracle soundness is that if each thread is verified with the Hoare rules, and is loaded onto a single concurrent machine, then if the machine does not get stuck and if it halts then all of the postconditions hold.

Erasure One can justly observe that our concurrent semantics is not especially realistic; *e.g.*, we: explicitly track resource ownership permissions (*i.e.*, our semantics is *unerased*); have

[1] We change Appel and Blazy's definition so that our Hoare tuple guarantees that the allocation pool is available for verifying the Hoare rule for $x{:=}\,\texttt{new}\ e$.

an unrealistic memory allocator/deallocator and scheduler; ignore issues of byte-addressable memory; do not store code in the heap; and so forth. We believe that we could connect our semantics to a more realistic semantics that could handle each of these issues, but most of them are orthogonal to barriers. For brevity we will comment only on erasing the resource accounting since it forms the heart of our soundness result.

We have defined, in Coq, the *erased* sequential and concurrent semantics presented in §2.3.2. An erased memory is simply a pair of a break address and a function from addresses to values. The run-time state of an erased barrier is simply a pair of naturals: the first tracking the number of threads currently waiting on the barrier, and the second giving the final number of threads the barrier is waiting for. We define a series of erase functions that take an unerased type (memory/barrier status/thread/etc.) to an erased one by "forgetting" all permission information. The sequential erased semantics is quite similar to the unerased one, with the exception that we do not check if we have read/write permission before executing a load/store. The concurrent erased semantics is much simpler than the complicated accounting-enabled semantics explained above since all that is needed to handle the barrier is incrementing/resetting a counter, plus some modest management of the thread list to suspend/resume threads. Critically, our erased semantics is a computable function, enabling program evaluation. Finally, we have proved that our unerased semantics is a conservative approximation to our erased one: that is, if our unerased concurrent machine can take a step from some state Σ to Σ', then our erased machine takes a step from $\mathsf{erase}(\Sigma)$ to $\mathsf{erase}(\Sigma')$.

4.6 Tool Support for Barriers

We have integrated our program logic for barriers into HIP/SLEEK, an existing program verification toolset [76, 38]. SLEEK is an entailment checker for separation logic and HIP applies Hoare rules to programs and uses SLEEK to discharge the associated proof obligations.

Currently HIP and SLEEK support all the features in the specification language described

in §2.4 except for fractional shares and barrier definitions. We proceeded as follows:

1. We developed an equational solver over the sophisticated fractional share model of Dockins *et al.* [28]. Permissions can be existentially or universally quantified and arbitrarily related to permission constants.

2. We integrated our equational solver over shares into SLEEK to handle fractional permissions on separation logic assertions (*e.g.*, points-to, etc.). We believe that SLEEK is the first automatic entailment checker for separation logic that can handle a sophisticated share model (although some other tools can handle simpler share models).

3. We developed an encoding of barrier definitions (diagrams) in SLEEK, which now automatically verifies the side conditions from §4.3.

4. We modified HIP to recognize barrier definitions (whose side conditions are then verified in SLEEK) and barrier calls and apply the barrier rule presented in §4.4.

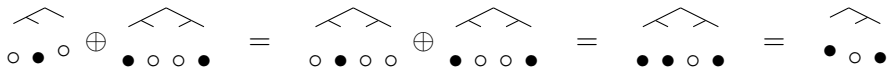
Next we describe our equational solver for the Dockins *et al.* share model before giving a more technical background to the HIP/SLEEK system and describing our modifications to it in detail.

4.6.1 A Solver for Shares

SLEEK discharges the heap-related proof obligations but relies on external decision procedures for the pure logical fragments it extracts from separation logic formulae. For example, SLEEK utilizes Omega for Presburger arithmetic, Redlog for arithmetic in $\mathbb{R}$, and MONA for monadic second-order logic. Adding fractional permissions required an appropriate equational solver for fractional shares.

Solvers for simple fraction share models such as rationals between 0 and 1 need only solve systems of linear equations. The more sophisticated fractional share model of Dockins *et al.* [28] requires a more sophisticated solver.

Dockins *et al.* represent shares as binary trees with boolean-valued leaves. The full share ■ is a tree with one true leaf • and the empty share □ is a tree with one false leaf ∘. The left-half share ◩ is a tree with two leaves, one true and one false: $\overset{\frown}{\bullet\ \circ}$; similarly, the right-half share ◪ is a tree with two leaves, one false and one true: $\overset{\frown}{\circ\ \bullet}$. The trees can continue to be split indefinitely: for example, the right half of ◩ is $\overset{\frown}{\circ\ \bullet\ \circ}$. Joining is defined by structural induction on the shape of the trees with base cases $\circ \oplus \circ = \circ$, $\bullet \oplus \circ = \bullet$, and $\circ \oplus \bullet = \bullet$ (emphasis: $\oplus$ is partial). When two trees do not have the same shape, they are unfolded according to the rules $\bullet \cong \overset{\frown}{\bullet\ \bullet}$ and $\circ \cong \overset{\frown}{\circ\ \circ}$; for example:



SLEEK takes a formula in separation logic with fractional shares and extracts a specialized formula over **strictly positive** shares whose syntax is as follows:

$$\phi \ ::= \ \exists v.\phi \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \mid v_1 \oplus v_2 = v_3 \mid v_1 = v_2 \mid v = \chi$$

Our share formulae ϕ contain share variables v, existentials $\exists$, conjunctions $\wedge$, disjunctions $\vee$, join facts $\oplus$, equalities between variables, and assignments of variables to constants χ. The tool also recognizes $v{\in}[\chi_1, \chi_2]$, pronounced "v is bounded by χ_1 and χ_2", which is semantically equal to:

$$((v = \chi_1) \vee (\exists v'.\ \chi_1 \oplus v' = v)) \ \wedge \ ((v = \chi_2) \vee (\exists v''.\ v \oplus v'' = \chi_2))$$

Disjunctions are needed because share variables can only be instantiated with positive shares: $\forall v.\ \nexists v'.v \oplus v' = v$. Handling bounds checks "natively" rather than compiling them into semantic definitions increases efficiency by reducing the number of existentials and disjunctions.

SLEEK asks the solver questions of the following forms:

1. (UNSAT) Is a given formula ϕ unsatisfiable?

2. ($\exists$-ELIM) Given a formula of the form $\exists v.\ \phi(v)$, is there a unique constant χ such that $\exists v.\ \phi(v)$ is equivalent to $\phi(\chi)$?

3. (IMPL) Given two formulae ϕ_1 and ϕ_2, does ϕ_1 entail ϕ_2?

All of these questions can be reduced to solving a series of constraint systems whose equations are of the form $v_1 \oplus v_2 = v_3$, $v \in [\chi_1, \chi_2]$, and $v = \chi$. Solving constraint systems in separation algebras (*i.e.*, cancellative partial commutative monoids) is not as straightforward as it might seem because many of the traditional algebraic techniques do not apply. Our lightweight constraint solver finds an overapproximation to the solution, returning either (a) the constant `UNSAT` or (b) for each variable v_i either an **assignment** $v_i = \chi$ or a **bound** $v_i \in [\chi_1, \chi_2]$ such that:

- (FALSE) If the algorithm returns `UNSAT`, then the formula is unsatisfiable. The algorithm will return `UNSAT` if it discovers a bound whose "lower value" is higher than its "upper value", or if it discovers a falsehood (*e.g.*, after constant propagation one of the equations becomes $\blacksquare \oplus \blacksquare = \blacksquare$).

- (COMPLETE) All solutions to the system (if any) lie within the bounds.

- (SAT-PRECISE) A solution is *precise* when all variables are given assignments. If a solution is precise, then the formula is satisfiable.

Note that, for this presentation we have chosen an incomplete but fast method for solving these constraints. However in a separate investigation, we have proven the problem to be decidable and provided the first sound and complete solver, albeit considerable slower, for constraints over this share domain [66].

SLEEK queries are given in share formulae that must be transformed into the equational systems understood by our constraint solver. To do this transformation, first we put the relevant

formulae into disjunctive normal form (DNF). Each disjunct becomes an independent system of equations. Given one disjunct we form this system by simply treating each basic constraint (*i.e.*, $v = v'$, $v = \chi$, $v \in [\chi_1, \chi_2]$, and $v_1 \oplus v_2 = v_3$) as an equation. Our solver approximates each system independently and can then answer SLEEK's questions as follows:

- (UNSAT): Return `False` when the algorithm returns `UNSAT` for each constraint system obtained from the formula; otherwise return `True`.
- ($\exists$-ELIM): If the variable v has the same assignment in all constraint systems derived from the DNF, then return that value. It is sound to substitute that value for v and eliminate the existential. (If the formula is satisfiable, then that is the unique assignment that makes it so; if the formula is false then after the assignment it will still be false.)
- (IMPL): Return `True` only when either:
  - the solver returns `UNSAT` for all systems derived from the antecedent
  - the solver returns a precise solution for each system of equations derived from the antecedent, and the solver also returns **the same** precise solution for at least one of the consequent systems.

The constraint solver works by eliminating one class of constraints at a time:

1. First we substitute $v = \chi$ constraints into the remaining equations.
2. We handle $\oplus$ constraints with exactly one variable as follows:
   - $\chi_1 \oplus \chi_2 = v$: we check if the join is defined, and if so substitute the sum for v in the remaining equations; otherwise, we return `UNSAT`.
   - $\chi_l \oplus v = \chi_r$ or $v \oplus \chi_l = \chi_r$: we check if χ_r contains χ_l, and if so substitute the difference $\chi_r - \chi_l$ for v in the remaining equations; otherwise return `UNSAT`. ("$-$" has the property that if $\chi_1 - \chi_2 = \chi_3$ then $\chi_3 \oplus \chi_2 = \chi_1$).

3. Constraints involving constants ($\chi_1 \oplus \chi_2 = \chi_3$ and $\chi \in [\chi_1, \chi_2]$) are dismissed if the equality/inequalities hold; otherwise return `UNSAT`.

4. We attempt to dismiss certain kinds of unsatisfiable systems via a consistency check as follows. We first compute the transitive closure of variable substitutions, resulting in facts of the form $v_1 \oplus \ldots \oplus v_n \oplus \chi_1 \oplus \ldots \oplus \chi_m = \chi$. Nonempty shares **cannot** join with themselves. Therefore, if the v_i contain duplicates we return `UNSAT`. We also return `UNSAT` if the constants χ_i do not join or if χ does not contain $\chi_1 \oplus \ldots \oplus \chi_m$.

5. Variables in the remaining constraints are given initial domains of ($\square$, $\blacksquare$).

6. Each $\in$ constraint is used to restrict the domain of its corresponding variable.

7. At this point only $a_1 \oplus a_2 = a_3$ constraints involving at least two variables remain. The algorithm then proceeds by iteratively selecting an equation, checking it for consistency, and then refining the associated domains via a forward and backward propagation. The algorithm iterates until either a fixpoint is reached or a consistency check fails. To check an equation for consistency, the algorithm verifies that:

   - for each variable, the lower bound is less than the upper bound
   - the current lower bounds of the LHS variables join together
   - the join of the LHS lower bounds is below the RHS upper bound
   - the join of the LHS upper bounds is above the RHS lower bound

   Forward propagation consists of (Fa) lowering the upper bound of the RHS by intersecting away any subtree that does not appear in the upper bounds of the LHS, and (Fb) increasing the lower bound of the RHS by unioning all subtrees present in the lower bounds in the LHS. Backwards propagation consists of (Ba) lowering the upper bounds of the LHS by intersecting away any subtree that does not appear in the upper bound on the RHS. Increasing the lower bounds of the LHS (Bb) is trickier since we do not know

which operand should be increased. There are several possibilities we could have taken, but we selected the simplest: we simply leave the bounds as they were unless one of the operands has been determined to be a constant, in which case we can calculate exactly what the lower bound for the other variable should be. After each forward/backwards propagation, if we have refined a domain to a single point, the variable is substituted for a constant value of that point in the remaining equations.

Once we reach a fixpoint, the resulting variable bounds represent an over approximation of the solution.

4.6.2 An Introduction to SLEEK

SLEEK checks entailments in separation logic with user-defined predicates [78]. The antecedent may cover more of the heap than the consequent, in which case SLEEK returns this residual heap together with the pure portion of the antecedent. SLEEK can also discover instantiations for certain existentials in the consequent, a feature that we elide here; details may be found in [20, 38].

The core of the SLEEK entailment works by algorithmically discharging the heap obligations and then referring any remaining pure constraints to other provers. Entailments in SLEEK are written as follows: $\Delta_A \vdash^{\kappa}_{V} \mathsf{Q}_C * \Delta_R$, which is shorthand for $\kappa * \Delta_A \vdash \exists V \cdot (\kappa * \mathsf{Q}_C) * \Delta_R$ The entailment checks whether the consequent heap nodes Q_C are covered by heap nodes in antecedent Δ_A, and if so, SLEEK returns the residual heap Δ_R, which consists of the antecedent nodes that were not used to cover Q_C. The implementation performs a proof search and thus returns a set of residues. For simplicity, assume that only one residue is computed. In the entailment, κ is the history of nodes from the antecedent that have been used to match nodes from the consequent, V is the list of existentially quantified variables from the consequent. Note that κ and V are discovered iteratively: entailment checking begins with $\kappa = emp$ and $V = \emptyset$.

The initial system behavior was described in detail in [78, 20, 38]. The main rules for matching, folding, unfolding, and discharging of pure constraints are given here. The initial

$$\mathsf{XPure}(emp) \equiv (\texttt{true}, \emptyset)$$

$$\frac{\mathit{IsData}(c)}{\mathsf{XPure}(p{::}c\langle\overrightarrow{v}\rangle) \equiv (p{\neq}0; \{p\})}$$

$$\frac{\mathit{IsPred}(c) \qquad (\texttt{pred c}\langle\overrightarrow{v}\rangle \equiv \mathsf{Q}\ \texttt{inv}\ (\pi_1, \pi_2)) \in V}{\mathsf{XPure}(p :: c\langle\overrightarrow{v_p}\rangle) \equiv ([p/\mathsf{self}, \overrightarrow{v_p}/\overrightarrow{v}]\pi_1, [p/\mathsf{self}, \overrightarrow{v_p}/\overrightarrow{v}]\pi_2)}$$

$$\frac{\mathsf{XPure}(\kappa_1) \equiv (f_1, s_1) \qquad \mathsf{XPure}(\kappa_2) \equiv (f_2, s_2)}{\mathsf{XPure}(\kappa_1{*}\kappa_2) \equiv (f_1 \wedge f_2, s_1 \cup s_2)}$$

Figure 4.4: *XPure* : Translating to Pure Form

main entailment checking rules are given in Figure 4.5. Later we show how we modified these rules to accommodate fractional shares.

Entailment between separation formulae is reduced to entailment between pure formulae by matching heap nodes in the RHS to heap nodes in the LHS (possibly after a fold/unfold). Once the RHS is pure, the remaining LHS heap formula is soundly approximated to a pair of pure formula and set of disjoint pointers by function XPure as defined in Figure 4.4. The functions *IsData*(c) and *IsPred*(c) decide respectively if c is a data structure or a predicate. The procedure successively pairs up heap nodes that it proves are aliased. SLEEK keeps the successfully matched nodes from the antecedent in κ for better precision in the next iteration.

SLEEK discharges heap obligations in three ways: heap node matching, predicate folding, and predicate unfolding. All three heap reducing steps start by establishing that there is a heap node on the LHS of the entailment that is aliased with the RHS heap node that is to be reduced ($p_1 = p_2$). In order to prove the aliasing, the LHS heap together with the previously consumed nodes are approximated to a pure formula, and together with the LHS pure formula the $p_1 = p_2$ implication is checked. Similarly, when a match occurs (rule MATCH), equality between node arguments needs to be proven.

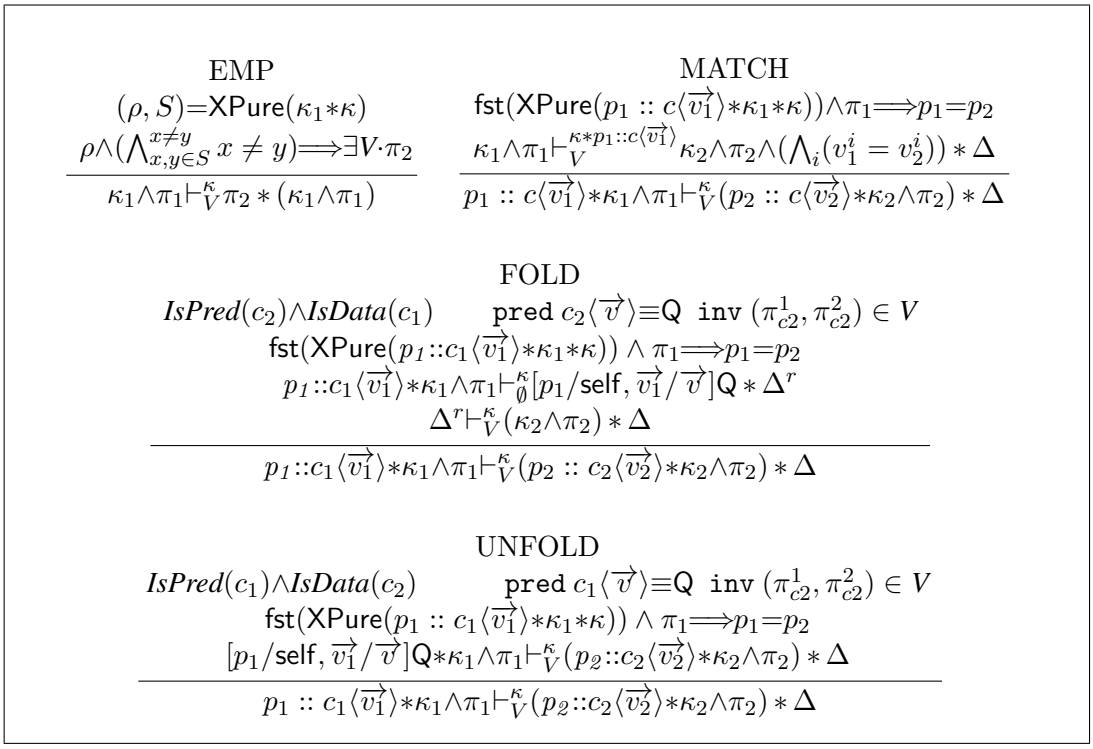


Figure 4.5: Separation Constraint Entailment

Fold/unfold operations handle inductive predicates in a deductive manner. SLEEK can unfold a predicate instance that appears in the LHS if the unfolding exposes a heap node that can match with a node in the RHS. Similarly, several LHS nodes can be folded into a predicate instance if the resulting predicate instance can be matched with a RHS node. To guarantee termination, SLEEK needs to ensure that (i) each predicate fold or unfold must be immediately followed by a match, and (ii) no two fold operations for the same predicate are performed in order to match one node. These restrictions ensure that each successful fold, unfold, and match operation decreases the number of RHS nodes. Well-formedness conditions imposed on the predicate definitions ensure that after a fold or unfold a matching always takes place; these conditions have been elided for this presentation. The unfold rule presents the replacement of a predicate instance in which the predicate definition is reduced to a disjunctive form and

$$\mathsf{FXPure}(emp, \tau) \equiv (\mathtt{true}, \emptyset)$$

$$\frac{\mathit{IsData}(c) \qquad \tau \Rightarrow v_f = cs}{\mathsf{FXPure}(p{::}c^{v_f}\langle\overrightarrow{v}\rangle, \tau) \equiv (p{\neq}0; \{(p, cs)\})}$$

$$\frac{\mathsf{FXPure}(\kappa_1, \tau) \equiv (f_1, s_1) \qquad \mathsf{FXPure}(\kappa_2, \tau) \equiv (f_2, s_2)}{\mathsf{FXPure}(\kappa_1{*}\kappa_2, \tau) \equiv (f_1 \wedge f_2, s_1 \cup s_2)}$$

$$\frac{\begin{array}{c}\mathit{IsPred}(c) \qquad (\mathsf{c}\langle\overrightarrow{v}\rangle \equiv \mathsf{Q}\ \mathtt{inv}\ (\pi_1, \pi_2)) \in P \\ \tau \Rightarrow v_f = cs \qquad \pi_1' = [p/\mathsf{self}]\pi_1 \qquad \pi_2' = \{\forall v \in \pi_2, ([p/\mathsf{self}]v, cs)\}\end{array}}{\mathsf{XPure}(p :: c^{v_f}\langle\overrightarrow{v}\rangle, \tau) \equiv (\pi_1', \pi_2')}$$

Figure 4.6: FXPure*:* XPure *with shares*

in which the arguments have been substituted. The fold step requires the LHS to entail the predicate definition. The residue of this entailment is then used as the new LHS for the rest of the original entailment. Chin *et al.* give a more detailed explanation of the SLEEK entailment process in [20].

4.6.3 Entailment Procedure for Separation Logic with Shares

Adding fractional permissions required several modifications to the entailment process.

- **Empty heap**. In a separation logic without shares, whenever $(\exists a, b.x{\mapsto}a{*}y{\mapsto}b)$ then $x \neq y$. In SLEEK, this fact is captured in the EMP rule, which tries to prove the pure part of the consequent after enriching the antecedent pure formula with pure information collected from the previously consumed heap and the remaining LHS heap. It extracts both the invariants of the heap nodes and constructs a formula that ensures that all pointers in the heap are distinct.

  Introducing fractional permissions requires the relaxation of this constraint because $\exists a, b.x\overset{x_f}{\longmapsto}a{*}y\overset{y_f}{\longmapsto}b$ implies $x \neq y$ only if the x_f and y_f shares overlap. We extended the XPure function to (a) take into account fractional shares constraints, given through an extra argument τ, and (b) return a pair of a pure formula, and a set of pairings be-

$$\text{FOLD}$$
$$\frac{\begin{array}{c}\mathit{IsPred}(c_2)\wedge\mathit{IsData}(c_1)\quad c_2\langle\overrightarrow{v}\rangle{\equiv}\mathsf{Q}\ \ \mathtt{inv}(\pi^1_{c2},\pi^2_{c2})\in V\\ \mathsf{fst}(\mathsf{FXPure}(p_1{::}c_1^{f_1}\langle\overrightarrow{v_1}\rangle{*}\kappa_1{*}\kappa,\tau_1))\wedge\pi_1{\Longrightarrow}p_1{=}p_2\\ \mathsf{Q}'=\mathsf{set\_shares}([p_1/\mathsf{self},\overrightarrow{v_1}/v]\mathsf{Q},f_2)\\ p_1{::}c_1^{f_1}\langle\overrightarrow{v_1}\rangle{*}\kappa_1{\wedge}\pi_1{\wedge}\tau_1{\vdash}^{\kappa}_{\emptyset}\mathsf{Q}'*\Delta^r\\ \Delta^r{\vdash}^{\kappa^r}_{V}(\kappa_2{\wedge}\pi_2{\wedge}\tau_2)*\Delta\end{array}}{p_1{::}c_1^{f_1}\langle\overrightarrow{v_1}\rangle{*}\kappa_1{\wedge}\pi_1\wedge\tau_1{\vdash}^{\kappa}_{V}(p_2::c_2^{f_2}\langle\overrightarrow{v_2}\rangle{*}\kappa_2{\wedge}\pi_2\wedge\tau_2)*\Delta}$$

$$\text{UNFOLD}$$
$$\frac{\begin{array}{c}c_1\langle\overrightarrow{v}\rangle{\equiv}\mathsf{Q}\ \ \mathtt{inv}(\pi^1_{c2},\pi^2_{c2})\in V\quad \mathit{IsPred}(c_1)\wedge\mathit{IsData}(c_2)\\ \mathsf{fst}(\mathsf{FXPure}(p_1::c_1^{f_1}\langle\overrightarrow{v_1}\rangle{*}\kappa_1{*}\kappa,\tau_1))\wedge\pi_1{\Longrightarrow}p_1{=}p_2\\ \mathsf{Q}'=\mathsf{set\_shares}([p_1/\mathsf{self},\overrightarrow{v_1}/\overrightarrow{v}]\mathsf{Q},f_1)\\ \mathsf{Q}'{*}\kappa_1{\wedge}\pi_1{\wedge}\tau_1{\vdash}^{\kappa}_{V}(p_2{::}c_2^{f_2}\langle\overrightarrow{v_2}\rangle{*}\kappa_2{\wedge}\pi_2{\wedge}\tau_2)*\Delta\end{array}}{p_1::c_1^{f_1}\langle\overrightarrow{v_1}\rangle{*}\kappa_1{\wedge}\pi_1{\wedge}\tau_1{\vdash}^{\kappa}_{V}(p_2{::}c_2^{f_2}\langle\overrightarrow{v_2}\rangle{*}\kappa_2{\wedge}\pi_2{\wedge}\tau_2)*\Delta}$$

Figure 4.7: Folding/Unfolding in the presence of shares

tween pointers and the concrete fractional shares associated with the memory location the pointers point to. The new formulation for FXPure is given in Figure 4.6. This extension allowes the EMP rule to be rewritten to enforce inequality only between pointers that have conflicting shares:

$$\frac{\begin{array}{c}(\rho,S){=}\mathsf{FXPure}(\kappa_1{*}\kappa,\tau)\\ \rho{\wedge}(\bigwedge^{x\neq y\wedge(\neg\exists z\cdot x_f\oplus y_f=z)}_{(x,x_f),(y,y_f)\in S}\cdot x\neq y){\Longrightarrow}\exists V{\cdot}\pi_2\end{array}}{\kappa_1\wedge\pi_1\wedge\tau_1{\vdash}^{\kappa}_{V}\pi_2*(\kappa_1\wedge\pi_1\wedge\tau_1)}\ \mathsf{EMP}$$

- **Folding/unfolding**. By convention, all the heap nodes abstracted by a predicate instance are owned with the same fractional permission as the predicate instance. Therefore, unfolding a node first replaces the permissions of the nodes in the predicate definition with the permission of that LHS node. Then the updated predicate definition replaces the predicate instance. Similarly, folding a node replaces the permissions of all nodes in

the definition with the permission of that RHS node before trying to entail the predicate definition. The $\mathsf{set\_shares}(\mathsf{Q}, v)$ function sets the permissions of all heap nodes in Q to v. The new set of rules is shown in Figure 4.7.

- **Matching**. In order to properly handle a match in the presence of fractional shares, the entailment process needs to (a) reduce both LHS and RHS nodes entirely, or (b) split the LHS node and reduce one side, or (c) split the RHS and reduce one side.

$$\frac{\begin{array}{c}\big(\mathsf{fst}(\mathsf{FXPure}(p_1 :: c^{f_1}\langle\overrightarrow{v_1}\rangle{*}\kappa_1, \tau_1)) \wedge \pi_1\big) \implies p_1 = p_2 \\ \kappa' = \kappa{*}p_1 :: c^{f_1}\langle\overrightarrow{v_1}\rangle \\ \rho = f_1{=}f_2 \wedge (\bigwedge_i (v_1^i = v_2^i)) \\ \kappa_1 \wedge \pi_1 \wedge \tau_1 \vdash^{\kappa'}_V \kappa_2 \wedge \pi_2 \wedge \tau_2 \wedge \rho * \Delta\end{array}}{p_1 :: c^{f_1}\langle\overrightarrow{v_1}\rangle{*}\kappa_1 \wedge \pi_1 \wedge \tau_1 \vdash^{\kappa}_V (p_2 :: c^{f_2}\langle\overrightarrow{v_2}\rangle{*}\kappa_2 \wedge \pi_2 \wedge \tau_2) * \Delta}\ \mathsf{FULL\text{-}MATCH}\ (a)$$

$$\frac{\begin{array}{c}\mathsf{fst}(\mathsf{FXPure}(p_1 :: c^{f_1}\langle\overrightarrow{v_1}\rangle{*}\kappa_1, \tau_1)){\wedge}\pi_1 \implies p_1{=}p_2 \\ \tau_1' = \tau_1 \wedge f_{c1} \oplus f_{r1} = f_1 \\ \kappa' = \kappa{*}p_1 :: c^{f_{c1}}\langle\overrightarrow{v_1}\rangle \\ \rho = f_{c1}{=}f_2 {\wedge} (\bigwedge_i (v_1^i = v_2^i)) \\ p_1 :: c^{f_{r1}}\langle\overrightarrow{v_1}\rangle{*}\kappa_1{\wedge}\pi_1{\wedge}\tau_1' \quad \vdash^{\kappa'}_V \kappa_2{\wedge}\pi_2{\wedge}\tau_2{\wedge}\rho * \Delta\end{array}}{p_1 :: c^{f_1}\langle\overrightarrow{v_1}\rangle{*}\kappa_1{\wedge}\pi_1{\wedge}\tau_1 \vdash^{\kappa}_V (p_2 :: c^{f_2}\langle\overrightarrow{v_2}\rangle{*}\kappa_2{\wedge}\pi_2{\wedge}\tau_2) * \Delta}\ \mathsf{LEFT\text{-}SPLIT\text{-}MATCH}\ (b)$$

$$\frac{\begin{array}{c}\mathsf{fst}(\mathsf{FXPure}(p_1 :: c^{f_1}\langle\overrightarrow{v_1}\rangle{*}\kappa_1, \tau_1)){\wedge}\pi_1 \implies p_1{=}p_2 \\ V' = \texttt{if } f_2 \in V \texttt{ then } V \cup \{f_{c2}, f_{r2}\} \texttt{ else } V \\ \tau_1' = \texttt{if } f_2 \in V \texttt{ then } \tau_1 \texttt{ else } (\tau_1{\wedge}f_{c2} \oplus f_{r2} = f_2) \\ \tau_2' = \texttt{if } f_2 \in V \texttt{ then } (\tau_2{\wedge}f_{c2} \oplus f_{r2} = f_2) \texttt{ else } \tau_2 \\ \kappa' = \kappa{*}p_1 :: c^{f_1}\langle\overrightarrow{v_1}\rangle \\ \rho = f_1{=}f_{c2}{\wedge}(\bigwedge_i (v_1^i = v_2^i)) \\ \kappa_1{\wedge}\pi_1{\wedge}\tau_1' \vdash^{\kappa'}_{V'} \; p_2 :: c^{f_{r2}}\langle\overrightarrow{v_2}\rangle{*}\kappa_2{\wedge}\pi_2{\wedge}\tau_2' \wedge \rho{*}\Delta\end{array}}{p_1 :: c^{f_1}\langle\overrightarrow{v_1}\rangle{*}\kappa_1{\wedge}\pi_1{\wedge}\tau_1 \vdash^{\kappa}_{V} (p_2 :: c^{f_2}\langle\overrightarrow{v_2}\rangle{*}\kappa_2{\wedge}\pi_2{\wedge}\tau_2) * \Delta}$$

RIGHT-SPLIT-MATCH (c)

Because the search can be computationally expensive, we have devised an aggressive pruning technique. We try to determine to what extent the fractional constraints restrict the fractional variables. It may be that (a) $f_1 = f_2$, in which case only FULL-MATCH is feasible, or (b) f_1 is included in f_2, in which case RIGHT-SPLIT-MATCH is feasible, or (c) f_1 includes f_2, in which case only
LEFT-SPLIT-MATCH is feasible.

4.6.4 Proving Barrier Soundness

The fractional share solver and enhancements to SLEEK's entailment procedures discussed above help with any program logic that needs fractional shares (*e.g.*, concurrent separation logic with locks, sequential separation logic with read-only data). In contrast, our other enhancements are specific to the logic for Pthreads-style barriers. Our initial goal is to automatically check the consistency of barrier definitions—that is, whether a barrier definition meets the side conditions presented in §4.3. The first step is to describe a barrier diagram to SLEEK.

Although the barrier diagrams presented in §4.3 are intuitive and concise, programs need a more textual representation. Barrier diagrams describe the possible transitions a barrier state can make and the specifications associated with those transitions. In a sense, a barrier definition can be viewed as a disjunctive predicate definition where the body is a disjunction of possible

transitions.

SLEEK already contains user-defined predicates so it is easy to introduce the "is a barrier" assertion $bn{::}bname^{v_f}\langle\overrightarrow{v}\rangle$. We extended SLEEK's input language to accept barrier diagrams in the form:

$$\begin{array}{lll} \mathsf{bdef} & ::= & \texttt{barrier}\ bname[n]\ <\overrightarrow{v}>==\overrightarrow{\mathsf{transition}} \\ \mathsf{transition} & ::= & (from\_state\ ,\ to\_state\ ,\ \overrightarrow{\mathsf{pre\text{-}post\text{-}spec}}) \\ \mathsf{pre\text{-}post\text{-}spec} & ::= & (\Phi_{pre}\ ,\ \Phi_{post}) \end{array}$$

Where n denotes the number of threads synchronized by any instance of this barrier definition. We have chosen this rather verbose form in order to allow the input to resemble more the graphical diagrams presented in the previous sections rather than the more compact form from Figure 2.5 in §2.4:

$$[\mathsf{self}::]bname\langle\overrightarrow{v}\rangle \equiv \bigvee (\texttt{requires}\ \Phi_{pre}\ \texttt{ensures}\ \Phi_{post})$$

However the latter form is much more suitable for the verification process. It is straight forward to translate from the former to the latter formulation and in fact SLEEK does that as a preprocessing step.

SLEEK can now automatically check the well-formedness conditions on the barrier definitions as follows:

- All transitions must have exactly n specifications, one for each thread
- For each transition, let $from$ and to be the state labels, then:
  - for each specification $(\Phi_{pre}\ ,\ \Phi_{post})$
    1. Φ_{pre} contains a fraction of the barrier in state $from$:
       $\Phi_{pre} \vdash \mathsf{self} :: bn^{v_f}\langle from\rangle * \Delta$
    2. Φ_{post} contains a fraction of the barrier in state to:

$\Phi_{pre} \vdash \mathsf{self} :: bn^{v_f}\langle to\rangle * \Delta$

3. $\Phi_{pre} * \Phi_{pre} \vdash \mathsf{False}$

   The soundness proof assumes that each precondition P is precise. Unfortunately, precision is not very easy to verify automatically. We believe that the logic will be sound if we can assume the (strictly) weaker property "token": $P \star P \vdash \mathsf{False}$ instead of precision. At this stage, our prototype extension to SLEEK verifies that preconditions are tokens rather than that they are precise. We are in the process of attempting to update our soundness proof to require that preconditions be tokens rather than precise; if we are unable to do so then one solution would be for SLEEK to output a Coq file stating lemmas regarding the precision of each precondition. Users would then be required to prove these lemmas manually to be sure that their barrier definitions were sound. In our experience, the Coq proofs of precision were very easily discharged as compared with the soundness proof of the entire barrier definition. So the savings from using SLEEK to verify the soundness of a barrier definition should still be quite substantial. Another choice would be to devise a heuristic algorithm for determining precision; we suspect that such an algorithm could handle the examples from this chapter.

– the star of all the preconditions contains the full barrier (recall that the entailment $\vdash$ check in SLEEK can produce a residue)
  $*_{i=1}^{n}\Phi_{pre}^{i} \vdash \mathsf{self} :: bn^{\blacksquare}\langle from\rangle * \Delta$
– the star of all the postconditions contains the full barrier
  $*_{i=1}^{n}\Phi_{post}^{i} \vdash \mathsf{self} :: bn^{\blacksquare}\langle to\rangle * \Delta$
– the star of all the preconditions equals the star of all postconditions modulo the barrier state change for a transition. We check this constraint by carving the full barrier out of the total heap using the residues Δ_{pre} and Δ_{post} of the entailments given in the previous constraints. Δ_{pre} and Δ_{post} are then tested for equality by

requiring bi-entailment with empty residue. That is, given

$*_{i=1}^{n}\Phi_{pre}^{i} \vdash \mathsf{self} :: bn^{\blacksquare}\langle from\rangle * \Delta_{pre}$

and

$*_{i=1}^{n}\Phi_{post}^{i} \vdash \mathsf{self} :: bn^{\blacksquare}\langle to\rangle * \Delta_{post}$,

we check

$\Delta_{pre} \vdash \Delta_{post}$ and $\Delta_{post} \vdash \Delta_{pre}$ **with empty residues**.

– For states with more than one successor, we check mutual exclusion for the pre-conditions as required by §4.3 by verifying that for any two preconditions of two distinct transitions must entail False. This check was extremely tedious but SLEEK can do it easily.

Once SLEEK has verified each of the above conditions, the barrier definition is well-formed according to the well-formed Definitions 4.3.1 and 4.3.2 given §4.3 (modulo precision).

4.6.5 Extension to Program Verification

Integrating our Hoare rule for barriers into HIP was the easiest part of adding our program logic to HIP/SLEEK. As mentioned, following the concept of structured specifications [38], we transform our barrier diagrams into disjunctions of the form

$$\bigvee(\texttt{requires}\ \Phi_{pre}\quad \texttt{ensures}\ \Phi_{post}),$$

where the disjunction contains all specifications in all transitions in the barrier definition. The benefit of such an encoding lies in the insight that SLEEK's entailment procedure is flexible enough to discharge entailment obligations even if the consequent is presented in this complex form [38].

In short and informal, given the entailment $\Delta \vdash \bigvee(\texttt{requires}\ \Phi_{pre}\ \texttt{ensures}\ \Phi_{post})$, SLEEK will first check which preconditions Φ_{pre} are entailed by Δ , compute the residue for each entailment, add the corresponding postcondition Φ_{post} to the residue and return the

possible outcomes as a disjunctive formula.

With this formulation the barrier rule presented in §4.4 becomes even simpler. In order to verify a barrier call `barrier` b, it prescribed two operations: locating the proper barrier definition in the barrier repository Γ and executing the $lookup\_spec$. In HIP these operations correspond to: first locating in the current state, a node of the form $b{::}bname^{\pi}\langle\overrightarrow{v}\rangle$ which indicate that b points to a barrier of type $bname$; secondly retrieving the $bname$ barrier definition, in the disjunctive form presented earlier and substituting the formal arguments substituted with the $\overrightarrow{v}$ arguments; thirdly, the $lookup\_spec$ is performed through a SLEEK entailment check between a formula abstracting the current states and the barrier definition.

Thus ensuring that in one entailment check, the proper pre/post condition pair is selected, the precondition is consumed, and the postcondition is added, in effect applying the barrier Hoare rule in one simple go.

Note that the bulk of the work behind the verification of barrier calls elegantly reduces to an entailment. However, based on the example application of the barrier rule in §4.4, one might suggest that a simpler method be used, one with a more syntactic flavour. Unfortunately, despite the fact that the entailment process is expensive, for the general case a syntactic filtering is not sufficient, as the values critical for selecting the proper specification might not be given as plain text constants in Δ. Thus a very costly, full entailment check is required.

As a final observation, the performance cost lies in the fact that entailment checks need to take place for each precondition in the barrier definition. For our simple running example this translates to eight entailments for each barrier call. In the next section we will show what are the performance penalties induced when verifying programs with barriers. Also in Chapter 5 we will introduce a general technique which can greatly improve the performance by reducing this disjunctive explosion.

4.6.6 Tool Performance Outline

We have developed a small set of benchmarks for our HIP/SLEEK with barriers prototype. Our SLEEK tests divide into two categories: entailment checks for separation logic formulae containing fractional permissions and checking barrier consistency constraints as described in §4.6.4. Individual entailment checks are quite speedy and our benchmark covers a number of interesting cases (*e.g.*, inductively defined predicates). Barrier wellformedness checks take more time but the performance is more than adequate: for a set of 54 human generated entailments focusing on corner cases of fractional share reasoning SLEEK took 2.1 seconds, while for a set of 279 machine generated entailments obtained during the well-formedness checks of various barrier definitions SLEEK took 3.69 seconds.

One of the barrier definitions used is the example barrier given in Figure 4.1. For a similar example, it took **2,700 (highly tedious) lines of code** and 48 seconds of verification time to convince Coq that the example barrier definition met the soundness requirements[1]. SLEEK verifies this example barrier definition and analyzes five others (some sound, others not) in 3.69 seconds **without any interaction from the user**[2].

We have also developed a HIP benchmark suite. Through this suite we outline the impact several parameters have on the verification time for programs with barriers. The parameters we are interested in, are the number of threads synchronized, the number of memory resources shared, the pattern of communications (both the barrier definition shape and the complexity of the resource reshuffling) and finally the complexity of the arithmetic constraints imposed on the content of the shared memory.

In Figure 4.8 we list the HIP verification times for several tests. For each test we also note the number of threads and the number of shared memory cells. We note that the most important variation between the examples lie in the structure of the barrier definitions, the

[1] Techniques such as those developed by Braibant *et al.* [15], Nanevski *et al.* [73], and Gonthier *et al.* [40] can probably eliminate some (but not all) of the tedium of reasoning about the associativity and commutativity of $*$. Unfortunately, proofs of mutual exclusion for barrier transitions seem less tractable.

[2] As explained in §4.6.4, SLEEK verifies properties that are slightly different from those listed in §4.3.

Test	Threads	Mem cells	CODE LOC	SPEC LOC	Time (s)
video	2	5	43	36	59.17
	3	7	68	60	215
video-weak	2	5	43	36	60.35
video-strong	2	5	43	36	61.12
MISD (without feedback)	2	1	36	26	0.81
	3	1	36	31	1.33
	4	1	36	37	1.97
	5	1	36	43	2.81
	6	1	36	51	3.88
MISD (with feedback)	2	2	47	27	3.22
	3	3	52	28	4.04
	4	4	87	42	9.28
	5	5	105	51	69.51
	6	6	127	60	177
multi-loops	3	4	102	58	12.82
	4	5	135	73	132
	5	6	165	90	364
pipeline	3	3	35	19	1.43
	4	4	44	26	2.52
	5	5	56	33	4.25
	6	6	66	40	7.08
Horner(v1)	3	7	73	43	181
Horner(v2)	3	4	74	37	8.04
	4	5	95	48	97
	5	6	118	58	279

Figure 4.8: Verification times for HIP with barriers

pattern of permission reshuffling, and the number of guarded memory locations and threads. The examples are grouped in five classes of tests included in our HIP benchmark:

- various instantiations of a simplified video encoding algorithm that encodes a video frame by frame. Encoding a frame consists of splitting the frame among several worker threads and piecing together the result once the threads finish. The main challenge in

verifying this type of algorithms lies in the fact that the encoding of each frame piece depends both on the piece itself and on the entire previous frame, thus inducing a more complex resource handling protocol. The outline of the synchronization pattern is:

```
while (cond)                          ||  ...  ||  while (cond)
{  compute & write to memory;         ||       ||  {  compute & write to memory;
   barrier;                           ||       ||     barrier;
   read results of all threads;       ||       ||     read results of all threads;
   barrier;                           ||       ||     barrier;
}                                     ||       ||  }
```

- a general communication architecture with one thread generating data that is fed into several workers each executing a set of operations functionally equivalent yet structurally distinct. This architecture resembles the multiple instructions/single data(MISD) structure defined the Flynn taxonomy[36]. Such an architecture is often used in safety critical systems in which redundancy is required. We construct two types of examples based on this architecture: i)one in which the output of the working threads is fed into subsequent stages without interfering with the generator, ii) the worker output is also fed as feedback into the generator. The synchronization pattern can be outlined as follows:

```
while (true)          ||  while (true)          ||     ||  while (true)
{  generate;          ||  {                     ||     ||  {
   write to memory;   ||                        ||     ||
   barrier;           ||     barrier;           ||  ...||     barrier;
                      ||     read and compute;  ||     ||     read and compute;
   barrier;           ||     barrier;           ||     ||     barrier;
}                     ||  }                     ||     ||  }
```

- various instantiations of a row-by-row matrix traversal in which several threads work together to examine each element of the matrix. The main feature of this test consists in a complex barrier definition, one containing two imbricated loops which mirror the program code.

- a linear systolic machine for computing polynomials of various degrees by using Horner's algorithm.

- a generic pipeline architecture, each stage in the pipeline is modelled by a thread, the output of one stage is used by the next stage. The barrier ensures that the threads advance in a lockstep manner.

The video encoding tests use two threads that share 5 memory cells. Each test has a different level of precision with respect to the barrier and method specifications: in the `video` tests, we verify a trivial correctness property *i.e.*, we verify the postcondition of `True`, meaning that if the barrier calls succeed then the program finishes safely; we also verified two more complex postconditions by using two more finely-grained barrier definitions: in `video-weak`, we verified a general relationship between the input frame piece and the resulting encoding; finally, in `video-strong` we verified the precise values in the result after the loop terminates. As expected, the tighter bounds require more verification time; however, the differences are relatively small because most of the work is dealing with the heap constraints as opposed to the pure constraints. Part of the time for each example is spent verifying the correctness of the included barrier definition.

We also experimented with various thread count and shared memory footprints. For each of the five test classes described, we include in Figure 4.8 verification timings for implementations with various numbers of threads and shared memory cells. We observe that the thread count by itself does not greatly influences the verification speed, as suggested by the MISD without feedback tests. Neither does the increase in the memory footprint greatly affect the verification speed, as indicated by the pipeline examples. This is due to the fact that both sets of examples

rely on a relatively simple ownership change pattern.

However, for more complex examples like the MISD with feedback or the Horner implementations or the multi-loops, where the ownership patterns or the barrier definition graphs are considerably more complex we can observe a sharp degradation of performance with the increase in the number of tracked memory cells. This is because with more individual resources being tracked while they are pushed through a complex ownership change pattern, the generated heap constraints become more complex leading to more expensive entailments.

The previous results clearly point out that even for relatively small programs, in the current form, barrier calls are fairly computationally expensive to verify due to the need to check multiple, possibly complex, entailments. We believe that the performance can be greatly improved by applying the novel predicate specialization technique described in Chapter 5, thus ensuring that our verification logic is both expressive end efficient enough to warrant wider adoption.

4.7 Summary

We have designed a sound program logic for Pthreads-style barriers. Our development includes a formal design for barrier definitions and a series of soundness conditions to verify that a particular barrier can be used safely. Our Hoare rules can verify threads independently, enabling a thread-modular approach. We have modified the verification toolset HIP/SLEEK to use our logic to verify concurrent programs that use barriers.

Our prototype HIP/SLEEK verification tool is available at:

`www.comp.nus.edu.sg/~cristian/projects/barriers/tool.html`

Acknowledgements. We thank Christian Bienia for showcasing numerous example programs containing barriers, Christopher Chak for help on an early version of this work, Jules Villard for useful comments in general and in particular on the relation of our logic to the logic of his Heap-Hop tool, and Bart Jacobs for discovering how to verify our example program in his

VeriFast tool.

Chapter 5

Effective Verification through Predicate Pruning

We have shown a Hoare logic that successfully verifies programs with exceptions and barriers and we have shown how it can be included in an automatic verifier. However, in order for our proposal to gain acceptance, it is not sufficient to work; it needs to work fast.

The last essential component of our proposal consists of a general predicate pruning targeting disjunctive predicates. In this chapter we will describe the novel predicate pruning calculus and showcase how it can be applied to our verification logic with impressive results.

5.1 Motivation

Abstraction mechanisms are important for modelling and analyzing programs. Recent developments allow richer classes of properties to be expressed via user-defined predicates for capturing commonly occurring patterns of program properties. Separation logic-based abstraction mechanisms represent one such development. As an example, the following predicate captures an abstraction of a sorted doubly-linked list.

$$\begin{array}{l}\textbf{data}\ \texttt{node}\ \{\ \texttt{int val};\ \texttt{node prev};\ \texttt{node next};\ \}\\ \mathit{self}{::}\texttt{dll}\langle p, n, S\rangle \equiv \texttt{self}{=}\texttt{null} \wedge \texttt{n}{=}0 \wedge \texttt{S}{=}\{\}\\ \quad \vee \exists \texttt{v}, \texttt{q}, \texttt{S}_1 \cdot \mathit{self}{::}\texttt{node}\langle v, p, q\rangle * q{::}\texttt{dll}\langle \mathit{self}, n{-}1, S_1\rangle\\ \qquad \wedge \texttt{S} = \texttt{S}_1 \cup \{\texttt{v}\} \wedge \forall \texttt{a} \in \texttt{S}_1 \cdot \texttt{v} \leq \texttt{a} \qquad \textbf{inv}\ \texttt{n} \geq 0;\end{array}$$

In this definition `self` denotes a pointer into the list, `n` the length of the list, `S` represents its set of values, whereas `p` denotes a backward pointer from the first node of the doubly-linked list. The invariant $\texttt{n} \geq 0$ must hold for all instances of this predicate.

We clarify the following points. Firstly, this abstraction mechanism is inherently infinite, due to recursion in predicate definition. Secondly, a predicate definition is capable of capturing multiple features of the data structure it models, such as its size and set of values. While this richer set of features can enhance the precision of a program analysis, it inevitably leads to larger disjunctive formulas.

This chapter focuses on a novel way of handling disjunctive formulas, in conjunction with abstraction via user-defined predicates. While disjunctive forms are natural and expressive, they are major sources of redundancy and inefficiency. The goal of this work is to ensure that disjunctive predicates can be efficiently supported in a program analysis setting, in general, and program verification setting, in particular.

To achieve this, we propose a *specialization calculus* for disjunctive predicates that supports symbolic pruning of infeasible states within each predicate instance. This allows for the implementation of both *incremental pruning* and *memoization* techniques. As a methodology, predicate specialization is not a new concept, since general specialization techniques have been extensively used in the optimization of logic programs [84, 83, 69]. The novelty of our approach stems from applying specialization to a new domain, namely program verification, with its focus on pruning infeasible disjuncts, rather than a traditional focus on propagating static information into callee sites. This new use of specialization yields a fresh approach towards optimising program verification. This approach has not been previously explored, since

pervasive use of user-defined predicates in analysis and verification has only become popular recently (e.g. [75]).

Our key contributions are:

- We propose a *new specialization calculus* that leads to more effective program verification. Our calculus specializes proof obligations produced in the program verification process, and can be used as a preprocessing step before the obligations are fed into third party theorem provers or decision procedures.

- We adapt *memoization* and *incremental pruning* techniques to obtain an optimized version of the specialization calculus.

- We present a prototype implementation of our specialization calculus, integrated into an existing program verification system. The use of our specializer yields significant reductions in verification times, especially for larger problems.

- We adapt the specialization calculus for barrier definitions. We outline the impressive speedup in verifying programs with barriers observed after the introduction of specialization.

5.2 Examples

Program states that are built from predicate abstractions are more concise, but may require properties that are hidden inside predicates. As an example, consider :

$$x{::}\texttt{dll}\langle p_1, n, S_1\rangle * y{::}\texttt{dll}\langle p_2, n, S_2\rangle \wedge \texttt{x}{\neq}\texttt{null}$$

This formula expresses the property that the two doubly-linked lists pointed to by `x` and `y` have the same length. Ideally, we should augment our formula with the property: `y`$\neq$`null`, `n`$>$`0`,

$\mathtt{S_1}{\neq}\{\}$ and $\mathtt{S_2}{\neq}\{\}$, currently hidden inside the two predicate instances but may be needed by the program verification tasks at hand.

A naive approach would be to unfold the two predicate instances, but this would blow up the number of disjuncts to four, as shown:

$$\begin{array}{l}
\mathtt{x{=}null} \wedge \mathtt{y{=}null} \wedge \mathtt{n{=}0} \wedge \mathtt{S_1{=}\{\}} \wedge \mathtt{S_2{=}\{\}} \wedge \mathtt{x{\neq}null} \\
\vee\ y{::}\mathtt{node}\langle v_2, p_2, q_2\rangle * q_2{::}\mathtt{dll}\langle y, n{-}1, S_4\rangle \wedge \mathtt{x{=}null} \\
\quad \wedge \mathtt{S_1{=}\{\}} \wedge \mathtt{n{=}0} \wedge \mathtt{S_2{=}\{v_2\}} \cup \mathtt{S_4} \wedge \mathtt{n{-}1{\geq}0} \wedge \mathtt{x{\neq}null} \\
\vee\ x{::}\mathtt{node}\langle v_1, p_1, q_1\rangle * q_1{::}\mathtt{dll}\langle x, n{-}1, S_3\rangle \wedge \mathtt{y{=}null} \wedge \mathtt{n{=}0} \\
\quad \wedge \mathtt{S_1{=}\{v_1\}} \cup \mathtt{S_3} \wedge \mathtt{S_2{=}\{\}} \wedge \mathtt{n{-}1{\geq}0} \wedge \mathtt{x{\neq}null} \\
\vee\ x{::}\mathtt{node}\langle v_1, p_1, q_1\rangle * y{::}\mathtt{node}\langle v_2, p_2, q_2\rangle * q_1{::}\mathtt{dll}\langle x, n{-}1, S_3\rangle \\
\quad * q_2{::}\mathtt{dll}\langle y, n{-}1, S_4\rangle \wedge \mathtt{S_1{=}\{v_1\}} \cup \mathtt{S_3} \wedge \mathtt{S_2{=}\{v_2\}} \cup \mathtt{S_4} \wedge \mathtt{n{-}1{\geq}0} \wedge \mathtt{x{\neq}null}
\end{array}$$

As contradictions occur in the first three disjuncts, we can simplify our formula to:

$$\begin{array}{l}
x{::}\mathtt{node}\langle v_1, p_1, q_1\rangle * y{::}\mathtt{node}\langle v_2, p_2, q_2\rangle * q_1{::}\mathtt{dll}\langle x, n{-}1, S_3\rangle \\
\quad * q_2{::}\mathtt{dll}\langle y, n{-}1, S_4\rangle \wedge \mathtt{S_1{=}\{v_1\}} \cup \mathtt{S_3} \wedge \mathtt{S_2{=}\{v_2\}} \cup \mathtt{S_4} \wedge \mathtt{n{-}1{\geq}0} \wedge \mathtt{x{\neq}null}
\end{array}$$

After removing infeasible disjuncts, the propagated properties are exposed in the above more *specialized* formula. However, this naive approach has the shortcoming that unfolding leads to an increase in the number of disjuncts handled, and its associated costs.

A better approach would be to avoid predicate unfolding, but instead apply predicate specialization to prune infeasible disjuncts and propagate hidden properties. Given a predicate $v{::}\mathtt{pred}\langle\overrightarrow{v}\rangle$ that is defined by `k` disjuncts, we shall denote each of its specialized instances by $v{::}\mathtt{pred}\langle\overrightarrow{v}\rangle@\mathtt{L}$, where `L` denotes a subset of the disjuncts, namely $\mathtt{L} \subseteq \{1 \ldots k\}$, that have not been pruned. Initially, we can convert each predicate instance $v{::}\mathtt{pred}\langle\overrightarrow{v}\rangle$ to $v{::}\mathtt{pred}\langle\overrightarrow{v}\rangle@\{1 \ldots k\}$, its most general form, while adding the basic invariant of the predicate to its context. As an illustration, we may view the definition of `dll` as a predicate with two

disjuncts, labelled informally by 1: and 2: prior to each of its disjuncts, as follows:

$$\begin{array}{l} \mathit{self}::\mathtt{dll}\langle p, n, S\rangle \equiv \mathtt{1}{:}(\mathtt{self}{=}\mathtt{null} \wedge \mathtt{n}{=}\mathtt{0} \wedge \mathtt{S}{=}\{\}) \\ \vee\ \mathtt{2}{:}(\mathit{self}::\mathtt{node}\langle v, p, q\rangle * q::\mathtt{dll}\langle \mathit{self}, n{-}1, S_1\rangle \wedge \mathtt{S} = \mathtt{S_1}{\cup}\{\mathtt{v}\} \wedge \forall \mathtt{a}{\in}\mathtt{S_1} \cdot \mathtt{v}{\leq}\mathtt{a}) \end{array}$$

We may convert each dll predicate by adding its invariant n≥0, as follows:

$$x::\mathtt{dll}\langle p, n, S\rangle \implies x::\mathtt{dll}\langle p, n, S\rangle@\{1,2\} \wedge \mathtt{n}{\geq}\mathtt{0}$$

With our running example, this would lead to the following initial formula after the same invariant n≥0 (from the two predicate instances) is added.

$$x::\mathtt{dll}\langle p_1, n, S_1\rangle@\{1,2\} * y::\mathtt{dll}\langle p_2, n, S_2\rangle@\{1,2\} \wedge \mathtt{x}{\neq}\mathtt{null} \wedge \mathtt{n}{\geq}\mathtt{0}$$

This predicate may be further specialized with the help of its context by pruning away disjuncts that are found to be infeasible. Each such pruning would allow more states to be propagated by the specialized predicate. By using the context, x≠null, we can specialize the first predicate instance to $x::\mathtt{dll}\langle p_1, n, S_1\rangle@\{2\}$ since this context contradicts the first disjunct of the dll predicate.

In a general case, it would be desirable to have a pre-computed set of pruning conditions which encode rules of the form: when a condition is contradicted, a set of labes can be discarded. In the previous case, the desired pruning condition would be: $x = null{\leftarrow}2$, if x is guaranteed not to be $null$ then label 2 is infeasible.

With the previous specialization, we may strengthen the context with a propagated state, namely n>0 ∧ $\mathtt{S_1}{\neq}\{\}$, that is implied by its specialized instance, as follows:

$$x::\mathtt{dll}\langle p_1, n, S_1\rangle@\{2\} * y::\mathtt{dll}\langle p_2, n, S_2\rangle@\{1,2\} \wedge \mathtt{x}{\neq}\mathtt{null} \wedge \mathtt{n}{>}\mathtt{0} \wedge \mathtt{S_1}{\neq}\{\}$$

Note that n≥0 is removed when a stronger constraint n>0 is added. The new constraint n>0

now triggers a pruning of the second predicate instance, since its first disjunct can be shown to be infeasible. This leads to a specialization of the second predicate, with more propagation of atomic formulas, as follows:

$$x{::}\texttt{dll}\langle p_1, n, S_1\rangle@\{2\} * y{::}\texttt{dll}\langle p_2, n, S_2\rangle@\{2\}$$
$$\wedge\ \texttt{x}{\neq}\texttt{null} \wedge \texttt{n}{>}0 \wedge \texttt{S}_1{\neq}\{\} \wedge \texttt{y}{\neq}\texttt{null} \wedge \texttt{S}_2{\neq}\{\}$$

As another example of predicate specialization, consider the following more general formula:

$$x{::}\texttt{dll}\langle p_1, n, S_1\rangle@\{1,2\} * y{::}\texttt{dll}\langle p_2, n, S_2\rangle@\{1,2\} \wedge \texttt{n}{\geq}0$$

Taken separately, neither of the two predicate instances can be further specialized. However, by analyzing the two predicate instances, we would notice that they share a pair of common pruning conditions, namely $\texttt{n}{=}0$ or $\texttt{n}{>}0$, that could be employed to simultaneously specialize both predicates. Furthermore, $\texttt{n}{=}0 \vee \texttt{n}{>}0$ is implied by the current context $\texttt{n}{\geq}0$. We can therefore perform a case analysis on the above formula to obtain two distinct program states, as shown below:

$$x{::}\texttt{dll}\langle p_1, n, S_1\rangle@\{1,2\} * y{::}\texttt{dll}\langle p_2, n, S_2\rangle@\{1,2\} \wedge \texttt{n}{=}0$$
$$\vee\ x{::}\texttt{dll}\langle p_1, n, S_1\rangle@\{1,2\} * y{::}\texttt{dll}\langle p_2, n, S_2\rangle@\{1,2\} \wedge \texttt{n}{>}0$$

The above case analysis has managed to strengthen each of the two program states in isolation. This strengthening can be harnessed to further specialize both program states as follows:

$$x{::}\texttt{dll}\langle p_1, n, S_1\rangle@\{1\} * y{::}\texttt{dll}\langle p_2, n, S_2\rangle@\{1\} \wedge \texttt{n}{=}0$$
$$\wedge\texttt{x}{=}\texttt{null} \wedge \texttt{y}{=}\texttt{null} \wedge \texttt{S}_1{=}\{\} \wedge \texttt{S}_2{=}\{\}$$
$$\vee\ x{::}\texttt{dll}\langle p_1, n, S_1\rangle@\{2\} * y{::}\texttt{dll}\langle p_2, n, S_2\rangle@\{2\} \wedge \texttt{n}{>}0$$
$$\wedge\texttt{x}{\neq}\texttt{null} \wedge \texttt{y}{\neq}\texttt{null} \wedge \texttt{S}_1{\neq}\{\} \wedge \texttt{S}_2{\neq}\{\}$$

We refer to this general mechanism as *case specialization*. This mechanism is applicable when-

ever we have a set of pruning conditions $\mathtt{S}$ that are common across two or more predicate instances, and under the same context C as follows:

- Each of the pruning conditions is neither implied by, nor does it contradict the context C. That is $\forall \alpha \in \mathtt{S} \cdot \neg(C \implies \alpha \vee C \implies \neg\alpha)$. This condition ensures that all specialization steps have been applied to each of the selected pruning conditions.

- Each pair of pruning conditions are mutually exclusive. That is:

$$\forall \alpha_{\mathtt{i}} \in \mathtt{S} \cdot \forall \alpha_{\mathtt{j}} \in (\mathtt{S} - \{\alpha_{\mathtt{i}}\}) \cdot \alpha_{\mathtt{i}} \wedge \alpha_{\mathtt{j}} \implies \mathtt{false}$$

  This condition ensures that each program state that is being split from the case analysis is also disjoint from every other state.

- The context implies a disjunction for the selected set of pruning conditions for case analysis. That is $C \implies \bigvee_{\alpha \in \mathtt{S}} \alpha$. This allows us to introduce the splitting required by case analysis.

While case specialization allows more pruning to occur for some of the predicate instances, it causes the program state to be represented as multiple disjuncts. One advantage is that more of the infeasible disjuncts from the predicate instances are being pruned from our program state. Also, when compared to unfolding, the original abstraction for each of the predicate instances is still being kept. This allows the converse process of abstraction from the multiple disjuncts, if required, to be more easily carried out.

In a nutshell, the goal of our approach is to apply aggressive specialization to our predicate instances, without the need to resort to predicate unfolding, in the hope that infeasible disjuncts are pruned, where possible. In the process, our specialization technique is expected to propagate states that are consequences of each of the specialized predicate instances. We expect this proposal to support more efficient manipulation of program states, whilst keeping the original abstractions intact where possible.

spred	$::= p(\overrightarrow{v}) \equiv \hat{\Phi}; \mathcal{I}; \mathcal{R}$
$\hat{\Phi}$	$::= \bigvee (\exists \overrightarrow{w} \cdot \hat{\sigma} \mid C)^*$ $\quad\quad$ $\hat{\sigma} ::= \hat{\kappa} \wedge \pi$
$\hat{\kappa}$	$::= emp \mid v::c^v\langle\overrightarrow{v}\rangle \mid v::p^v\langle\overrightarrow{v}\rangle@L\#R \mid \hat{\kappa}_1 * \hat{\kappa}_2$
where	p, v, w, c, π *denote the same as in Figure 2.5;*
	$\mathcal{I}$ *is a family of invariants*;
	$\mathcal{R}$ *is a set of pruning conditions*.
	C is a pure formula denoting a context computed in the specialization process.

Figure 5.1: The Annotated Specification Language.

5.3 Formal Preliminaries

In this section we introduce the basic concepts and terminology that will be used in this chapter.

First we would like to reiterate an observation related to the interplay of predicates and fractional ownership. Given a predicate instance $v::p^{v_f}\langle\overrightarrow{v}\rangle$ it is implied that all resources abstracted by this predicate instance, as described by the predicate definition p are owned with the same share v_f. Therefore ownership annotations within predicate definitions are irrelevant.

Unannotated formulas, as described in Chapter 2 become annotated in the specialization process. Figure 5.1 defines the syntax of the *annotated* specification language. Annotated predicate definitions are generated by the nonterminal *spred*.

Definition 5.3.1 (Annotated Predicates and Formulas)**.**
Given a predicate definition $p(\overrightarrow{v}) \equiv \bigvee (\exists \overrightarrow{w} \cdot \kappa \wedge \pi)^*$*, the corresponding* annotated predicate definition *has the form* $p(\overrightarrow{v}) \equiv \bigvee (\exists \overrightarrow{w} \cdot \hat{\kappa} \wedge \pi \mid C)^*; \mathcal{I}; \mathcal{R}$*, where* $\mathcal{I}$ *is a family of invariants, and* $\mathcal{R}$ *is a set of pruning conditions. Each disjunct* $\exists \overrightarrow{w} \cdot \hat{\kappa} \wedge \pi \mid C$ *now contains the annotated counterpart* $\hat{\kappa}$ *of* κ*, and is augmented with a context C, which is a pure formula for which* $C \rightarrow \pi$ *always holds. Intuitively, C captures also the consequences of the specialized states of* $\hat{\kappa}$*. An* annotated formula *is a formula where all the predicate instances are annotated. An* annotated predicate instance *is of the form* $v::p^v\langle\overrightarrow{v}\rangle@L\#R$*, where* $L \subseteq \{1, .., n\}$ *is a set of labels denoting the unpruned disjuncts, and where* $R \subseteq \mathcal{R}$ *is a set of remaining pruning conditions.*

The set of invariants $\mathcal{I}$ *is of the form* $\{(L{\rightarrow}\pi_L) \mid \emptyset{\subset}L{\subseteq}\{1,..,n\}\}$. *For each set of labels L,* π_L *represents the invariant for the specialized predicate instance* $v{::}p^v\langle\overrightarrow{v}\rangle@L$. *For a given annotated predicate instance* $v{::}p^v\langle\overrightarrow{v}\rangle@L\#R$, *it is possible for* $L = \emptyset$. *When this occurs, it denotes that none of the predicate's disjuncts are satisfiable. Moreover, we have* $\pi_\emptyset{=}false$ *which will contribute towards a false state (or contradiction) for its given context.*

Definition 5.3.2 (Pruning Condition)**.** *A* pruning condition *is a pair of an atomic predicate instance* α *and a set of labels L, written* $\alpha{\leftarrow}L$. *Its intuitive meaning is that the disjuncts in L should be kept if* α *is satisfiable in the current context. The symbol* $\mathcal{R}$ *denotes a finite set of such pruning conditions.*

In the case of the $\texttt{dll}\langle p, n, S\rangle$ predicate, a pruning condition can be $S \neq \{\}{\leftarrow}2$. That is, if the set of elements in the list is shown to be empty, then the inductive branch, labeled 2, can be discarded as it is infeasible. Similarly $n = 0{\leftarrow}1$ indicates that if the number of elements in the list is not 0 then the base case, the first disjunct is infeasible. Lastly, we observe there is no explicit limit on the pruning condition guards. They could be expressed on pointers as well: $\texttt{self} = null{\leftarrow}1$. The practical restriction is however that the conditions allow a fast satisfiability check.

Given a predicate definition $p(\overrightarrow{v}) \equiv \bigvee_{i=1}^{n}(\exists\overrightarrow{w}{\cdot}\hat{\sigma_i} \mid C_i)$; $\mathcal{I}$; $\mathcal{R}$, we call $\texttt{D}_i =_{df} (\exists\overrightarrow{w}{\cdot}\hat{\sigma_i} \mid C_i)$ the i^{th} disjunct of p ; i will be called the *label* of its disjunct. We shall use $\texttt{D}_i$ freely as the i^{th} disjunct of the predicate at hand whenever there is no risk of confusion. We employ a notion of *closure* for a given conjunctive formula. Consider a formula $\pi(\overrightarrow{w}) = \exists\overrightarrow{v}{\cdot}\alpha_1{\wedge}\cdots{\wedge}\alpha_m$, where α_i are atomic predicates, and variables $\overrightarrow{w}$ appear free. We denote by $\texttt{S} = closure(\pi(\overrightarrow{w}))$ a set of atomic predicates (over the free variables $\overrightarrow{w}$) such that each element $\alpha \in \texttt{S}$ is entailed by $\pi(\overrightarrow{w})$. Some of the variables $\overrightarrow{w}$ may appear free in α but *not* $\overrightarrow{v}$. To ensure this closure set be finite, we also impose a requirement that weaker atomic constraints are never present in the same set, as follows: $\forall\alpha_i \in \texttt{S}\cdot\neg(\exists\alpha_j \in \texttt{S}\cdot i{\neq}j \wedge \alpha_i \Longrightarrow \alpha_j)$. Ideally, *closure*$(\pi(\overrightarrow{w}))$ contains *all* stronger atomic formulas entailed by $\pi(\overrightarrow{w})$, though depending on the abstract domain used, this set may not be computable. A larger closure set leads to more aggressive pruning.

We extend the semantic model of the assertion language, as presented in §2.4.1 to also cover annotated predicates and formulas as follows:

Definition 5.3.3 (Extension of Assertion Model)**.**

$$
\begin{array}{lll}
s,h,b \models \exists \overrightarrow{v} \cdot \hat{\sigma} \mid \pi & \texttt{iff} & s,h,b \models \exists \overrightarrow{v} \cdot \hat{\sigma} \wedge \pi \\
s,h,b \models \exists \overrightarrow{v} \cdot \hat{\kappa} \wedge \pi & \texttt{iff} & \exists \overrightarrow{\nu} \cdot s[\overrightarrow{v} \mapsto \overrightarrow{\nu}], h, b \models \hat{\kappa} \textit{ and } s[\overrightarrow{v} \mapsto \overrightarrow{\nu}] \models \pi \\
s,h,b \models \hat{\kappa}_1 * \hat{\kappa}_2 & \texttt{iff} & \exists h_1,h_2,b_1,b_2 \cdot (s,h_1,b_1) \oplus (s,h_2,b_2) = (s,h,b) \\
 & & \textit{and } s,h_1,b_1 \models \hat{\kappa}_1 \textit{ and } s,h_2,b_2 \models \hat{\kappa}_2 \\
s,h,b \models v_0 :: p^{v_f} \langle \overrightarrow{v} \rangle @L\#R & \texttt{iff} & (p(\overrightarrow{w}) \equiv \bigvee_{i=1}^{n} (\exists \overrightarrow{u} \cdot \hat{\sigma}_i | \pi_i); \mathcal{I}; \mathcal{R}) \textit{ and} \\
 & & s,h,b \models \bigvee_{i \in L} (\exists \overrightarrow{u} \cdot set\_perm(\hat{\sigma}_i, v_f) | \pi_i) \, [\overrightarrow{v}/\overrightarrow{w}]
\end{array}
$$

Here $v :: p^{v} \langle \overrightarrow{v} \rangle @L\#R$ denotes an annotated predicate instance.

Our specialization calculus (§ 5.4) is based on the annotated specification language. We have an initialization and inference process (§ 5.5) to automatically generate all annotations (including $\mathcal{I}, \mathcal{R}$) that are required by specialization. For simplicity of presentation, we only include normalized linear constraints in our language. Our system currently supports both arbitrary linear arithmetic constraints, as well as set constraints. This is made possible by integrating the Omega [85] and MONA solvers [63] into the system. In principle, the system may support arbitrary constraint domains, provided that a suitable solver is available for the domain of interest. Such a solver should be capable of handling conjunctions efficiently, as well as computing approximations of constraints that convert disjunctions into conjunctions (e.g. hulling).

5.4 A Specialization Calculus

Our specialization framework detects infeasible disjuncts in predicate definitions without explicitly unfolding them, and computes a corresponding strengthening of the pure part while preserving satisfiability. We present this as a calculus consisting of *specialization rules* that

can be applied exhaustively to convert a non-specialized annotated formula[1] into a fully specialized one, with stronger pure parts, that can be subsequently extracted and passed on to a theorem prover for satisfiability/entailment checking. Apart from being syntactically correct, annotated formulas must satisfy the following well-formedness conditions.

Definition 5.4.1 (Well-formedness). *For each annotated predicate* $v{::}p^v\langle\overrightarrow{v}\rangle@L\#R$ *in the formula at hand, assuming the definition* $p(\overrightarrow{v})\equiv\bigvee_{i=1}^{n}\mathtt{D_i};\mathcal{I};\mathcal{R}$, *we have that (a)* $L\subseteq\{1,\ldots,n\}$ *; (b)* $R\subseteq\mathcal{R}$ *; and (c) forall* $\alpha{\leftarrow}L_0\in R$ *we have* $L\cap L_0\neq\emptyset$.

Definition 5.4.2 (Specialization Step). *A* specialization step *has the form* $\hat{\Phi}_1\mid C_1\ \text{–}sf\!\!\rightarrow\ \hat{\Phi}_2\mid C_2$, *and denotes the relation allows the annotated formula* $\hat{\Phi}_1$ *with context* C_1 *to be transformed into a more specialized formula* $\hat{\Phi}_2$ *with context* C_2.

Our calculus produces specialization steps, which are applied in sequence, exhaustively, to produce *fully specialized* formulas (a formal definition of such formulas will be given below). Relation –sf→ depends on relation –sp→, which produces predicate specialization steps defined by the following:

Definition 5.4.3 (Predicate Specialization Step). *A* predicate specialization *step has form*

$$(1)\qquad v{::}p^v\langle\overrightarrow{v}\rangle@L_1\#R_1\mid C_1\ \text{–}sp\!\!\rightarrow\ v{::}p^v\langle\overrightarrow{v}\rangle@L_2\#R_2\mid C_2.$$

and signifies that annotated predicate $v{::}p^v\langle\overrightarrow{v}\rangle@L_1\#R_1\mid C_1$ *can be specialized into* $v{::}p^v\langle\overrightarrow{v}\rangle@L_2\#R_2\mid C_2$, *where* $L_2\subseteq L_1$, $R_2\subset R_1$, *and* C_2 *is stronger than* C_1.

Here, the sets L_1 and L_2 denote sets of disjuncts of $v{::}p^v\langle\overrightarrow{v}\rangle$ that have not been detected to be infeasible. Each specialization step aims at detecting new infeasible disjuncts and removing them during the transformation. Thus L_2 is expected to be a subset of L_1.

The sets of pruning conditions R_1 and R_2 may be redundant, but are instrumental in making specialization efficient. They record incremental changes to the state of the specializer, and

[1] The conversion of non-annotated formulas into annotated ones shall be presented in § 5.5.

represent information that would be expensive to re-compute at every step. Essentially, a pruning condition $\alpha \leftarrow L_0$ states that whenever $\neg\alpha$ is entailed by the current context, the disjuncts whose labels are in L_0 can be pruned. The initial set of pruning conditions is derived when converting formulas into annotated formulas, and is formally discussed in § 5.5.

In a nutshell, each specialization step of the form (1) detects (if possible) a pruning condition $\alpha \leftarrow L_0 \in R$ such that if $\neg\alpha$ is entailed by the current context, then the disjuncts whose labels occur in L_0 are infeasible and can be pruned. Given the notations in (1), this is achieved by setting $L_2 = L_1 - L_0$. Subsequently, the current set of pruning conditions is reduced to contain only elements of the form $\alpha' \leftarrow L_0'$ such that $L_0' \cap L_2 \neq \emptyset$. Thus, the well-formedness of the annotated formula is preserved

A key aspect of a specialization step is that, as a result of pruning, the context of a predicate can often be strengthened due to the fact that fewer disjuncts are now potentially feasible. The strengthening procedure shall be discussed later in this section, when the calculus rules are introduced. However, at this point, we would like to emphasize the important role that context strengthening plays in the overall process of formula specialization. After having specialized a predicate in the formula of interest, we can use the resulting stronger context to specialize a second predicate. The new strengthening may create a new opportunity for specializing the first predicate again, which in turn will strengthen the context enough to lead to yet another specialization of the second predicate, and so on. In fact, context strengthening helps reveal and prune *mutually infeasible* disjuncts in groups of predicates, which leads to a more aggressive optimization as compared to the case where predicates are specialized in isolation.

Definition 5.4.4 (Fully Specialized Formula; Complete Specialization)**.** *An annotated formula is* fully specialized *w.r.t a context when all its annotated predicates have empty pruning condition sets. If the initial pruning condition sets are computed using a notion of* strongest closure*, then for each predicate in the fully specialized formula, all the remaining labels in the predicate's label set denote feasible disjuncts with respect to the current context, and in that sense, the specialization is* complete*.*

$$\frac{R_f = \{(\alpha \leftarrow L_0) \mid (\alpha \leftarrow L_0) \in R,\ (L \cap L_0 = \emptyset) \vee (C \Longrightarrow \alpha)\}}{C, L \vdash R - \text{filter}\rightarrow (R - R_f)} \quad [\textbf{SP}-[\textbf{FILTER}]]$$

$$\frac{C \wedge \alpha \Longrightarrow \texttt{false} \quad (\alpha \leftarrow L_0) \in R \quad L \cap L_0 \neq \emptyset \quad L_2 = L - L_0 \quad C_1 = \mathcal{I}nv(v{::}p^v\langle\overrightarrow{v}\rangle, L_2) \quad C \wedge C_1, L_2 \vdash R - \text{filter}\rightarrow R_1}{v{::}p^v\langle\overrightarrow{v}\rangle@L\#R \mid C \ \text{–sp}\rightarrow\ v{::}p^v\langle\overrightarrow{v}\rangle@L_2\#R_1 \mid C \wedge C_1} \quad [\textbf{SP}-[\textbf{PRUNE}]]$$

$$\frac{C, L \vdash R - \text{filter}\rightarrow \emptyset \qquad R \neq \emptyset}{v{::}p^v\langle\overrightarrow{v}\rangle@L\#R \mid C \ \text{–sp}\rightarrow\ v{::}p^v\langle\overrightarrow{v}\rangle@L\#\emptyset \mid C} \quad [\textbf{SP}-[\textbf{FINISH}]]$$

Figure 5.2: Single-step Predicate Specialization

This procedure is formalized in the calculus rules given in Figures 5.2 and 5.3. Figure 5.2 defines the predicate specialization relation –sp→. This relation has two main components: the one represented by the rule [SP–[FILTER]], which restores the well-formedness of an annotated predicate, and the one represented by the rule [SP–[PRUNE]], which detects infeasible disjuncts and removes the corresponding labels from the annotation. A third rule, [SP–[FINISH]] produces the fully specialized predicate. The rules in Figure 5.3 turn predicate specialization steps into formula specialization steps. Let us examine the rules in detail.

The rule [SP–[FILTER]] restores the well-formedness of an annotated predicate, and is necessary because, as the next rule will reveal, once a new set of labels L, and a new context C have been inferred during a specialization step, some of the elements in the current pruning condition set R become irrelevant.

The $-\text{filter}\rightarrow$ relation collects pruning conditions that are no longer relevant to the newly specialized set of labels into the set R_f and then removes them from the original set R.

The rule [SP–[PRUNE]] defines the part of the relation –sp→ that performs the actual prun-

ing. In transforming the annotated predicate $v{::}p^v\langle\overrightarrow{v}\rangle@L\#R$, the rule picks a pruning condition $\alpha{\leftarrow}L_0 \in R$ such that $C \wedge \alpha$ is unsatisfiable, with the added requirement that L and L_0 not be disjoint. Having found such a pruning condition, it is necessarily the case that those disjuncts of predicate p whose labels occur in L_0 are infeasible in the current context C. Thus, replacing L with $L - L_0$ will not change the satisfiability of the predicate. Consequently, due to the fact that we have fewer potentially feasible disjuncts, a new, stronger context can be computed for $v{::}p^v\langle\overrightarrow{v}\rangle$. This new context is based on the procedure $\mathcal{I}nv$ that is detailed in § 5.5. Intuitively, this procedure computes a conjunction of atomic formulas entailed by all the disjuncts whose labels are in L_2. Obviously, it is desirable for this conjunction to be as strong as possible, and the choices available for computing it shall be discussed in due course. For the time being, however, we shall only assume that the result of $\mathcal{I}nv$ is stronger when the procedure is applied to fewer disjuncts. The new context is obtained by conjoining the old one with the result of the $\mathcal{I}nv$ procedure. Finally, the well-formedness of the predicate is restored by applying the $-$filter$\rightarrow$ relation. The set of pruning conditions effectively shrinks as a result, since the $\alpha{\leftarrow}L_0$ element of R is indeed filtered out.

The rule [SP$-$[FINISH]] handles the case when all remaining pruning conditions are irrelevant for the current context and set of labels. It defines the part of the $-$sf$\rightarrow$ relation that transforms a predicate into a fully specialized one.

The predicate specialization relation can be weaved into the formula specialization relation given in Figure 5.3. The first rule, [SF$-$[PRUNE]], defines the part of the $-$sf$\rightarrow$ relation which picks a predicate in a formula and transforms it using the $-$sp$\rightarrow$ relation, leaving the rest of the predicates unchanged. However, the transformation of the predicate's context is incorporated into the transformation of the formula's context. This rule realizes the potential for cross-specialization of predicates, eliminating disjuncts of different predicates that are mutually unsatisfiable.

The rule [SF$-$[CASE$-$SPLIT]] allows further specialization via case analysis. It defines the part of the $-$sf$\rightarrow$ relation that produces two instances of the same formula, joined in a

$$\frac{[\textbf{SF}\text{–}[\textbf{PRUNE}]]}{v{::}p^v\langle\overrightarrow{v}\rangle@L\#R \mid C \text{ –sp→ } v{::}p^v\langle\overrightarrow{v}\rangle@L_2\#R_2 \mid C_2}{v{::}p^v\langle\overrightarrow{v}\rangle@L\#R * \hat{\kappa} \mid C \text{ –sf→ } v{::}p^v\langle\overrightarrow{v}\rangle@L_2\#R_2 * \hat{\kappa} \mid C_2}$$

$$\frac{[\textbf{SF}\text{–}[\textbf{CASE}\text{–}\textbf{SPLIT}]]}{\vdash C \Longrightarrow \alpha_1 \vee \alpha_2 \quad \vdash \alpha_1 \wedge \alpha_2 \Longrightarrow \texttt{false} \quad \forall i \in \{1,2\} \cdot \hat{\kappa} \mid C \wedge \alpha_i \text{ –sf→ } \hat{\kappa}_i \mid C_i}{\hat{\kappa} \mid C \text{ –sf→ } (\hat{\kappa}_1 \mid C_1) \vee (\hat{\kappa}_2 \mid C_2)}$$

$$\frac{[\textbf{SF}\text{–}[\textbf{OR}]]}{\hat{\kappa}_1 \mid C_1 \text{ –sf→ } \hat{\kappa}_3 \mid C_3}{(\hat{\kappa}_1 \mid C_1) \vee (\hat{\kappa}_2 \mid C_2) \text{ –sf→ } (\hat{\kappa}_3 \mid C_3) \vee (\hat{\kappa}_2 \mid C_2)}$$

Figure 5.3: Single-step Formula Specialization

disjunction, each of the new formulas having a stronger context. Each stronger context is produced by conjunction with an atom α_i, $i \in \{1, 2\}$, with the requirement that the two atoms be disjoint and their disjunction cover the original context C. This rule is instrumental in guaranteeing that all predicates reach a fully specialized status. Indeed, whenever an annotated predicate has a pruning condition $\alpha \leftarrow L_0$, such that α is not entailed by the context C, yet $\alpha \wedge C$ is satisfiable, the only way to further specialize the predicate is by case analysis with the atoms α and $\neg\alpha$. Finally, the rule [SF–[OR]] handles formulas with multiple disjunctions.

In the remainder of this section, we formalize the notion that our calculus produces terminating derivations, and is sound and complete.

Property 1. *Relations –sp→ and –sf→ preserve well-formedness. Thus, given two annotated predicate instances* $v{::}p^v\langle\overrightarrow{v}\rangle@L_1\#R_1$ *and* $v{::}p^v\langle\overrightarrow{v}\rangle@L_2\#R_2$*, if*

$$v{::}p^v\langle\overrightarrow{v}\rangle@L_1\#R_1 \mid C_1 \text{ –sp→ } v{::}p^v\langle\overrightarrow{v}\rangle@L_2\#R_2 \mid C_2$$

can be derived from the calculus, and $v{::}p^v\langle\overrightarrow{v}\rangle@L_1\#R_1$ *is well-formed, then*

$v{::}p^v\langle\overrightarrow{v}\rangle@L_2\#R_2$ is well-formed as well. Moreover, for all annotated formulas $\hat{\Phi}_1$ and $\hat{\Phi}_2$, if

$$\hat{\Phi}_1 \mid C_1 \dashv sf\mapsto \hat{\Phi}_2 \mid C_2$$

can be derived from the calculus, and $\hat{\Phi}_1$ is well-formed, then $\hat{\Phi}_2$ is well formed as well.

Proof sketch: The relation $-\text{filter}\mapsto$ is responsible for maintaining the well-formedness of annotated formulas by filtering out exactly those pruning conditions that violate the well-formedness requirements with respect to a newly computed set of labels. Filtering of the pruning condition set is employed in both the $[\underline{\textbf{SP}-[\textbf{PRUNE}]}]$ and $[\underline{\textbf{SP}-[\textbf{FINISH}]}]$ rules, ensuring that their output is well-formed.

In the case of the $[\underline{\textbf{SF}-[\textbf{PRUNE}]}]$, $[\underline{\textbf{SF}-[\textbf{CASE}-\textbf{SPLIT}]}]$, and $[\underline{\textbf{SF}-[\textbf{OR}]}]$ rules, the proof follows from the assumption that the relations given as a premises have the preservation property, and the fact that neither of these rules make explicit changes in the predicate annotations. □

A *predicate specialization sequence* is a sequence of annotated predicates such that each pair of consecutive predicates is in the relation $\dashv\text{sp}\mapsto$. A *formula specialization sequence* is a sequence of annotated formulas such that each pair of consecutive formulas is in the $\dashv\text{sf}\mapsto$ relation.

Definition 5.4.5 (Canonical Specialization Sequence)**.** *A canonical specialization sequence is a formula specialization sequence where:*

1. *the first element is well-formed;*
2. *specialization rules are applied exhaustively ;*
3. *the $[\underline{\textbf{SF}-[\textbf{CASE}-\textbf{SPLIT}]}]$ relation is only applied as a last resort (i.e. when no other relation is applicable);*
4. *the case analysis atoms for the $[\underline{\textbf{SF}-[\textbf{CASE}-\textbf{SPLIT}]}]$ relation must be of the form α, $\neg\alpha$, where $\alpha\leftarrow L_0$ is a pruning condition occurring in an annotated predicate $v{::}p^v\langle\overrightarrow{v}\rangle@L\#R$*

of the formula, such that $L \cap L_0 \neq \emptyset$.

Property 2 (Termination)**.** *All canonical specialization sequences are finite and produce either fully specialized formulas, or formulas whose context is unsatisfiable.*

Proof sketch The proof is divided into two parts. The first part addresses the finiteness of specialization sequence. We note that the "driving" steps of the calculus are provided by the rules $[\underline{\textbf{SP}-[\textbf{PRUNE}]}]$, $[\underline{\textbf{SP}-[\textbf{FINISH}]}]$, and $[\underline{\textbf{SF}-[\textbf{CASE}-\textbf{SPLIT}]}]$. The rest of the rules are "weaving" rules that establish how the "driving" steps are to operate in larger contexts. Thus, it is sufficient to show that these three steps produce relations where some measure decreases. It is obvious that in both the $[\underline{\textbf{SP}-[\textbf{PRUNE}]}]$ and $[\underline{\textbf{SP}-[\textbf{FINISH}]}]$ rules the size of the R annotation is decreased for the predicate at hand. Since a formula has only a finite number of predicates, each with a finite number of pruning conditions, the relations produced by these rules can only be applied a finite number of times.

The $[\underline{\textbf{SF}-[\textbf{CASE}-\textbf{SPLIT}]}]$ rule poses a somewhat more complicated problem, since each application of the resulting relation increases the size of the formula at hand. However, each split will use up one pruning condition of the form $\alpha{\leftarrow}L_0$, which cannot be subsequently reused. To understand this, let us examine how the two branches, whose contexts are strengthened respectively with $\neg\alpha$ and α, behave after the split. In the branch strengthened with $\neg\alpha$, a specialization step will be immediately applicable, removing the labels in L_0 from the predicate's L annotation. As a consequence, $\alpha{\leftarrow}L_0$ will be filtered out as well, and subsequent case analysis steps on this branch will not be able to reuse this pruning condition. On the other hand, in the branch strengthened with α, α is now entailed by the context and further ineligible for case analysis, at least not without violating the canonical specialization sequence requirements. Thus, even though a formula increases in size as a result of case analysis, it can only do so a finite number of times for each predicate. As a result, the specialization sequence is necessarily finite.

For the second part of the proof, let us assume a canonical specialization sequence whose last element is a formula that is not fully specialized (i.e. $R{\neq}\emptyset$ for some predicate), and whose

context is satisfiable. According to Property 1, this last element must be well formed. Since the set R is not empty, one of the following cases must hold:

- There exists a pruning condition $\alpha \leftarrow L_0$ that occurs in the annotation of some predicate $v::p^v\langle\overrightarrow{v}\rangle@L\#R \mid C$, such that $\alpha \wedge C$ and $\neg\alpha \wedge C$ are both satisfiable, and also $L \cap L_0 \neq \emptyset$. In this case, a case-analysis step is applicable.

- There exists a pruning condition $\alpha \leftarrow L_0$ that occurs in the annotation of some predicate $v::p^v\langle\overrightarrow{v}\rangle@L\#R \mid C$, such that $\alpha \wedge C$ is not satisfiable, and also $L \cap L_0 \neq \emptyset$. In this case, a pruning step is applicable.

- Finally, either all the pruning conditions have α entailed by the current context, or $L_0 \cap L = \emptyset$. In this case, the $[\underline{\textbf{SP}-[\textbf{FINISH}]}]$ is applicable.

This shows that for any well-formed formula that is not fully specialized, some rule is applicable. This contradicts the hypothesis that the specialization sequence was canonical. Thus, each canonical sequence must end with a fully specialized formula. □

Property 3 (Soundness)**.** *The $-sp\rightarrow$ and $-sf\rightarrow$ relations preserve satisfiability. Thus: if $v::p^v\langle\overrightarrow{v}\rangle@L_1\#R_1 \mid C_1 \; -sp\rightarrow \; v::p^v\langle\overrightarrow{v}\rangle@L_2\#R_2 \mid C_2$ can be derived from the calculus, then for all heaps h, stacks s and barrier maps b, it follows that $s,h,b \models v::p^v\langle\overrightarrow{v}\rangle@L_1\#R_1 \mid C_1$ iff $s,h,b \models v::p^v\langle\overrightarrow{v}\rangle@L_2\#R_2 \mid C_2$. Moreover, if $\hat{\Phi}_1 \mid C_1 \; -sf\rightarrow \hat{\Phi}_2 \mid C_2$ can be derived from the calculus, then $s,h,b \models \hat{\Phi}_1|C_1$ iff $s,h,b \models \hat{\Phi}_2|C_2$.*

Proof It is sufficient to prove that each rule produces a relation that preserves the satisfiability of its arguments. The satisfiability of a formula depends on the satisfiability of its annotated predicates, which in turn depends on the set of labels L that represents the disjuncts that have yet to be checked for infeasibility. Since the rule $[\underline{\textbf{SP}-[\textbf{PRUNE}]}]$ is the only one that specifically removes elements from L, it is sufficient to prove that this rule produces only relations

that preserve the satisfiability of the predicates given as arguments. This rule selects a pruning condition $\alpha \leftarrow L_0$ such that, given the pruning context C, we have that $C \wedge \alpha$ is unsatisfiable. Since, by the construction[1] of R, α is entailed by all the pure parts of disjuncts whose labels are in L_0, it follows immediately that those disjuncts are infeasible in the context C. Thus, by replacing L with $L_2 = L - L_0$, the satisfiability of the predicate is preserved with respect to the current context C. The next step is the strengthening of the context, which is performed by computing an invariant C_2 of the *remaining* disjuncts. Since C_2 is entailed by all the disjuncts represented in L_2, conjoining it with the current context C will produce an equisatisfiable formula. □

We note here that the set R does not play a role in the way an annotated predicate is interpreted. Mishandling R (as long as no elements are added) may result in lack of termination or incompleteness, but does not affect soundness.

Finally, we address the issue of completeness. This property, however, is dependent on how "complete" the conversion of a predicate into its annotated form is. Thus, we shall first give an ideal characterization of such a conversion, after which we shall endeavour to prove the completeness property. Realistic implementations of this conversion shall be discussed in §5.6.

Definition 5.4.6 (Strongest Closure)**.** *The strongest closure of unannotated formula Φ, denoted by sclosure(Φ), is the largest set of atoms α with the following properties: (a) for all stacks s, $s \models \alpha$ whenever there exists h, b such that $s, h, b \models \Phi$, and (b) there exists no atom α' strictly stronger than α – that is, it is not the case that for all s, $s \models \alpha$ whenever $s \models \alpha'$. For practical and termination reasons, we shall assume only closures which return finite sets in our formulation.*

[1] A subtle point here is that the set R continue to enjoy this property after repeated applications of specialization steps. This is indeed true since, via the $-$filter$\rightarrow$ relation, each specialization step only removes elements from R.

In our conversion of an unannotated predicate definition for $p(\overrightarrow{v})$ into the annotated definition $p(\overrightarrow{v}) \equiv \bigvee_{i=1}^{n} \mathtt{D}_i; \mathcal{I}; \mathcal{R}$, we compute the following sets:

- $\mathtt{G}_i = \mathit{sclosure}(\mathtt{D}_i \wedge \pi)$, for $i = 1, .., n$,
- $\mathtt{H}_L = \{\alpha \mid \text{forall } i \in L, \text{exists } \alpha' \in \mathtt{G}_i \text{ s.t. forall } s, s \models \alpha' \text{ whenever } s \models \alpha\}$
- $\mathcal{I} = \{L \rightarrow \pi \mid L \subseteq \{1...n\},\ \pi = \bigwedge_{\alpha \in \mathtt{H}_L} \alpha\}$, and
- $\mathcal{R} = \{\alpha \leftarrow L \mid L \text{ is the largest set s.t. } \alpha \in \bigcap_{i \in L} \mathtt{G}_i\}$.

Moreover, we introduce the notation $\mathcal{I}nv(v{::}p^v\langle\overrightarrow{v}\rangle, L) = \pi_L$, where $(L \rightarrow \pi_L) \in \mathcal{I}$. This notation is necessary in applying the rule [SP−[PRUNE]].

In practice, either the assumption holds, or the closure procedure computes a close enough approximation to the strongest closure so that very few, if any, infeasible disjuncts are left in the specialized formula.

Property 4 (Completeness)**.** *Let $v{::}p^v\langle\overrightarrow{v}\rangle@L\#\emptyset * \hat{\sigma}|C$ be a fully specialized formula that resulted from a specialization process that started with an annotated formula. Denote by π_i, $1 \leq i \leq n$ the pure parts of the disjuncts in the definition of $\mathtt{p}(\overrightarrow{v})$, and assume that C is satisfiable. Then, for all $i \in L$, $\pi_i \wedge C$ is satisfiable.*

Proof Let us assume that we have a formula specialization sequence whose last element, while fully specialized, has a predicate with an annotation that contains the label of an infeasible disjunct. That is, the fully specialized formula has the form $v{::}p^v\langle\overrightarrow{v}\rangle@L\#\emptyset * \hat{\sigma}|C$, and it is the case that $\mathtt{D_i} \wedge C$ is unsatisfiable. Since $\mathcal{R}$ is part of a well-formed annotation, there must exist a set of pruning conditions of the form $(\alpha \leftarrow L_0) \in \mathcal{R}$ such that the following three conditions hold simultaneously: $\mathtt{D_i}$ entails α, $i \in L_0$, and $\alpha \wedge C$ is unsatisfiable. The fact that neither of these pruning conditions could be used to prune the i^{th} disjunct $\mathtt{D_i}$ can only be attributed to the fact that all these pruning conditions were filtered out before any of them had a chance to be picked by a pruning step.

Let us consider the step in the specialization sequence where the last of these pruning conditions was filtered out. Assume that the transformation performed at this step relies on the following predicate specialization relation:

$$v{::}p^{v}\langle\overrightarrow{v}\rangle@L_1\#R_1 \mid C_1 \multimap \mathrm{sp} \rightarrow v{::}p^{v}\langle\overrightarrow{v}\rangle@L_2\#R_2 \mid C_2,$$

and that the pruning condition of interest is filtered out in the process of computing R_2 from R_1 and L_2. For the filtering operation to proceed as described, either of the following conditions (preconditions to the rule $[\underline{\mathbf{SP}-[\mathbf{FILTER}]}]$) must hold:

- $L_2 \cap L_0 = \emptyset$; this can only happen if the disjunct has already been pruned, which contradicts the hypothesis that it was still present in the last element.
- $C_2 \Longrightarrow \alpha$; this contradicts the hypothesis that the context C of the fully specialized formula, which can only be stronger than C_2, would have the property that $C \wedge \alpha$ is unsatisfiable.

Thus, the hypothesis that the fully specialized formula has infeasible disjuncts is false. □

5.5 Inferring Specializable Predicates

We present inference techniques that must be applied to each predicate definition so that they can support the specialization process. We refer to this process as *inference for specializable predicates*. A predicate is said to be *specializable* if it has multiple disjuncts and it has a non-empty set of pruning conditions. These two conditions would allow a predicate instance to be specializable from one form to another specialized form. Our predicates are processed in a bottoms-up order with the following key steps:

- Transform each predicate definition to its specialized form.
- Compute an *invariant* (in conjunctive form) for each predicate.

- Compute a *family of invariants* to support all specialized instances of the predicate.
- Compute a *set of pruning conditions* for the predicate.
- Specialize recursive invocations of the predicate, if possible.

As a running example for this inference process, let us consider the following predicate which could be used to denote a list segment of singly-linked nodes, for conciseness we use x instead of $self$:

$$\textbf{data}\ \texttt{snode}\ \{\ \texttt{int val};\ \texttt{snode next};\}$$
$$x{::}\texttt{lseg}\langle p, n\rangle\ \equiv\ \texttt{x=p}\wedge\texttt{n=0}\vee\exists \texttt{q},\texttt{m}\cdot x{::}\texttt{snode}\langle\_, q\rangle * q{::}\texttt{lseg}\langle p, m\rangle\wedge\texttt{m=n-1}$$

Our inference technique derives the following specializable predicate definition:

$$\begin{array}{l}\texttt{x}{::}\texttt{lseg}\langle p, n\rangle \equiv \texttt{x=p}\wedge\texttt{n=0}\mid\texttt{x=p}\wedge\texttt{n=0}\vee\\ \exists\texttt{q},\texttt{m}\cdot x{::}\texttt{snode}\langle\_, q\rangle{*}q{::}\texttt{lseg}\langle p, m\rangle\wedge\texttt{m=n-1}\mid\texttt{x}\neq\texttt{null}\wedge\texttt{n}>0;\\ \mathcal{I}=\{\{1\}\rightarrow\texttt{x=p}\wedge\texttt{n=0},\ \{2\}\rightarrow\texttt{x}\neq\texttt{null}\wedge\texttt{n}>0,\ \{1,2\}\rightarrow\texttt{n}\geq 0\};\\ \mathcal{R}=\{\texttt{x=p}\leftarrow\{1\},\ \texttt{n=0}\leftarrow\{1\},\texttt{x}\neq\texttt{null}\leftarrow\{2\},\ \texttt{n}>0\leftarrow\{2\}\}\end{array}$$

Note that we have a family of invariants, named $\mathcal{I}$, to cater to each of the specialized states. The most general invariant for a $lseg$ predicate instance is :

$$\mathcal{I}\texttt{nv}(x{::}\texttt{lseg}\langle p, n\rangle, \{1,2\}) = \texttt{n}\geq 0$$

This is computed by a fix-point analysis [24] on the body of the predicate. If we determine that a particular predicate instance can be specialized to $x{::}\texttt{lseg}\langle p, n\rangle@\{2\}$, we may use a stronger invariant $\mathcal{I}\texttt{nv}(x{::}\texttt{lseg}\langle p, n\rangle, \{2\}) = \texttt{x}\neq\texttt{null}\wedge\texttt{n}>0$ to propagate this constraint from the specialized instance. Such a family of invariants allows us to enrich the context of the predicate instances that are being progressively specialized.

Furthermore, we must process the predicate definitions in a bottom-up order, so that predicates lower in the definition hierarchy are inferred before predicates higher in the hierarchy. This is needed since we intend to specialize the body of each predicate definition with the help of specialized definitions that were inferred earlier. In the case of a set of mutually-recursive predicate definitions, we shall process this set of predicates simultaneously. Initially, we shall assume that the set of pruning conditions for each recursive predicate is empty, which makes its recursive instances unspecializable. However, once its set of pruning conditions has been determined, we can apply further specialization so that the recursive invocations of the predicate are specialized as well.

$$\frac{}{\kappa \wedge \pi \ -\text{if}\rightarrow \hat{\kappa} \wedge \pi \mid C_1 \qquad \hat{\kappa} \mid C_1 \ -\text{sf}\rightarrow^* \hat{\kappa}_1 \mid C_2}{\kappa \wedge \pi \ -\text{msf}\rightarrow \hat{\kappa}_1 \wedge \pi \mid C_2} \ [\textsc{init}\text{-}[\textsc{multi}\text{-}\textsc{spec}]]$$

$$\frac{spred_{old} \ = \ (\mathtt{p}(\overrightarrow{v}) \equiv \bigvee_{i=1}^{n}(\exists \overrightarrow{u}_i \cdot \sigma_i)) \qquad \forall i \in \{1,..,n\} \cdot \sigma_i \ -\text{msf}\rightarrow \hat{\sigma}_i \mid C_i \quad spred_{new} \ = \ (\mathtt{p}(\overrightarrow{v}) \equiv \bigvee_{i=1}^{n}(\exists \overrightarrow{u}_i \cdot \hat{\sigma}_i \mid C_i))}{spred_{old} \ -\text{isp}\rightarrow \ spred_{new}} \ [\textsc{isp}\text{-}[\textsc{spec}\text{-}\textsc{body}]]$$

$$\frac{spred_{old} \ = \ (\mathtt{p}(\overrightarrow{v}) \equiv \bigvee_{i=1}^{n}(\exists \overrightarrow{u}_i \cdot \hat{\sigma}_i \mid C_i)) \qquad \rho = [inv_p(\overrightarrow{\forall}) \mapsto \mathtt{fix}(\bigvee_{i=1}^{n} \exists \overrightarrow{u}_i \cdot C_i)] \quad \mathcal{I} = \{(L \rightarrow hull(\bigvee_{i \in L} \exists \overrightarrow{u}_i \cdot \rho C_i) \mid \emptyset \subset L \subset \{1..n\}\} \cup \{\{1..n\} \rightarrow \rho(inv_p(\overrightarrow{\forall}))\} \quad spred_{new} \ = \ (\mathtt{p}(\overrightarrow{v}) \equiv \bigvee_{i=1}^{n}(\exists \overrightarrow{u}_i \cdot \hat{\sigma}_i \mid \rho C_i);\ \mathcal{I})}{spred_{old} \ -\text{isp}\rightarrow \ spred_{new}} \ [\textsc{isp}\text{-}[\textsc{build}\text{-}\textsc{inv}\text{-}\textsc{family}]]$$

$$\frac{spred_{old} \ = \ (\mathtt{p}(\overrightarrow{v}) \equiv \bigvee_{i=1}^{n}(\exists \overrightarrow{u}_i \cdot \hat{\sigma}_i \mid C_i);\ \mathcal{I}) \qquad G = \bigcup_{i=1}^{n} closure(\mathcal{I}(\{i\})) \quad \mathcal{R} = \bigcup_{\alpha \in G}\{\alpha \leftarrow \{i \mid 1 \leq i \leq n \ \wedge \ \mathcal{I}(\{i\}) \Longrightarrow \alpha\}\} \quad \forall i \in \{1,..,n\} \cdot \hat{\sigma}_i \mid C_i \ -\text{sf}\rightarrow^* \hat{\sigma}_{i,2} \mid C_{i,2} \quad spred_{new} \ = \ (\mathtt{p}(\overrightarrow{v}) \equiv \bigvee_{i=1}^{n}(\exists \overrightarrow{u}_i \cdot \hat{\sigma}_{i,2} \mid C_{i,2});\ \mathcal{I};\ \mathcal{R})}{spred_{old} \ -\text{isp}\rightarrow \ spred_{new}} \ [\textsc{isp}\text{-}[\textsc{build}\text{-}\textsc{prune}\text{-}\textsc{cond}]]$$

where $-\text{sf}\rightarrow^*$ *is the transitive closure of* $-\text{sf}\rightarrow$; *and* $\mathcal{I}(\{i\}) = \pi_i$, *given* $(\{i\} \rightarrow \pi_i) \in \mathcal{I}$.

Figure 5.4: Inference Rules for Specializable Predicates

The formal rules for inferring each specializable predicate are given in Figure 5.4. The rule $[\textsc{init}\text{-}[\textsc{multi}\text{-}\textsc{spec}]]$ converts an unannotated formula into its corresponding special-

ized form. It achieves this by an initialization step via the $-\text{if}\rightarrow$ relation given in Figure 5.5, followed by a multi-step specialization using $-\text{sf}\rightarrow^*$, without resorting to case specialization (that would otherwise result in an outer disjunctive formula). This essentially applies a transitive closure of $-\text{sf}\rightarrow$ until no further reduction is possible.

The rule [ISP−[SPEC−BODY]] converts the body of each predicate definition into its specialized form. For each recursive invocation, it will initially assume a symbolic invariant, named $inv_{\mathtt{p}}(\overrightarrow{v})$, without providing any pruning conditions. This immediately puts each recursive predicate instance in the fully-specialized form.

After the body of the predicate definition has been specialized, we can proceed to build a constraint abstraction for its predicate's invariant, denoted by $inv_{\mathtt{p}}(\overrightarrow{v})$, in the [ISP−[BUILD−INV−FAMILY]] rule. For example, we may denote the invariant of predicate $x{::}\mathtt{lseg}\langle p, n\rangle$ symbolically using $inv_{\mathtt{lseg}}(\mathtt{x},\mathtt{p},\mathtt{n})$, before building the following recursive constraint abstraction:

$$inv_{\mathtt{lseg}}(\mathtt{x},\mathtt{p},\mathtt{n}) \equiv \mathtt{x}{=}\mathtt{p}\wedge\mathtt{n}{=}0 \vee \exists \mathtt{q},\mathtt{m}\cdot \mathtt{x}{\neq}\mathtt{null}\wedge\mathtt{n}{=}\mathtt{m}{+}1\wedge inv_{\mathtt{lseg}}(\mathtt{q},\mathtt{p},\mathtt{m})$$

If we apply a classical fix-point analysis to the above abstraction, we would obtain a closed-form formula as the invariant of the lseg predicate, that is $inv_{\mathtt{lseg}}(\mathtt{x},\mathtt{p},\mathtt{n}) = \mathtt{n}{\geq}0$. With this predicate invariant, we can now build a family of invariants for each proper subset L of disjuncts, namely $0{\subset}L{\subset}\{1..\mathtt{n}\}$. This is done with the help of the convex hull approximation. The size of this family of invariants is exponential to the number of disjuncts. While this is not a problem for predicates with a small number of disjuncts, it could pose a problem for unusual predicates with a large number of disjuncts. To circumvent this problem, we could employ either a lazy construction technique or a more aggressive approximation to cut down on the number of invariants generated. For simplicity, this aspect is not considered in the present work.

Our last step is to build a set of pruning conditions for the disjunctive predicates using the [ISP−[BUILD−PRUNE−COND]] rule. This is currently achieved by applying a closure operation over the invariant $\mathcal{I}(\{\mathtt{i}\})$ for each of the disjuncts. To obtain a more complete set of pruning conditions, we are expected to generate a set of strong atomic constraints for each of the closure operations. For example, if we currently have a formula $\mathtt{a}{>}\mathtt{b} \wedge \mathtt{b}{>}\mathtt{c}$, a strong closure operation over this formula may yield the following set of atomic constraints $\{\mathtt{a}{>}\mathtt{b},\ \mathtt{b}{>}\mathtt{c},\ \mathtt{a}{>}\mathtt{c}{+}\mathtt{1}\}$ as pruning conditions and omit weaker atomic constraints, such as $\mathtt{a}{>}\mathtt{c}$.

$$\frac{}{emp \ -\text{ih}\rightarrow\ emp \mid \mathtt{true}}\ [\textsc{init}-[\textsc{emp}]] \qquad \frac{}{x{::}p\langle\overrightarrow{v}\rangle \ -\text{ih}\rightarrow\ x{::}p\langle\overrightarrow{v}\rangle \mid x{\neq}null}\ [\textsc{init}-[\textsc{cell}]]$$

$$\frac{p(\overrightarrow{v}) \equiv (\bigvee_{i=1}^{n}(\exists\overrightarrow{u}_i{\cdot}\hat{\sigma}_i \mid C_i));\mathcal{I};\mathcal{R} \quad C = \mathcal{I}nv(v{::}p\langle\overrightarrow{v}\rangle,\{1..n\})}{v{::}p\langle\overrightarrow{v}\rangle \ -\text{ih}\rightarrow\ v{::}p\langle\overrightarrow{v}\rangle@\{1..n\}\#\mathcal{R} \mid C}\ [\textsc{init}-[\textsc{pred}]]$$

$$\frac{p(\overrightarrow{v}) \equiv (\bigvee_{i=1}^{n}(\exists\overrightarrow{u}_i{\cdot}\hat{\sigma}_i \mid C_i));\mathcal{I};\mathcal{R} \quad (L{\rightarrow}C) \in \mathcal{I}}{\mathcal{I}nv(v{::}p\langle\overrightarrow{v}\rangle,L) = C}\ [\textsc{inv}-\textsc{def}]$$

$$\frac{\forall i{\in}\{1,2\}{\cdot}\kappa_i \ -\text{ih}\rightarrow\ \hat{\kappa}_i \mid C_i}{\kappa_1{*}\kappa_2 \ -\text{ih}\rightarrow\ \hat{\kappa}_1{*}\hat{\kappa}_2 \mid C_1{\wedge}C_2}\ [\textsc{init}-[\textsc{heap}]] \qquad \frac{\kappa \ -\text{ih}\rightarrow\ \hat{\kappa} \mid C}{\kappa{\wedge}\pi \ -\text{if}\rightarrow\ \hat{\kappa}{\wedge}\pi \mid C{\wedge}\pi}\ [\textsc{init}-[\textsc{formula}]]$$

Figure 5.5: Initialization for Specialization

Definition 5.5.1 (Sound invariant and sound pruning condition)**.** *Given a predicate definition* $p(\overrightarrow{v}){\equiv}\bigvee_{i=1}^{n}\mathtt{D_i};\ \mathcal{I};\ \mathcal{R}$*:*

(1) an invariant $L{\rightarrow}\pi$ *is said to be sound w.r.t. the predicate* p *if*

(1.a) $\emptyset \subset L \subseteq \{1,..,n\}$*, and*

(1.b) $v{::}p\langle\overrightarrow{v}\rangle@L\#_{\_} \models \pi$*.*

(2) a family of invariants $\mathcal{I}$ *is sound if every invariant from* $\mathcal{I}$ *is sound and the domain of* $\mathcal{I}$ *is the set of all non-empty subsets of* $\{1,..,n\}$*;*

(3) a pruning condition $(\alpha{\leftarrow}L)$ *is sound w.r.t. the predicate* p *if*

(3.a) $\emptyset \subset L \subseteq \{1,..,n\}$*,*

(3.b) $vars(\alpha) \subseteq \{\overrightarrow{v}\}$, *and*

(3.c) $\forall i \in L \cdot \mathtt{D_i} \models \alpha$.

(4) a set of pruning conditions R is sound if every pruning condition in R is sound w.r.t. the predicate p.

Property 5. *For each predicate* $p(\overrightarrow{v})$*, the family of invariants and the set of pruning conditions derived for p by our inference process are sound, assuming the fixpoint analysis and the hulling operation used by the inference are sound.*

Proof Suppose the inference process produces the annotated predicate definition:

$$p(\overrightarrow{v}) \equiv \bigvee_{i=1}^{n} \mathtt{D_i};\ \mathcal{I};\ \mathcal{R},\ \ where\ \ \mathtt{D_i} = \exists \overrightarrow{w_i} \cdot \hat{\sigma}_i | C_i$$

(1) To prove the soundness of $\mathcal{I}$, by Definition 5.5.1, we need to show the soundness of each invariant in $\mathcal{I}$. Given $(L \rightarrow \pi_L) \in \mathcal{I}$, by the same definition, it is sufficient to show that:

- (1.a) $\emptyset \subset L \subseteq \{1, .., n\}$

  obvious from the [ISP–[BUILD–INV–FAMILY]] rule for constructing $\mathcal{I}$

- (1.b) $v{::}p\langle\overrightarrow{v}\rangle@L\#_{\_} \models \pi_L$.

  This follows from an induction on $|L|$.

  Base case ($|L|$=1): Let $L = \{i\}$. $v{::}p\langle\overrightarrow{v}\rangle@L\#_{\_} = \mathtt{D_i}$ and $\mathcal{I}(\{i\}) = \rho C_i$. (1.b) clearly holds since $\mathtt{D_i} \models C_i$ and $C_i \models \rho C_i$. The latter comes from the fact that $inv_p(\overrightarrow{v}) \Longrightarrow \mathtt{fix}(inv_p(\overrightarrow{v}))$.

  Inductive case ($|L|$=k+1): there exist some L_0 and some i such that L=$L_0 \cup \{i\}$ and $|L_0| = k$. By induction hypothesis, we have $v{::}p\langle\overrightarrow{v}\rangle@L_0\#_{\_} \models \mathcal{I}(L_0)$. We also have $v{::}p\langle\overrightarrow{v}\rangle@\{i\}\#_{\_} \models \mathcal{I}(\{i\})$ from base case. Therefore, $(v{::}p\langle\overrightarrow{v}\rangle@L\#_{\_} \ \vee\ v{::}p\langle\overrightarrow{v}\rangle@L_0\#_{\_}) \models (\mathcal{I}(L_0) \vee \mathcal{I}(\{i\}))$, from which (1.b) follows (in addition to the soundness of the hulling operation).

(2) To prove the soundness of $\mathcal{R}$, by Definition 5.5.1, we need to show the soundness of each pruning condition in $\mathcal{R}$. Given $(\alpha \leftarrow L) \in \mathcal{R}$, by the same definition, we need to show:

- $\emptyset \subset L \subseteq \{1, .., n\}$, follows from [ISP–[BUILD–PRUN–COND]]
- $vars(\alpha) \subseteq \{\overrightarrow{v}\}$, follows from [ISP–[BUILD–PRUN–COND]]
- $\forall i \in L \cdot \exists \overrightarrow{w} \cdot \mathtt{D_i} \models \alpha$, follows from the soundness of $\mathcal{I}(\{i\})$ (which gives $\mathtt{D_i} \models \mathcal{I}(\{i\})$) and that $\mathcal{I}(\{i\} \implies \alpha$(from the aforementioned rule)

□

5.6 Specialization for Program Verification

We have designed our specialization calculus originally to improve an existing separation logic-based program verification system. To support predicate specialization, we must provide a set of translation rules that would normalize and specialize each given formula. A core set of the rules is given in Fig 5.6. The first rule ([NORM–[PURE–AND]]) shows how a pure term is added to the context of its annotated formula, prior to its specialization via the $-\text{sf}\rightarrow^*$ transition. The second rule ([NORM–[SEP–AND]]) shows how two heap terms are spatially conjoined together, with a conjunctive strengthening of their combined contexts. Specialization is also applied to eliminate infeasible disjuncts, where possible. The third rule ([NORM–[UNFOLDING]]) can be used to unfold a specialized predicate by revealing only the feasible disjuncts. This unfolding rule may be invoked by our entailment procedure to help materialize a predicate instance. The disjunctive formula itself is distributed and recursively normalized by the rule ([NORM–[DIST–OR]]).

The final step towards our goal consists in linking the specializer into the verification logic we proposed in the previous chapters. One of the first changes we made to our verification

$$\frac{[\textsc{norm}\text{-}[\textsc{pure}\text{-}\textsc{and}]]\quad \hat{\kappa} \mid C \wedge \pi_2 \multimap \hat{\kappa}_2 \mid C_2}{(\hat{\kappa} \wedge \pi \mid C) \wedge \pi_2 \multimap \hat{\kappa}_2 \wedge (\pi \wedge \pi_2) \mid C_2}$$

$$\frac{[\textsc{norm}\text{-}[\textsc{sep}\text{-}\textsc{and}]]\quad (\hat{\kappa}_1 * \hat{\kappa}_2) \mid C_1 \wedge C_2 \multimap \hat{\kappa}_3 \mid C_3}{(\hat{\kappa}_1 \wedge \pi_1 \mid C_1) * (\hat{\kappa}_2 \wedge \pi_2 \mid C_2) \multimap \hat{\kappa}_3 \wedge (\pi_1 \wedge \pi_2) \mid C_3}$$

$$\frac{[\textsc{norm}\text{-}[\textsc{unfolding}]]\quad \mathtt{p}(\overrightarrow{v}) \equiv \bigvee_{i=1}^{n} (\exists \overrightarrow{u_i} \cdot \hat{\sigma}_i \mid C_i)\quad J \equiv (\bigvee_{i \in L} \exists \overrightarrow{u_i} \cdot \hat{\sigma}_i) \mid C_i \qquad J * (\hat{\sigma} \mid C) \multimap \hat{\Phi}}{((v{::}p\langle \overrightarrow{v} \rangle @ L \# R) * \hat{\sigma}) \mid C \multimap \hat{\Phi}}$$

$$\frac{[\textsc{norm}\text{-}[\textsc{dist}\text{-}\textsc{or}]]\quad \forall i \in \{1..n\} \cdot (\hat{\sigma}_i * \hat{\sigma}) \mid C \wedge C_i \multimap \hat{\sigma}_{i,2} \mid C_{i,2}}{(\bigvee_{i=1}^{n} \exists \overrightarrow{u_i} \cdot (\hat{\sigma}_i \mid C_i)) * (\hat{\sigma} \mid C) \multimap \bigvee_{i=1}^{n} \exists \overrightarrow{u_i} \cdot (\hat{\sigma}_{i,2} \mid C_{i,2})}$$

Figure 5.6: Normalizing Specialized Separation Logic

system was to transform pre/post specifications into specialized form, as shown below:

$$\frac{\Phi_{pr} \multimap \hat{\Phi}_{pr} \qquad \Phi_{po} \multimap \hat{\Phi}_{po}}{\begin{array}{c} \mathit{mn}\ ((t\ v)^*; (\mathtt{ref}\ t\ v)^*)\ \mathit{requires}\ \Phi_{pr}\ \mathit{ensures}\ \Phi_{po} \cdots \\ \Longrightarrow\ \mathit{mn}\ ((t\ v)^*; (\mathtt{ref}\ t\ v)^*)\ \mathit{requires}\ \hat{\Phi}_{pr}\ \mathit{ensures}\ \hat{\Phi}_{po} \cdots \end{array}}$$

This pre-processing step has several benefits. Firstly, it is able to strengthen the contexts of our specifications. Secondly, the specialized formula is reusable at each of its caller's site, reducing computation costs. Lastly, we may eliminate constraints from the specification that are already implied by the specialized predicates. This can help reduce some proof obligations needed. As an example of these benefits, consider a specification formula, $x{::}\mathtt{dll}^{\mathtt{v}}\langle p, n, S\rangle \wedge \mathtt{n}{>}2 \wedge \mathtt{x} \neq \mathtt{null}$.

Applying our normalization and specialization to this formula would initially yield:

$$x{::}\texttt{dll}^{\texttt{v}}\langle p, n, S\rangle@\{2\}{\wedge}\texttt{n}{>}2{\wedge}\texttt{x}{\neq}\texttt{null} \mid \texttt{n}{>}0{\wedge}\texttt{x}{\neq}\texttt{null}{\wedge}\texttt{S}{\neq}\{\}$$

We can remove the constraint $\texttt{x}{\neq}\texttt{null}$ as it is already implied by invariant of the specialized predicate $x{::}\texttt{dll}^{\texttt{v}}\langle p, n, S\rangle@\{2\}$,thus obtaining a more compact formula:

$$x{::}\texttt{dll}^{\texttt{v}}\langle p, n, S\rangle@\{2\}{\wedge}\texttt{n}{>}2 \mid \texttt{n}{>}0{\wedge}\texttt{x}{\neq}\texttt{null}{\wedge}\texttt{S}{\neq}\{\}.$$

With the specialized pre/post specifications, we can now present Hoare-style forward verification rules $\vdash \{\hat{\Phi}_1\}\, e\, \{\hat{\Phi}_2\}$, where we expect $\hat{\Phi}_1$ to be given before computing $\hat{\Phi}_2$. We highlight some verification rules in Figure 5.7. They are used to track heap states as accurately as possible with path-sensitivity ($[\underline{\textbf{FV}{-}[\textbf{IF}]}]$), flow-sensitivity and context-sensitivity ($[\underline{\textbf{FV}{-}[\textbf{CALL}]}]$). For each call site, $[\underline{\textbf{FV}{-}[\textbf{CALL}]}]$ ensures that its method's precondition is satisfied, while its postcondition is spatially conjoined with the prevailing program state. In all these rules, we always normalize/specialize after each monotonic change to the program state. This allows the specialized formula to be shared on a timely basis. Our new entailment checking procedure is optimized to take advantage of this specialized form.

5.7 Improved Specialization

Thus far, we have described the core concepts behind predicate specialization, and how it enables a more optimal representation of program states for static reasoning. In this section, we explore two key optimization techniques that can make predicate specialization a practical tool for program reasoning. The first technique, called *memoization*, is based on a trade-off between re-proving a result with respect to a given context, or recalling its stored result from a memoized set. The second technique, called *incremental pruning*, relies on incremental computation that is made possible through our approach. Both optimizations are detailed in

$$\frac{\hat{\Phi}\wedge v' -\text{norm}\rightarrow \hat{\Phi}_1 \quad \vdash \{\hat{\Phi}_1\}\, e_1\, \{\hat{\Phi}_3\} \qquad \hat{\Phi}\wedge\neg v' -\text{norm}\rightarrow \hat{\Phi}_2 \quad \vdash \{\hat{\Phi}_2\}\, e_2\, \{\hat{\Phi}_4\}}{\vdash \{\hat{\Phi}\}\ \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2\ \{\hat{\Phi}_3\vee\hat{\Phi}_4\}}\ [\textbf{FV}-[\textbf{IF}]]$$

$$\frac{t_0\ m(t_1\ v_1,..,t_n\ v_n)\ \textit{requires}\ \hat{\Phi}_{pr}\ \textit{ensures}\ \hat{\Phi}_{po}\ \{..\} \quad \rho{=}[v_i'/v_i] \quad \hat{\Phi} -\text{norm}\rightarrow \hat{\Phi}_0 \quad \hat{\Phi}_0\vdash \rho\hat{\Phi}_{pr} * \hat{\Phi}_1 \quad W=\{v_1,..,v_n\} \quad (\hat{\Phi}_1 *_W \rho\hat{\Phi}_{po}) -\text{norm}\rightarrow \hat{\Phi}_2}{\vdash \{\hat{\Phi}\}\, m(v_1..v_n)\, \{\hat{\Phi}_2\}}\ [\textbf{FV}-[\textbf{CALL}]]$$

Figure 5.7: Some Verification Rules

Figure 5.8 and shall be discussed in the remainder of this section.

5.7.1 Memoization

As the cost for invoking automated provers accounts for the bulk of the time taken by program verification system, in general, we can improve our system by minimising the number of invocations to these provers. One simple way to achieve this is to maintain two memoization sets for each context C using a so-called *syntactic context*: $\mathbb{S} = (I, F)$. The first element I denotes a set of atomic constraints that are implied by the context C, that is $\forall\alpha{\in}I \cdot C \Longrightarrow \alpha$. The second element F denotes a set of atomic constraints that are in contradiction with context C, that is $\forall\alpha{\in}F \cdot C \Longrightarrow \neg\alpha$. With these two memoization sets, we may now approximate the check $C \Longrightarrow \alpha$ by a membership test $\alpha{\in}I$, and also approximate the contradiction check $C \Longrightarrow \neg\alpha$ by its corresponding membership test $\alpha{\in}F$. These two membership tests are only sound approximations of the corresponding implication checks. In case both membership tests fail for a given pruning condition α, we could turn to automated provers (as a last resort) to help determine $C \Longrightarrow \neg\alpha$.

Furthermore, the syntactic context $\mathbb{S}$ currently keeps track of both strong and weak atomic

constraints for memoization purposes, whereas for implication checking we only need to use the stronger constraints. To support this feature of using *only* the stronger atomic constraints, where possible, our system keeps two implication sets, namely $I = (I_S, I_W)$. The I_S set is expected to capture a set of stronger atomic constraints, while I_W set can keep the weaker atomic constraints that are already implied by I_S.

As an example of how these two implication sets are maintained, consider a simple example $x{::}\texttt{dll}^{\texttt{v}}\langle p, n, S\rangle \wedge \texttt{x}{\neq}\texttt{null}$. In the beginning, our specialization mechanism will add the predicate invariant $\texttt{n}{\geq}\texttt{0}$ to its current context. After specializing the predicate to $x{::}\texttt{dll}^{\texttt{v}}\langle p, n, S\rangle@\{2\}$, our mechanism can add a stronger atomic constraint $\texttt{n}{>}\texttt{0}$ to the context. This specialization step would cause the earlier atomic constraint $\texttt{n}{\geq}\texttt{0}$ to be marked as redundant. When this happens, we keep each new stronger atomic constraint (computed as π_o in rule [OSF–[ADD–INV]]) in the I_S set, while each weaker atomic constraint (computed as π_i in the same rule) will be moved into the I_W set. Both the memoization sets, I_S and I_W, are for membership tests to approximate implication, but only the stronger set I_S is used by the automated prover. This allows us to use a more efficient *Strong-Imply*$(\mathcal{S}) \Longrightarrow \neg\alpha$ instead of *Imply*$(\mathcal{S}) \Longrightarrow \neg\alpha$ where *Strong-Imply*$(\mathcal{S})$ denotes $\bigwedge_{\alpha\in I_S} \alpha$, while *Imply*$(\mathcal{S})$ denotes $\bigwedge_{\alpha\in(I_S\cup I_W)} \alpha$.

5.7.2 Incremental Pruning

To support the early elimination of infeasible states, our calculus is designed to prove if each atomic constraint α contradicts a given context C by the implication $C \Longrightarrow \neg\alpha$. Furthermore, the context C is allowed to evolve to a monotonically stronger context C_1, such that $C_1 \Longrightarrow C$. Hence, if indeed $C \Longrightarrow \neg\alpha$, we can be assured that $C_1 \Longrightarrow \neg\alpha$ will also hold. This monotonic change of the context is the basis of the *memoization* optimization that allows previous outcomes of implications and contradictions to be reused.

Another advantage of the above approach is that we can easily *slice out* relevant constraints from (a satisfiable) C that are needed to prove an atomic constraint α. This slicing technique

is allowed by our approach because we detect infeasible branches by proving only one atomic pruning constraint at a time. For example, consider a context $\texttt{x}{\neq}\texttt{null} \wedge \texttt{n}{>}\texttt{0} \wedge \texttt{S}{\neq}\{\}$. If we need to prove its contradiction with $\texttt{n}{=}\texttt{0}$, a naive solution is to use $(\texttt{x}{\neq}\texttt{null} \wedge \texttt{n}{>}\texttt{0} \wedge \texttt{S}{\neq}\{\}) \Longrightarrow \neg(\texttt{n}{=}\texttt{0})$. A better solution is to slice out just the constraint $\texttt{n}{>}\texttt{0}$, and then proceed to prove the contradiction using $\texttt{n}{>}\texttt{0} \Longrightarrow \neg(\texttt{n}{=}\texttt{0})$. Slicing out constraints from the context is an integral part of incremental pruning and is enabled by our decision to use atomic constraints as pruning conditions, and relying on monotonically stronger contexts.

To support this optimization, we propose to partition each context into sets of connected constraints. Two atomic constraints in a context C are said to be *connected* if they satisfy the following relation.

$$\texttt{connected}(\alpha_1, \alpha_2) \text{ :- } (\mathit{vars}(\alpha_1) \cap \mathit{vars}(\alpha_2)) \neq \{\}$$
$$\texttt{connected}(\alpha_1, \alpha_2) \text{ :- } \exists \alpha \in \texttt{C} \cdot \texttt{connected}(\alpha_1, \alpha) \wedge \texttt{connected}(\alpha_2, \alpha)$$

With a continuous partitioning of connected constraints for the contexts, our system can easily slice out a set of constraints (from the context) that are connected to each pruning condition. This slicing mechanism has been shown to reduce analysis time significantly in our experiments.

One other important optimization that is possible in our calculus is to take advantage of provers with incremental solving capability, particularly from SMT provers like Yices [32] and Alt-Ergo [12]. In these provers, it is possible to incrementally add new constraints into an existing context before checking for satisfiability of the combined formula, and later pop out constraints based on a stack-like discipline. The pre-processing done for each context is always kept when the prover returns to a prior state. In the case of our specialization calculus, the strong constraints from our earlier context would have been added by incremental provers. To check if each new pruning condition α contradicts with the current context, we need only push constraint α into the prover before checking for a contradiction. This same α constraint can later be popped out, before we explore the next pruning conditions in turn. The context may

also be monotonically strengthened with new constraints, but this has to be checked for contradiction too, since a `false` context should be detected explicitly. Furthermore, we may use a set of simultaneous incremental provers, one for each set of connected constraints from the context. This approach yields an even better performance gain for our specialization calculus.

In the improved specialization, we would attempt to minimize on the number of calls to the prover. This is achieved by memoising on all atomic constraints that are either implied by or contradicted with the current context. We represent each current context $\mathcal{S}$ by (I, F) where I is the set of atomic constraints that are implied by the context, while F is the set of atomic constraints that are in contradiction with the context. That is $\forall \alpha \in I \cdot \mathcal{S} \Longrightarrow \alpha$ and $\forall \alpha \in F \cdot \mathcal{S} \Longrightarrow \neg\alpha$.

Furthermore, in the implied set I, we distinguish between strong and weak atomic constraints. A constraint from I is said to be strong if it is not implied by a different constraint in I. In theory, we can compute the set of strong constraint by the following operator

$$\textit{strong-imply}(I) =_{df} \{\alpha_1 | \alpha_1 \in I, \neg(\exists \alpha_2 \in I \cdot \alpha_1 \neq \alpha_2 \wedge \alpha_2 \Longrightarrow \alpha_1)\}$$

As this computation can be expensive, we shall instead use a marker to identify each atomic constraint in I that is known to be redundant. The operator to select stronger atomic constraints will simply omit those constraints that are marked as redundant.

5.8 Experiments

We have built a prototype system for our specialization calculus inside HIP [75].

Figure 5.9 summarizes a suite of programs tested which included the 17 small programs (comprised of various methods on singly, doubly, sorted and circular linked lists, selection-sort, insertion-sort and methods for handling heaps, and perfect trees). Due to similar outcomes, we present the average of the performances for these 17 programs. We also experimented with a set of medium-sized programs that included complex shapes and invariants, to support full

$$\frac{L{\subseteq}L_0 \vee \alpha{\in}\mathit{Imply}(\mathcal{S})}{v{::}p^v\langle\overrightarrow{v}\rangle@L\#R \mid \mathcal{S} \xrightarrow{\text{osp}} v{::}p^v\langle\overrightarrow{v}\rangle@L_2\#R_2 \mid \mathcal{S}_2}\ [\textbf{OSP-[FILTER]}]$$
$$\frac{}{v{::}p^v\langle\overrightarrow{v}\rangle@L\#(\{\alpha{\leftarrow}L_0\}\cup R) \mid \mathcal{S} \xrightarrow{\text{osp}} v{::}p^v\langle\overrightarrow{v}\rangle@L_2\#R_2 \mid \mathcal{S}_2}$$

$$\frac{\alpha \in \mathit{FailSet}(\mathcal{S})}{v{::}p^v\langle\overrightarrow{v}\rangle@(L{-}L_0)\#R \mid \mathcal{S} \xrightarrow{\text{osp}} v{::}p^v\langle\overrightarrow{v}\rangle@L_2\#R_2 \mid \mathcal{S}_2}\ [\textbf{OSP-[PRUNE1]}]$$
$$\frac{}{v{::}p^v\langle\overrightarrow{v}\rangle@L\#(\{\alpha{\leftarrow}L_0\}\cup R) \mid \mathcal{S} \xrightarrow{\text{osp}} v{::}p^v\langle\overrightarrow{v}\rangle@L_2\#R_2 \mid \mathcal{S}_2}$$

$$\frac{\vdash \mathit{Strong\text{-}Imply}(\mathcal{S}) \Longrightarrow \neg\alpha}{v{::}p^v\langle\overrightarrow{v}\rangle@(L{-}L_0)\#R \mid \mathcal{S}{\wedge}\neg\alpha \xrightarrow{\text{osp}} v{::}p^v\langle\overrightarrow{v}\rangle@L_2\#R_2 \mid \mathcal{S}_2}\ [\textbf{OSP-[PRUNE2}]$$
$$\frac{}{v{::}p^v\langle\overrightarrow{v}\rangle@L\#(\{\alpha{\leftarrow}L_0\}\cup R) \mid \mathcal{S} \xrightarrow{\text{osp}} v{::}p^v\langle\overrightarrow{v}\rangle@L_2\#R_2 \mid \mathcal{S}_2}$$

$$\frac{v{::}p^v\langle\overrightarrow{v}\rangle@L\#R \mid \mathcal{S} \xrightarrow{\text{osp}} v{::}p^v\langle\overrightarrow{v}\rangle@L\#R_2 \mid \mathcal{S}_1 \quad \kappa \mid \mathcal{S}_1 \xrightarrow{\text{osf}} \hat{\kappa} \mid \mathcal{S}_2}{v{::}p^v\langle\overrightarrow{v}\rangle@L\#R*\kappa \mid \mathcal{S} \xrightarrow{\text{osf}} v{::}p^v\langle\overrightarrow{v}\rangle@L\#R_2*\hat{\kappa} \mid \mathcal{S}_2}\ [\textbf{OSF-[HEAP]}]$$

$$\frac{L_2{\subset}L \quad v{::}p^v\langle\overrightarrow{v}\rangle@L\#R \mid \mathcal{S} \xrightarrow{\text{osp}} v{::}p^v\langle\overrightarrow{v}\rangle@L_2\#R_2 \mid \mathcal{S}_1 \quad \pi_i = \mathit{Inv}(p(\overrightarrow{v}),L_2) - \mathit{Inv}(p(\overrightarrow{v}),L) \quad \pi_o = \mathit{Inv}(p(\overrightarrow{v}),L) - \mathit{Inv}(p(\overrightarrow{v}),L_2) \quad I_2 = \mathit{ImplySet}(\mathcal{S}_1){\ominus}\pi_o \uplus \pi_i \quad F_2 = \mathit{FailSet}(\mathcal{S}_1) \cup \{\neg\alpha \mid \alpha \in \pi_i\} \quad \kappa \mid (I_2,F_2) \xrightarrow{\text{osf}} \hat{\kappa} \mid \mathcal{S}_3}{v{::}p^v\langle\overrightarrow{v}\rangle@L\#R*\kappa \mid \mathcal{S} \xrightarrow{\text{osf}} v{::}p^v\langle\overrightarrow{v}\rangle@L_2\#R_2*\hat{\kappa} \mid \mathcal{S}_3}\ [\textbf{OSF-[ADD-INV]}]$$

$$\mathit{For}\ \mathcal{S} = ((I_S,I_W),F),\ \mathit{Imply}(\mathcal{S}) =_{df} \bigwedge_{\alpha\in(I_S\cup I_W)} \alpha;$$

$$\mathit{ImplySet}(\mathcal{S}){=}_{df}(I_S,I_W);\ \mathit{FailSet}(\mathcal{S}){=}_{df}F;$$

$$(I_S,I_W){\ominus}\pi = (I_S{-}\pi, I_W{\cup}\pi);\ (I_S,I_W){\uplus}\pi = (I_S{\cup}\pi, I_W);$$

$$\mathit{For}\ \pi = \bigwedge_{i=1}^{m}\alpha_i,\ I{-}\pi{=}_{df}I{-}\{\alpha_1..\alpha_m\},\ I{\cup}\pi{=}_{df}I{\cup}\{\alpha_1..\alpha_m\},$$

$$((I_S,I_W),F){\wedge}\pi{=}_{df}((I_S{\cup}\{\alpha_1,..,\alpha_m\},I_W),F{\cup}\{\neg\alpha_1,..,\neg\alpha_m\})$$

Figure 5.8: Improved Specialization

Progs (specified props)	LOC	HIP	HIP+Spec
		Time(s)	Time(s)
17 small progs (size)	87	0.86	0.80
Bubble sort (size, sets)	80	2.20	2.23
Quick sort (size, sets)	115	2.43	2.13
Merge sort (size, sets)	128	3.10	2.15
Complete(size,minh)	137	5.01	2.94
AVL (height, size, bal)	160	64.1	16.4
Heap Trees(size, maxel)	208	14.9	4.62
AVL (height, size)	340	27.5	13.1
AVL (height, size, sets)	500	657	60.7
Red Black (size, height)	630	25.2	15.6

Progs (specified props)	HIP			HIP+Spec		
	Count	Disj	Size	Count	Disj	Size
17 small progs (size)	229	1.63	12.39	612	1.13	2.97
Bubble sort (size, sets)	296	2.13	18.18	876	1.09	2.79
Quick sort (size, sets)	255	3.29	17.97	771	1.27	3.08
Merge sort (size, sets)	286	2.04	16.74	1079	1.07	2.99
Complete(size,minh)	463	3.52	43.75	2134	1.11	10.10
AVL (height, size, bal)	764	2.90	85.02	6451	1.07	9.66
Heap Trees(size, maxel)	649	2.10	56.46	2392	1.02	8.68
AVL (height, size)	704	2.98	70.65	7078	1.09	10.74
AVL (height, size, sets)	1758	8.00	86.79	14662	1.91	10.11
Red Black (size, height)	2225	3.84	80.91	7697	1.01	3.79

Figure 5.9: Verification Times and Proof Statistics (Proof Counts, Avg Disjuncts, Avg Size)

functional correctness. We measured the verification times taken for the original HIP system, and also the enhanced system, called HIP+Spec, with predicate specialization. For the suite of simple programs, the verifier with specializer runs about 7% faster. For programs with more complex properties (with the exception of bubble sort), predicate specialization manages to reduce verification times by between 12% and 90%. These improvements were largely due to the presence of smaller formulae with fewer disjuncts, as captured in Figure 5.10. This graph compares the characteristics (e.g. average disjuncts, sizes and timings) of formula encountered by HIP+Spec, as a percentage relative to the same properties of the original HIP system. For example, the average number of disjuncts per proof encountered went down from 3.2 to 1.1; while the size of each proof (based on as the number of atomic formulae) also decreased from an average of 48.0 to 6.5. This speed-up was achieved despite a six fold increase in the proof

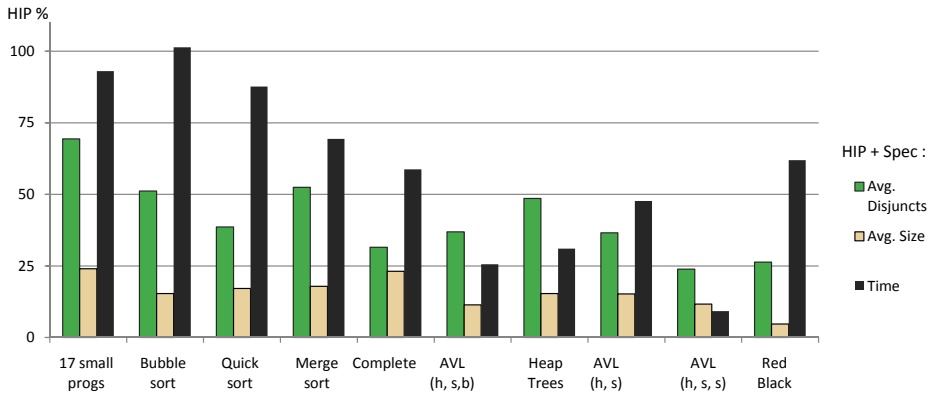


Figure 5.10: Characteristic (disjunct, size, timing) of HIP+Spec compared to the Original HIP

counts per program from 763 to 4375 used by the specialization and verification processes. We managed to achieve this improvement despite the overheads of a memoization mechanism and the time taken to infer annotations for specializable predicates. We believe this is due to smaller and simpler proof obligations generated with the help of our specialization process.

We also investigated the effects of various optimizations on the specialization mechanism. Memoizing implications and contradictions saves 3.47%, while memoizing each context for state change saves 22.3%. For incremental pruning, we have utilized the slicing mechanism which saves 48% on average. We have not yet exploited the incremental proving capability based on strengthening of contexts since our current solvers, Omega and MONA, do not support such a feature. These optimizations were measured separately, with no attempt made to study their correlation.

5.9 Barrier Logic with Specialization

Our novel specialization calculus is a great general purpose tool for improving verification efficiency in the presence of disjunctive predicates. In the previous section we presented several compelling results showcasing the efficiency improvements the specialization calculus can generate. However we believe it could also produce impressive results when applied to our

verification logic with exceptions and barriers.

In this section we will present the extensions necessary for applying the specialization calculus to barrier definitions. As support, we will also present the considerable improvements specialization generated in the performance of the HIP/SLEEK for the barrier examples.

We have already pointed out that barrier definitions and predicate definitions are similar in that they both describe properties associated with resources, heaps and barriers. However they differ in that predicate definitions describe a property of the current state while barrier definitions describe how a barrier call would behave given the program state. Therefore, while predicate specialization eagerly dismisses infeasible states thus decreasing the penalty of reasoning with unfolded predicates, barrier specialization will do an eager pruning of the infeasible transitions minimizing the barrier call penalty.

As a quick reminder, in §4.6 we have shown that barrier definitions can be encoded as a set of formula pairs :

$$[\mathsf{self}::]bname\langle\overrightarrow{v}\rangle \equiv \bigvee (\texttt{requires}\ \Phi_{pre}\ \texttt{ensures}\ \Phi_{post})$$

We have also shown that applying the barrier verification rule for a barrier call `barrier` b translates into locating the barrier node $b::bname^{\pi}\langle\overrightarrow{v}\rangle$ corresponding to b, retrieving the barrier definition for $bname$ and issuing a SLEEK entailment check with the barrier body as the consequent.

This behaviour is very similar to the unfolding of disjunctive predicates. And unfortunately induces a similar performance penalty. In the style of predicate definition annotations we enrich the barrier definitions with the set of pruning conditions $\mathcal{R}$:

$$[\mathsf{self}::]bname\langle\overrightarrow{v}\rangle \equiv \left(\bigvee (\texttt{requires}\ \hat{\Phi}_{pre}\ \texttt{ensures}\ \hat{\Phi}_{post})\right);\mathcal{R}$$

As with predicate definitions we use $\mathcal{R} = \{\alpha{\leftarrow}L\}$ to express the conditions in which a particular specification in the barrier definition can be eagerly discarded before actually executing

the entailment. In order to compute the $\mathcal{R}$ pruning conditions we apply a similar specialization inference relation to the one presented for predicates in §5.5.

$$get\_inv(\exists \overrightarrow{u} \cdot \hat{\kappa} \wedge \pi \mid C) = [inv_p(\overrightarrow{v}) \mapsto \texttt{fix}(\exists \overrightarrow{u} \cdot C_i)]C$$

$$get\_inv(\Phi_1 \vee \Phi_2) = hull(\ get\_inv(\Phi_1) \vee\ get\_inv(\Phi_2))$$

$$\frac{bdef_{old} \ = \ bname\langle \overrightarrow{v} \rangle \equiv \left(\bigvee_i (\texttt{requires}\ \Phi^i_{pre}\ \texttt{ensures}\ \Phi^i_{post}) \right) \quad \Phi^i_{pre} \text{ –msf→ } \hat{\Phi}^i_{pre} \quad G = \bigcup_{i=1}^{n} closure(get\_inv(\hat{\Phi}^i_{pre})) \quad \mathcal{R} = \bigcup_{\alpha \in G} \{\alpha \leftarrow \{i \mid 1 \leq i \leq n \ \wedge\ get\_inv(\hat{\Phi}^i_{pre})) \Longrightarrow \alpha\}\} \quad \Phi^i_{post} \text{ –msf→ } \hat{\Phi}^i_{post} \quad bdef_{new} \ = \ bname\langle \overrightarrow{v} \rangle \equiv \left(\bigvee_i (\texttt{requires}\ \hat{\Phi}^i_{pre}\ \texttt{ensures}\ \hat{\Phi}^i_{post}) \right); \mathcal{R}}{bdef_{old} \text{ –isp→ } bdef_{new}} \ [\underline{\textbf{ISP–[BUILD–PRUNE–COND]}}]$$

Figure 5.11: Inference Rules for Specializable Barriers

Note that the inference process takes into account only information from the preconditions, which is intuitive since the preconditions contain the link to the current state. In the process of building the barrier pruning conditions the invariants corresponding to the preconditions are built as an intermediary step, however, they are not exposed. In the case of barrier definitions, there are no invariant families, as barrier definitions describe the behaviour of an operation rather than the properties of a state.

Once the pruning conditions have been computed, the actual barrier instance specialization step varies little from the predicate specialization. The only change lies in the $[\underline{\textbf{SP–[PRUNE]}}]$ step and consists in not adding any invariant after a pruning has been triggered.

$$\frac{C \wedge \alpha \Longrightarrow \texttt{false} \quad (\alpha \leftarrow L_0) \in R \quad L \cap L_0 \neq \emptyset \quad L_2 = L - L_0 \quad C, L_2 \vdash R \text{ –filter→ } R_1}{v{::}bname^v\langle \overrightarrow{v} \rangle @ L \# R \mid C \text{ –sp→ } v{::}bname^v\langle \overrightarrow{v} \rangle @ L_2 \# R_1 \mid C \wedge C_1} \ [\underline{\textbf{SP–[PRUNE]}}]$$

We would like to point out that so far we have considered only linear arithmetic constraints as pruning conditions. However, as mentioned in the previous sections, it is possible to extend the domain simply by ensuring a hulling operation for the desired domain. In particular we are interested in also using fractional share constraints as pruning conditions. Although a proper hulling could be devised, in our investigations we have observed that even a trivial approach in which the hull returns the weakest result, $true$, works sufficiently well in practice and is strong enough to prune all the necessary branches for our examples. This not surprising, since most of the specifications could be pruned based on arithmetic constraints. For example, given the context: $b::bname \blacksquare \langle 2, x_1, x_2, y_1, y_2, i\rangle$ where $bname$ denotes the barrier definition from our running example in §4.2, the arithmetic conditions would eliminate all specifications not pertaining to the transition from state 2 to state 1 however this still leaves two specifications. By adding the fractional shares pruning conditions, the specializer is able to properly reduce to only one applicable specification (the second move). And thus cutting the search space from 8 possible specifications to only one.

Experimental validation. We run HIP and SLEEK with pruning support for barrier definitions on the benchmarks used in §4.6.6 to test the initial extension for barrier verification.

Unsurprisingly, the timings obtained for the SLEEK example without barriers are slightly higher, mainly due to the preprocessing overhead for predicate definitions: for the 54 entailments focused on the fractional share support, the timing increased from 2.1 seconds to 2.34. However, for the 279 entailments generated during the well-formedness checks of several barrier definitions, the timing dropped from 3.69 seconds to 2.45 seconds.

Furthermore, in Figure 5.12, we list the timings obtained with pruning versus the timings without pruning, for the HIP examples presented in §4.6.6 .

Note that for the HIP examples , specialization generated an average speedup of 12.5. Another important observation is that the highest gain were attained for the examples with more complex ownership transfer patterns and larger barrier definitions (e.g. multi-loops, Horner

Test	Threads	Mem cells	LOC code+ spec	No Prun (s)	With Prun (s)	Speedup
video	2	5	79	59.17	2.38	24.86
	3	7	128	215	6.06	35.59
video-weak	2	5	79	60.35	2.89	20.88
video-strong	2	5	79	61.12	2.99	20.44
MISD (without feedback)	2	1	62	0.81	0.69	1.17
	3	1	67	1.33	0.95	1.40
	4	1	73	1.97	1.25	1.58
	5	1	79	2.81	1.65	1.70
	6	1	87	3.88	2.12	1.83
MISD (with feedback)	2	2	74	3.22	1.64	1.96
	3	3	80	4.04	1.95	2.07
	4	4	129	9.28	3.21	2.89
	5	5	156	69.51	5.18	13.42
	6	6	187	177	7.9	22.41
multi-loops	3	4	160	12.82	3.8	3.37
	4	5	208	132	6.31	20.92
	5	6	255	364	9.76	37.3
pipeline	3	3	54	1.43	1.03	1.39
	4	4	70	2.52	1.25	2.02
	5	5	89	4.25	2.08	2.04
	6	6	106	7.08	2.88	2.46
Horner(v1)	3	7	116	181	5.22	34.67
Horner(v2)	3	4	111	8.04	2.6	3.09
	4	5	143	97	4.3	22.56
	5	6	176	279	7.41	37.65

Figure 5.12: Verification times for HIP with specialized barriers

algorithm). This is extremely important as these factors often lead to massive performance degradation.

Furthermore, note that introducing specialization to barrier reasoning makes it almost as efficient as traditional verification for reasonable complex sequential programs of comparable size (e.g. AVL or red black trees).

5.10 Summary

In this chapter we have proposed a specialization calculus for disjunctive predicates in a separation logic-based abstract domain. Our specialization calculus is proven sound and is terminating. It supports symbolic pruning of infeasible states within each predicate instance, under monotonic changes to the program context. We have designed inference techniques that can automatically derive all annotations required for each specializable predicate. We have also proposed various ways to optimize the specialization process, including memoization and incremental pruning. Initial experiments have confirmed speed gains from the deployment of our specialization mechanism to handle separation logic specifications in program verification. Nevertheless, our calculus is more general, and is useful for program reasoning over any abstract domain that supports disjunctive predicates. Apart from its efficiency benefits, predicate specialization has the added advantage of providing an *abstraction barrier* that would allow a modular approach to the program verification process. Similar to modular code development practices, the sets of pruning conditions and family of invariants, once computed, can be seen as an "interface" of the predicate, whereas the predicate definition can be seen as its "implementation". Program verification with specialized predicates requires only the use of the "interface". By not unfolding the predicate's definition, we have made the "implementation" opaque. In a modular setting, a large verification problem could be split up not only by code regions, but also by specifications, utilizing pre-specialized pre/post conditions and predicate interfaces between modules. This modular approach to verification is being enabled by predicate specialization.

Finally, we have shown how the specialization calculus naturally extends to barrier definitions and that it ensures that the barrier reasoning mechanism does not incur any considerable performance penalty.

Chapter 6

Comparative Remarks

Next, we will outline some of the recent works relevant to our proposal and comment on their respective merits. We will divide the discussion in three topics following the main contributions of this thesis.

6.1 Barrier Verification

O'Hearn's concurrent separation logic focused on programs that used critical regions [80, 16]; subsequent work by Hobor *et al.* and Gotsman *et al.* added first-class locks and threads [51, 43, 50]. Our basic soundness techniques (unerased semantics tracks resource accounting;oracle semantics isolates sequential and concurrent reasoning from each other; etc.) follow Hobor *et al.* Recently both Villard *et al.* and Bell *et al.* extended concurrent separation logic to channels [7, 96]. The work on channels is similar to ours in that both Bell and Villard track additional dynamic state in the logic and soundness proof. Bell tracks communication histories while Villard tracks the state of a finite state automaton associated with each communication channel. Of all of the previous soundness results, only Hobor *et al.* had a machine-checked soundness proof, albeit an incomplete one.

An interesting question is whether is it possible to reason about barriers in a setting with

locks or channels. The question has both an operational and a logical flavor. Speaking operationally, in a practical sense the answer is no: for performance reasons barriers are not implemented with channels or locks. If we ignore performance, however, it **is** possible to implement barriers with channels or locks[1]. The logical part of the question then becomes, are the program logics defined by O'Hearn, Hobor, Gotsman, Villard, or Bell (including their coauthors) strong enough to reason about the (implementation of) barriers in the style of the logic we have presented? As far as we can tell each previous solution is missing at least one required feature, so in a strict sense, the answer here is again no.

For illustration we examine what seems to be the closest solution to ours: the copyless message passing channels of Villard *et al.* Operationally speaking, the best way to implement barriers seems to be by adding a central authority that maintains a channel with each thread using a barrier. When a thread hits a barrier, it sends "waiting" to the central authority, and then waits until it receives "proceed". In turn, the central authority waits for a "waiting" message from each thread, and then sends each of them a "proceed" message. Fortunately Villard allows the central authority to wait on multiple channels simultaneously.

The question then becomes a logical one. Although it should not pose any fundamental difficulty, their logic would first need to be enhanced with fractional permissions; in fact we believe that Villard's Heap-Hop tool already uses the same fractional permission model (by Dockins *et al.*) that we do. Since Villard uses automata to track state, we think it probable, but not certain, that our barrier state machines can be encoded as a series of his channel state machines.

There are some problems to solve. Villard requires certain side conditions on his channels; we require other kinds of side conditions on our barriers; these conditions do not seem fully compatible [2]. Assuming that we can weaken/strengthen conditions appropriately, we reach a second problem with the side conditions: some of our side conditions (*e.g.*, mutual exclusion)

[1] Indeed, it is possible to implement channels and locks in terms of each other.

[2] For example, Villard requires determinacy whereas we do not; he would also require that the postconditions of barriers be precise whereas we do not; etc.

are restrictions on the shape of the entire diagram; in Villard's setting the barrier state diagram has been partitioned into numerous separate channel state machines. Verifying our side conditions seems to require verification of the relationships that these channel state machines have to each other; the exact process is unclear.

Once the matter of side conditions is settled, there remains the issue of verifying the individual threads and the central authority. Villard's logic seems to have all that is required for the individual threads; the question is how difficult it would be to verify the central authority. Here we are less sure but suspect that with enough ghost state/instructions it can be done.

There remains a question as to whether it is a good idea to reason about barriers via channels (or locks). We suspect that it is not a good idea, even ignoring the fact that actual implementations of barriers do not use channels. The main problem seems to be a loss of intuition: by distributing the barrier state machine across numerous channel state machines and the inclusion of necessary ghost state, it becomes much harder to see what is going on. We believe that one of the major contributions of our work is that our barrier rule is extremely simple; with a quick reference to the barrier state diagram it is easy to determine what is going on. There is a secondary problem: we believe that our barrier rule will look and behave essentially the same way in a setting with first-class barriers in which it is possible to define functions that are polymorphic over the barrier diagram; even assuming a channel logic enriched in a similar way, the verification of a polymorphic central authority seems potentially formidable.

One interesting question is how our barrier rule would interact with the rules of other flavors of concurrent separation logic (*e.g.*, with locks or channels). We believe that the answer is yes, at least in the context of a logic of partial correctness[1], as long as the primitives used remain strongly synchronizing (*i.e.*, coarse-grained). It is not clear how our barrier rule might interact with the kind of fine-grained concurrency that is the subject of Vafeiadis and Parkinson [95], Dodds *et al.* [29], or Dinsdale-Young *et al.* [26]. We believe that our barrier rule is sound on a

[1]Of course, the more concurrency primitives a programmer has, the easier it is to get into a deadlock. We hypothesize that concurrent program logics of total correctness may not be as compositional as concurrent program logics of partial correctness.

machine with weak memory as long as all of the concurrency is strongly synchronized.

Finally, work on concurrent program analysis is in the early stages; Gotsman *et al*., Calcagno *et al*., and Villard *et al*. give techniques that cover some use cases involving locks and channels but much remains to be done [42, 18, 97].

Connection to a result by Jacobs and Piessens. We recently learned that Jacobs and Piessens have an impressive result on modular fine-grained concurrency [59]. Jacobs was able to reason about our example program using his VeriFast tool by designing an implementation of barriers using locks and reducing our barrier diagram to a large disjunction for a resource invariant.

However, VeriFast has some disadvantages compared to the HIP/SLEEK approach we presented. First, HIP/SLEEK required far less input from the user. In the case of our 30-line example program, more than 600 lines of annotation were required in VeriFast, not including the code/annotations for the barrier implementation itself. HIP/SLEEK were able to verify the same program with approximately 30 lines of annotation (mostly the barrier definition). Second, it was harder to gain insight into the program from the disjunction-form of the invariant; in contrast we find our barrier diagrams straightforward to understand. Finally, it is unclear to us whether the reduction is always possible or whether it was only enabled by the relative simplicity of our example program. That said, Jacobs and Piessens have the only logic and tool proven to be able to reason about barriers as derived from a more general mechanism.

6.2 Specialization Calculus

Traditionally, specialization techniques [61, 83, 69] have been used for code optimization by exploiting information about the context in which the program is executed. Classical examples are the partial evaluators based on binding-time analysers that divide a program into static parts to be specialized and dynamic parts to be residualized. However, our work focusses on a different usage domain, proposing a predicate specialization for program verification to prune infeasible disjuncts from abstract program states. In contrast, partial evaluators [61]

use unfolding and specialised methods to propagate static information. More advanced partial evaluation techniques which integrate abstract interpretation have been proposed in the context of logic and constraint logic programming [84, 83, 69].They can control the unfolding of predicates by enhancing the abstract domains with information obtained from other unfolding operations. Our work differs in its focus on minimizing the number of infeasible states, rather than on code optimization. This difference allows us to use techniques, such as memoization and incremental pruning, that were not previously exploited for specialization.

Several symbolic pruning techniques [39, 2, 34, 44] have been proposed in the area of model checking for ameliorating the state explosion problem, especially in the context of verifying concurrent systems. Thus partial order reduction techniques [39, 2, 34] are used to prune away redundant thread interleavings within each trace equivalence class, while property-driven symbolic methods [62] are used to prune away property specific redundant interleavings between different trace classes. Similarly, symbolic model checking techniques, e.g. [23, 44, 62], rely on underlying SAT or SMT solvers to prune infeasible states. SAT solvers usually use a conflict analysis [92] that records the causes of conflicts so as to recognize and preempt the occurrences of similar conflicts later on in the search. Modern SMT solvers (e.g. [79, 32]) use analogous analyses to reduce the number of calls to underlying theory solvers. Compared to our pruning approach, conflict analysis [92] is a backtracking search technique that discovers contexts leading to conflicts and uses them to prune the search space. These techniques are mostly complementary since they did not consider predicate specialization, which is important for expressive logics. From a program verification perspective, our work falls in line with recent trends of having an expressive specification mechanism. This is important for verification systems, such as [101, 102], that aim for full functional correctness to be guaranteed.

The primary goal of our work is to provide a more effective way to handle disjunctive predicates for separation logic [75, 77]. The proper treatment of disjunction (to achieve a trade-off between precision and efficiency) is a key concern of existing shape analyses based on separation logic [27, 65]. One research direction is to design parameterized heap materialization

mechanisms (also known as focus operation) adapted to specific program statements and to specific verification tasks [88, 71, 87, 8, 82]. Another direction is to design partially disjunctive abstract domains with join operators that enable the analysis to abstract away information considered to be irrelevant for proving a certain property [45, 100, 19]. Techniques proposed in these directions are currently orthogonal to the contribution of this work and it would be interesting to investigate if they could benefit from predicate specialization, and vice-versa.

6.3 Exception Verification

Some proposals, [64, 60] have considered a core language with exceptions by adding both a `throw e` construct and a simplified `try` $\mathtt{e}_1$ `catch` (`c v`) $\mathtt{e}_2$ construct from the Java language. However, this simplified feature was not able to *succinctly* handle more advanced features, such as `try-finally` nor `try-with-multiple-catches`. Another proposal [31] directly adds these advanced features in their core language, but this is done at the price of a more complex formalization. [60] employs two analyses of different granularity, at the expression and method levels, to collect constraints over the set of exception types handled by each Java program. By solving the constraint system using techniques from [33], they can obtain a fairly precise set of uncaught exceptions. To overcome the above shortcomings, we proposed a more expressive specification logic, which complements the type system through selective tracking on types that are linked to control flows.

A recent approach towards exception handling in a higher-order setting, is taken in [11]. Exceptions are represented as sums and exception handlers are assigned polymorphic, extensible row types. Furthermore, following a translation to an internal language, each exception handler is viewed as an alternative continuation whose domain is the sum of all exceptions that could arise at a given program point. Though CPS can be used to uniformly handle the control flows, it increases the complexity of the verification process as continuations are first class values.

Spec# also has a specification mechanism for exceptions however exceptional specifications are currently useable only by the runtime checking module. The current Spec# prototype for static verification would unsoundly approximate each exception as false, denoting an unreachable program state [68].

Another impressive verification system, based on the Java language, is known as the KeY approach [6]. This approach is more complex ,due to their use of dynamic logic, since rewriting rules would have to be specified for each programming construct that is allowed in the dynamic logic. For example, to support exception handling, a set of rewriting rules (that are meaning-preserving) would have to be formulated to deal with raise, break and try-catch-finally constructs, in addition to rewriting rules for block and conditionals and their possible combinations. The KeY approach is meant to be a semi-automated verification system that occasionally prompts the user for choice of rewriting rules to apply. In contrast, our approach is meant to be fully-automated verification system, once pre/post specifications have been designed.

At a foundational level, Ancona et al [3] have added a limited form of exceptions into Featherweight Java [56] to formally study the interaction between inheritance and exceptions. Others [64, 31, 13] have provided a richer core language with exceptions but have still to provide an unified view over the myriad of control flows, including that for normal execution. These proposals have formalised type systems that have been proven sound, though [64] skipped the tracking of uncaught exceptions. Another important dimension is the inference of uncaught exceptions [60, 33, 81]. For example, [60] employs two analyses of different granularity, at the expression and method levels, to collect constraints over the set of exception types handled by each Java program. By solving the constraint system using techniques from [33], they can obtain a fairly precise set of uncaught exceptions.

Exception-based programs are generally harder to analyse, since they introduce extra control flows to the programs. Many previous approaches handle this challenge at a lower-level of abstraction. For example, [93] describes the effects of exceptions on a control-flow, data-flow and control dependence analysis and suggested using a control flow graph that will contain

exceptional flows. For this approach to be concise, a prior type inference for exception types must also be done. Along the same line, [22] introduced a form of a compact control flow graph that includes exceptional control flow and demonstrated that it could be used to construct an improved interprocedural analysis. Weimer and Necula [99] use a data flow analysis to detect broken specifications for resource usage. Even in the verification setting, Barnett and Leino [5] resorted to lower-level unstructured programs (with global variables) to model and analyse exception-based programs. Though lower-level constructs can be made to work, they are typically harder to formulate and prove correct. Our proposal to unify both normal and abnormal control flows in a core calculus is aimed at providing a higher-level of abstraction for reasoning about exception-based programs, including those needed by advance program analyses.

To support deeper reasoning of exception-based programs, Huisman et al. [55] proposed an extension for Hoare-style verification to handle exceptions and abnormal control flows. Separate logical rules were formulated to support reasoning on total and partial correctness, and for handling each kind of abnormal control flows. However, the myriad of logical rules used can be complex to implement. In the case of our core calculus, we were able to provide a unified specification mechanism for the outcomes of both normal and abnormal executions, without having to treat exceptions in a special way.

Chapter 7

Conclusions

In this thesis we have introduced a sound verification logic designed to efficiently handle programs with complex control flow patterns. On one hand, in the sequential setting, the logic explicitly and precisely tracks control flows essential to the program verification. On the other hand, it incorporates abstractions (e.g. barrier definitions) required to maintain modularity and avoid the exponential blowup when reasoning in a multithreaded setting in the presence of sophisticated synchronization mechanisms. In order to maintain the tractability of verification in the presence of barriers, we have introduced a novel calculus for pruning disjunctive predicates. We have also described several contributions related to the integration of our verification logic in order to broaden the applicability of the HIP/SLEEK verification tool chain, e.g. the first solver for the distinct fractional shares domain.

7.1 Results Summary

Exception Verification. In Chapters 2 and 3 we have presented a new approach to the verification of exception-handling programs based on a specification logic that can uniformly handle exceptions, program errors and other kinds of control flows. During the implementation process of our verification logic we have found it beneficial to make a fundamental language

re-design. The unification of flow type handling, has allowed us to formalise a core calculus that enables an elegant formulation of the verification rules.

The specification logic is currently built on top of the formalism of separation logic, as the latter can give precise description to heap-based data structures. Our main motivation for proposing this new specification logic is to adapt the verification method to help ensure exception safety in terms of the four guarantees of increasing quality introduced in [94] and extended in [70], namely no-leak guarantee, basic guarantee, strong guarantee and no-throw guarantee. During the evaluation process, we found the strong guarantee to be restrictive for some scenarios, as it always forces a recovery mechanism on the callee, should exceptions occur. Hence, we propose to generalise the definition of strong guarantee for exception safety.

Barrier Verification. In Chapter 4, we have extended the exception logic with support for Pthreads-style barriers. Our development includes a formal design for barrier definitions and a series of soundness conditions to verify that a particular barrier can be used safely. Our Hoare rules can verify threads independently, enabling a thread-modular approach.

Efficient Verification. In Chapter 5, we have proposed a sound specialization calculus for disjunctive predicates in a separation logic-based abstract domain. The calculus supports symbolic pruning of infeasible states within each predicate instance, under monotonic changes to the program context. We have designed inference techniques that can automatically derive all annotations required for each specializable predicate. We have also proposed various ways to optimize the specialization process, including memoization and incremental pruning. Initial experiments have confirmed speed gains from the deployment of our specialization mechanism to handle separation logic specifications in program verification.

We have shown how the specialization calculus naturally extends to barrier definitions and that it ensures that the barrier reasoning mechanism does not incur any considerable performance penalty over traditional verification of sequential programs.

Our approach has been formalised and implemented in a prototype extension of HIP/SLEEK,

and tested on a suite of exception-handling and barrier synchronized concurrent examples. We hope it would eventually become a useful tool to help programmers build more robust software.

7.2 Future Work

We observe several avenues for further research:

Barrier analysis With respect to barrier reasoning , we have described the necessary mechanisms for capturing and reasoning about barrier behaviour. However, the current assumptions were that both barrier call placement and barrier definitions were given. We believe that it could be possible to go two steps further: to first develop a program analysis that could infer the sufficient set of program points at which barrier calls are needed and secondly to infer , at least to some extent, both barrier definition shape and partial specifications.

Synchronization logic A second avenue of investigation is spurred by the observation that a myriad of highly specialized, synchronization related, verification logics have been proposed, each targeted at a specific synchronization technique: barriers, locks, semaphores. More so, several others have not been investigated: various styles of monitors, dynamic barriers (X10-style clocks), phasers[90]. One open question that arises is: could a general synchronization mechanism together with a corresponding verification logic be designed such that they would encompass the commonly used synchronization mechanisms and thus have one simple, practical, unifying formalism?

Reverse specialization One motivation for the specialization calculus was the fact that unfold operations are on one hand essential for obtaining more precise information and on the other hand are quite expensive. We point out that the folding operation could also benefit from the specialization calculus. For the sake of conciseness or abstraction in most instances information is commonly discarded during folding. By making use of the specialization calcu-

lus, this loss of information could be alleviated while still obtaining the conciseness required. Apart from program verification, we suspect program analysis could also greatly benefit from "reverse pruning".

Specialization optimisation In §5.7.2 we have briefly discussed a secondary optimization facilitated by the pruning mechanism. Although it did allow for significant gains by generating a partition of the constraints into dependent groups and thus permitting an easy filtering of relevant constraints and effectively shrinking the proof obligations sent to the provers we believe there is still room for improvement. The observation is that the current construction is very conservative in its partitioning due to the definition of the `connected` relation.

One proposal is to allow for a more flexible partitioning scheme in which the user is allowed to partially sketch the partitions. For example he could specify that constraints pertaining to certain variables be in distinct partitions. Simple annotations can be fitted into the specification languages such that providing partitioning hints would be effortless while unannotated constraints would be processed by the current partitioning mechanism thus requiring the user to specify only the partitions of interest.

Bibliography

[1] SPECjvm2008 Benchmarks. "http://www.spec.org/jvm2008/". 66

[2] R. ALUR, R. K. BRAYTON, T. A. HENZINGER, S. QADEER, AND S. K. RAJAMANI. Partial-order reduction in symbolic state-space exploration. *Form. Methods Syst. Des.*, **18**[2]:97–116, 2001. 159

[3] DAVIDE ANCONA, GIOVANNI LAGORIO, AND ELENA ZUCCA. A Core Calculus for Java Exceptions. In *Conference on Object-Oriented*, pages 16–30, 2001. 161

[4] ANDREW W. APPEL AND SANDRINE BLAZY. Separation logic for small-step C minor. In *TPHOLs*, pages 5–21, 2007. 81, 87

[5] MICHAEL BARNETT AND K. RUSTAN M. LEINO. Weakest-precondition of unstructured programs. In *PASTE*, pages 82–87, 2005. 162

[6] BERNHARD BECKERT, REINER HÄHNLE, AND PETER H. SCHMITT, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007. 66, 161

[7] CHRISTIAN J. BELL, ANDREW W. APPEL, AND DAVID WALKER. Concurrent separation logic for pipelined parallelization. In *SAS*, 2010. 155

[8] J. BERDINE, C. CALCAGNO, B. COOK, D. DISTEFANO, P. W. O'HEARN, T. WIES, AND H. YANG. Shape analysis for composite data structures. In *CAV*, pages 178–192, 2007. 160

[9] J. BERDINE, C. CALCAGNO, AND P. W. O'HEARN. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, Springer LNCS 4111, pages 115–137, 2006. 3

[10] CHRISTIAN BIENIA. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, Department of Computer Science, Princeton, NJ, December 2010. 4, 70

[11] MATTHIAS BLUME, UMUT A. ACAR, AND WONSEOK CHAE. Exception handlers as extensible cases. In *APLAS*, pages 273–289, 2008. 160

[12] FRANÇOIS BOBOT, SYLVAIN CONCHON, EVELYNE CONTEJEAN, AND STÉPHANE LESCUYER. Implementing Polymorphism in SMT solvers. In *SMT 2008: 6th International Workshop on Satisfiability Modulo*, 2008. 144

[13] EGON BÖRGER AND WOLFRAM SCHULTE. A practical method for specification and analysis of exception handling - a java/jvm case study. *IEEE Trans. Software Eng.*, **26**[9]:872–887, 2000. 161

[14] RICHARD BORNAT, CRISTIANO CALCAGNO, PETER O'HEARN, AND MATTHEW PARKINSON. Permission accounting in separation logic. In *POPL*, pages 259–270, 2005. 31, 71

[15] THOMAS BRAIBANT AND DAMIEN POUS. Tactics for reasoning modulo AC in Coq. In *CPP*, pages 167–182, 2011. 106

[16] STEPHEN D. BROOKES. A semantics for concurrent separation logic. In *CONCUR*, pages 16–34, 2004. 155

[17] DAVID R. BUTENHOF. *Programming with POSIX Threads*. Addison-Wesley, 1997. 70

[18] CRISTIANO CALCAGNO, DINO DISTEFANO, AND VIKOR VAFEIADIS. Bi-abductive resource invariant synthesis. In *APLAS*, 2009. 158

[19] B.-Y. E. CHANG AND X. RIVAL. Relational inductive shape analysis. In *POPL*, pages 247–260, 2008. 160

[20] WEI-NGAN CHIN, CRISTINA DAVID, HUU HAI NGUYEN, AND SHENGCHAO QIN. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Science of Computer Programming*, 2010. 6, 9, 95, 98

[21] WEI-NGAN CHIN, CRISTIAN GHERGHINA, RAZVAN VOICU, QUANG LOC LE, FLORIN CRACIUN, AND SHENGCHAO QIN. A specialization calculus for pruning disjunctive predicates to support verification. In *CAV*, pages 293–309, 2011. 9

[22] JONG-DEOK CHOI, DAVID GROVE, MICHAEL HIND, AND VIVEK SARKAR. Efficient and precise modeling of exceptions for the analysis of java programs. *SIGSOFT Softw. Eng. Notes*, **24**[5]:21–31, 1999. 162

[23] BYRON COOK, DANIEL KROENING, AND NATASHA SHARYGINA. Symbolic model checking for asynchronous boolean programs. In *SPIN*, 2005. 159

[24] P. COUSOT AND R. COUSOT. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM Symposium on Principles of Programming Languages*, 1977. 134

[25] C. DAVID, C. GHERGHINA, AND W. N. CHIN. Translation and optimization for a core calculus with exceptions. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*. ACM Press, 2009. 7

[26] THOMAS DINSDALE-YOUNG, MIKE DODDS, PHILIPPA GARDNER, MATTHEW J. PARKINSON, AND VIKTOR VAFEIADIS. Concurrent abstract predicates. In *ECOOP*, pages 504–528, 2010. 157

[27] D. DISTEFANO, P. W. O'HEARN, AND H. YANG. A local shape analysis based on separation logic. In *TACAS*, pages 287–302, 2006. 159

[28] ROBERT DOCKINS, AQUINAS HOBOR, AND ANDREW W. APPEL. A fresh look at separation algebras and share accounting. In *APLAS*, 2009. 6, 9, 31, 71, 90

[29] MIKE DODDS, XINYU FENG, MATTHEW J. PARKINSON, AND VIKTOR VAFEIADIS. Deny-guarantee reasoning. In *ESOP*, pages 363–377, 2009. 157

[30] MARK DOWSON. The ariane 5 software failure. *SIGSOFT Softw. Eng. Notes*, **22**[2]:84–, March 1997. 1

[31] SOPHIA DROSSOPOULOU AND TANYA VALKEVYCH. Java Exceptions Throw No Surprises. Technical report, Imperial College of Science, Technology and Medicin, March 2000. 160, 161

[32] B. DUTERTRE AND L. M. DE MOURA. A Fast Linear-Arithmetic Solver for DPLL(T). In *CAV*, pages 81–94, 2006. 144, 159

[33] MANUEL FÄHNDRICH AND ALEXANDER AIKEN. Program analysis using mixed term and set constraints. In *SAS '97: Proceedings of the 4th International Symposium on Static Analysis*, pages 114–126, London, UK, 1997. Springer-Verlag. 160, 161

[34] CORMAC FLANAGAN AND PATRICE GODEFROID. Dynamic partial-order reduction for model checking software. In *ACM Symposium on Principles of Programming Languages*, 2005. 159

[35] ROBERT W. FLOYD. Assigning meanings to programs. In *Proc. Amer. Math. Soc. Symposia in Applied Mathematics*, **19**, pages 19–31, 1967. 2

[36] MICHAEL J. FLYNN AND KEVIN W. RUDD. Parallel architectures. *ACM Comput. Surv.*, **28**[1]:67–70, March 1996. 108

[37] CRISTIAN GHERGHINA AND CRISTINA DAVID. A specification logic for exceptions and beyond. In *ATVA*, pages 173–187, 2010. 8

[38] CRISTIAN GHERGHINA, CRISTINA DAVID, SHENGCHAO QIN, AND WEI-NGAN CHIN. Structured specifications for better verification of heap-manipulating programs. In *FM*, 2011. 89, 95, 104

[39] PATRICE GODEFROID. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag, 1996. 159

[40] GEORGES GONTHIER, BETA ZILIANI, ALEKSANDAR NANEVSKI, AND DEREK DREYER. How to make ad hoc proof automation less ad hoc. In *ICFP*, pages 163–175, 2011. 106

[41] JOHN B. GOODENOUGH. Structured exception handling. In *POPL '75: Proceedings of the 2nd ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 204–224, New York, NY, USA, 1975. ACM. 57

[42] ALEXEY GOTSMAN, JOSH BERDINE, AND BYRON COOK. Interprocedural Shape Analysis with Separated Heap Abstractions. In *SAS*, pages 240–260, 2006. 3, 158

[43] ALEXEY GOTSMAN, JOSH BERDINE, BYRON COOK, NOAM RINETZKY, AND MOOLY SAGIV. Local reasoning for storable locks and threads. In *APLAS*, pages 19–37, 2007. 3, 69, 77, 155

[44] ORNA GRUMBERG, FLAVIO LERDA, OFER STRICHMAN, AND MICHAEL THEOBALD. Proof-guided underapproximation-widening for multi-process systems. In *ACM Symposium on Principles of Programming Languages*, pages 122–131, 2005. 159

[45] B. GUO, N. VACHHARAJANI, AND D. I. AUGUST. Shape analysis with inductive recursion synthesis. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 256–265, 2007. 160

[46] J. V. GUTTAG AND J. J. HORNING, editors. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993. 2

[47] C. A. R. HOARE. An axiomatic basis for computer programming. *Commun. ACM*, **12**[10]:576–580, 1969. 2

[48] TONY HOARE. Verified software: Theories, tools, experiments. In *International Conference on Engineering of Complex Computer Systems*, 2008. 2

[49] TONY HOARE AND JAYADEV MISRA. Verified software: Theories, tools, experiments vision of a grand challenge project. In *VSTTE*, pages 1–18, 2005. 2

[50] AQUINAS HOBOR. Oracle semantics. Technical Report TR-836-08, Princeton, 2008. 3, 69, 77, 85, 88, 155

[51] AQUINAS HOBOR, ANDREW W. APPEL, AND FRANCESCO ZAPPA NARDELLI. Oracle semantics for concurrent separation logic. In *ESOP*, pages 353–367, 2008. 3, 21, 69, 81, 155

[52] AQUINAS HOBOR AND CRISTIAN GHERGHINA. Barriers in concurrent separation logic. In *ESOP*, pages 276–296, 2011. 8

[53] AQUINAS HOBOR AND CRISTIAN GHERGHINA. Barriers in concurrent separation logic: Now with tool support! *Logical Methods in Computer Science*, **8**[2], 2012. 8

[54] P. HUDAK AND ET AL. Report on the programming language Haskell: A non-strict, purely functional language. *ACM SIGPLAN Notices*, **27**[5], May 1992. 43

[55] MARIEKE HUISMAN AND BART JACOBS. Java Program Verification via a Hoare Logic with Abrupt Termination. In *FASE*, pages 284–303, 2000. 162

[56] A. IGARASHI, B. PIERCE, AND P. WADLER. Featherweight Java: A Minimal Core Calculus for Java and GJ. In *ACM OOPSLA*, Denver, Colorado, Nov 1999. 161

[57] S. ISHTIAQ AND P.W. O'HEARN. BI as an assertion language for mutable data structures. In *ACM Symposium on Principles of Programming Languages*, pages 14–26, London, Jan 2001. 2

[58] B. JACOBS, J. SMANS, AND F. PIESSENS. A Quick Tour of the VeriFast Program Verifier. In *APLAS*, pages 304–311, 2010. 3

[59] BART JACOBS AND FRANK PIESSENS. Expressive modular fine-grained concurrency specification. In *POPL '11*, page To appear, 2011. 158

[60] JANG-WU JO, BYEONG-MO CHANG, KWANGKEUN YI, AND KWANG-MOO CHOE. An uncaught exception analysis for Java. *Journal of Systems and Software*, **72**[1]:59–69, 2004. 17, 160, 161

[61] N.D. JONES, C.K. GOMARD, AND P. SESTOFT. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993. 158

[62] VINEET KAHLON, AARTI GUPTA, AND NISHANT SINHA. Symbolic model checking of concurrent programs using partial orders and on-the-fly transactions. In *CAV*, pages 286–299, 2006. 159

[63] N. KLARLUND AND A. MOLLER. MONA Version 1.4 - User Manual. BRICS Notes Series, Jan 2001. 122

[64] GERWIN KLEIN AND TOBIAS NIPKOW. A machine-checked model for a Java-like language, virtual machine, and compiler. *ACM Trans. Program. Lang. Syst.*, **28**[4]:619–695, 2006. 17, 160, 161

[65] V. LAVIRON, B.-Y. EVAN CHANG, AND X. RIVAL. Separating shape graphs. In *ESOP*, pages 387–406, 2010. 159

[66] XUAN BACH LE, CRISTIAN GHERGHINA, AND AQUINAS HOBOR. Decision procedures over sophisticated fractional permissions. In *APLAS*, 2012. 92

[67] K. RUSTAN M. LEINO AND WOLFRAM SCHULTE. Exception Safety for C#. In *SEFM '04: Proceedings of the Software Engineering and Formal Methods, Second International Conference*, pages 218–227, Washington, DC, USA, 2004. IEEE Computer Society. 58

[68] RUSTAN LEINO. personal communication, Jan 2009. 161

[69] M. LEUSCHEL. A framework for the integration of partial evaluation and abstract interpretation of logic programs. *ACM Trans. Program. Lang. Syst.*, **26**[3]:413–463, 2004. 114, 158, 159

[70] XIN LI, H. JAMES HOOVER, AND PIOTR RUDNICKI. Towards automatic exception safety verification. In *FM*, pages 396–411, 2006. iv, 6, 8, 59, 68, 164

[71] R. MANEVICH, J. BERDINE, B. COOK, G. RAMALINGAM, AND M. SAGIV. Shape analysis by graph decomposition. In *TACAS*, pages 3–18, 2007. 160

[72] ROY A. MAXION AND ROBERT T. OLSZEWSKI. Improving software robustness with dependability cases. In *28th International Symposium on Fault Tolerant Computing*, pages 346–355, 1998. 57

[73] ALEKSANDAR NANEVSKI, VIKTOR VAFEIADIS, AND JOSH BERDINE. Structuring the verification of heap-manipulating programs. In *POPL*, pages 261–274, 2010. 106

[74] CHARLES GREGORY NELSON. *Techniques for program verification*. PhD thesis, Stanford University, 1980. 2

[75] H.H. NGUYEN, C. DAVID, S.C. QIN, AND W.N. CHIN. Automated Verification of Shape And Size Properties via Separation Logic. In *VMCAI*, Nice, France, Jan 2007. 3, 33, 115, 145, 159

[76] HUU HAI NGUYEN AND WEI-NGAN CHIN. Enhancing program verification with lemmas. In *CAV*, pages 355–369, 2008. 89

[77] HUU HAI NGUYEN AND WEI-NGAN CHIN. Enhancing program verification with lemmas. In *CAV*, 2008. 159

[78] HUU HAI NGUYEN, CRISTINA DAVID, SHENGCHAO QIN, AND WEI-NGAN CHIN. Automated verification of shape and size properties via separation logic. In *VMCAI*, pages 251–266, 2007. 6, 95

[79] R. NIEUWENHUIS, A. OLIVERAS, AND C. TINELLI. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *J. ACM*, **53**[6]:937–977, 2006. 159

[80] PETER W. O'HEARN. Resources, concurrency and local reasoning. *Theoretical Computer Science*, **375**[1]:271–307, May 2007. 3, 69, 70, 77, 155

[81] FRANCOIS PESSAUX AND XAVIER LEROY. Type-based analysis of uncaught exceptions. In *Symposium on Principles of Programming Languages*, pages 276–290, 1999. 161

[82] A. PODELSKI AND T. WIES. Counterexample-guided focus. In *ACM Symposium on Principles of Programming Languages*, pages 249–260, 2010. 160

[83] G. PUEBLA AND M. HERMENEGILDO. Abstract specialization and its applications. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 29–43, 2003. 114, 158, 159

[84] G. PUEBLA, M. HERMENEGILDO, AND J. P. GALLAGHER. An integration of partial evaluation in a generic abstract interpretation framework. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 75–84, January 1999. 114, 159

[85] W. PUGH. The Omega Test: A fast practical integer programming algorithm for dependence analysis. *Communications of the ACM*, **8**:102–114, 1992. 122

[86] J. REYNOLDS. Separation Logic: A Logic for Shared Mutable Data Structures. In *IEEE LICS*, pages 55–74, Copenhagen, Denmark, Jul 2002. 2, 33

[87] N. RINETZKY, A. POETZSCH-HEFFTER, G. RAMALINGAM, M. SAGIV, AND E. YAHAV. Modular shape analysis for dynamically encapsulated programs. In *ESOP*, pages 220–236, 2007. 160

[88] M. SAGIV, T. REPS, AND R. WILHELM. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, **24**[3]:217–298, 2002. 160

[89] S. B. SANJABI AND C.-H. L. ONG. Fully abstract semantics of additive aspects by translation. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, pages 135–148, New York, NY, USA, 2007. ACM. 43

[90] JUN SHIRAKO, DAVID M. PEIXOTTO, VIVEK SARKAR, AND WILLIAM N. SCHERER. Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. In *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, pages 277–288, 2008. 165

[91] OLIN SHIVERS AND DAVID FISHER. Multi-return function call. In *ICFP '04: Proceedings of the ninth ACM SIGPLAN international conference on Functional programming*, pages 79–89, New York, NY, USA, 2004. ACM. 12, 16

[92] J. P. MARQUES SILVA AND K. A. SAKALLAH. GRASP—a new search algorithm for satisfiability. In *International Conference on Computer-Aided Design*, pages 220–227, 1996. 159

[93] SAURABH SINHA AND MARY JEAN HARROLD. Analysis and testing of programs with exception handling constructs. *IEEE Trans. Software Eng.*, **26**[9]:849–871, 2000. 161

[94] BJARNE STROUSTRUP. Exception safety: Concepts and techniques. In *Advances in Exception Handling Techniques*, pages 60–76, 2000. iv, 6, 8, 59, 62, 64, 67, 68, 164

[95] VIKTOR VAFEIADIS AND MATTHEW J. PARKINSON. A marriage of rely/guarantee and separation logic. In *CONCUR*, pages 256–271, 2007. 157

[96] JULES VILLARD, ÉTIENNE LOZES, AND CRISTIANO CALCAGNO. Proving copyless message passing. In *APLAS*, pages 194–209, 2009. 155

[97] JULES VILLARD, ÉTIENNE LOZES, AND CRISTIANO CALCAGNO. Tracking heaps that hop with heap-hop. In *TACAS*, pages 275–279, 2010. 158

[98] EELCO VISSER. Program transformation with stratego/xt. rules, strategies, tools, and systems in stratego/xt 0.9. Technical Report UU-CS-2004-011, Department of Information and Computing Sciences, Utrecht University, 2004. 35

[99] WESTLEY WEIMER AND GEORGE C. NECULA. Exceptional situations and program reliability. *ACM Trans. Program. Lang. Syst.*, **30**[2]:1–51, 2008. 162

[100] H. YANG, O. LEE, J. BERDINE, C. CALCAGNO, B. COOK, D. DISTEFANO, AND P. W. O'HEARN. Scalable shape analysis for systems code. In *CAV*, pages 385–398, 2008. 160

[101] KAREN ZEE, VIKTOR KUNCAK, AND MARTIN RINARD. Full functional verification of linked data structures. *SIGPLAN Not.*, **43**[6]:349–361, 2008. 159

[102] KAREN ZEE, VIKTOR KUNCAK, AND MARTIN C. RINARD. An integrated proof language for imperative programs. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 338–351, New York, NY, USA, 2009. ACM. 159