A Framework for Automated Concurrency Verification

Matthew Bernard Windsor

Ph.D

University of York
Computer Science

April 2019

Abstract

Reasoning systems based on Concurrent Separation Logic make verifying complex concurrent algorithms readily possible. Such algorithms contain subtle protocols of permission and resource transfer between threads; to cope with these intricacies, modern concurrent separation logics contain many moving parts and integrate many bespoke logical components.

Verifying concurrent algorithms by hand consumes much time, effort, and expertise. As a result, computer-assisted verification is a fertile research topic, and fully automated verification is a popular research goal. Unfortunately, the complexity of modern concurrent separation logics makes them hard to automate, and the proliferation and fast turnover of such logics causes a downward pressure against building tools for new logics. As a result, many such logics lack tooling.

This dissertation proposes Starling: a scheme for creating concurrent program logics that are automatable by construction. Starling adapts the existing *Concurrent Views Framework* for sound concurrent reasoning systems, overlaying a framework for reducing concurrent proof outlines to verification conditions in existing theories (such as those accepted by off-the-shelf sequential solvers).

This dissertation describes Starling in a bottom-up, modular manner. First, it shows the derivation of a series of general concurrency proof rules from the Views framework. Next, it shows how one such rule leads to the Starling framework itself. From there, it outlines a series of increasingly elaborate *frontends*: ways of decomposing individual Hoare triples over atomic actions into verification conditions suitable for encoding into backend theories. Each frontend leads to a concurrent program logic.

Finally, the dissertation presents a tool for verifying C-style concurrent proof outlines, based on one of the above frontends. It gives examples of such outlines, covering a variety of algorithms, backend solvers, and proof techniques.

Contents

Listings

Acknowledgements

This work would not have been possible without the input and guidance from my two supervisors, **Mike Dodds** (from 2014 to 2017) and **Radu Calinescu** (2017 onwards). Both have gone above and beyond the call of duty to help me in my journey — especially in times where the path has been unclear, or fraught. I am grateful to them for not giving up on me, especially when times have been difficult — and whenever I was a challenging student!

I'm grateful to **Jeremy Jacob** and **Ana Cavalcanti** for several reasons. Jeremy, my internal assessor, has been a reassuring and stabilising force throughout my PhD journey; Ana, my undergraduate project supervisor in 2014, helped me to start the journey in the first place. My gratitude goes back earlier, though: Jeremy and Ana's *Programming: Correctness by Construction* course taught me the beauty and elegance of formal methods, and was a fundamental influence on the direction I have taken ever since.

I also thank **Matthew Parkinson** for his invaluable contributions throughout the PhD. This dissertation builds extensively on both his published work and insights he and Mike have shared with me over the four years. When I did not understand part of the *Views* framework, or was struggling with a piece of Starling's theory, 'MattP' was often the first port of call.

Many more people have helped me on my journey, and I cannot thank them all in one page. I especially thank: **Ben Simner** for his work as an undergraduate intern on Starling$_{\text{tool}}$; **Claudio Russo** for hosting *me* as an intern at Microsoft Research, Cambridge; **John Wickerson** and **Alastair Donaldson**, my bosses at Imperial College, London; my office-mates **Ivaylo**, **Rudy**, **Michael**, **Gia**, **Tim**, and **José**; and all of my friends who have helped me survive my PhD.

I gratefully note that an **EPSRC** Doctoral Training Grant funded my PhD. I also thank the **Interface Reasoning for Interacting Systems** project, which funds my post-doctoral work.

Finally, I thank my family, especially my parents **Angela** (1966–2018) and **Stephen**. They never gave up on me, even at my worst and my lowest, always seeing the best in me. I hope this dissertation does them proud.

To Mum

Declaration

I, Matthew Bernard Windsor, declare: that this dissertation presents original work; that I identify parts of said work developed in collaboration, and the collaborating parties, below this declaration; that I am the sole author of the dissertation, though parts of its text draw heavily from publications written in collaboration with other authors, which I also identify below this declaration; that I acknowledge all sources, including said publications, as references; and that this work has not been presented for a previous award at this, or any other, university.

Collaboration history

- Starling, in general, develops a previously unpublished idea by Matthew Parkinson and Mike Dodds;
- The definitions of the thread-local semantic judgement and multi-threaded action judgement in § 5.2 stem from conversations between the author and Matthew Parkinson in March 2018. The local-state action judgement is original work, but based on a thread-aware version suggested by Matthew Parkinson;
- Discussion of Starling's relationship with thread modularity, and its contrasts with that of Owicki-Gries, stem from a conversation with Andreas Podelski at CAV'2017;
- $\text{Starling}_{\text{tool}}$ is mainly the author's own work, but contains code written by Mike Dodds (including initial GRASShopper support, and examples including the lock-coupling list and allocatable ARC); Matthew Parkinson (including efficiency optimisations, bug-fixes, and initial phase timing support); and Ben Simner (including colourised error formatting, optimisation passes, continuous integration and regression testing infrastructure, initial support for iterated views, and work on the ARC proof)[1];
- The mechanisation is entirely the author's own work, except for the adapted use of parts of the original mechanisation of the Starling proof rule contributed by Matthew Parkinson. The mechanisation's treatment of the Concurrent and Local Views frameworks take inspiration, but not direct code, from the original Concurrent Views Framework Coq mechanisation.

[1] See also: `https://github.com/MattWindsor91/starling-tool/graphs/contributors`.

Publication history

The following parts of the dissertation have been previously published, in some form, as the paper *Starling: Lightweight Concurrency Verification with Views* [2]:

- the deriving-views proof rule (Definition 3.12), and its derivation from *Views* axiom soundness;
- discussion of $\text{Starling}_{\text{tool}}$'s Z3 and GRASShopper backends (included here in the relevant case studies);
- the ARC and CLH lock proofs (Listing 8.3, Listing 8.5, Listing 8.7)

Chapter 1

Introduction

> Your free lunch will soon be over. What can you do about it? What *are* you doing about it?
>
> — Herb Sutter [3]

In 2005, Sutter [3] predicted that years of constant performance gain through advances in CPU design — the 'free lunch' — were about to end. Soon after, CPUs hit their acceptable heat and power limits, and CPU designers changed tack: instead of faster cores, we now have more of them. To exploit these *multi-core* systems, we write *concurrent* programs: programs that let more than one thread of control exist at a time [4]. These threads *may* run in parallel on separate cores[1]; this lets concurrent programs make more efficient use of multi-core CPUs than sequential (non-concurrent) programs.

Concurrency is *not* a free lunch. Programmers pay through new challenges, including:

Deadlock Threads block, waiting for each other to perform tasks;

Livelock Threads are not blocked, but do not perform useful work;

Race conditions Changes to the scheduling of threads' actions change the behaviour of the program. Some race conditions are tolerable, but unexpected race conditions *may* cause undesirable results. Race conditions include *data races*, where one thread writes to a location at the same time as another accesses the same location [5]. Such access can be a write, causing the final value to be non-deterministic; or a read, causing non-determinism in whether none, part, or all of the write appears in the value that is read.

These challenges can lead to bugs, breaking software in ways both hard-to-predict and strikingly different from sequential breakages. For instance, sequential bugs are often *safety* issues where programs perform some *unwanted* action [6, §1], but deadlock and livelock are *liveness* issues that prevent programs from performing any *wanted* actions.

This said, concurrency safety bugs occur. Worse, they can kill: the *Therac-25* medical accelerator [7] is a well-known example. Its software, which used non-atomic shared-variable concurrency with no proper synchronisation, exhibited harmful data races. These, and other issues such as integer overflows, led to at least six radiation overdoses and three deaths.

[1] Concurrency does not enforce this: we can also sequence threads onto the same core.

Program proof. To avoid such disasters, we can prove concurrent programs correct against some specification. This specification can range from a loose collection of required properties up to a formal mapping to a sequential model. Such proofs are not straightforward; timing differences between cores, scheduling decisions, and other such factors mean that one program can exhibit many possible behaviours, and our proofs must consider each. Inter-thread interactions can make writing *compositional* proofs — where the parallel composition of two proven-correct programs is known correct without further proof — challenging.

These issues give rise to *correctness conditions*: ways to check concurrent behaviours against sequential specifications. *Linearisability* [8] is one such condition: it requires that each high-level operation in the specification appears 'to "take effect" instantaneously' in the concurrent program, and 'the order of nonconcurrent operations should be preserved'. Linearisability is compositional, but burdensome to prove, more so if the points where operations 'take effect' are subtle. Techniques such as *aspect-oriented linearisability proofs* [9] help with, but do not fully solve, these problems.

While proving safety properties such as memory safety and mutual exclusion does not give the same strong guarantees as linearisability or other full correctness conditions, it still boosts our ability to trust concurrent code. Even so, such proofs remain challenging — and interesting! This dissertation focuses on *automatically* proving safety properties over *atomic-action* (or 'fine-grained') concurrency; let us now discuss these qualifiers.

Atomic actions. Modern programming languages often offer high-level concurrency primitives. Go [10] and Rust push concurrency models based on threads sending messages through channels; other approaches include Haskell's *Par* monad [11]. These models shield programmers from data races by restricting the types of data access, and can be more intuitive than low-level concurrency. They *do not* prevent all concurrency bugs, for a variety of reasons:

- We must first build them, using low-level concurrency. (In fact, to avoid bottlenecks in high-level concurrent code, we must use efficient — and risky — primitives and techniques when doing so.) This leads to the insight that, given a system for reasoning about low-level concurrency, we can build high-level reasoning on top of it, instead of building a new system for each new high-level primitive [12].

- High-level abstractions cannot always express all correct concurrent algorithms. For example, Par works with *deterministic parallelism*, but some algorithms are inherently non-deterministic. Data-race prevention tactics can make correct, but racy, algorithms inexpressible. Message-passing systems cannot natively express mutual exclusion [13].

- High-level abstractions come with overhead, either by adding some directly or by preventing certain optimisations and techniques. There are examples in the literature of programmers gaining noteworthy speed-ups from using low-level tactics [14].

This dissertation considers low-level concurrency primitives. It focuses on small-scale primitives (changing one or two memory words at a given time) that are *atomic* (neither we, nor any part of the concurrent system, can observe their effect in an incomplete form). The

main challenge of atomic-action concurrency is to sequence these small actions to perform large state changes without causing concurrency bugs.

Automation. Proving properties of concurrent programs is hard, and the burden of doing so has limited the adoption of formal methods in software engineering. As a result, finding ways to shift the proof burden from humans to computers is a common research topic — as we see in § 8.4, which explores some of the existing work.

What can we improve in a well-explored field? From a high-level perspective, which Chapter 2 explores further, this work aims to build tools for fully automating proofs in the *Concurrent Separation Logic* tradition. Such proof systems cope well with the unique issues of shared-memory atomic-action concurrency, but are historically hard to automate. This is because they are complex; they contain much bespoke logical machinery; and tool production lags behind the fast pace of logic design.

This dissertation's goal is to find a way to balance the power and rapid development of such logics with automation friendliness, and build tools to make proving properties of atomic-action concurrent programs more straightforward.

1.1 Contribution

This dissertation explores the following thesis:

> Automatic concurrency verification by building tools for existing *Concurrent Separation Logic*-style program logics is hard: they are often complex, with much bespoke meta-theory. We propose a new approach: adapting an existing framework for proving soundness of program logics into a framework for building sound concurrent program logics that are automatable by construction. We hypothesise that this approach is flexible, expandable, and can produce practical tools for proving safety properties of real-world concurrency algorithms.

Following this thesis, this dissertation contributes the following:

1. *Starling*, a framework for deriving sound program logics for safety reasoning on concurrent programs. Starling reduces concurrent proof outlines to verification conditions in existing sequential theories.

2. A series of program logics built using Starling, which we provide as *frontends* independent of the target sequential theory;

3. *Starling*$_{tool}$, a proof-of-concept tool for automating the generation and discharging of Starling proof obligations using existing sequential solvers: the Z3 SMT solver, HSF Horn-clause solver, and GRASShopper reachability logic solver.

4. A partial mechanisation of parts of Starling and Starling$_{\text{tool}}$ in Coq.

1.2 Structure

The rest of this dissertation consists of the following chapters:

Chapter 2 Technical background needed to follow the rest of the dissertation;

Chapter 3 Discussion of how to build general proof rules for atomic-action concurrency by progressively adapting the *axiom soundness* result from the Views framework;

Chapter 4 The Starling framework for building automatable program logics in terms of *frontends* that implement the above proof rules, and *backends* that interface with underlying sequential logic theories;

Chapter 5 Local-state reasoning for Starling by extending the Views framework, and a prototype frontend for handling proofs with local state;

Chapter 6 Work towards practical frontends that balance automation with local-state reasoning, resulting in the $_{\mathrm{g}}$Starling frontend that forms a basis for the next chapters;

Chapter 7 $\mathrm{Starling}_{\mathrm{tool}}$ (a tool for proving properties of concurrent programs) and $\mathrm{C}_{\mathrm{view}}$ (its C-like input language);

Chapter 8 Validation of $\mathrm{Starling}_{\mathrm{tool}}$ and $\mathrm{C}_{\mathrm{view}}$ by working through case studies, as well as the use of unit tests and Coq mechanisations;

Chapter 9 Review of the above contributions, comparing them against the thesis, and discussion of avenues for work going forwards.

Chapter 2

Background

> The usual way in which we plan today for tomorrow is in yesterday's vocabulary. We do so, because we try to get away with the concepts we are familiar with and that have acquired their meanings in our past experience. Of course, the words and the concepts don't quite fit because our future differs from our past, but then we stretch them a little bit.
>
> Edsger W. Dikjstra, *EWD 1036* [15]

This chapter provides technical background needed to understand the rest of the dissertation. It starts, in § 2.1, by discussing the type of concurrency that this dissertation concerns. It then introduces the field of formal methods for concurrency in § 2.2, before focusing specifically on Floyd/Hoare-style program logics (the formal method type used in the rest of the dissertation) in § 2.3. In closing, it discusses two specific fields this dissertation bridges together: in § 2.4 it outlines the *Views Framework*, a unifying theory of concurrent reasoning; and in § 2.5 it discusses possible approaches for automating program logics.

2.1 Atomicity and atomic actions

This dissertation focuses on atomic-action concurrency. This section expands on this field, as well as the more general idea of *atomicity*.

Atomicity

Atomicity is 'the abstraction that an operation takes effect at a single, discrete instant in time' [16]. That certain actions appear atomic is a commonly required property of concurrent algorithms; one reason is that atomicity prevents data races by disallowing simultaneous accesses to resources being updated. Atomicity can be abstract: an operation that appears atomic at a high level of abstraction may in fact take multiple steps in its implementation.

This dissertation does not directly consider abstract forms of atomicity. Instead, it focuses on proving safety properties of programs that use primitive *atomic actions*. Such actions are guaranteed to be atomic in their manipulation of individual shared-state locations. Showing that these programs establish *abstract* atomicity is out of scope for this dissertation.

Action	C_{view} syntax (§ 7.1)	Sequential equivalent
Fetch	⟨*V*⟩ '=' ⟨*S*⟩	`V = S;`
Store	⟨*S*⟩ '=' ⟨*E*⟩	`S = E;`
Add	⟨*S*⟩ '+=' ⟨*E*⟩	`S = S + E;`
Subtract	⟨*V*⟩ '−=' ⟨*E*⟩	`S = S − E;`
Increment	⟨*S*⟩ '++'	`S = S + 1;`
Decrement	⟨*V*⟩ '−−'	`S = S − 1;`
Fetch-and-increment	⟨*V*⟩ '=' ⟨*S*⟩ '++'	`V = S; S = S + 1;`
Fetch-and-decrement	⟨*V*⟩ '=' ⟨*S*⟩ '−−'	`V = S; S = S − 1;`
Compare-and-swap	'Cas' '(' ⟨*S*⟩ ',' ⟨*V*⟩ ',' ⟨*E*⟩ ')'	`if S == V { S = E; }` `else { V = S; }`

Table 2.1: Examples of atomic actions as used in the dissertation.

Atomic actions

CPU support for atomic actions has existed for decades. The System/370 architecture of the 1970s includes such actions for use 'by programs sharing common storage areas in either a multiprogramming or multiprocessing environment' [17], and, in 1986, Treiber [18] used its compare-and-swap operations to implement a non-blocking stack. This dissertation focuses on a small set of atomic actions seen in modern CPU architectures (see Table 2.1).

Without CPU support, we can still 'mock up' atomic actions using locked sequences of more primitive actions. While there are algorithms, such as Peterson's [19], for making locked critical regions using only atomic assignments, these come with the performance detriment of performing multiple instructions to replace one.

2.2 Formal methods

We can contain the threats that concurrency bugs pose by finding ways to check the correctness of concurrent programs. There are two main approaches: *testing*, where we show that our programs give the expected behaviour in a given set of inputs and environments; and *formal methods*, which are 'mathematically-based languages, techniques, and tools' [20] for specifying and verifying computer systems.

While testing is an open research area, and both approaches are complementary [21] (tests can show faults in formal specifications, for instance), this dissertation focuses on the use of formal methods to specify and verify concurrent programs.

Properties

This dissertation both compares existing formal methods and proposes new ones. We need, then, a set of criteria for evaluating formal methods. This section proposes a set of functional and non-functional properties for doing so.

Expressiveness. Formal methods are only useful if they can express our proofs. They must be expressive enough to encode the program we are proving and any assertions, properties, and mathematics needed for its proof. Too much expressiveness can be as bad as too little, though, as Jones et al. [22] argue through analogy with unstructured programming languages.

Safety and termination correctness. Some methods only prove safety properties (sometimes called 'partial correctness' [23]); they show that programs never produce incorrect results. Other methods can prove termination properties (or 'total correctness'), showing that programs will always, eventually, produce correct results.

Compositionality and modularity. Vafeiadis [24] calls a method *compositional* if we can combine the proofs of a system's sub-components to form a proof of the full system, and *modular* if it allows re-use of proofs in all valid contexts. Not all compositional methods are modular, as Vafeiadis explains; some methods allow for combination of proofs only if each proof exhaustively claims compatibility with each detail of the others.

Compositionality and modularity help us prove at scale. Compositional methods let us make proofs incrementally, component by component. Modular methods let us replace components without breaking the proof of the rest of the system.

Verify-while-develop. Formal methods literature often deals with the checking of existing programs against specifications [25]. This works well when such programs are correct, but not when we must fix mistakes: we must go around a loop of fully developing the program, trying to prove it, finding bugs, and starting afresh.

Instead, de Roever advocates the *verify-while-develop* paradigm: while developing a program from a specification, we prove the correctness of each design decision at the moment we take that decision. We must be able to frame away parts of the program where we have not made those decisions yet, assuming that they behave as specified.

Compositionality and modularity help us achieve verify-while-develop. This is because, for verify-while-develop reasoning, we must break a system up into its key decision points, and prove them in the absence of full proofs of the other points.

Reasoning capabilities

To refine our idea of expressivity, let us consider potential capabilities of formal methods.

Thread modularity. Some methods require that proofs be *thread-modular*. In such methods, each thread must have a separate proof that does not depend on any information about what each other thread is doing, except for general interference invariants [26].

Other methods let us reason more closely about assertion-level thread interactions. Hoenicke *et al.* [27] generalise thread modularity accordingly; a proof is *thread-modular at level* k if it is built from inductive assertions over products of k threads and a non-interference clause specifying that the execution of a $k+1$th thread cannot invalidate any such assertion.

Interference. To reason about correctness of concurrent programs, we must show that anything a thread does only affects other threads in specific, permitted ways. If not, we have no guarantee that any action one thread takes has not been the subject of an unexpected data race or other such interaction from a different thread.

Approaches to interference differ in granularity. At one extreme, some methods do not support concurrency at all. The next step up is *disjoint parallelism*, where we assume that threads have no interference at all. Both of these systems have low expressivity. The other extreme is to force the user to prove manually, for each thread, that each action the thread takes preserves the proof of each other action on each other thread. This scales badly and hurts both modularity and compositionality.

Most methods take a middling approach where we must write a specific protocol that each thread must obey when accessing shared state. This scales, so long as we assume any new operations on shared state obey the protocol.

Separation. Some methods support *separation*: splitting shared resources such that operations on one sub-resource cannot affect the other sub-resource. Given proofs on one such sub-resource, we can *frame* on the other sub-resource to make proofs over the full resource.

Separation helps us establish local and modular reasoning. Under separation, if one thread holds access to one section of memory and a second thread accesses another, then any overlaps between those sub-heaps are in the context of a set of permitted actions. This lets us reason only about whether each thread establishes its own obligations, and whether the permitted actions' effects meet the other threads' expectations.

We usually associate this concept with *separation logic* [28], where separation splits shared heaps into disjoint sub-heaps. However, others have since generalised it and applied it to a wide range of other resources and separation models.

Higher-order and impredicative reasoning. Sometimes, we must parametrise the properties we are proving with other properties. For example, properties over a lock may need to carry the invariant properties of the resource the lock protects. For this, we need *higher-order reasoning*. If these properties are self-referential (for example, the resource can access the same lock that protects it) we need *impredicative reasoning*. Many modern program logics have higher-order reasoning; some, such as *iCAP* [29], have impredicativity.

These features are not essential for concurrent reasoning, and increase the complexity of logics that support them. This said, they enhance our ability to reason about more complex concurrent systems while not ruling out first-order reasoning in the same logic.

Formal method types

There are many types of formal method, each with different approaches to satisfying (or not satisfying) the above properties. This dissertation mainly concerns *program logics*, so the rest of this background focuses on them.

2.3 Floyd/Hoare-style program logics

Program logics use logical reasoning to reduce and discharge proofs that programs obey certain properties. They combine axioms based on the semantics of primitive commands with laws for assembling said commands into a control flow. Program logics are expressive — modern logics support concurrency, interference, separation, and modularity —, but often need a separate effort from program development, and can be hard to automate.

The program logic tradition upon which this work builds derives from the early work of Floyd [30] and Hoare [31]; the rest of this dissertation calls such logics *Floyd/Hoare-style logics*. These logics combine two languages: a command language (normally more abstract than a real programming language), and a mathematical assertion language.

In these logics, we associate some command C with precondition (P) and postcondition (Q) assertions, forming a *Hoare triple* $\{P\}\ C\ \{Q\}$. The meaning of such triples depends on the logic. The original interpretation dealt with safety properties, not liveness: if P holds at the beginning of C, *and* C *terminates*, then Q must hold. In this reading ($\Vdash_{\mathsf{FH}}$), P is a predicate over the state before C, and Q a predicate over the state afterwards[1].

Definition 2.1. If P and Q are predicates on states, and C a relation on states, the Floyd/Hoare-style safety judgement $\Vdash_{\mathsf{FH}}$ of the triple $\{P\}\ C\ \{Q\}$ is:

$$\Vdash_{\mathsf{FH}} \{P\}\ C\ \{Q\} \stackrel{\text{def}}{\iff} \forall \sigma, \sigma'.\, P(\sigma) \wedge C(\sigma, \sigma') \implies Q(\sigma')$$

Floyd/Hoare-style logics have inference rules that let us combine Hoare triples along control flows. Most give rules for repetition, selection, and sequential composition (and some add non-deterministic choice and parallel composition). For example, logics typically have the following sequential-composition rule:

$$\frac{\{P\}\ C_1\ \{R\} \qquad \{R\}\ C_2\ \{Q\}}{\{P\}\ C_1; C_2\ \{Q\}}$$

This lets us treat two agreeing operations in sequence as a 'black box' that respects the first and last conditions. In sequential logics, this is valid because we can assume that the environment has no way to observe or modify the state between C_1 and C_2.

Floyd/Hoare-style reasoning involves proving a specification (a pair of precondition P and postcondition Q) of a command C. We can do so by applying inference rules to show that we can legally compose the primitive commands of C, in the form of triples containing their inherent specifications, into the triple $\{P\}\ C\ \{Q\}$.

Proof outlines. Proof outlines are a compact way to present Floyd/Hoare-style proofs. They consist of the program code to be proven, with assertions inserted between actions and around control flows. Such outlines, resembling Floyd's flowcharts [30], cover each Hoare triple in the program proof (from those of each primitive command to the composed triples for compound statements) while occupying minimal space. Proof outlines feature heavily in this dissertation: for example, in Listing 2.1.

[1] Some treatments, including more recent ones by Hoare [32], support two-state relational postconditions.

'Hoare logic'

The first Floyd/Hoare-style logic was Hoare's *axiomatic basis for computer programming* [31] (or, informally, *Hoare logic*). This logic gives us a simple framework for using predicate logic to reason about programs in a toy sequential programming language. It provides inference rules for assignment, repetition, selection, and sequential composition, amongst others.

Hoare's original logic is intuitive, lightweight, and mature, but supports neither concurrency nor separation. As the conditions of a command are predicate-logic assertions, specifications say nothing about any state not explicitly mentioned — meaning that framing is not possible, and safety reasoning in the face of possible aliasing is difficult and scales badly —, nor can they easily model intangible ideas such as permissions and resources.

Owicki-Gries

The elegance and immediate usability of Hoare's logic shaped early attempts to provide axiomatic reasoning for concurrent programs, including those by Hoare himself [33]. To support such reasoning, logics must address the problem of showing that concurrent actions do not interfere with the proof of other parts of the program. Hoare's work in *Towards a Theory of Parallel Programming* does so by requiring each such action to inhabit a conditional critical region, which is unsuitable for low-level needs.

The *Owicki-Gries* method [34], a more flexible extension of Hoare's work, influenced many program logics in use today. Owicki-Gries adds Dijkstra-style [35] parallel composition (**cobegin** C1 // ... // Cn **coend**); atomicity at the individual-statement level[2]; conditional critical regions (**await** B **then** C), to model synchronisation primitives; and proof rules for the above, including the parallel-composition rule (Definition 2.2).

Definition 2.2 (Owicki-Gries parallel). For commands C1, ..., Cn, the following holds:

$$\frac{\{P_1\}\ \texttt{C1}\ \{Q_1\}, \ldots, \{P_n\}\ \texttt{Cn}\ \{Q_n\} \text{ are interference-free}}{\{P_1 \wedge \cdots \wedge P_n\}\ \textbf{cobegin}\ \texttt{C1 // ... // Cn}\ \textbf{coend}\ \{Q_1 \wedge \cdots \wedge Q_n\}}$$

With this rule, we can prove each triple $\{P_n\}$ Cn $\{Q_n\}$ in a **cobegin** sequentially, so long as we also prove non-interference. To do so, we check that each atomic command (or **await** body) preserves the precondition of any triple that is not its own.

As Owicki–Gries proofs are sequential proofs combined with a non-interference check, they must be thread-modular. This means that, to prove algorithms with interaction between threads, we often have to add auxiliary variables.

[2] This reduces to atomicity at the memory-reference level if programs follow certain conventions.

Consider Listing 2.1, a classic concurrency example where two loops each atomically increment one counter 20 times. Let us prove that the final counter value is 40.

Listing 2.1: Incomplete Owicki–Gries proof of multiple counter increment.

$\{C = 0\}$ **cobegin**
  $\{C \geqslant 0\}$
  `i := 0;` **while** `i < 20` **do** $\{C \geqslant i\}$ `C := C + 1;` $\{C \geqslant i+1\}$ **end**
  $\{C \geqslant 20\}$
//
  $\{C \geqslant 0\}$
  `j := 0;` **while** `j < 20` **do** $\{C \geqslant j\}$ `C := C + 1;` $\{C \geqslant j+1\}$ **end**
  $\{C \geqslant 20\}$
coend $\{C = 40\}$

We cannot prove this using Definition 2.2, as $C \geqslant 20 \wedge C \geqslant 20 \not\Longrightarrow C = 40$. We must make each thread's contribution explicit in the proof, as in listing 2.2.

Listing 2.2: Corrected Owicki–Gries proof of multiple counter increment.

$\{C = 0\}$
`C1 := 0; C2 := 0;`
$\{C = C1 + C2 \wedge C1 = 0 \wedge C2 = 0\}$ **cobegin**
  $\{C = C1 + C2 \wedge C1 = 0\}$
  `i := 0;`
  $\{C = C1 + C2 \wedge C1 = i \wedge i = 0\}$
  while `i < 20` **do**
    $\{C = C1 + C2 \wedge C1 = i \wedge i < 20\}$
    await true then begin `C := C + 1; C1 := C1 + 1;` **end**
    $\{C = C1 + C2 \wedge C1 = i+1 \wedge i < 20\}$
    `i := i + 1`
    $\{C = C1 + C2 \wedge C1 = i+1 \wedge i \leqslant 20\}$
  end
  $\{C = C1 + C2 \wedge C1 = 20\}$
//
  $\{C = C1 + C2 \wedge C2 = 0\}$
  `j := 0;`
  $\{C = C1 + C2 \wedge C2 = j \wedge j = 0\}$
  while `j < 20` **do**
    $\{C = C1 + C2 \wedge C2 = j \wedge j = 0\}$
    await true then begin `C := C + 1; C2 := C2 + 1;` **end**
    $\{C = C1 + C2 \wedge C2 = j+1 \wedge j < 20\}$
    `j := j + 1`
    $\{C = C1 + C2 \wedge C2 = j+1 \wedge j \leqslant 20\}$
  end
  $\{C = C1 + C2 \wedge C2 = 20\}$
coend $\{C = 40\}$

Owicki-Gries was a key step towards the concurrent program logics of today. Its logic is clean and simple, extending Hoare's logic directly rather than replacing it. It adds no bespoke logical constructs, and, while its rules generate a large amount of side-conditions, none of them are particularly difficult to discharge. The method is still under active research: a 2015 paper gave a strengthening of the non-interference property for relaxed memory models [36].

Owicki-Gries has drawbacks compared with modern program logics. Its non-interference check makes it neither compositional nor modular: each action must be checked against the precondition of each other action, and composing another proof would make the check incomplete. Owicki-Gries, then, does not scale to large, verify-while-develop-style proofs. The thread-modularity of Owicki-Gries proofs also causes problems: many thread protocols are not expressible without the use of auxiliary variables or abstraction leakage.

Rely/Guarantee

In 1981, Jones introduced the *rely/guarantee* method [37, 38]. Rely/guarantee gives a more compositional treatment of interference than Owicki–Gries. To do so, Jones adds two new assertions to each Hoare triple: the *rely* $\mathtt{R}$, which specifies the environment interactions under which the triple is stable, and the *guarantee* $\mathtt{G}$, which captures the interference the triple can cause to the environment. Each new assertion is a relation from states before interaction to states after interaction. Though various conventions on how to write the resulting tuples exist, this discussion uses the form $\{\mathtt{P}, \mathtt{R}\}\ \mathtt{C}\ \{\mathtt{Q}, \mathtt{G}\}$ as per Jones et al. [22].

Specifying interactions inside process specifications means that non-interference checks are local and compositional. We see this in definition 2.3, the rely/guarantee parallel rule.

Definition 2.3 (Rely/guarantee parallel rule)**.**

$$\frac{\{\mathtt{P},\ \mathtt{R} \cup \mathtt{G}_2\}\ \mathtt{C}_1\ \{\mathtt{Q}_1,\ \mathtt{G}_1\} \qquad \{\mathtt{P},\ \mathtt{R} \cup \mathtt{G}_1\}\ \mathtt{C}_2\ \{\mathtt{Q}_2,\ \mathtt{G}_2\}}{\{\mathtt{P},\ \mathtt{R}\}\ \mathtt{C}_1 \| \mathtt{C}_2\ \{\mathtt{Q}_1 \wedge \mathtt{Q}_2,\ \mathtt{G}_1 \cup \mathtt{G}_2\}}$$

This is an adaptation of Vafeiadis's [24] parallel-rule interpretation to a Hoare-style presentation. Many other statements of the rule exist, such as that in Jones's original paper. Some generalise the rule to allow different preconditions and relies.

We usually represent the rely and guarantee as two-state predicates. This means we can use predicate reasoning to make parts of a proof fit together, strengthening guarantees and weakening relies until the composition works.

Rely/guarantee forms the basis of many other logics and is still under active research [39]. Modern rely/guarantee sheds the Floyd/Hoare-style formulation for an algebra based on Morgan's refinement calculus [40], where users embed relies and guarantees into the program code. This recasting seeks to yield a cleaner expression of the system and its laws.

Separation logics

Separation logics support native separation reasoning over combinations of *distinct* resources. They contain a *separating conjunction* operator $*$ that is distinct from ordinary conjunction $\wedge$ (which joins two assertions about the *same* resource). Such operators obey a *frame rule*:

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

This rule states that if we have an observation about a command C modifying a resource P into a resource Q, we can put the command and resources into any disjoint context R. Going backwards, should we have a resource $P * R$, we can *frame off* R and apply C, resulting in $Q * R$. This provides the idea of separation we saw earlier.

The original 'separation logic' [28] concerns the separation and framing of heaps. More abstractly, we can form separation logics over *separation algebras*: cancellative, partial commutative monoids over $*$ [41]. This idea of commutative monoids (sets with an identity element and commutative, associative binary operator) as an abstract representation of knowledge appears frequently both in concurrent verification [42, 43] and this dissertation (as Definition 2.5). Cancellativity allows for framing, and partiality lets us handle the possibility of ill-formed combinations of resources.

This more abstract treatment works with models that barely resemble the original heap set-up. Hoare's *graphical models of separation logic* [44], which adapts abstract separation logic to the task of reasoning about programs as trace sets, is an exotic example.

Permissions accounting

Modelling the transfer of intangible *permissions* in program logics is useful for proving algorithms where thread roles change dynamically[3]. Permissions accounting can take many forms, from simple counting models to elaborate algebras and tree-based approaches.

Boyland's *fractional permissions* [45] have been particularly influential on concurrent separation logics such as CAP [46]). Such permissions are unforgeable tokens which can be split and recombined. In the original model, a full permission grants the ability to read and write a value; splitting one creates multiple read-only permissions, preventing data-races.

Marrying rely/guarantee and separation logic

Combining separation and interference in one logic is appealing but non-trivial. Vafeiadis's *RGSep* [24] does so through the 'marriage' of rely/guarantee and separation logic.

Vafeiadis observes that separation logic is useful for reasoning about state (by framing off unchanged areas, for instance) but not interference, and that rely/guarantee works well with interference but requires global reasoning in its specifications. RGSep applies each system according to these strengths: rely/guarantee for shared state, and separation logic for thread-local state. We can see this combination in Figure 2.1, the RGSep parallel rule.

$$\frac{\{P_1,\ R \cup G_2\}\ C_1\ \{Q_1,\ G_1\} \qquad \{P_2,\ R \cup G_1\}\ C_2\ \{Q_2,\ G_2\}}{\{P_1 * P_2,\ R\}\ C_1 \| C_2\ \{Q_1 * Q_2,\ G_1 \cup G_2\}}$$

Figure 2.1: The RGSep parallel rule.

[3] We see one such algorithm, the *atomic reference counter*, in § 8.1.

RGSep originally targeted a toy language called GPPL. The language this dissertation explores descends indirectly from GPPL through the similar *Views* language (Figure 2.3).

Concurrent Abstract Predicates

Based on RGSep, *Concurrent Abstract Predicates* (for short, CAP) aims to improve modularity in concurrent separation logics [46]. CAP is based on *abstract predicates*, a way of separating assertions from their shared-state interpretation. For example, we can state that 'x is a lock and *lock(x)* locks it while keeping it as a lock', instead of 'x points to *false* and *lock(x)* changes it to *true*'. Abstract predicates can take resources, stating some abstract fact on that resource without leaking knowledge of its contents.

CAP's main idea is the *fiction of disjointness*: two separate abstract predicates can refer to parts of the same resource, unlike normal separation logic reasoning which enforces strict disjointness under $*$. Instead of explicit rely and guarantee relations, CAP structures interference in terms of a *interference relation* on shared state, as well as a Boyland-style permissions system used to guard interference transitions. By taking full permission on an interference action, a thread can state that it expects no other thread to perform it, making the rely and guarantee concepts implicit.

CAP's modularity story lets us write proofs under one definition Δ of the abstract predicates, then reuse them under a weaker definition Δ'. This idea gives rise to the *let rule*:

$$\frac{\Delta \vdash \{P_1\}\ C_1\ \{Q_1\} \qquad \dots \qquad \Delta \vdash \{P_n\}\ C_n\ \{Q_n\} \qquad \Delta \vdash \Delta' \qquad \Delta';\ \{P_1\}\ f_1\ \{Q_1\};\ \cdots;\ \{P_n\}\ f_n\ \{Q_n\} \ \vdash\ \{P\}\ C\ \{Q\}}{\{P\}\ \text{let } f_1 = C_1, \dots, f_n = C_n \text{ in } C\ \{Q\}}$$

CAP has tool support through Caper [47], which § 8.4 discusses further.

2.4 The *Views* framework

Though the program logics above share much common structure, such as their Floyd/Hoare heritage, each differs in how it observes the shared state in assertions and context. Each logic needs its own soundness proof; the resulting redundancy has caused concern [12].

The *Concurrent Views Framework* [42], or just *Views*, addresses this problem. Its main idea is the *view*: an abstract unit of information about the shared state of a program, as well as a grant of rights to change it. Views abstract logical formulae, abstract predicates, type judgements, and many other types of observation and permission.

Given a views set, and some other logical parameters defining a reasoning system, *Views* reduces the proof burden from full soundness to a smaller *axiom soundness* property. This property is easier to prove, and the *Views* paper shows this with examples capturing separation logics, type systems, Owicki-Gries reasoning, and rely/guarantee reasoning. That said, *Views*'s imposed structure on reasoning systems makes it unsuitable for some logics: for example, those that use higher-order reasoning, or reasoning over liveness properties.

Views in summary

This section presents *Views* in terms the dissertation uses throughout. The *Views* paper [42] gives a detailed and authoritative account with which this summary is *broadly* compatible.

Views algebras. *Views* uses algebraic structures to abstract over the shape of views in reasoning systems. It also separates the shared-state meaning of facts from their abstract form: systems must provide a *reification function* to map between the two.

Unlike the paper, but in keeping with the Coq mechanisation of *Views*, views sets must here have an equivalence relation $\equiv$ (formally making them views *setoids*, as in Definition A.1), and $\equiv$ appears in many cases where the Views paper used Leibniz equality $=$. This change makes working with views in intuitionistic settings, such as Coq, easier.

Views assumes that the views in a concurrent reasoning system can be combined in some commutative, associative way. This models the combination of facts from local assertions with facts from an external context: for instance, other threads running in parallel with the local code. *Views* represents this requirement using *views semigroups*.

> **Definition 2.4.** A *views semigroup* $(V, \bullet, \equiv)$ is a commutative semigroup with operation $\bullet$, where commutativity and associativity are defined over $\equiv$, and $\bullet$ is compatible with $\equiv$ ($\forall x, y, z : V, . \, x \equiv y \implies x \bullet z \equiv y \bullet z$).
> (Coq: ViewsSemigroup in Starling.Views.Classes)

Reasoning systems sometimes allow for a *global invariant*: a view capturing the minimum knowledge and permissions all part of a program can hold onto at all times. We can represent this idea by adding a unit element to our semigroup. This gives us *views monoids*.

> **Definition 2.5.** A *views monoid* $(V, \bullet, \varepsilon, \equiv)$ is a commutative monoid with operation $\bullet$ and unit ε, with commutativity, associativity, compatibility, and unit laws over $\equiv$.
> (Coq: ViewsMonoid in Starling.Views.Classes)

States and reification. *Views* makes a distinction between the concrete set of shared states and the abstract set of views that map onto them.

> **Definition 2.6.** A *state set* S is a set of all possible shared states in a reasoning system. There are no constraints on the structure of the states.

As views are abstract, and can map to concrete states in complex ways, a *Views* instance must provide a *reification function* that maps each view to the set of states that satisfy it.

> **Definition 2.7.** A *reification function* $\lfloor - \rfloor : V \to \mathbb{P}(S)$, where $(V, \equiv)$ is a setoid, is a function from views to state sets such that $\forall x, y \in V. \; x \equiv y \implies \lfloor x \rfloor = \lfloor y \rfloor$.
> (Coq: Reifier in Starling.Views.Frameworks.Common.Reifier)

```
if (P1) {                          {
   C1();                              assume P1; C1();
} else if (P2) {                   } or {
   C2();                              assume (!P1 && P2); C2();
} else {                           } or {
   C3();                              assume (!P1 && !P2); C3();
}                                  }
```

Figure 2.2: These control flows are equivalent under safety analysis.

⟨Prog⟩ ::= `skip`	no-operation
| < ⟨A⟩ >	atomic action
| ⟨Prog⟩ `;` ⟨Prog⟩	sequential composition
| ⟨Prog⟩ `||` ⟨Prog⟩	parallel composition
| ⟨Prog⟩ + ⟨Prog⟩	nondeterministic choice
| ⟨Prog⟩ *	looping

Figure 2.3: *Views* command language, parametric over an atomic action language A. (Coq: Prog in Starling.Views.Frameworks.Common.Language)

Programming language. Normally, a reasoning system designer would need to prove the system's soundness across the full program language on which the system can reason. However, most languages used in imperative reasoning systems have the same basic control flows: sequential and parallel composition, repetition, and selection.

As this dissertation only considers the proof of safety properties, we can exploit divergence to reduce the control flow set. Conditional branching, for instance, becomes equivalent to deciding nondeterministically which branch to take, then diverging if we chose wrongly. To demonstrate, let `or` be nondeterministic choice, and **`assume`** `P` a command that diverges if `P` is false; then, the programs in fig. 2.2 are equivalent under safety reasoning.

To take advantage of this commonality, *Views* supplies its own, Dijkstra-style programming language. This language (fig. 2.3) resembles GPPL, but without basic commands, and parametrised by an atomic action language provided by the reasoning system.

> **Definition 2.8.** An *atomic action language* A, ranged over by α, is a set of atomic actions supported by a reasoning system. We need not constrain the shape of A, as other definitions impose further structure.

Atomic actions map onto state transformers through a total *semantic function*.

> **Definition 2.9.** A *semantic function* $[\![-]\!] : A \to S \to \mathbb{P}(S)$ maps from atomic actions in language A to non-deterministic state transformers on shared states in state set S.

Lifted sets and functions. The *Views* paper defines lifted sets and functions for use in the rest of the instantiation. One lifting (Definition 2.10), from atomic actions to labels in *Views*'s transition-based semantics, expands the action language to contain an identity label id.

While id appears in many parts of the *Views* meta-theory, it *does not* exist in the programming language. This means that <id> is not expressible; instead, we would use `skip`[4].

Definition 2.10. The *atomic label language* A^{id} is $\mathrm{A} \cup \{\,\mathrm{id}\,\}$, where id is a unique atomic action not in A. Let $\widehat{\alpha}$ denote an arbitrary atomic label (which, unlike α, may be id).

We can lift $[\![-]\!]$ over atomic labels, giving id the expected identity semantics.

Definition 2.11. The *label semantic function* $[\![-]\!]^{\mathsf{id}} : \mathrm{A}^{\mathsf{id}} \to \mathrm{S} \to \mathbb{P}(\mathrm{S})$ lifts $[\![-]\!]$ to labels: $[\![-]\!]^{\mathsf{id}} \stackrel{\text{def}}{=} [\![-]\!] \cup \{\,\mathrm{id} \mapsto \lambda\sigma.\{\,\sigma\,\}\}$.

We often need to apply $[\![\widehat{\alpha}]\!]^{\mathsf{id}}$ to a state *set*, the result being all possible output states we can reach by applying $\widehat{\alpha}$ to any one of the input states. This leads to another lifting over $[\![-]\!]$.

Definition 2.12. The *lifted semantic function* $[\![-]\!]^{*} : \mathrm{A}^{\mathsf{id}} \to \mathbb{P}(\mathrm{S}) \to \mathbb{P}(\mathrm{S})$ lifts $[\![-]\!]^{\mathsf{id}}$ to the domain of state sets: $[\![\widehat{\alpha}]\!]^{*}(\mathrm{S}) \stackrel{\text{def}}{=} \bigcup\{\,[\![\widehat{\alpha}]\!]^{\mathsf{id}}(\sigma) \mid \sigma \in \mathrm{S}\,\}$.

Signatures. The views semigroup, reification function, atomic actions, and semantic function together characterise a reasoning system's proof language. We can see these parameters as distinct from, but closely linked to, the actual proof rule the reasoning system implements.

This dissertation refers to the group of four parameters mentioned above as a *views signature*, as they represent the outwardly visible signature of a reasoning system. This distinction (not present in the original *Views* development) becomes useful later on, as parts of the Starling meta-theory manipulate the two sides of the reasoning system separately.

Definition 2.13. A *views signature* is a tuple $(\mathrm{V}, \mathrm{A}, \mathrm{S}, \bullet, \equiv, \lfloor - \rfloor, [\![-]\!])$, where:

- A is an atomic action language (Definition 2.8);
- $(\mathrm{V}, \bullet, \equiv)$ is a views semigroup (Definition 2.4);
- $\lfloor - \rfloor$ is a reification function (Definition 2.7);
- S is a state set (Definition 2.6);
- $[\![-]\!]$ is a semantic function (Definition 2.9).

Let $\mathsf{Sig}(\mathrm{V}, \mathrm{A}, \mathrm{S})$, ranged over by s, be the set of all signatures over V, A, and S.

(Coq: Signature in Starling.Views.Frameworks.Common.Signatures)

When the signature from which we draw a *Views* parameter x is ambiguous, this dissertation uses the notation $\mathrm{s}.\mathrm{x}$ to show that it comes from signature s. For example, $\mathrm{s}.\lfloor \nu \rfloor$ expresses the reification of a view ν, but explicitly using s's reification.

[4] One reason for this distinction between id and `skip` is semantics: `skip` behaves as the terminating element of the language's structural operational semantics, while atomic actions have an explicit transition to `skip`.

Axioms. Once the reasoning system designer has supplied *Views* with an signature, the next step is to define the possible axioms of the reasoning system. These are Hoare triples that witness each safe combination of views and atomic action. Since Views provides the control flows that combine atomic actions into programs, it also provides the inference rules needed to turn the axioms into a full program logic.

In *Views*, axioms are always a triple of a precondition view, a label, and a postcondition view. This dissertation refers to such triples, axioms or otherwise, as *atomic Hoare triples*.

Definition 2.14. Given a views semigroup with carrier V and an atomic action language A, the *atomic Hoare triple* set $\mathsf{AHoare}(V, A) \stackrel{\text{def}}{=} V \times A^{\mathsf{id}} \times V$.

Let us use the notation $\langle p \rangle\ \widehat{\alpha}\ \langle q \rangle$ to refer to a triple $(p, \widehat{\alpha}, q) \in \mathsf{AHoare}(V, A)$.

To instantiate *Views*, we must provide an *axiomatisation*: the set of all valid axioms of the reasoning system. We need not give the set extensionally; typically, one would instead define axiom schemata for each atomic action.

Definition 2.15. An *axiomatisation* is a set $T \subseteq \mathsf{AHoare}(V, A)$ of atomic Hoare triples that the atomic proof rule of some reasoning system admits.

Axiom soundness. The final step is to show that every axiom $\langle p \rangle\ \widehat{\alpha}\ \langle q \rangle$ in the axiomatisation is *axiom sound*. Informally, this means that when we run $\widehat{\alpha}$ in a state satisfying p, we get a state satisfying q (so, local safety); in addition, where the initial state satisfies $p \bullet v$ for any external context v, we get a state satisfying $q \bullet v$ (so, context preservation).

The axiom soundness definition comes in two steps. First, we consider the *action judgement*, which tells us when a single axiom is axiom sound. Then, we can define axiom soundness by quantifying the action judgement over all axioms in the axiomatisation.

Both parts of the action judgement have the form 'when we run $\widehat{\alpha}$ in a state satisfying x, we get y'. This, effectively, lifts the judgement $\Vdash_{\mathsf{FH}}$ to views. To make the definition of the action judgement clearer, let us give this sub-judgement its own notation:

Definition 2.16. The *views–Floyd/Hoare judgement* $s, \widehat{\alpha} \Vdash_{\mathsf{VFH}} \{p\}\{q\}$ holds for a signature $s : \mathsf{Sig}(V, A, S)$ and atomic Hoare triple $\langle p \rangle\ \widehat{\alpha}\ \langle q \rangle$ when $[\![\widehat{\alpha}]\!]^*(\lfloor p \rfloor) \subseteq \lfloor q \rfloor$.

The views–Floyd/Hoare judgement has several useful properties:

$$\lfloor p \rfloor \subseteq \lfloor q \rfloor \iff s, \mathrm{id} \Vdash_{\mathsf{VFH}} \{p\}\{q\} \qquad \text{(entailment)}$$

$$((\lfloor p \rfloor \subseteq \lfloor p' \rfloor) \wedge\ s, \widehat{\alpha} \Vdash_{\mathsf{VFH}} \{p'\}\{q\}) \implies s, \widehat{\alpha} \Vdash_{\mathsf{VFH}} \{p\}\{q\} \qquad \text{(left consequence)}$$

$$((\lfloor q' \rfloor \subseteq \lfloor q \rfloor) \wedge\ s, \widehat{\alpha} \Vdash_{\mathsf{VFH}} \{p\}\{q'\}) \implies s, \widehat{\alpha} \Vdash_{\mathsf{VFH}} \{p\}\{q\} \qquad \text{(right consequence)}$$

$$((p \equiv p') \wedge (q \equiv q') \wedge s, \widehat{\alpha} \Vdash_{\mathsf{VFH}} \{p'\}\{q'\}) \implies s, \widehat{\alpha} \Vdash_{\mathsf{VFH}} \{p\}\{q\} \qquad \text{(proper)}$$

A definition of the action judgement, and then axiom soundness, follows. They differ from the original presentation by making the signature an explicit parameter.

Definition 2.17. The *action judgement* $s, \widehat{\alpha} \Vdash \{p\}\{q\}$ holds for a signature $s : \mathsf{Sig}(V, A, S)$ and atomic Hoare triple $\langle p \rangle\ \widehat{\alpha}\ \langle q \rangle$ when:

$$(s, \widehat{\alpha} \Vdash_{\mathsf{VFH}} \{p\}\{q\}) \quad \wedge \quad (\forall v \in V.\ s, \widehat{\alpha} \Vdash_{\mathsf{VFH}} \{p \bullet v\}\{q \bullet v\})$$

Definition 2.18. *Axiom soundness* holds for a signature s and axiomatisation T if:

$$\forall \langle p \rangle\ \widehat{\alpha}\ \langle q \rangle \in T. \quad s, \widehat{\alpha} \Vdash \{p\}\{q\}$$

As axiom soundness depends only on showing a modified, interference-aware version of $\Vdash_{\mathsf{FH}}$, it concerns only safety. As a consequence, *Views*-based reasoning systems can only soundly reason about safety properties, not liveness.

Views instances. Pairs of signatures and axiomatisations yield *Views* framework instances.

Definition 2.19. A *views instance* is a tuple (s, T) where, for some V, A, and S, s is a signature in $\mathsf{Sig}(V, A, S)$, and T an axiomatisation (subset of $\mathsf{AHoare}(V, A)$) over it. Let $\mathsf{Inst}(V, A, S)$, ranged over by i, be the set of such instances over a particular V, A, and S.

If an instance's axioms are axiom-sound, we call that instance *sound*. Sound instances correspond, through the *Views* metatheory, to sound reasoning systems.

As with signatures, we use $i.x$ to clarify that parameter x comes from instance i: for example, $i.T$ represents i's axiomatisation. For brevity, we can write i in any place where we would write $i.s$: for any x other than s and T, $i.x$ means $i.s.x$.

Monoidal action judgement. If V carries a views monoid, we must consider the context $v = \varepsilon$ in the action judgement. In that case, the right-hand side of the judgement is $\widehat{\alpha} \Vdash_{\mathsf{VFH}} \{p \bullet \varepsilon\}\{q \bullet \varepsilon\}$, which is equivalent to $\widehat{\alpha} \Vdash_{\mathsf{VFH}} \{p\}\{q\}$, and thus establishes the left-hand side of the judgement. We can then simplify the action judgement:

Definition 2.20. The *monoidal action judgement* $s, \widehat{\alpha} \Vdash_{\mathrm{m}} \{p\}\{q\}$ holds when:

$$\forall v \in V.\ s, \widehat{\alpha} \Vdash_{\mathsf{VFH}} \{p \bullet v\}\{q \bullet v\}$$

This gives us a corresponding simplification of axiom soundness: an instance (s, T) has axiom soundness if s is over a views monoid and $\forall \langle p \rangle\ \widehat{\alpha}\ \langle q \rangle \in T.\ s, \widehat{\alpha} \Vdash_{\mathrm{m}} \{p\}\{q\}$. While this dissertation does not use these simplifications directly, Chapter 3 does use similar results.

The *Views* program logic

By defining its own programming language given a set of atomic actions and their semantics, *Views* can provide a sound program logic over said language for each sound instance. This logic maps each axiom in the axiomatisation to a primitive judgement, and includes many of the standard rules of Concurrent Separation Logic. Figure 2.4 gives the logic's main rules.

$$\frac{}{\{p\}\ \mathsf{skip}\ \{p\}}$$

(a) Skip

$$\frac{\langle p\rangle\ \alpha\ \langle q\rangle \in T}{\{p\}\ <\alpha>\ \{q\}}$$

(b) Atomic

$$\frac{\{p\}\ C\ \{q\}}{\{p \bullet f\}\ C\ \{q \bullet f\}}$$

(c) Frame rule

$$\frac{\{p\}\ C\ \{p\}}{\{p\}\ C{*}\ \{p\}}$$

(d) Iteration

$$\frac{\{p\}\ C_1\ \{q\} \qquad \{p\}\ C_2\ \{q\}}{\{p\}\ C_1{+}C_2\ \{q\}}$$

(e) Nondeterminism

$$\frac{\{p\}\ C_1\ \{r\} \qquad \{r\}\ C_2\ \{q\}}{\{p\}\ C_1 ; C_2\ \{q\}}$$

(f) Sequential composition

$$\frac{\{p_1\}\ C_1\ \{q_1\} \qquad \{p_2\}\ C_2\ \{q_2\}}{\{p_1 \bullet p_2\}\ C_1 \,||\, C_2\ \{q_1 \bullet q_2\}}$$

(g) Parallel composition

$$\frac{\langle p\rangle\ \mathsf{id}\ \langle p'\rangle \in T \qquad \{p'\}\ C\ \{q\}}{\{p\}\ C\ \{q\}}$$

(h) Left consequence

$$\frac{\{p\}\ C\ \{q'\} \qquad \langle q'\rangle\ \mathsf{id}\ \langle q\rangle \in T}{\{p\}\ C\ \{q\}}$$

(i) Right consequence

Figure 2.4: The main rules of the *Views* program logic [42, Def. 8].

Figures 2.4(c) and 2.4(g) are interesting for two reasons. First, they neatly capture the main concepts from Concurrent Separation Logic. Second, they show that, in *Views*, the act of combining views from multiple threads and that of adding more context views within the same thread are the same operation, $\bullet$. In fact, *Views* has no concept of threads at all, but reasoning systems can encode them if needed.

Consequence in both directions. From left and right consequence, we can derive a rule that applies consequence in both directions simultaneously:

$$\frac{\langle p\rangle\ \mathsf{id}\ \langle p'\rangle \in T \qquad \{p'\}\ C\ \{q'\} \qquad \langle q'\rangle\ \mathsf{id}\ \langle q\rangle \in T}{\{p\}\ C\ \{q\}}$$

Generalised parallel composition. Though fig. 2.4(g) only permits the composition of two processes, the associativity of both parallel composition and the views semigroup means that we can trivially construct an Owicki-Gries-style n-process parallel rule:

$$\frac{\{p_1\}\ C_1\ \{q_1\} \qquad \dots \qquad \{p_n\}\ C_n\ \{q_n\}}{\{p_1 \bullet \dots \bullet q_n\}\ C_1 \,||\, \dots \,||\, C_n\ \{q_1 \bullet \dots \bullet q_n\}}$$

2.5 Automating program-logic proofs

This dissertation aims to present automation-friendly proof rules for concurrency. Doing so requires understanding possible schemes for automating such rules. These schemes include:

- expanding the rule into a set of predicates decidable by some satisfiability solver, quantifying over any shared state, and asking the solver to try to refute each predicate;

- expanding the rule into a constraint system, and using a constraint solver;
- partial automation by embedding the rule into an interactive prover.

Automation by satisfiability solver

In this approach, we first decompose the proof into the applications of the atomic proof rule, by applying the program logic and expanding out quantifications where possible. Then, we expand the proof rule applications into *verification conditions* in some decidable theory (for example: Boolean logic, or linear arithmetic) and send them to a solver for that theory.

One possible sub-approach is to use a *satisfiability modulo theories* (SMT) solver, such as *Z3* [48]. These solvers are fast, accept predicates over a broad variety of theories, and are widely used either directly or through *intermediate verification languages* such as *Boogie* [49].

Automation by constraint solver

In this approach, we decompose the proof into a system of partially uninterpreted basic constraints. We then give the *whole system* to a constraint solver, which tries to solve the system in one go by finding definitions for the uninterpreted constraints.

The advantage to this scheme is that it can infer definitions for uninterpreted terms — in practice, this means that we can omit auxiliary assertions, or leave assertions open for strengthening and weakening to make the proof work. On the other hand, this approach needs a closed set of proof terms, which makes it hard to modify and compose proofs.

One sub-approach is to use *Horn clauses* (disjunctions in which at most one literal is in a positive position) as the shape of each constraint [50]. Some or all of the terms in each Horn clause can be left abstract, in which case the solver will try to find definitions such that the system reaches a fixed point. Solvers such as *Threader* [51] use this approach.

Horn clauses are restrictive in shape, and it can be hard to arrange proof terms into pure Horn clauses. Some solvers, such as *HSF* [52], accept more loosely-structured clauses, but still impose some structure to allow fixed-point solution.

Interactive automation

While the above two approaches give us full automation, they rely on an unambiguous decomposition of proofs into predicates or constraints. Automating this decomposition restricts the expressivity of reasoning systems, as we see in § 3.4.

Instead, we can trade-off automation for expressivity, embedding the program logic into an interactive prover that gives the user access to the logic's inference rules, but leaves decisions about when to use them (at least partly) to the user.

This approach can either target a general theorem prover such as Coq (for example, *FCSL* [53]) or a custom prover for the logic (for example, *Caper* [47]).

Chapter 3

Proof-Specific Views Instances

> We are motivated not by an abstract ideal of elegance, but by the practical problem of reasoning about real algorithms. Rigorous reasoning is the only way to avoid subtle errors in concurrent algorithms, and we want to make reasoning as simple as possible by making the underlying formalism simple.
>
> Leslie Lamport, on *TLA* [54]

To achieve our goal of automated concurrency reasoning, we can design a new program logic, then prove it sound (using the *Views* framework mentioned in § 2.4), then build tooling for it. By optimising the logic for automation over expressivity, we can achieve fuller automation than efforts over existing logics (such as Caper [47]).

This approach has risks. We risk our system becoming obsolete with the development of more advanced logics. We risk not having the flexibility to adapt our system to the needs of industrial programmers. We also risk making the wrong trade-off between automation, expressivity, and elegance. For example, while the Owicki-Gries rule in § 2.3 is elegant and straightforward to automate, it has many issues (thread modularity, lack of compositionality, and so on) that a *practical* automatable logic must overcome.

This chapter proposes a new approach for building program logics. The approach uses *Views* in an unusual way: instead of building single views instances for existing reasoning systems, it derives a new one for each program proof, using it as a core part of the proof argument. This lets us then apply the resulting *Views* program logic; § 3.2 discusses how to do so automatically, reducing proof outlines with a particular structure to a set of atomic Hoare triples that we can check against our instance's axiomatisation.

Each instance built must be axiom-sound. As we control the construction of the instances, we can explore routes to axiom soundness that exploit properties of the instances' structure. *Free views instances*, where we define the axiomatisation directly over the *Views* action judgement, appear in § 3.4. In § 3.5, this approach leads to *axiomatisation templates*: functions from proof-specific signatures to axiomatisations, built so that soundness depends on a well-defined interface to which the signature must adhere.

Most of the later chapters use ideas from this chapter. Chapters 4 to 6 use its results to develop Starling, a framework for building automatable program logics, and a family of increasingly complex logics based on it. These logics then lead to $\text{Starling}_{\text{tool}}$ in Chapter 7.

3.1 Disposable views instances

We normally build views instances as part of the meta-theory development of a reasoning system, to prove that every expressible statement in that system is sound. As such, we let V be the set of *all* legal facts from that system; A the set of *all* atomic actions; and T the set of *all* possible atomic Hoare triples over those parameters. If we prove axiom soundness, *Views* gives us our desired soundness result.

Instead of proving soundness for a whole reasoning system with one instance, we can build a new, 'disposable', instance for each proof. The way we carry out the proof — showing that the proof decomposes into a set of axioms that inhabit the instance's axiomatisation, and showing that the instance is axiom-sound — is effectively the same as if we used *Views* in the usual manner. The difference is that we need not fully define a reasoning system and prove it sound: we can just define the sets of views, atomic actions, and axioms we need for one proof (with caveats, which we see below).

This re-casting of *Views* gives us flexibility in several different areas. We can just check axiom-soundness for the axioms that the proof uses (a relatively small, and potentially bounded, set), rather than the set of all expressible axioms of a whole reasoning system. As each instance need not be generally applicable, we can more easily apply approaches and techniques that depend on the specifics of the views and atomic actions in use.

Which views semigroup? We cannot always use the set of all assertions in a program proof as its views semigroup. To see why, consider this triple from a hypothetical reference counter[1]:

$$\{\text{reference}\}\ \langle \text{refs} := \text{refs} + 1 \rangle\ \{\text{reference} \bullet \text{reference}\}$$

Here, the set of assertions is $\{\,\text{reference}, \text{reference} \bullet \text{reference}\,\}$; this set cannot form a semigroup as it is not closed over $\bullet$. We must take an over-approximation: for example,

$$V = \left\{ \overbrace{\text{reference} \bullet \cdots \bullet \text{reference}}^{n \text{ times}} \;\middle|\; n \in \mathbb{Z}^+ \right\}$$

Another consideration is that our choice of semigroup determines the set of contexts we consider when deciding non-interference. This gives us compositionality: by including the views from a program Q when proving program P, and vice versa, we show that P and Q can run in parallel without violating each other's assertions. For example, we could include the views of some client of the reference counter in our proof, to show that increasing the reference count cannot violate any of that client's assertions.

3.2 Decomposing proof outlines

To achieve our goal of automatic verification of whole-program proofs, we need a way to decompose those proofs into obligations that we can discharge against the proofs' disposable views instances. This decomposition must be sound and automatable.

[1] § 8.1 investigates an actual proof of such a counter.

```
⟨P⟩ ::= { ⟨V⟩ } skip { ⟨V⟩ }                                  no-operation
      | { ⟨V⟩ } < ⟨A⟩ > { ⟨V⟩ }                               atomic action
      | { ⟨V⟩ } ( ⟨P⟩ ) * { ⟨V⟩ }                             iteration
      | { ⟨V⟩ } frame ⟨V⟩ in ( ⟨P⟩ ) { ⟨V⟩ }                  frame rule
      | { ⟨V⟩ } ( ⟨P⟩ ; ⟨P⟩ ) { ⟨V⟩ }                         sequential composition
      | { ⟨V⟩ } ( ⟨P⟩ || ⟨P⟩ ) { ⟨V⟩ }                        parallel composition
      | { ⟨V⟩ } ( ⟨P⟩ + ⟨P⟩ ) { ⟨V⟩ }                         nondeterministic choice
```

(Coq: Outline in Starling.ProgramProof)

Figure 3.1: *Views* proof outline grammar.

Views already gives us a program logic over a GPPL-style programming language. Let us then define the decomposition as a function that, given proof outlines over that language, returns a set of atomic Hoare triples. If each triple forms an axiom in a sound views instance, the proof goes through. This method will need adapting to target more realistic programming languages later, but the general approach remains the same.

The next task is to define the shape of proof our automated rules will accept. This leads to a process for reducing proofs in that shape to atomic Hoare triples. The appendices give the formal reasoning for said development.

Proof outlines

Proof outlines, as we saw in § 2.3, are a compact way to present proofs in Floyd/Hoare-style logics. This section considers proof outlines as first-class structures on which we can base automated proof rules and program logics.

Basing a program logic on proof outlines is not a new idea. De Roever et al., for example, propose it as a way to handle the non-interference requirements of Owicki-Gries reasoning [25, §10.4]. As *Views* handles non-interference in an elegant, abstract way, modulo the correct application of a set of inference rules, we can instead use proof outlines to help us decide which rules to apply when, allowing for full automation.

Structure. To begin, let us impose a rigid structure on outlines: a precondition and postcondition must surround each unit of the *Views* language seen in Figure 2.3. As these outlines place assertions at each control flow in the program, they correspond to de Roever et al.'s *annotated programs* [25, §10.4]. Figure 3.1 shows the resulting grammar.

The rigidity of this grammar makes automation easier at the expense of making the proofs harder to write. Chapters 4 and 7 consider relaxed forms of this structure.

Examples. The proof outline $\{p\}\ (\{r\}\ \langle a\rangle\ \{s\}; \{s\}\ \langle b\rangle\ \{t\})\ \{q\}$ stands for the Hoare triples:

$$\{r\}\ \langle a\rangle\ \{s\}, \qquad \{s\}\ \langle b\rangle\ \{t\}, \qquad \{p\}\ (\langle a\rangle; \langle b\rangle)\ \{q\}$$

To see how to interpret the frame-rule construct in Figure 3.1, consider the following:

$$\{p\}\ (\{r\}\ \langle a\rangle\ \{s\}; \{s\}\ \mathsf{frame}\ v\ \mathsf{in}\ (\{x\}\ \langle b\rangle\ \{y\})\ \{t\})\ \{q\}$$

This proof states that we: split the view s into $x \bullet v$; use the frame rule to preserve v across the proof of b, giving us $y \bullet v$; then recombine $y \bullet v$ into t. In Hoare triples, this is:

$$\{x\}\ \langle b\rangle\ \{y\}, \xrightarrow{\text{frame with } v} \{x \bullet v\}\ \langle b\rangle\ \{y \bullet v\}, \xrightarrow{\text{split/combine}} \begin{array}{l}\{r\}\ \langle a\rangle\ \{s\},\\ \{p\}\ (\langle a\rangle; \langle b\rangle)\ \{q\}\end{array}$$

Access notation. Let us define notation to access parts of an outline without unfolding it:

Definition 3.1. For all outlines $o = \{p\}\ c\ \{q\}$, let $o.p \stackrel{\text{def}}{=} p$, $o.c \stackrel{\text{def}}{=} c$, and $o.q \stackrel{\text{def}}{=} q$.

Proving proof outlines automatically

To prove outlines by hand, we show that there exists some series of applications of the program logic that deconstructs the outline into its primitive operations, such that the Hoare triple at each stage matches the corresponding triple in our outline. We then show that each primitive operation is valid; in *Views*, this means showing that its triple is a valid axiom.

To prove $\{p\}\ (\{r\}\ \langle a\rangle\ \{s\}; \{s\}\ \langle b\rangle\ \{t\})\ \{q\}$ this way, we first unfold it by *Views*-logic rules:

$$\dfrac{\dfrac{\langle p\rangle\ \mathrm{id}\ \langle r\rangle \in \mathsf{T} \qquad \dfrac{\dfrac{\langle r\rangle\ a\ \langle s\rangle \in \mathsf{T}}{\vdash \{r\}\ \langle a\rangle\ \{s\}}\ \text{Ax.} \quad \dfrac{\langle s\rangle\ b\ \langle t\rangle \in \mathsf{T}}{\vdash \{s\}\ \langle b\rangle\ \{t\}}\ \text{Ax.}}{\vdash \{r\}\ \langle a\rangle; \langle b\rangle\ \{t\}}\ \text{SC}}{\vdash \{p\}\ \langle a\rangle; \langle b\rangle\ \{t\}}\ \text{LC} \qquad \langle t\rangle\ \mathrm{id}\ \langle q\rangle \in \mathsf{T}}{\vdash \{p\}\ \langle a\rangle; \langle b\rangle\ \{q\}}\ \text{RC}$$

Then, we show that $\langle r\rangle\ a\ \langle s\rangle$, $\langle s\rangle\ b\ \langle t\rangle$, $\langle p\rangle\ \mathrm{id}\ \langle r\rangle$, and $\langle t\rangle\ \mathrm{id}\ \langle q\rangle$ are valid axioms; this step depends on the views instance.

For automation, this approach has two problems. First, we need two decision processes: one to apply program-logic steps, and another for the axioms. Second, arbitrary application of program-logic steps is hard to automate: we may need to apply constructs like frame rule and consequence in the proof without syntactic cues. How do automated techniques choose when to apply them, and which parameters to use?

Instead, we can use the outline's rigid structure to choose which logic rule to apply at each step. We then automatically apply the rule of consequence where needed; the resulting id-axioms discharge any obligations we need to apply the control-flow rules. This lets us reduce the proof outline into a set of uniform axioms, which we can then hand to a solver.

Let us define a function, oflat, that takes a proof outline and produces a set of atomic Hoare triples. This set combines the atomic Hoare triples contained in the outline with the entailments $\langle p\rangle\ \mathrm{id}\ \langle q\rangle$ resulting from the automatic application of the rule of consequence when stepping through control flows. Appendix A.4 contains a formal derivation of oflat using the *Views* program logic.

Definition 3.2. The *outline flattening function* $\mathsf{oflat} : \mathsf{P} \to \mathbb{P}((V \times A \times V))$ recursively reduces a proof outline into a set of atomic Hoare triples, as follows:

$$
\begin{aligned}
\mathsf{oflat}(\{p\}\ \mathsf{skip}\ \{q\}) = &\quad \{\langle p\rangle\ \mathrm{id}\ \langle q\rangle\} \\
\mathsf{oflat}(\{p\}\ \langle\alpha\rangle\ \{q\}) = &\quad \{\langle p\rangle\ \alpha\ \langle q\rangle\} \\
\mathsf{oflat}(\{p\}\ (\{p'\}\ C\ \{q'\})^*\ \{q\}) = &\quad \mathsf{oflat}(\{p'\}\ C\ \{q'\}) \\
\mathsf{oflat}(\{p\}\ (\mathsf{frame}\ r\ \mathsf{in}\ (\{p'\}\ C\ \{q'\})\ \{q\}) = &\quad \mathsf{oflat}(\{p'\}\ C\ \{q'\}) \\
&\cup \{\langle p\rangle\ \mathrm{id}\ \langle p' \bullet r\rangle, \langle q' \bullet r\rangle\ \mathrm{id}\ \langle q\rangle\} \\
&\cup \{\langle p\rangle\ \mathrm{id}\ \langle p'\rangle, \langle p'\rangle\ \mathrm{id}\ \langle q\rangle, \langle q'\rangle\ \mathrm{id}\ \langle p'\rangle\} \\
\mathsf{oflat}(\{p\}\ \{p'\}\ C_1\ \{r\}; \{s\}\ C_2\ \{q'\})q\ \{)\} = &\quad \mathsf{oflat}(\{p'\}\ C_1\ \{r\}) \\
&\cup\ \mathsf{oflat}(\{s\}\ C_2\ \{q'\}) \\
&\cup\ \{\langle p\rangle\ \mathrm{id}\ \langle p'\rangle, \langle r\rangle\ \mathrm{id}\ \langle s\rangle, \langle q'\rangle\ \mathrm{id}\ \langle q\rangle\} \\
\mathsf{oflat}(\{p\}\ \{p_1\}\ C_1\ \{q_1\} \parallel \{p_2\}\ C_2\ \{q_2\}\ \{q\}) = &\quad \mathsf{oflat}(\{p_1\}\ C_1\ \{q_1\}) \\
&\cup\ \mathsf{oflat}(\{p_2\}\ C_2\ \{q_2\}) \\
&\cup\ \{\langle p\rangle\ \mathrm{id}\ \langle p_1 \bullet p_2\rangle, \langle q_1 \bullet q_2\rangle\ \mathrm{id}\ \langle q\rangle\} \\
\mathsf{oflat}(\{p\}\ \{p_1\}\ C_1\ \{q_1\} + \{p_2\}\ C_2\ \{q_2\}\ \{q\}) = &\quad \mathsf{oflat}(\{p_1\}\ C_1\ \{q_1\}) \\
&\cup\ \{\langle p\rangle\ \mathrm{id}\ \langle p_1\rangle, \langle q_1\rangle\ \mathrm{id}\ \langle q\rangle\} \\
&\cup\ \mathsf{oflat}(\{p_2\}\ C_2\ \{q_2\}) \\
&\cup\ \{\langle p\rangle\ \mathrm{id}\ \langle p_2\rangle, \langle q_2\rangle\ \mathrm{id}\ \langle q\rangle\}
\end{aligned}
$$

(Coq: vcs in Starling.ProgramProof)

Applying oflat to $\{p\}\ (\{r\}\ \langle a\rangle\ \{s\}; \{s\}\ \langle b\rangle\ \{t\})\ \{q\}$ gives us the axioms:

$$
\begin{aligned}
&\mathsf{oflat}(\{p\}\ (\{r\}\ \langle a\rangle\ \{s\}; \{s\}\ \langle b\rangle\ \{t\})\ \{q\}) \\
&= \begin{pmatrix} \{\langle p\rangle\ \mathrm{id}\ \langle r\rangle, \langle s\rangle\ \mathrm{id}\ \langle s\rangle, \langle q\rangle\ \mathrm{id}\ \langle t\rangle, \langle t\rangle\ \mathrm{id}\ \langle q\rangle\} \\ \cup\ \mathsf{oflat}(\{r\}\ \langle a\rangle\ \{s\})\ \cup\ \mathsf{oflat}(\{s\}\ \langle b\rangle\ \{t\}) \end{pmatrix} \\
&= \{\langle p\rangle\ \mathrm{id}\ \langle r\rangle, \langle s\rangle\ \mathrm{id}\ \langle s\rangle, \langle q\rangle\ \mathrm{id}\ \langle t\rangle, \langle t\rangle\ \mathrm{id}\ \langle q\rangle, \langle r\rangle\ a\ \langle s\rangle, \langle s\rangle\ b\ \langle t\rangle\}
\end{aligned}
$$

Except for the trivial entailment $\langle s\rangle\ \mathrm{id}\ \langle s\rangle$, these axioms are those reached by hand-proof.

3.3 Case study: Peterson's algorithm

This part of the dissertation uses Peterson's algorithm [19] (*Peterson*, for short) as a running example[2]. *Peterson* is a classic solution to the problem of two-thread mutual exclusion: ensuring that only one of two threads can access a given shared resource. It uses three variables: two flags capturing intent to access the resource, and a turn counter capturing which thread has the ability to do so.

Figure 3.2 quotes the original, high-level, algorithm text. On the left, we have thread A; on the right, thread B. Each line has a number Pl, where P is the thread and l the adjacent

[2] We do not yet consider a proof outline for the algorithm; Chapter 8 introduces one as a C_{view} case study.

A1 *QA* := true	**B1** *QB* := true
A2 *TURN* := B	**B2** *TURN* := A
A3 wait until !*QB* or *TURN*=A	**B3** wait until !*QA* or *TURN*=B
(Critical Section)	
A4 *QA* := false	**B4** *QB* := false

Figure 3.2: Peterson's algorithm [19], annotated with thread and line numbers.

line in that thread's code. Let Linum be the set of such numbers; each corresponds to the moment of time just after its corresponding line has executed. The state after line 4 is the same as that before line 1, so we can use A4 and B4 in both cases.

A specification for Peterson's algorithm

Let us write down the specification for *Peterson* that we intend to prove. For *Peterson* (and other mutual exclusion algorithms that we see later), the specification is:

> There exists an abstract resource Lock such that each thread gains a copy of Lock *immediately before*, and retains it throughout, its critical section; and there is no execution of any number of threads in parallel such that more than one simultaneous copy of Lock can exist.

The uniqueness and duration of a Lock implies that at most one thread may be in its critical section at any given time. This specification *does not* require Locks release at the end of the critical section; this is a liveness property. It also says nothing about whether holding Lock permits invariant-violating access to any other resource; while program logics such as *iCAP* [29] do support this form of reasoning, it is out of scope for this dissertation.

In *Peterson*, a Lock arises when the current thread's Q flag is true, and either the other thread's Q flag is false or the turn counter has given the current thread priority.

A signature for Peterson's algorithm

Peterson serves to show various applications of this chapter's ideas, each resulting in views instances that capture proofs for the algorithm. These instances share the same signature (views semigroup, atomic actions, state set, and action semantics); let us now build it.

Let S be the set of records with type $(\mathtt{QA} : \mathbb{B}, \mathtt{QB} : \mathbb{B}, \mathtt{TURN} : \{\mathtt{A}, \mathtt{B}\})$, assuming that the atomic actions have the expected semantics over these sets. Next, consider the assertions we need for the proof (leading to a view algebra). Proofs of *Peterson* rely on the following relationships over the threads' positions in the algorithm and the shared state:

Flags At A1 up to and including A3, we know that QA is `true` (and similarly for thread B and QB). At A4, we know that it is `false` (similarly for B). Since each thread maintains its own flag, these assertions are always stable.

Turn priority When thread A is in the critical section (at $A3$) and thread B is waiting to enter (at $B2$), $\texttt{TURN}$ must be A, as thread B started waiting *after* thread A last modified $\texttt{TURN}$. The reverse also holds.

Singleton threads To assert that the code of a thread T cannot run on more than one actual thread, we must forbid all pairs (Tx, Ty) for arbitrary x and y (including where $x = y$).

Mutual exclusion To assert that the critical section has mutual exclusion, we must forbid the presence of the pair $(A3, B3)$ (both threads are in the critical section).

Our view algebra should be able to capture both the assertions in the proof text (which relate to one thread only) and the relationships given by the above statements (which relate to zero, one, or two threads). This algebra must also be a monoid, which complicates any direct encoding as line-number pairs. One approach is to let V be the set of multisets[3] over Linum, with the empty multiset as ε and multiset sum as $\bullet$. The 'singleton threads' statement ensures that at most one position for each thread appears in any views considered.

To define $\lfloor - \rfloor$, we must consider every possible combination of positions. Though this set is unbounded, we can translate our assertions into a small, bounded number of mappings between 'defining' multisets (effectively, patterns that we can observe in the views we are trying to define) and state sets. This can take the form of a partial function $d : V \rightharpoonup S$:

$$
\mathrm{d} \stackrel{\text{def}}{=} \left\{ \begin{array}{ll}
\text{Flags:} & \\
\quad \wr A1 \int \mapsto \{\, \sigma \mid \sigma.\mathtt{QA} \,\} & \quad \wr B1 \int \mapsto \{\, \sigma \mid \sigma.\mathtt{QB} \,\} \\
\quad \wr A2 \int \mapsto \{\, \sigma \mid \sigma.\mathtt{QA} \,\} & \quad \wr B2 \int \mapsto \{\, \sigma \mid \sigma.\mathtt{QB} \,\} \\
\quad \wr A3 \int \mapsto \{\, \sigma \mid \sigma.\mathtt{QA} \,\} & \quad \wr B3 \int \mapsto \{\, \sigma \mid \sigma.\mathtt{QB} \,\} \\
\quad \wr A4 \int \mapsto \{\, \sigma \mid \neg\sigma.\mathtt{QA} \,\} & \quad \wr B4 \int \mapsto \{\, \sigma \mid \neg\sigma.\mathtt{QB} \,\} \\
\text{Turn priority:} & \\
\quad \wr A3, B2 \int \mapsto \{\, \sigma \mid \sigma.\mathtt{TURN} = A \,\} & \quad \wr A2, B3 \int \mapsto \{\, \sigma \mid \sigma.\mathtt{TURN} = B \,\} \\
\text{Singleton threads:} & \\
\quad \forall x, y.\, \wr Ax, Ay \int \mapsto \emptyset & \quad \forall x, y.\, \wr Bx, By \int \mapsto \emptyset \\
\text{Mutual exclusion:} & \quad \wr A3, B3 \int \mapsto \emptyset
\end{array} \right\}
$$

Each mapping encodes part of our informal assertion set: the one-position mappings capture the flag assertions, and the three sets of two-position mappings capture turn priority, singleton-threads, and mutual exclusion assertions. Let us define the reifier as the conjunction of all such definitions that match some sub-multiset of the reified view:

$$\lfloor \nu \rfloor \stackrel{\text{def}}{=} \bigcap \{\, \sigma \mid (u, \sigma) \in \mathrm{d} \wedge u \subseteq_m \nu \,\}$$

Under this scheme, for example, $\wr A3, B2 \int$ would collect the mappings for $\wr A3 \int$, $\wr B2 \int$, and $\wr A3, B2 \int$; after taking the intersection of all four definitions and simplifying, we get an effective definition of $\{\, \sigma \mid \sigma.\mathtt{QA} \wedge \sigma.\mathtt{QB} \wedge \sigma.\mathtt{TURN} = A \,\}$. This method of building a reifier plays a major role later on (see Definition 3.12).

[3] We explore multisets as views further in § 6.1.

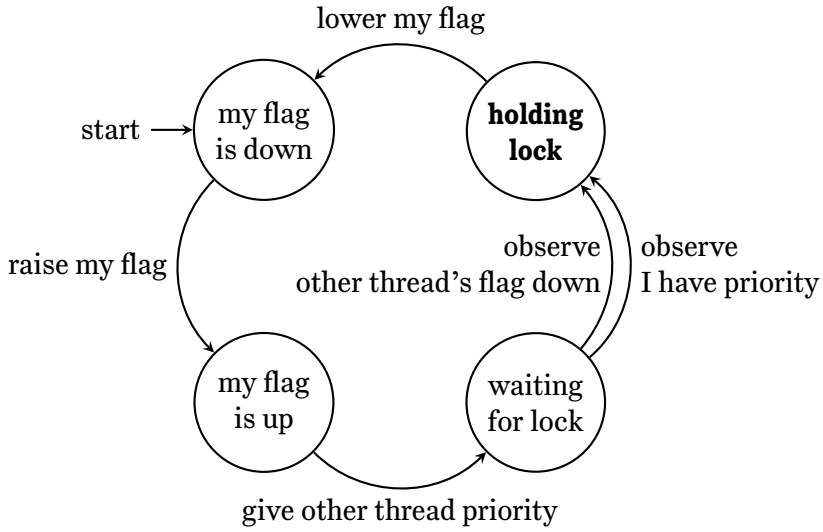


Figure 3.3: The finite-state machine underlying a single thread in Peterson's algorithm.

Peterson's algorithm as a finite-state machine

As well as the program-counter-based treatment of *Peterson* above, we can view the algorithm as consisting of two threads with mirror images of the same finite-state automaton (Figure 3.3). This high-level view becomes useful for more sophisticated instances of this scheme (specifically, in § 6.4), where thread state machines often form a source of proof assertions.

3.4 Free views instances

The action judgement (Definition 2.17) captures the essence of concurrent atomic-action correctness. It shows that an atomic action is both sequentially safe and *also* does not interfere with any views held by the context *when combined with local assertions*:

$$\mathsf{s}, \mathsf{c} \Vdash \{\mathsf{p}\}\{\mathsf{q}\} \stackrel{\text{def}}{=} \overbrace{\mathsf{s}, \mathsf{c} \Vdash_{\mathsf{VFH}} \{\mathsf{p}\}\{\mathsf{q}\}}^{\text{Sequential safety}} \wedge \overbrace{\forall \nu \in \mathsf{s}.V.\ \mathsf{s}, \mathsf{c} \Vdash_{\mathsf{VFH}} \{\mathsf{p} \bullet \nu\}\{\mathsf{q} \bullet \nu\}}^{\text{Non-interference modulo local observations}}$$

This view of soundness almost directly generalises the Owicki-Gries method[4]. As Owicki-Gries works as a standalone proof rule, we can infer that the action judgement itself forms the core of a proof rule, expressed as the axiomatisation of a views instance. Such *free views instance*s — so named as they are the minimal sound instance over a given signature — capture the elegance of Owicki-Gries, while generalising it to arbitrary signatures.

Definition 3.3. For all signatures $\mathsf{s} : \mathsf{Sig}(V, A, S)$, we define the *free views instance* $\mathsf{finst}(\mathsf{s})$ as the instance $(\mathsf{s}, \{\, \langle \mathsf{p} \rangle\ \mathsf{c}\ \langle \mathsf{q} \rangle \mid \mathsf{c} \Vdash \{\mathsf{p}\}\{\mathsf{q}\} \,\})$.

Free views instances are interesting for two reasons. First, no matter how we fix its remaining parameters, a free views instance is axiom-sound by construction.

[4] One difference is that the non-interference clause must show the state ends in $\mathsf{q} \bullet \nu$: informally, we must both preserve ν *and* show that locally finishing in q is consistent with ν.

Theorem 3.1. All free views instances are axiom-sound.

Second, free instances are *maximal*: the axiomatisation of each sound views instance subsets that of the free instance over the same signature.

Theorem 3.2. For all sound views instances $\mathtt{i}$, $\mathtt{i}.\mathsf{T} \subseteq \mathsf{finst}(\mathtt{i}).\mathsf{T}$.

We can show this by noting that an axiom is in $\mathsf{finst}(\mathtt{i})$ if, and only if, it passes the action judgement: $\mathtt{i}$ is sound, so all of its axioms do so. (Appendix B.1 formalises this property.)

Using free views instances in practice

Consider the free views instance over the *Peterson* signature we sketched in § 3.3, and the triple $\langle A4 \rangle$ `QA := true` $\langle A1 \rangle$. This triple captures thread A's first action: raising its flag. To prove this, we show that it forms a free-instance axiom by the action judgement:

$$[\![\mathtt{QA := true}]\!]^*(\lfloor \wr A4 \int \rfloor) \subseteq \lfloor \wr A1 \int \rfloor \quad \wedge \quad \forall \nu.\, [\![\mathtt{QA := true}]\!]^*(\lfloor \wr A4 \int \bullet \nu \rfloor) \subseteq \lfloor \wr A1 \int \bullet \nu \rfloor$$

The first part is straightforward: after applying $\mathtt{d}$ and substituting out, we arrive at:

$$\left\{ \sigma' \;\middle|\; \begin{array}{l} \exists \sigma.(\sigma, \sigma') \in [\![\mathtt{QA := true}]\!] \\ \wedge\, \sigma.\mathtt{TURN} \in \{1,2\} \wedge \sigma.\mathtt{QA} = \mathrm{false} \end{array} \right\} \subseteq \left\{ \sigma' \;\middle|\; \begin{array}{r} \sigma'.\mathtt{TURN} \in \{1,2\} \\ \wedge\, \sigma'.\mathtt{QA} = \mathrm{true} \end{array} \right\}$$

Unless `QA := true` has a surprising semantics, this obligation checks out.

The non-interference part is troublesome, as we must discharge it *for every* ν *in* V. As V is the carrier of a views semigroup, it must close over $\bullet$; for every u and ν in the set, $u \bullet \nu$ is *also* in the set. In algebras such as multisets, where this construction almost always yields a new and distinct view, this makes the quantification unbounded.

If we try to automate free-instance reasoning, the quantification may expand indefinitely, and the automated process may diverge. Avoiding this requires finding ways to restrict the non-interference quantification to a bounded set. § 3.6 discusses this, but let us first take a diversion to discuss the set-up for doing so in a sound manner.

3.5 Axiomatisation templates

The disposable-views approach reduces each program proof to an axiom-soundness proof over the created instance. Producing such proofs from first principles every time would neither scale nor be amenable to automation. To address this, we impose some structure on how we construct the instances; we move closer to the original *Views* idea of instances per reasoning systems, but keep much of the flexibility of our approach.

Recall that each views instance contains a signature (Definition 2.13) and axiomatisation. The signature collects the parts of an instance that represent the structure of assertions, actions, and their translation into state sets; the axiomatisation tells us which observations (over signature elements) are valid according to some reasoning system. We can, then, see the signature as the main variable between proofs, and focus on constraining the axiomatisation.

Let us construct each disposable-instance axiomatisation by taking a subset of an instantiation of a signature-parametrised *axiomatisation template*. This lets us reduce the

axiom-soundness proof to showing that the construction is sound over all possible signatures, while keeping some flexibility in how we define the signature itself.

Definition 3.4. An *axiomatisation template* $\mathsf{P} : \mathsf{Sig}(V, A, S) \nrightarrow \mathsf{T}$ maps each signature s that satisfies the particular template's requirements to an axiomatisation. We call an axiomatisation template *sound* if $(s, \mathsf{P}(s))$ is axiom-sound for all s in $\mathrm{dom}\ \mathsf{P}$.

This chapter develops three templates: the *free* (Definition 3.5), *adjoint* (Definition 3.6), and *defining-views* (Definition 3.12) templates. Each imposes progressively stronger restrictions on signatures, but adds properties that are useful for automation.

Soundness by subsetting, and the free template

When building templates, we must show that any views instances they generate are axiom-sound. We can do so from first principles on each template, but this becomes tedious as we make increasingly large-scale changes. Instead, we can show that, over all compatible signatures, the template yields a subset of the axioms produced by an existing, sound template.

Theorem 3.3. For all axiomatisation templates P, P':

$$\frac{\forall s \in \mathrm{dom}\ \mathsf{P}.\ \mathsf{P}(s) \subseteq \mathsf{P}'(s) \qquad \text{P' sound}}{\text{P sound}}$$

This argument relies on the existence of a known-sound template. As a result, we need at least one template with a full soundness argument. As lifting the action judgement to an axiomatisation produces sound-by-construction, maximal free views instances, we can just re-cast the lifting as a (free) template.

Definition 3.5. The *free axiomatisation template* freeT is $\lambda s. \{\, \langle p \rangle\ c\ \langle q \rangle) \mid s, c \Vdash \{p\}\{q\} \,\}$.

3.6 Automation-friendly templates

We can transform the free template, step-by-step, to ease its automation. The first step is to re-arrange the non-interference part of the template to have a single, universally quantified view in the head, making further rewrites easier and opening up automation by Horn solver. The second is to bound the context set over which we quantify the non-interference part by restricting the set of views that carry information not carried by their sub-views alone.

Unlike the free template, soundness for these new templates requires extra structure in the target signatures. This comes, when needed, in the form of new algebraic classes.

The adjoint template

The non-interference part of the free template conjoins the atomic Hoare triple's precondition and postcondition with the context view v. As we do not constrain the effects of $\bullet$, we must recompute both sides of the non-interference condition each time we choose a new v, wasting effort. Worse, quantifying on *part* of the final view $q \bullet v$ means that we cannot judge whether to consider v without also considering q, making context reduction difficult.

We can rewrite the free template by introducing a new *goal view*, $g = q \bullet v$ — so named as it captures the final intended view of this state. We then re-express both $q \bullet v$ and $p \bullet v$ in terms of g, and quantify over g instead of v. Replacing $p \bullet v$ requires new structure in the views semigroup: an operator $\setminus$ that acts like a semantic adjoint to $\bullet$ (behaving similarly to *monus* or *residual* operators in some varieties of commutative monoid).

With such an operator, we can express v as $g \setminus q$, and so $p \bullet v$ becomes $p \bullet (g \setminus q)$. This construct strongly resembles a *weakest precondition*, and so the dissertation refers to it as such throughout. Applying these changes gives us the *adjoint* template.

Definition 3.6. The *adjoint template*, defined over all s with a *subtractive* $s.V$, is:

$$\lambda s.\, \{\, \langle p \rangle\ c\ \langle q \rangle \mid (s, c \Vdash_{\mathsf{VFH}} \{p\}\{q\}) \ \wedge\ (\forall g \in V.\ s, c \Vdash_{\mathsf{VFH}} \{p \bullet (g \setminus q)\}\{g\})\, \}$$

We define $\setminus$, and subtractive views semigroups, below.

Defining $\setminus$. To avoid overly constraining $\setminus$, we can define it through an adjoint relationship with $\bullet$. The obvious relationship is $a \equiv b \bullet c \iff a \setminus b \equiv c$; this property holds on, for example, integers ($a = b + c \iff a - b = c$), and admits the rewrite we need (from $p \bullet v$ to $p \bullet (g \setminus q)$). This is stronger than we need, and rules out certain otherwise-valid views models: for example, constructive sets cannot guarantee the full adjoint property.

The actual definition weakens the above in two ways. First, as we are subsetting the template, and the rewrite occurs in a negative position, we still preserve soundness even if the rewrite *expands* the view — so long as the reification expands monotonically with it. As such, the definition concerns view *inequalities*, not equivalences. This mirrors separation logic [28], where $a \implies b * c \iff a \mathrel{-\!\!*} b \implies c$. Second, the property need not hold in the backwards direction, as we need only show the soundness (not completeness) of the rewrite.

The adjoint operator uses a similar residuation property to that of separation logic, but over $\setminus$ and a new *view inclusion* operator $\sqsubseteq$. Each of these operators induces a new class of views algebra: *ordered* (for $\sqsubseteq$) and *subtractive* (for $\setminus$) semigroups.

Ordered and subtractive views semigroups

In some semigroups, we can order views by the abstract quantities of information they represent. In our *Peterson* example, we can order $\wr A1, B2 \int$ (telling us two pieces of information: the positions of both threads) above $\wr A1 \int$, but below $\wr A1, A1, B2 \int$. This order is partial: for instance, $\wr A1 \int$ and $\wr B2 \int$ represent disjoint pieces of information.

We can capture this with *ordered views semigroups*: views semigroups with a pre-order[5], $\sqsubseteq$, representing such an information ordering. In ordered views semigroups, $\equiv$ becomes the induced equivalence over $\sqsubseteq$: two items have equivalence if, and only if, $\sqsubseteq$ orders them both ways. This gives us an antisymmetry-style property: $a \sqsubseteq b \wedge b \sqsubseteq a \iff a \equiv b$. Ordered views semigroups also require that conjoining a view to both sides of an ordering preserves that ordering, and $\sqsubseteq$ orders each view below any $\bullet$ involving it.

[5] As the reasoning does not use view equality, it makes no sense to require antisymmetry (partial order).

Definition 3.7. An algebra $(V, \bullet, \sqsubseteq, \equiv)$ is an *ordered views semigroup* if $(V, \bullet, \equiv)$ is a views semigroup, and the following laws hold:

$$\begin{array}{rcll} & & a \sqsubseteq a & (\sqsubseteq\text{-reflexivity}) \\ a \sqsubseteq b \ \wedge\ b \sqsubseteq c & \implies & a \sqsubseteq c & (\sqsubseteq\text{-transitivity}) \\ a \equiv b & \implies & a \sqsubseteq b & (\equiv\text{-}\sqsubseteq) \\ a \sqsubseteq b \ \wedge\ b \sqsubseteq a & \implies & a \equiv b & (\sqsubseteq\text{-}\equiv) \\ a \sqsubseteq b & \implies & (a \bullet c) \sqsubseteq (b \bullet c) & (\bullet\text{-}\sqsubseteq\text{-increasing}) \\ & & a \sqsubseteq (a \bullet b) & (\bullet\text{-}\sqsubseteq\text{-inflation}) \end{array}$$

(Coq: PreOrder in Coq.Classes.RelationClasses)

(Coq: OrderedViewsSemigroup in Starling.Views.Classes)

Subtractive views semigroups. Not all view algebras have a subtraction operator, and so those that do form a new class.

Definition 3.8. An algebra $(V, \bullet, \setminus, \sqsubseteq, \equiv)$ is a *subtractive views semigroup* if $(V, \bullet, \sqsubseteq, \equiv)$ is an ordered views semigroup, and $\setminus : V \to V \to V$ obeys the following laws:

$$\begin{array}{rcll} a \sqsubseteq b & \implies & (a \setminus c) \sqsubseteq (b \setminus c) & (\setminus\text{-}\sqsubseteq\text{-increasing}) \\ a \sqsubseteq b \bullet c & \implies & (a \setminus b) \sqsubseteq c & (\sqsubseteq\text{-residual-forwards}) \end{array}$$

These laws are weak (for example, they define no relationships between a and $(a \setminus b) \bullet b$), but suffice for now. The later *separating views semigroup* class (Definition 6.1) adds further laws that bring $\setminus$ closer to the usual definition of subtraction operators.

Compatibility. Showing that the adjoint template yields subsets of the free template (for Theorem 3.3) depends on one more property: removing q from $p \bullet (q \bullet v)$ must preserve any information that v contributes to the reification. As this property concerns reification (and we can achieve it by constraining $\lfloor - \rfloor$ independently of the views algebra itself), its definition is separate from the views algebra classes.

Definition 3.9. A signature is *adjoint compatible* if:

$$\forall p, q, v \in V.\ \lfloor p \bullet v \rfloor \subseteq \lfloor p \bullet ((q \bullet v) \setminus q) \rfloor$$

(Coq: adjoint_compat in Starling.ProofRules)

Theorem 3.4. For all adjoint compatible signatures s, the adjoint template yields subsets of the free template. (Coq: adjoint_strengthens_free in Starling.ProofRules)

To get compatibility, we can add structure to either the reification or the views algebra. If $x \sqsubseteq y$ always implies that $\lfloor y \rfloor \subseteq \lfloor x \rfloor$, then the properties of adjoint semigroups give us compatibility without needing to further constrain the algebra itself. Conversely, if $x \sqsubseteq (y \bullet x) \setminus y$, we gain compatibility through the algebra itself: as the adjoint property already

entails $(y \bullet x) \setminus y \sqsubseteq x$ (through $y \bullet x \sqsubseteq x \bullet y$), we get an equivalence, and can then use this with $\bullet$-compatibility to show that $p \bullet v \equiv p \bullet ((q \bullet v) \setminus q)$.

Running example. To use this new template in *Peterson*, we must show that multisets are subtractive, ordered views semigroups. The appropriate views algebra instances for multisets appear in § 6.1; for now, let us assume that two operators exist: $\setminus_m$, the adjoint of multiset union (serving as $\setminus$); and $\subseteq_m$, the inclusion order on multisets (serving as $\sqsubseteq$).

Next, we need adjoint compatibility. If we construct reification from the intersection of $d(u)$ for all $u \subseteq_m v$, as v expands so does the number of matching definitions, and the resulting set of states satisfying $\lfloor - \rfloor(v)$ contracts monotonically. As a result, we get compatibility without investigating the views algebra.

We can again try to discharge $\langle \wr A4 \int \rangle$ `Q1 := true` $\langle \wr A1 \int \rangle$; this time, the obligation is:

$$[\![\texttt{Q1 := true}]\!]^*(\lfloor \wr A4 \int \rfloor) \subseteq \lfloor \wr A1 \int \rfloor \quad \wedge \quad \forall g.\, [\![\texttt{Q1 := true}]\!]^*(\lfloor \wr A4 \int \bullet (g \setminus \wr A1 \int) \rfloor) \subseteq \lfloor g \rfloor$$

Recall that g here is the goal view we saw earlier. We can explore examples of non-interference sub-obligations generated by choosing specific values of g. First, let $g = \wr A1 \int$:

$$\begin{aligned} & [\![\texttt{Q1 := true}]\!]^*(\lfloor \wr A4 \int \bullet (\wr A1 \int \setminus \wr A1 \int) \rfloor) && \subseteq \lfloor \wr A1 \int \rfloor \\ = \; & [\![\texttt{Q1 := true}]\!]^*(\lfloor \wr A4 \int \rfloor) && \subseteq \lfloor \wr A1 \int \rfloor \end{aligned}$$

This is the same as the local-correctness obligation from earlier. Now, consider $g = \wr B4 \int$:

$$\begin{aligned} & [\![\texttt{Q1 := true}]\!]^*(\lfloor \wr A4 \int \bullet (\wr B4 \int \setminus \wr A1 \int) \rfloor) && \subseteq \lfloor \wr B4 \int \rfloor \\ = \; & [\![\texttt{Q1 := true}]\!]^*(\lfloor \wr A4, B4 \int \rfloor) && \subseteq \lfloor \wr B4 \int \rfloor \end{aligned}$$

This expands to:

$$\left\{ \sigma' \;\middle|\; \begin{array}{l} \exists \sigma.(\sigma, \sigma') \in [\![\texttt{Q1 := true}]\!] \\ \wedge\, \sigma.\texttt{TURN} \in \{1, 2\} \\ \wedge\, \sigma.\texttt{Q1} = \text{false} \wedge \sigma.\texttt{Q2} = \text{false} \end{array} \right\} \subseteq \left\{ \sigma' \;\middle|\; \begin{array}{r} \sigma'.\texttt{TURN} \in \{1, 2\} \\ \wedge\, \sigma'.\texttt{Q2} = \text{false} \end{array} \right\}$$

On the one hand, the adjoint template yields smaller terms that are closer in shape to constraint-solver input. On the other, the quantification over g remains unbounded and difficult to discharge in an automatable manner. We need further changes to the template.

The defining-views template

When automating the adjoint template, the quantification over g poses problems. Solvers do not understand views natively, and thus cannot discharge the quantification themselves; we must expand views into a form, such as Boolean formulae, that they do understand. As the adjoint rule assumes nothing about the relationship between each view and its meaning over shared states, we would need to expand out each possible context into such a form.

A problem with this approach is that g ranges over V, which is closed over $\bullet$ (for every g_1 and g_2 in V, $g_1 \bullet g_2$ is in V and distinct from both) and thus unbounded in size. We also assume nothing about the relationship between $\lfloor g_1 \rfloor$, $\lfloor g_2 \rfloor$, and $\lfloor g_1 \bullet g_2 \rfloor$, and so cannot use

results from larger contexts to prove results about smaller contexts and vice versa. The adjoint rule gives us no *context reduction*: restricting the values of g we must consider.

Owicki-Gries, in contrast, does have context reduction. If we check non-interference against a precondition P_1 in one thread, and also against a precondition P_2 in another thread, we need not also check against the combined context $P_1 \wedge P_2$, as it carries no extra obligations. We can, then, prove an outline for an unbounded number of threads by checking non-interference against the (finite) set of preconditions in the outline.

Context reduction in the style of Owicki-Gries puts strong restrictions on the shape of the views semigroup, which limit generality. We need some structure to have *any* context reduction, though, so we both rewrite the adjoint template *and* impose restrictions on the signature that let us infer such relationships.

Defining views. One approach to context reduction is to split the information and rights inside a views algebra into a finite number of pieces, then tie each to a specific *defining view*. Every time a view has a piece of information, it includes the respective defining view, and vice versa. This way, we define each view using a finite number of defining views.

Let us tie each defining view to a set of matching states and contain the resulting pair in a *definer*. For now, the definition of definers is abstract.

Definition 3.10. A *definer* $d : V \rightharpoonup \mathbb{P}(S)$ is a partial function mapping each defining view v to a set of states. Intuitively, each set $d(v)$ contains a particular state σ if, and only if, σ is compatible with the *specific* shared-state knowledge and rights contribution made by v — that is, not including those of its subviews.

This is precisely the role of the function d in our Peterson's algorithm example; the specific combinations of line numbers in its domain are, then, defining views.

Defining reification. When using a definer, the reification of each view v is the intersection of the state-sets of each defining view $u \sqsubseteq v$. This generalises the approach we used in § 3.3.

Definition 3.11. The *definer reification* function $\mathsf{dReify} : (V \leftrightarrow S) \to V \to \mathbb{P}(S)$ is:

$$\mathsf{dReify}(d)(v) \stackrel{\text{def}}{=} \bigcap \{\, d(u) \mid u \in \operatorname{dom} d \wedge u \sqsubseteq v \,\}$$

A state σ is in $\mathsf{dReify}(d)(v_1)$ if it is in the image, in d, of each defining view $u \sqsubseteq v_1$.

Defining the template. The defining-views template differs from the adjoint template in three ways. First, it assumes that the incoming signature's reifier is equivalent to $\mathsf{dReify}(d)$ for some d. Second, it applies context reduction in the quantification, taking g from the domain of d and not V. Third, instead of showing that the post-state inhabits $\mathsf{dReify}(d)(g)$, the new template checks it against one definition of g. This works since the former just checks all definitions for all $u \sqsubseteq g$, and the new quantification considers each such definition.

As the new template uses $d(g)$ instead of $\mathsf{dReify}(d)(g)$, it cannot use $\Vdash_{\mathsf{VFH}}$ in the non-interference part, and instead must range over state sets directly.

Definition 3.12. The *defining-views template*, given some definer d, is:

$$\lambda \mathrm{s}. \left\{ \langle \mathrm{p} \rangle\ \widehat{\alpha}\ \langle \mathrm{q} \rangle \;\middle|\; \begin{array}{l} \mathrm{s}, \widehat{\alpha} \Vdash_{\mathsf{VFH}} \{\mathrm{p}\}\{\mathrm{q}\} \\ \wedge\ (\forall (\mathrm{g}, \sigma_{\mathrm{g}}) \in \mathrm{d}.\ [\![\widehat{\alpha}]\!]^*(\mathsf{dReify}(\mathrm{d})(\mathrm{p} \bullet (\mathrm{g} \setminus \mathrm{q}))) \subseteq \sigma_{\mathrm{g}}) \end{array} \right\}$$

where $\mathrm{s}.V$ is a subtractive view semigroup, and $\mathrm{s}.\lfloor - \rfloor \equiv \mathsf{dReify}(\mathrm{d})$.

(Coq: defining_views_template in Starling.ProofRules)

Theorem 3.5. For all signatures s where, for some definer d, $\mathrm{s}.\lfloor v \rfloor = \mathsf{dReify}(\mathrm{d})(v)$ for all v, the defining-views template strengthens the adjoint template.

(Coq: defining_views_strengthens_adjoint in Starling.ProofRules)

Peterson. Let us apply the defining-views approach to our running example. Recall that § 3.3 gave $\lfloor - \rfloor$ in terms of a definer function d: this set-up is already compatible with defining-views. This time, the obligation for our example triple $\langle \wr A4 \int \rangle$ `Q1 := true` $\langle \wr A1 \int \rangle$ is:

$$\begin{array}{c} [\![\texttt{Q1 := true}]\!]^*(\mathsf{dReify}(\mathrm{d})(\wr A4 \int)) \subseteq \mathsf{dReify}(\mathrm{d})(A1) \\ \wedge \forall (\mathrm{g}, \sigma_{\mathrm{g}}) \in \mathrm{d}.\ [\![\texttt{Q1 := true}]\!]^*(\mathsf{dReify}(\mathrm{d})(\wr A4 \int \bullet (\mathrm{g} \setminus A1))) \subseteq \sigma_{\mathrm{g}} \end{array}$$

The quantifier over g in this non-interference obligation expands into 31 conditions: eight corresponding to each single line number, three corresponding to forbidden pairs of numbers across both threads, and twenty corresponding to cases where the same thread has two line numbers active (accounting for symmetry). If we, once again, take $\mathrm{g} = \wr A1 \int$ as an example, the defining-views template gives us:

$$\begin{array}{rl} & [\![\texttt{Q1 := true}]\!]^*(\lfloor \wr A4 \int \bullet (\wr A1 \int \setminus \wr A1 \int) \rfloor) \subseteq \mathrm{d}(\wr A1 \int) \\ = & [\![\texttt{Q1 := true}]\!]^*(\lfloor \wr A4 \int \rfloor) \subseteq \mathrm{d}(\wr A1 \int) \end{array}$$

The main difference between this obligation and that produced by the adjoint template is the right-hand side: instead of requiring the set of final states to be a subset of the reification, we now require it to be a subset of the definition of $\wr A1 \int$. While this seemingly weakens the obligation, in practice the outer quantification over defining views does the job originally performed by the subview intersection inside $\lfloor - \rfloor$. This reduces redundancy in the verification conditions, and makes them more amenable to modelling as Horn clauses, but results in a less obvious connection between conditions and the contexts that produce them.

Notes on views monoids

This dissertation mostly explores signatures over views monoids, not arbitrary semigroups. This has two advantages: first, it provides a natural encoding for global invariants in a defining-views context; second, it lets us simplify the proof rule templates accordingly.

Global invariants. When using the defining-views template with views monoids, ε does not represent a complete lack of knowledge about the shared state, but instead the *baseline* knowledge. First, because $\varepsilon \sqsubseteq v$ for all views v, the reification of every view v includes the

definition of ε. Second, the defining reification of ε need not be S: in fact, by substituting ε for v in Definition 3.11, and noting that $u \sqsubseteq \varepsilon \iff u \equiv \varepsilon$, we arrive at the observation that:

$$\mathsf{dReify}(d)(\varepsilon) = \bigcap \{\, \sigma \mid (u, \sigma) \in d \wedge u \equiv \varepsilon \,\}$$

Defining-views monoids, then, let us define global invariants by mapping definitions to ε.

Simplifying templates. Views monoids let us express empty contexts as $v = \varepsilon$. If we substitute this into the non-interference part of the free template, we get $s, \alpha \Vdash_{\mathsf{VFH}} \{p\}\{q\}$, and can delete the sequential-safety part. This reflects the simplification we saw in Definition 2.20.

We can make a similar transformation to the adjoint and defining-views templates. As hinted-at in our running example, the right value of g for the adjoint template is q; we get this by substituting ε for v in the expansion of g. For the defining-views template, we must separately consider each sub-view of q.

3.7 Summary

This chapter used *Views* to create a general scheme for building concurrent proof rules. In it, we observed that we can build views instances on the fly by combining signatures taken from proofs with axiomatisations built from rule templates, and that we can build automatable templates by refining the action judgement.

The chapter explored three templates: free, adjoint, and defining-views. Given a compatible views signature, these templates prove sequential safety and non-interference of atomic actions. We designed defining-views, in particular, for automation through unfolding its obligations into verification conditions that an external solver can discharge.

The chapter concluded with a method to reduce whole-program proofs, given as structured Floyd-style outlines, into repeated applications of such a proof rule.

Joining these parts gives us a general scheme for automatic proof of concurrent programs. The next chapters expand upon this: Chapters 4 to 6 build Starling, a more elaborate framework for building automatable program logics, atop the defining-views template. Chapter 7 shows how to use Starling as the theory underlying a tool for verifying concurrent programs.

Chapter 4

The Starling Framework

While the scheme in Chapter 3 — particularly the *defining-views* template — helps us produce automatable reasoning systems with a degree of separation between proof-specific and proof-independent concerns, there are areas in which a more heavyweight logical framework would help further. For example, the scheme places views and atomic actions — elements specific to the proof — alongside the state model, which, when automating by targeting an existing solver, is part of that solver's underlying theory. Without some form of abstraction layer, this tightly couples proofs to backends.

This chapter proposes Starling, a design for a logical framework on top of the template-based rule scheme of Chapter 3. This framework builds in a separation of concerns between the *frontend* (the solver-independent logical machinery used by the proof author) and the *backend* (the solver theory, normally in the form of a decision procedure over $\Vdash_{\mathsf{FH}}$).

The frontend and backend combine, along with outline flattening § 3.2, to build a pipeline from proof to verification result. Each stage in the pipeline has a degree of interchangeability; for example, in § 8.1, we use several different backends with only minimal frontend changes.

This chapter starts by defining backends (§ 4.1) and frontends (§ 4.2). One question is how to prove Starling pipelines sound; this chapter considers doing so by building an axiom-sound Views instance such that verification of atomic triples in the pipeline entails the existence of corresponding axioms in the instance's axiomatisation. We can build these instances using templates, and the frontends in this dissertation closely resemble the templates on which their soundness is based. Figure 4.1 outlines the pipeline and soundness argument.

The rest of this chapter builds towards $_{\mu}$Starling, an example frontend. Frontends must re-express views and atomic actions in forms the backend solver will understand: *syntactic definers*, in § 4.3, form a scheme for doing so. The frontend itself appears in § 4.4. $_{\mu}$Starling is a prototype for the more expressive (but less general) $_{\mathrm{lo}}$Starling frontend built in Chapter 5, and the $_{\mathrm{g}}$Starling frontend built in Chapter 6.

Starling serves as the meta-theory justification for the tool, $\mathrm{Starling}_{\mathrm{tool}}$, discussed in Chapter 7. This tool uses a variation of $_{\mathrm{g}}$Starling as the frontend, and existing off-the-shelf solvers, such as Z3, as the backends.

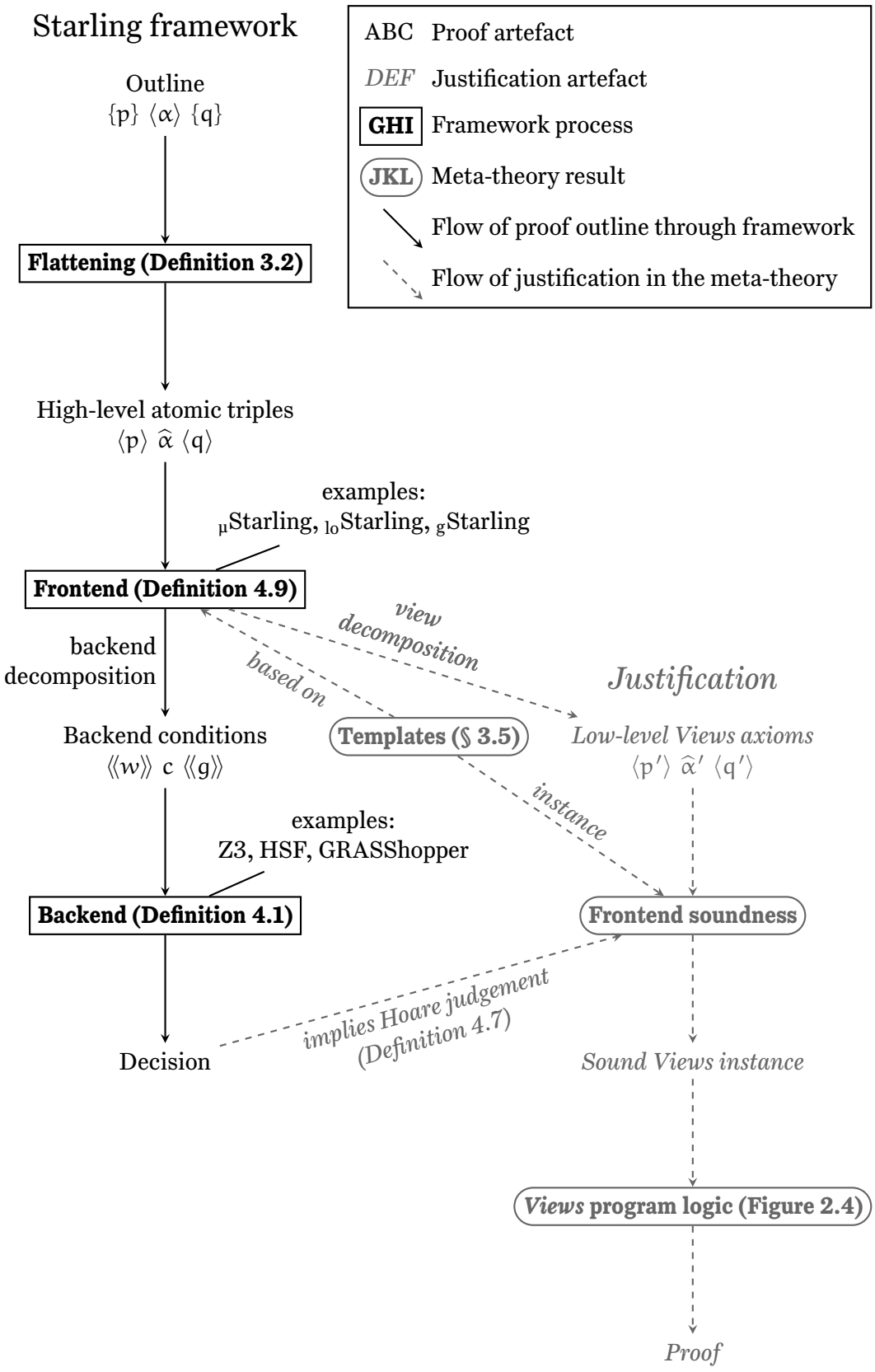


Figure 4.1: Interactions between the various parts of the Starling framework, including the results that make up its soundness justification. Templates, from the last chapter, form the base of both the frontends we construct and their soundness arguments.

4.1 Backends

To turn concurrent proof outlines into verification conditions that external solvers can check, we must reduce views and atomics into forms that those solvers understand. The last chapter followed the *Views* approach, expanding them into state sets. While this is appropriate for the meta-theory, it only works in practice if we can enumerate every possible state; this overly restricts the states, and solvers, we can use.

Solvers normally decide formulae over states at a higher level than state sets: as constraints in linear arithmetic, or bit-vectors, and so on. Starling captures these underlying theories as *backends*. A backend contains, amongst other things, a set of languages corresponding to the predicates and relations the solver understands, and an abstraction over the solver's decision procedure over conditions built using those languages. We can then define views and atomics over these languages[1] to take advantage of said procedure.

Abstracting over solver differences

Each solver differs in terms of which formulae it can decide, both syntactically and semantically. Any interface we define over them must account for the following areas of difference.

Input languages. If we target, for instance, an SMT solver that can decide predicates over certain theories, we must define Starling-level views and atomic actions using such predicates. Horn-clause solvers may use a similar, but more rigidly defined, predicate language. If we target a high-level sequential verification language, we may need to express conditions in its Boolean expression language, but commands in its primitive statement language.

Decision process. Each solver's process for deciding Hoare triples is different. For SMT solvers, we can encode a form of $\Vdash_{\mathsf{FH}}$ as a predicate. For Horn clauses, we may need to encode each judgement as a clause using a construct not available in the predicate language itself. High-level languages may need an elaborate encoding: we may need to lift the conditions into assume and assert commands, then pose each triple as a separate sequential program.

State model. While basic SMT solvers may only decide predicates over a fixed set of scalar shared variables, others may reason about sequences, arrays, records, and so on. Some solvers, like GRASShopper [55], model shared heaps: we must be able to use these with Starling to verify programs that handle dynamically allocated graph structures.

Abstraction approach

One way to define a backend is to build a specific set of languages, decision procedures, and state models, and assume that each solver we intend to use can implement them. Such an approach is hard to scale — what if we need to target a new solver with a different scheme of

[1] We cannot just give the solvers views and atomics directly; to do so, we would need to add theories of views and atomics to the backends, which would limit generality.

observations and commands? —, ties our logics to the concrete details of their solvers at an early stage, and makes using the full expressive power of our solvers hard.

Let us instead parametrise our logics with the set of languages, procedures, and models that the solver can understand, and connect these to the mechanisms of defining views and atomics. These components form a *backend*.

Definition 4.1. A *backend* is a tuple $(\mathsf{E_{Pr}}, \mathsf{E_{RI}}, S, \mathsf{Solve}, \mathsf{GCtx}, \mathsf{LCtx})$ of expression languages (Definitions 4.4 and 4.5), state set (Definition 2.6), solver predicate (Definition 4.8), and global and local context sets (Definitions 4.2 and 4.3).

Intuitively, $\mathsf{E_{Pr}}$ contains syntactic, single-state expressions that represent the final expansion of views; $\mathsf{E_{RI}}$ captures syntactic, two-state expressions that represent the semantics of atomic actions; Solve is a predicate abstractly reflecting the solver's decision process over said expressions; and the context sets let us capture additional quantifications (for example, on local states, or on constraints) that the solver must handle internally.

Running example

Let us consider automating the Peterson (§ 3.3) proof using a toy solver that understands propositional formulae over Boolean variables; primitive commands that *set* and *clear* variables; and an *assume* command that cause the program to diverge if a proposition does not hold[2]. The solver can also infer the bodies of uninterpreted Boolean functions $\mathtt{P}(\mathtt{x}, \mathtt{y}, \ldots, z)$, where $\mathtt{x}, \mathtt{y}, \ldots, z$ are also Boolean formulae. These functions have the same meaning throughout a solver instance, and we can refer to them in any place where a Boolean formula is allowed. Let us define $\mathsf{E_{Pr}}$ and $\mathsf{E_{RI}}$ with the following grammars:

$$\langle \mathsf{E_{Pr}} \rangle \quad ::= \langle \mathit{var} \rangle \mid \texttt{(fun } \langle \mathit{var} \rangle \langle \mathsf{E_{Pr}} \rangle^* \texttt{)} \mid \texttt{(not } \langle \mathsf{E_{Pr}} \rangle \texttt{)} \mid \texttt{(and } \langle \mathsf{E_{Pr}} \rangle^* \texttt{)} \mid \texttt{(or } \langle \mathsf{E_{Pr}} \rangle^* \texttt{)}$$

$$\langle \mathsf{E_{RI}} \rangle \quad ::= \texttt{(set } \langle \mathit{var} \rangle \texttt{)} \mid \texttt{(clear } \langle \mathit{var} \rangle \texttt{)} \mid \texttt{(assume } \langle \mathsf{E_{Pr}} \rangle \texttt{)}$$

The solver takes sets of Hoare triples over these languages as input. It tries to infer definitions for each *fun* such that each triple, with the definitions substituted in, satisfies the Hoare judgement. If a consistent system of definitions exists, the solver accepts the set.

Expression languages and their interpretation

With an informal example of a backend in hand, we can start formally defining its components. This section discusses expression languages; these, in turn, require us to discuss their interpretation using global and local contexts.

Contexts. Defining expression languages requires us to provide a state-set interpretation of the expressions. This is because, though backends deal directly with expressions, the *Views*-based metatheory, and so the Starling soundness argument, still expects raw state sets. These interpretations are purely theoretical, and need not be decidable.

[2] As we are concerned only with safety properties, this command can stand in for the conditional forms we need to implement the 'wait until…' parts of the algorithm.

For some solvers, the interpretation can rely on additional context that the solver builds over the whole program. We saw this in our running example: the set of inferred function definitions is one such piece of context. To model this at the state-set interpretation level, we can parametrise interpretations over a *global context set*.

Definition 4.2. A *global context set* GCtx, ranged over by x_g, is a set of internal states that a backend uses when interpreting *all* expressions belonging to the same program.

The interpretation of each *individual* verification condition may depend on some solver state besides S. While our running example has no such state, this idea becomes useful for reasoning about thread-local state in § 5.5. We can capture this using a *local context set*.

Definition 4.3. A *local context set* LCtx, ranged over by x_l, is a set of internal states over which a backend quantifies separately for each individual verification condition.

As we existentially quantify over contexts in the meta-theory, solvers that do not use them cannot define them as $\emptyset$. One tactic is to define them as a singleton set of some arbitrary object tt; the Coq development uses this encoding. Our running example defines $\mathsf{LCtx} = \{\,\mathsf{tt}\,\}$.

Propositions. In Starling, *proposition expression language*s define views. Their definition must consider the needs of frontends that implement a defining-views style of proof rule: these need the ability to conjoin definitions with some associative, commutative conjunction operator $\wedge_{\mathsf{Pr}}$, and represent a lack of matching definitions with a unit expression $\mathsf{true}_{\mathsf{Pr}}$.

Definition 4.4. A *proposition expression language* over states S, global contexts GCtx, and local contexts LCtx is an algebra $(\mathsf{E}_{\mathsf{Pr}}, \mathsf{true}_{\mathsf{Pr}}, \wedge_{\mathsf{Pr}}, [\![-]\!]_{\mathsf{Pr}})$ where $[\![-]\!]_{\mathsf{Pr}}$ is a function $\mathsf{E}_{\mathsf{Pr}} \to \mathsf{GCtx} \to \mathsf{LCtx} \to \mathbb{P}(S)$ interpreting expressions as the state-sets satisfying their propositions, and $(\mathsf{E}_{\mathsf{Pr}}, \mathsf{true}_{\mathsf{Pr}}, \wedge_{\mathsf{Pr}})$ is a monoid over $[\![-]\!]_{\mathsf{Pr}}$ such that $\wedge_{\mathsf{Pr}}$ over two expressions corresponds to the intersection of their interpretations:

$$\forall x, y, x_g, x_l.\, [\![x \wedge_{\mathsf{Pr}} y]\!]_{\mathsf{Pr}}(x_g)(x_l) = [\![x]\!]_{\mathsf{Pr}}(x_g)(x_l) \cap [\![y]\!]_{\mathsf{Pr}}(x_g)(x_l)$$

This definition is restrictive — for instance, it makes it hard to use a separation algebra as the expression language — but is necessary for key properties to hold later, and for our use of proposition expressions to be compatible with the defining-views template.

Relations. A relation expression set E_{Rl} represents atomic actions. We do not assume an algebraic structure for E_{Rl} yet, apart from requiring some special expressions to exist.

Definition 4.5. A *relation expression language* $(\mathsf{E}_{\mathsf{Rl}}, \mathrm{id}_{\mathsf{Rl}}, \emptyset_{\mathsf{Rl}}, [\![-]\!]_{\mathsf{Rl}})$ over states S and contexts GCtx is an algebra where $[\![-]\!]_{\mathsf{Rl}}$ is a function $\mathsf{E}_{\mathsf{Rl}} \to \mathsf{GCtx} \to \mathsf{LCtx} \to (S \leftrightarrow S)$ interpreting expressions as their underlying relations on states; and $\mathrm{id}_{\mathsf{Rl}}$ is the identity, and $\emptyset_{\mathsf{Rl}}$ the empty relation, under said interpretation:

$$[\![\mathrm{id}_{\mathsf{Rl}}]\!]_{\mathsf{Rl}}(x_g)(x_l) = \{\,(\sigma, \sigma) \mid \sigma \in S\,\} \qquad [\![\emptyset_{\mathsf{Rl}}]\!]_{\mathsf{Rl}}(x_g)(x_l) = \emptyset$$

Verification conditions

Proposition and relation expressions form the *verification conditions* we send to a backend solver. In Starling, such conditions arise from applying some decidable encoding of a proof rule — a *frontend* (§ 4.2) — to an atomic Hoare triple from an outline decomposition. Verification conditions are $\Vdash_{\mathsf{FH}}$-style judgements over sequentialised executions of the original atomic action modulo an environment, so we express them as Hoare triples.

Definition 4.6. For all E_{Pr} and E_{Rl}, a *verification condition* $\langle\langle w \rangle\rangle\ c\ \langle\langle g \rangle\rangle$ is a Hoare triple over a 'weakest pre-condition' predicate $w : \mathsf{E}_{\mathsf{Pr}}$, command $c : \mathsf{E}_{\mathsf{Rl}}$, and goal predicate $g : \mathsf{E}_{\mathsf{Pr}}$, representing a backend-theory query we generate from an atomic Hoare triple.

$\mathsf{VConds}(\mathsf{E}_{\mathsf{Pr}}, \mathsf{E}_{\mathsf{Rl}})$ is the set of all $\langle\langle w \rangle\rangle\ c\ \langle\langle g \rangle\rangle$ over E_{Pr} and E_{Rl}.

Solving verification conditions. The job of the backend theory is to decide the correctness of the verification conditions we generate from a proof outline. To let us use the scheme from Chapter 3 as a soundness argument for this set-up, the whole process must entail the application of the monoidal-defining-views rule to the decomposition of the outline.

Definition 4.7. The *verification-condition Hoare judgement* over $\langle\langle w \rangle\rangle\ c\ \langle\langle g \rangle\rangle$ is:

$$(x_g, c) \Vdash_{\mathsf{EVFH}} \{w\}\{g\} \stackrel{\text{def}}{=} \forall x_l \in \mathsf{LCtx}. \Vdash_{\mathsf{FH}} \{[\![w]\!]_{\mathsf{Pr}}(x_g)(x_l)\}\ [\![c]\!]_{\mathsf{Rl}}(x_g)(x_l)\ \{[\![g]\!]_{\mathsf{Pr}}(x_g)(x_l)\}$$

An ideal solver would implement this judgement directly. As $[\![-]\!]_{\mathsf{Pr}}$ and $[\![-]\!]_{\mathsf{Rl}}$ exist only at the theory level, and the exact format of σ and σ' is a black-box property of the theory, we cannot implement such solvers in practice. Worse, the theory may be unable to reason about individual verification conditions: for example, the inference our example solver does will depend on the entire closed system of conditions. Instead, we model the solver as a predicate on a verification condition *set*, and let it be stronger than Hoare reasoning.

Definition 4.8. A *solver predicate* is a predicate Solve over sets $T \in \mathbb{P}(\mathsf{VConds}(\mathsf{E}_{\mathsf{Pr}}, \mathsf{E}_{\mathsf{Rl}}))$ that, when true, implies that at least one global context exists such that the given set of verification conditions is correct under Hoare reasoning:

$$\mathsf{Solve}(T) \implies \exists x_g.\ \forall \langle\langle w \rangle\rangle\ c\ \langle\langle g \rangle\rangle \in T.\ (x_g, c) \Vdash_{\mathsf{EVFH}} \{w\}\{g\}$$

Free backends

If we consider Starling as being embedded in some meta-theory (for example, first-order logic, or Coq's calculus of constructions), then we can construct a 'free' backend in which proposition expressions are propositions, relation expressions are relations, $[\![x]\!]_{\mathsf{Pr}} = x$, $[\![x]\!]_{\mathsf{Rl}} = x$, and Solve is the construction on the right-hand side of Definition 4.8. § 8.3 outlines an implementation of this idea in the Coq mechanisation.

4.2 Frontends

The middle step in the Starling pipeline — a decomposition from atomic Hoare triples to backend conditions — is where we make the main soundness argument. These decompositions form the front-of-house logical machinery that arranges flattened proof outlines into a form the backend can process, so we call them *frontends*. Frontends are not fully decoupled from backends: they must re-express views as proposition expressions and atomic actions as relation expressions using systems such as *syntactic definers* (§ 4.3).

The frontend soundness argument involves a second decomposition, from atomic Hoare triples to sets of *Views* axioms. While we can map each triple to arbitrarily many axioms in which the views and atomic actions can differ from the originals, the obvious decomposition ($\lambda x.\{x\}$) is the one that most of this dissertation uses[3].

Soundness also requires that, if we can solve the system of backend conditions, the views-decomposition axioms inhabit the axiomatisation of a sound views instance. This instance can depend on the solver's global context: for example, different inferences for view definitions can change whether certain atomic actions preserve certain views.

Definition 4.9. A *frontend* is a triple

$$\begin{pmatrix} \mathsf{D_b} : \mathsf{AHoare}(V, A) & \leftrightarrow \mathsf{VConds}(\mathsf{E_{Pr}}, \mathsf{E_{Rl}}), \\ \mathsf{D_v} : \mathsf{AHoare}(V, A) & \leftrightarrow \mathsf{AHoare}(V', A'), \\ \mathrm{I} : \mathsf{GCtx} & \to \mathsf{Inst}(V', A', S) \end{pmatrix}$$

consisting of a *backend decomposition* $\mathsf{D_b}$ mapping each high-level atomic Hoare triple to zero or more backend conditions; a *views decomposition* $\mathsf{D_v}$ mapping the same triples to zero or more low-level *Views* axioms; and I, which maps a global context to a views instance over the low-level views semigroup and commands.

Frontends must have the property that, for all high-level triples $\langle p\rangle\ \widehat{\alpha}\ \langle q\rangle$ and global contexts x_g, if every verification condition $\langle\!\langle w\rangle\!\rangle\ c\ \langle\!\langle g\rangle\!\rangle$ in the triple's backend decomposition satisfies the verification-condition Hoare judgement, then every *Views* axiom in the triple's Views decomposition is a member of the axiomatisation of $\mathrm{I}(x_g)$:

$$\begin{aligned} &(\ \forall \langle\!\langle w\rangle\!\rangle\ c\ \langle\!\langle g\rangle\!\rangle\ .\ (\langle p\rangle\ \widehat{\alpha}\ \langle q\rangle)\ \mathsf{D_b}\ (\langle\!\langle w\rangle\!\rangle\ c\ \langle\!\langle g\rangle\!\rangle) \implies (x_g, c) \Vdash_{\mathsf{EVFH}} \{w\}\{g\}\ ) \\ \implies &(\forall \langle p'\rangle\ \widehat{\alpha}'\ \langle q'\rangle\ .\ (\langle p\rangle\ \widehat{\alpha}\ \langle q\rangle)\ \mathsf{D_v}\ (\langle p'\rangle\ \widehat{\alpha}'\ \langle q'\rangle) \implies \langle p'\rangle\ \widehat{\alpha}'\ \langle q'\rangle \in \mathrm{I}(x_g).\mathsf{T}) \end{aligned}$$

(Coq: covering in Starling.Frontend.Common)

While the definition above does not constrain the sets of axioms to which triples map — in fact, the most straightforward way to build a sound frontend is to let $\mathsf{D_v} = \emptyset$ — the soundness of outline flattening depends on a correspondence between outlines, the axioms generated by the frontend, and the *Views* program logic.

[3] This decomposition starts to become useful when we introduce the Local Views Framework in § 5.1.

From decompositions to templates

Suppose we have both views and backend decompositions. We can then build a template that contains precisely the set of axioms that correspond to views decompositions of atomic Hoare triples whose backend decompositions satisfy the Hoare judgement.

Definition 4.10. We define the *frontend-to-template* function fTemp as follows:

$$\begin{aligned}\mathsf{fTemp} : (\mathsf{AHoare}(V, A) &\to \mathsf{VConds}(\mathsf{E_{Pr}}, \mathsf{E_{RI}})) \\ &\to (\mathsf{AHoare}(V, A) \to \mathsf{AHoare}(V', A')) \\ &\to \mathsf{GCtx} \\ &\to (\mathsf{Sig}(V', A', S) \to \mathbb{P}(\mathsf{AHoare}(V', A')))\end{aligned}$$

$$\mathsf{fTemp}(\mathsf{D_b})(\mathsf{D_v})(s)(x_g) \stackrel{\text{def}}{=}$$

$$\left\{ \langle p' \rangle\ \alpha'\ \langle q' \rangle \;\middle|\; \begin{array}{l} \exists\, \langle p \rangle\ \alpha\ \langle q \rangle\,. \\ \quad \langle p' \rangle\ \alpha'\ \langle q' \rangle \in \mathsf{D_v}(\langle p \rangle\ \alpha\ \langle q \rangle) \\ \quad \wedge\, \mathsf{D_b}(\langle p \rangle\ \alpha\ \langle q \rangle) \subseteq \{\, \langle\!\langle w \rangle\!\rangle\ c\ \langle\!\langle g \rangle\!\rangle \mid (x_g, c) \Vdash_{\mathsf{EVFH}} \{w\}\{g\} \,\} \end{array} \right\}$$

(Coq: builder_to_template in Starling.Frontend.Common)

Lemma 4.1. Given relations $(\mathsf{D_b}, \mathsf{D_v}, \mathrm{I})$ where I maps from global contexts to views instances with axiomatisations built through $\mathsf{fTemp}(\mathsf{D_b})(\mathsf{D_v})(s)$, $(\mathsf{D_b}, \mathsf{D_v}, \mathrm{I})$ is a frontend.

4.3 Syntactic definers

Having discussed a way to express definitions of views and atomic actions in a backend theory, we now consider constructs for mapping views and atomics to such definitions. As the definitions are syntactic predicate and relation expressions from a backend interface, we call the constructs *syntactic definers*. There are two types of syntactic definer: the *syntactic view definer*, which maps views to proposition expressions; and the *syntactic atomic definer*, which maps atomics to relation expressions.

This section builds basic syntactic definers that map views and atomic actions directly to backend definitions. Chapter 6 shows that this direct mapping limits the expressivity of our logics, and explores more complex schemes based on pattern matching.

Syntactic view definitions

First, we need a syntactic means of defining views in terms of proposition expressions. In Definition 3.10, we defined semantic view definers as relations that map defining views to the sets of states they admit. In practice, we leave the state-set interpretation to the backend, so we cannot rely on the proof author to build such a definer. Instead, we show that a correspondence between our syntactic definers and the semantic equivalent exists in theory, and assume the backend handles the consequences in practice.

For now, let *syntactic definers* be partial functions $V \rightharpoonup \mathsf{E}_{\mathsf{Pr}}$ from views to proposition expressions, and gloss over the exact implementation. For an example of a more concrete realisation of syntactic definers, see Appendix A.3.

Syntactic reification and definition functions. Let us modify Definition 3.11 to account for the fact that we are building up a proposition expression, not a state set. The *Views* reification is, then, the interpretation of this expression. As the interpretation depends on the particular value of GCtx we pass as backend context, the reification — and, thus, the specific *Views* instance we construct as our soundness argument — depends on that value. For example, the *Views* instance can change depending on inferred meanings of uninterpreted functions.

Definition 4.11. The *syntactic definer reification* $\mathsf{sdReify} : (V \rightharpoonup V) \to V \to \mathsf{E}_{\mathsf{Pr}}$ is:

$$\mathsf{sdReify}(\mathrm{d})(v) = \bigwedge_{\mathsf{Pr}} \{\, \mathrm{d}(u) \mid u \sqsubseteq v \wedge u \in \mathrm{dom}\ \mathrm{d} \,\}$$

(Coq: sd_syn_reify in Starling.Frontend.SynDefiner)

Given a definer d and contexts x_g and x_l, we can lift sdReify to a *Views* reification:

$$\lfloor v \rfloor = \{\, \sigma \mid [\![\mathsf{sdReify}(\mathrm{d})(v)]\!]_{\mathsf{Pr}}(x_g)(x_l)(\sigma) \,\}$$

Syntactic atomic definitions

To interpret the atomic actions in the proof outline as relation expressions, we need another syntactic definer. There may be a potentially unbounded set of valid atomic actions, so the atomic syntactic definer is just a function.

Definition 4.12. An *syntactic atomic definer* ASDef is a function $A \to \mathsf{E}_{\mathsf{Rl}}$ that defines every possible atomic action as a relation expression.

In our running example, this function might contain definitions like this:

$$\begin{aligned}
\mathsf{ASDef}(\texttt{Q1 := false}) &= (\textbf{clear}\ \texttt{Q1}) \\
\mathsf{ASDef}(\texttt{Q1 := true}) &= (\textbf{set}\ \texttt{Q1}) \\
\mathsf{ASDef}(\texttt{TURN := 1}) &= (\textbf{clear}\ \texttt{TURN}) \\
\mathsf{ASDef}(\texttt{TURN := 2}) &= (\textbf{set}\ \texttt{TURN}) \\
\mathsf{ASDef}(\texttt{wait until !Q2 or TURN=2}) &= (\textbf{assume}\ \neg\texttt{Q1} \vee \texttt{TURN})
\end{aligned}$$

When translating atomic Hoare triples to backend verification conditions, we must handle commands given as A^{id}, not A. This means we must lift the syntactic atomic definer:

Definition 4.13. The *syntactic label definer* $\mathsf{ASDef}^{\mathsf{id}}$ over a syntactic atomic definer ASDef is the function $A^{\mathsf{id}} \to \mathsf{E}_{\mathsf{Rl}}$ defined piecewise as follows:

$$\mathsf{ASDef}^{\mathsf{id}}(\mathrm{id}) = \mathrm{id}_{\mathsf{Rl}} \qquad \mathsf{ASDef}^{\mathsf{id}}(\mathrm{c}) = \mathsf{ASDef}(\mathrm{c})$$

4.4 $_{\mu}$Starling

This section builds an initial frontend, $_{\mu}$Starling, by adapting the defining-views technique (Definition 3.12). Instead of defining views and atomic actions over state sets, $_{\mu}$Starling uses syntactic definers. This gives us finite, bounded sets of verification conditions, which we can send directly to an SMT solver, proof assistant, or other suitable backend solver.

The $_{\mu}$Starling frontend depends on three parameters. It needs a subtractive views semigroup from which we draw the target proof's assertions directly. It also assumes that said views map to proposition expressions through a syntactic view definer, and that atomic actions map to relation expressions through a syntactic atomic definer, per § 4.3.

The $_{\mu}$Starling backend decomposition

With a syntactic view definer d and atomic definer ASDef, we can build $_{\mu}$Starling's backend decomposition. This corresponds to applying the defining-views judgement over the triple, and we define it in several steps. We assume the definers are in scope throughout.

Building a single backend condition. Each $_{\mu}$Starling backend condition corresponds to a combination of an atomic Hoare triple $\langle \mathrm{p} \rangle\ \widehat{\alpha}\ \langle \mathrm{q} \rangle$ and a goal view g (which, as we saw earlier, combines q with a view from the outside context).

Definition 4.14. The function $\mathsf{D}^{\mu}_{\mathrm{g}} : (V \times A^{\mathsf{id}} \times V) \to V \to \mathsf{VConds}(\mathsf{E}_{\mathsf{Pr}}, \mathsf{E}_{\mathsf{Rl}})$ translates an atomic Hoare triple, given a goal view g, into a backend condition:

$$\mathsf{D}^{\mu}_{\mathrm{g}}(\langle \mathrm{p} \rangle\ \widehat{\alpha}\ \langle \mathrm{q} \rangle)(\mathrm{g}) \stackrel{\text{def}}{=} \langle\!\langle \mathsf{sdReify}(\mathrm{d})(\mathrm{p} \bullet (\mathrm{g} \setminus \mathrm{q})) \rangle\!\rangle\ \mathsf{ASDef}^{\mathsf{id}}(\widehat{\alpha})\ \langle\!\langle \mathsf{sdDef}(\mathrm{d})(\mathrm{g}) \rangle\!\rangle$$

(Coq: ms_decomp_ni_single in Starling.Logics.MicroStarling)

We can translate an atomic Hoare triple into a set of verification conditions, by taking the $\mathsf{D}^{\mu}_{\mathrm{g}}$ result for every g that is a defining view. When V is not a views monoid, we must add an extra verification condition per triple to check sequential safety. This gives us a new function, $\mathsf{D}^{\mu}_{\mathrm{b}}$. The result of $\mathsf{D}^{\mu}_{\mathrm{b}}$ is bounded and finite whenever the definer's domain is.

Definition 4.15. The function $\mathsf{D}^{\mu}_{\mathrm{b}}$ translates an atomic Hoare triple into a set of backend conditions that cover both sequential safety and non-interference:

$$\mathsf{D}^{\mu}_{\mathrm{b}}(\langle \mathrm{p} \rangle\ \widehat{\alpha}\ \langle \mathrm{q} \rangle) \stackrel{\text{def}}{=} \{\, \langle\!\langle \mathsf{sdReify}(\mathrm{d})(\mathrm{p}) \rangle\!\rangle\ \mathsf{ASDef}^{\mathsf{id}}(\widehat{\alpha})\ \langle\!\langle \mathsf{sdReify}(\mathrm{d})(\mathrm{q}) \rangle\!\rangle \,\}$$
(sequential safety)

$$\cup\, \{\, \mathsf{D}^{\mu}_{\mathrm{g}}(\langle \mathrm{p} \rangle\ \widehat{\alpha}\ \langle \mathrm{q} \rangle)(\mathrm{g}) \mid \exists e.\ \mathrm{d} = \mathrm{d}_1 +\!\!+ \langle (\mathrm{g}, e) \rangle +\!\!+ \mathrm{d}_2 \,\}$$
(non-interference)

(Coq: ms_decomp in Starling.Logics.MicroStarling)

The $_{\mu}$Starling views instances

D_b^{μ} gives us the backend decomposition for our $_{\mu}$Starling frontend. We next provide a sound views instance, and a decomposition into axioms in that instance. For $_{\mu}$Starling, the axiom decomposition is straightforward: as we have nothing in the assertion views and atomic actions that needs reducing, we just use $\lambda x.\{x\}$. With this set-up, we can build templates that capture the decision procedure that $_{\mu}$Starling and the solver implement together.

Definition 4.16. The $_{\mu}$*Starling template*, given a global context x_g and appropriate definers, is $\mathsf{fTemp}(D_b^{\mu})(\lambda x.\{x\})(x_g)$. (Coq: ms_template in Starling.Logics.MicroStarling)

This template yields instances in the usual way, through instantiation with a corresponding signature. To show that such instances are axiom-sound, we show that the template subsets the defining-views template in the presence of a compatible signature. Then, through the chains of subset results we built in § 3.6, we show that each subsets the free template, and, thus, produce axiom-sound instances.

Lemma 4.2. The $_{\mu}$Starling template produces subsets of the defining-views template. (Coq: ms_strengthens_defining in Starling.Logics.MicroStarling)

Corollary 4.2.1. The $_{\mu}$Starling template produces axiom-sound instances.

Through the definition of frontends (Definition 4.9), we know that when a solver accepts a set of atomic Hoare triples, the corresponding views axioms inhabit the axiomatisation of the appropriate $_{\mu}$Starling instance. Since, for $_{\mu}$Starling, said axiomatisation is sound, the mapping from the triples into it is direct, and our outline decomposition follows the rules of the *Views* framework, we can combine all of the pieces into a single proof rule. As before, the specific rule differs if we have a views monoid.

Definition 4.17. The $_{\mu}$*Starling outline rule* applies the Starling decision process, using the $_{\mu}$Starling frontend, to a *Views*-language outline o:

$$\vdash^{\mu} o \stackrel{\text{def}}{=} \mathsf{Solve}(\{\, D_b(\langle p\rangle\ \widehat{\alpha}\ \langle q\rangle) \mid \langle p\rangle\ \widehat{\alpha}\ \langle q\rangle \in \mathsf{oflat}(o)\,\})$$

Theorem 4.3. Rule $\vdash^{\mu}$ entails the *Views* semantic judgement on its respective outlines.

While this section does not justify it in depth, we can apply the simplification discussed in § 3.6. This frees us from needing to prove sequential safety for each triple explicitly.

4.5 Summary

This chapter introduced Starling, a framework for building automation-friendly concurrent program logics. Starling depends on two loosely-coupled components: a *backend* that abstracts over a Hoare-logic theory for which we have a solver, and a *frontend* that reduces atomic Hoare triples to a bounded set of verification conditions in said theory. In turn, Starling provides a sound process for verifying *Views*-language proof outlines.

The chapter then demonstrated frontend-building for Starling using μStarling, a direct lifting of the defining-views template from Definition 3.12. This frontend gives us a way to prove properties about programs where all state is shared, all actions are atomic and sequentially consistent, and where we can represent any shared-state assertions as a semigroup where parts of the semigroup map directly to shared-state subsets.

The next chapters expand μStarling's expressivity in various ways. In Chapter 5, we add native support for thread-local state to *Views*, and discuss how to adapt our frontend to make use of it while keeping full automation. This poses restrictions on how we can use thread-local state: so, in Chapter 6, we build a frontend that allows a limited, but automation-friendly, degree of local-state parametrisation. By adding ways to parametrise views, we also solve the problem, in μStarling, of needing to duplicate view structures any time we want to vary a view's interpretation.

Chapter 5

Adding Local-State Reasoning

To encode thread-local state in $_{\mu}$Starling, we must encode it into shared state. We then must, manually, separate each thread's local state (eg. with discrete variables, or arrays with separate thread IDs). This causes problems for encoding proofs like Listing 2.2, where we track threads' contributions to a counter through ghost variables. In $_{\mu}$Starling, we must encode these variables as shared state, and build their relation to the counter directly into each view's definition. This entangles the system's proof with its implementation, and pushes local observations into shared state.

This chapter explores how to add local-state reasoning to Starling. In §§ 5.1 to 5.3, it presents a modified *Views* framework based on work by Khyzha, Gotsman, and Parkinson. This framework restricts programs to a fixed top-level parallel composition of sequential threads, but lets us add native local state on top of *Views* — reusing a lot of meta-theory.

This chapter initially considers parametrising thread assertions directly over local state. (In the counter example, we can then use one view to relate the global counter to the current thread's local counter; the proof of each thread is then identical, making the proof cleaner and more compositional.) The resulting set-up models assertions as functions from local state to shared-state views. Proving non-interference on a thread uses only these shared views, and never depends on other threads' local state.

In § 5.5, we explore adding local state to the outline–frontend–backend pipeline by lifting backend expressions to functions over local state. Then, in § 5.6, we use this approach to build $_{\text{lo}}$Starling, a lightweight extension of $_{\mu}$Starling with local state. We find that, while $_{\text{lo}}$Starling is sound, it does not generate a bounded verification-condition set: Chapter 6 considers restricting the shape of local-state assertions to restore boundedness.

Local-state actions cannot interfere with other threads' local state, even when non-atomic. As we see in § 5.7, we can sometimes use the proof of a complex local action, modelled atomically, to prove the same action modelled as a non-atomic sequential composition without making strong assumptions about semantics.

Running example

Until now, our attempts to prove Peterson's algorithm have required us to specify both threads as two mirror-image programs, even though the only differences are tied directly to the identity of the thread (whether we look at flag Q1 or flag Q2; whether we write 1 or 2

```
   t1  Q[t] := true
   t2  TURN := t̄
   t3  wait until !Qt̄ or TURN=t
  ──────────────────────────────
            (Critical Section)
  ──────────────────────────────
λt. t4  Q[t] := false
```

Figure 5.1: One-thread-proof version of *Peterson*. Here, $\bar{t}$ means 'the thread that is not t'.

to the turn variable; and the set of line numbers we use as views). This duplication makes eyeballing and hand-proving the proof tedious, and also slows down automation.

This chapter considers a modified version of Figure 3.2 where we model the global variables as two-place arrays, and parametrise the program-position assertions over the current thread ID. Figure 5.1 gives this revised version.

5.1 The *Local Views Framework*

To reason about local state in Starling, we must adapt the shared-state-centric *Views* framework. There is already a *Views*-based framework that supports local-state parametrisation: Khyzha *et al.*'s generic linearisability logic [56] (for short, 'the GLL'); as the name suggests, it targets linearisability proofs, which causes its action judgement to be too strong for our existing template infrastructure to target soundly.

Here, we present a modification of the GLL to target Hoare-style safety properties, and have an action judgement that more closely resembles that of *Views*. We can also consider the resulting framework as an extension of *Views* to add GLL-style support for local states, and so we call it the *Local Views Framework* (or LVF).

Like the GLL, we internally model local state inside shared state, as a map from thread IDs to states. Unlike the GLL, which embeds linearisation points into its action judgements, we just extend *Views*'s existing set-up with local-state tracking. This leads to a different soundness derivation, which we sketch in § 5.3.

While our smaller, more *Views*-like framework does not natively support the strong correctness guarantees about programs that the GLL does, the interface that it requires reasoning systems to implement — LVF axiom soundness — is smaller and easier to satisfy. Also, by only making small changes to *Views*, we make it easier to embed parts of the LVF into *Views*, letting us re-use much of our existing meta-theory.

We take a bottom-up approach to constructing the framework. First, we add a framework parameter for local states, and restructure action semantics to let actions modify both shared state and the current thread's local state[1]. Then, to allow local-state observations to influence proof assertions, we lift views to *view functions*, parametrised directly by the current local state. As local state only serves to select a specific shared-state view, knowledge of one thread's local state never leaks to other threads, and we retain a form of the *Views* frame rule. We then introduce a local form of the action judgement, and propagate it to the other *Views* constructs: signatures, axiomatisations, instances, and axiom soundness.

[1] We discuss ways to get around the implied requirement that all local actions are atomic in § 5.7.

The introduction of local state requires large changes to the *Views* programming language, program logic, and soundness argument. As a result, we leave these to later sections: we introduce the language in § 5.2, and discuss soundness in § 5.3.

We also show that, for every LVF instance, we can build a *Views* instance by making local state explicit in the instance's axioms, and that local axiom soundness of the LVF instance directly relates to axiom soundness of the *Views* instance.

Local parametrisation

This section introduces local state itself, and the ways in which we parametrise the high-level parts of the framework — atomic actions and views — by it.

Local states. Like shared state, we model local state as an abstract set of possible states. We refer meta-syntactically to such a set as L, just as we use S for shared state.

> **Definition 5.1.** A *local state set* is a set L of possible local states. When such a set is in scope, we use l to refer to a pre-state in the set, and l' to a post-state in the set.

In our modified *Peterson*, we track one local variable (the thread ID), so $\mathsf{L} = \{ \mathsf{A}, \mathsf{B} \}$.

Atomic actions over local and shared states. As before, we assume atomic actions form a symbolic set A — but we now allow them to modify local state. This means that we must modify the semantic function (Definition 2.9). To capture the semantics of actions that modify both local and shared state, we use a *local semantic relation*.

> **Definition 5.2.** A *local semantic relation* $[\![-]\!]_{lo} : \mathsf{A} \to ((\mathsf{L} \times \mathsf{S}) \leftrightarrow (\mathsf{L} \times \mathsf{S}))$, for some atomic action set A, local state set L, and shared state set S, is a relation between pairs of local and shared pre-state, and local and shared post-state.

In our example, we assume that each thread begins execution with the right ID stored locally, and that each atomic action behaves as the identity on the ID.

As with $[\![-]\!]$ in Definition 2.11, we can lift $[\![-]\!]_{lo}$ to atomic labels. Let $[\![\widehat{\alpha}]\!]^{\mathsf{id}}_{lo}$ stand for this lifting, which, when applied to id, maps every pair (l, σ) to itself.

View functions. In LVF axioms, we draw preconditions and postconditions from the function space $\mathsf{L} \to \mathsf{V}$. This lets us use local state to choose which shared-state view to assert at a given point in the program. When we build the LVF action judgement, we will supply the local pre-state to the precondition, and the local post-state to the postcondition.

This is a strict expressivity increase from *Views* axioms, as we can always encode a view ν with no local-state dependency with the constant function $\lambda \mathsf{x}.\, \nu$. We use this encoding so often that we introduce shorthand for it: $\mathsf{const}(\nu)$ (which we formally define in Definition A.2).

In our one-proof *Peterson*, we must encode our views into the function space $\{ \mathsf{A}, \mathsf{B} \} \to \mathsf{V}$ (for some V), so that the functions model the selection of a program location given a thread ID. Here, we let V be the same as our last *Peterson* example ($\{ \mathsf{A1}, \mathsf{A2}, \mathsf{A3}, \mathsf{A4}, \mathsf{B1}, \mathsf{B2}, \mathsf{B3}, \mathsf{B4} \}$), and model each assertion at location N in the form $\lambda \mathsf{t}.\, \mathsf{tN}$.

Lifted views algebras. View functions form a views semigroup whenever V is a views semigroup, and so on for views monoids, subtractive semigroups, and ordered semigroups. This lets us lift the views notation and logical tooling we've built in the previous chapters to such functions. This approach to local-state parametrisation resembles that which Appel *et al.* describe in their Coq mechanisation of separation logics [57].

Lemma 5.1. If V forms a views algebra, we can derive an algebra on $X \to V$ for any X:

$$x \bullet y \stackrel{\text{def}}{=} \lambda i.\, x(i) \bullet y(i) \qquad x \equiv y \stackrel{\text{def}}{=} \forall i.\, x(i) \equiv y(i) \qquad \varepsilon \stackrel{\text{def}}{=} \mathsf{const}(\varepsilon)$$
$$x \setminus y \stackrel{\text{def}}{=} \lambda i.\, x(i) \setminus y(i) \qquad x \sqsubseteq y \stackrel{\text{def}}{=} \forall i.\, x(i) \sqsubseteq y(i)$$

(Coq: Starling.Views.Transformers.Function)

The position multisets we used in our last *Peterson* exploration were subtractive, ordered views monoids, so $\{ A, B \} \to \text{bag } \{ A1, A2, A3, A4, B1, B2, B3, B4 \}$ inhabits the same classes.

Local signatures, axioms, and instances

For the LVF, we modify signatures to contain local state sets, and use local semantics relations. We make no other changes: reification, for instance, still acts directly on V (and, so, in our *Peterson* example, we still use the reification we built in § 3.3). This reflects the fact that local views, in our set-up, are just shared-state views lifted into local functions.

Definition 5.3. A *local signature* $s_{lo} : \mathsf{LSig}(V, A, L, S)$ is a tuple $(\bullet, \equiv, \lfloor - \rfloor, [\![-]\!]_{lo})$, where:

- A is an atomic action language (definition 2.8);
- $(V, \bullet, \equiv)$ is a views semigroup (definition 2.4);
- $\lfloor - \rfloor$ is a reification function over V (definition 2.7);
- L is a local state set (definition 5.1);
- S is a shared state set (definition 2.6);
- $[\![-]\!]_{lo}$ is a local semantic relation (definition 5.2).

(Coq: LocalSignature in Starling.Views.Frameworks.LVF.Signatures)

As with *Views*, we can combine signatures with axiomatisations to create instances of the local *Views* framework. We use the same definition for axiomatisations (Definition 2.15) but, now, each axiom takes view functions as assertions.

Definition 5.4. A *local instance* $i_l : \mathsf{LInst}(V, A, L, S)$ is a tuple (s_{lo}, T), where:

$$s_{lo} : \mathsf{LSig}(V, A, L, S) \qquad T \subseteq \mathsf{AHoare}(L \to V, A)$$

(Coq: LVFInstance in Starling.Views.Frameworks.LVF.Instances)

Local instances, like their *Views* counterparts, are only sound if their axiomatisations obey axiom soundness. Before we define the local *Views* form of axiom soundness, we first define the new form of action judgement on which it depends.

Local action judgements. Recall the *Views* action judgement over $p, q \in V$:

$$s, \alpha \Vdash \{p\}\{q\} \stackrel{\text{def}}{=} (s, \alpha \Vdash_{\mathsf{VFH}} \{p\}\{q\}) \wedge (\forall v \in s.V.\ s, \alpha \Vdash_{\mathsf{VFH}} \{p \bullet v\}\{q \bullet v\})$$
$$(s, \alpha \Vdash_{\mathsf{VFH}} \{p\}\{q\} \stackrel{\text{def}}{=} [\![\alpha]\!]^*(\lfloor p \rfloor) \subseteq \lfloor q \rfloor)$$

To extend this judgement to local state, we first alter the views–Floyd/Hoare judgement to quantify over local states and use our local semantics. As the new judgement performs the local quantification itself, we supply it views of type $L \to V$.

Definition 5.5. The *local views–Hoare judgement* $s_{lo}, \widehat{\alpha} \Vdash_{\mathsf{LVFH}} \{p\}\{q\}$, over views $p, q \in (L \to V)$ where V is the views set of s_{lo}, is the judgement:

$$s_{lo}, \widehat{\alpha} \Vdash_{\mathsf{LVFH}} \{p\}\{q\} \stackrel{\text{def}}{\iff} \forall \sigma, \sigma' \in S,\ l, l' \in L.\ \sigma \in \lfloor p(l) \rfloor \wedge ((l, \sigma), (l', \sigma')) \in [\![\widehat{\alpha}]\!]^{\mathsf{id}}_{lo} \implies \sigma' \in \lfloor q(l') \rfloor$$

(Coq: slhoare in Starling.Views.Frameworks.LVF.ActionJudgements)

In the action judgement, we also change the type of p and q to $L \to V$ — but leave v as V. This is for two reasons. First, as the context, it can depend on local states other than those currently in scope (so we would need to give it its own local-state quantification). Second, by quantifying over V, we over-approximate quantifications over both L and $L \to V$ at the same time, making any such split redundant.

There is a type mismatch here: we need to join p and q with v, but v is not a view function. Instead, we use $\mathsf{const}(v)$: since $\bullet$ is pointwise, the result of $p \bullet \mathsf{const}(v)$ is $\lambda l.\ p(l) \bullet v$, and we achieve the intended behaviour. We then define the local action judgement as follows:

Definition 5.6. The *local action judgement* $s_{lo}, \widehat{\alpha} \Vdash_{lo} \{p\}\{q\}$, over views $p, q \in (L \to V)$ where V is the views set of s_{lo}, is the judgement:

$$s_{lo}, \widehat{\alpha} \Vdash_{lo} \{p\}\{q\} \stackrel{\text{def}}{\iff} s_{lo}, \widehat{\alpha} \Vdash_{\mathsf{LVFH}} \{p\}\{q\} \wedge \forall v \in V.\ s_{lo}, \widehat{\alpha} \Vdash_{\mathsf{LVFH}} \{p \bullet \mathsf{const}(v)\}\{q \bullet \mathsf{const}(v)\}$$

(Coq: slactionj in Starling.Views.Frameworks.LVF.ActionJudgements)

Framing. The local action judgement has a similar framing property to the *Views* action judgement, though only for constant view functions.

$$(s, \alpha \Vdash \{p\}\{q\} \implies \forall v \in V.\ s, \alpha \Vdash \{p \bullet v\}\{q \bullet v\})$$
$$s_{lo}, \alpha \Vdash_{lo} \{p\}\{q\} \implies \forall v \in V.\ s_{lo}, \alpha \Vdash_{lo} \{p \bullet \mathsf{const}(v)\}\{q \bullet \mathsf{const}(v)\}$$

Local axiom soundness. Local axiom soundness follows directly from *Views* axiom soundness, with the requisite changes to the signature and action judgement.

Definition 5.7. A local signature s_{lo} and axiom set T are *locally axiom-sound* if:

$$\forall \langle p \rangle\ \alpha\ \langle q \rangle \in \mathsf{T}. \quad s_{lo}, \alpha \Vdash_{lo} \{p\}\{q\}$$

We can derive the same simplifications for monoidal local action judgement and axiom soundness as for *Views*. These forms are closer to those used by Khyzha *et al.*, whose model considers only views monoids.

Semantic judgements

Let us discuss the core program-level proof rules of the LVF, the *semantic judgements*. These rules are similar to the CVF semantic judgement, and have the same intuition: they determine that a program is safe with regards to a pair of view assertions. As with the CVF, the LVF program logic then takes the form of inference rules over these judgements.

For the LVF, as we assume a top-level thread conjunction, we need separate single-thread and multi-thread semantic judgements. A later step shows that the parallel composition of threads running single-thread-safe programs forms a multi-thread-safe program, and that any multi-thread-safe program is sound with respect to the LVF language semantics.

Single-thread. The *single-thread semantic judgement* states that a single thread of an LVF program is safe: each time we take a transition labelled $\widehat{\alpha}$ in that thread, we move from a view p to some other view r such that the movement satisfies the local action judgement, and the remaining program is safe. Once the thread reaches `skip`, we show that the final precondition entails the whole program's postcondition. The semantic judgement proves that the thread's transitions are locally correct and preserve any other thread's views.

The single-thread semantic judgement has two forms. The first is an *explicit* version, taking the current local state of the thread's program at each step. To let us thread local state through the judgement, we define it in terms of the CVF action judgement. This, in turn, lets us prove useful atomicity rules later on, in § 5.7.

Definition 5.8. The *explicit single-thread semantic judgement* $s_{lo} \Vdash^{l}_{lo} \{p\}\ c\ \{q\}$, for signature s_{lo}, local views p and q, program c, and initial local state l, is:

$$s_{lo} \Vdash^{l}_{lo} \{p\}\ c\ \{q\} \overset{\text{def}}{\iff} (c = \texttt{skip} \implies \forall l'.\ \mathsf{Sig}^{\uparrow}_{lo}(s_{lo}), (\mathsf{id}, l, l') \Vdash \{p(l)\}\{q(l')\}) \wedge \forall \widehat{\alpha}, c'. \begin{pmatrix} c \xrightarrow{\widehat{\alpha}}_{s} c' \implies \exists r. \forall l'. \\ \mathsf{Sig}^{\uparrow}_{lo}(s_{lo}), (\widehat{\alpha}, l, l') \Vdash \{p(l)\}\{r(l')\} \\ \quad \wedge\ (l \overset{\widehat{\alpha}}{\rightsquigarrow} l' \implies s_{lo} \Vdash^{l'}_{lo} \{r\}\ c'\ \{q\}) \end{pmatrix}$$

$$\text{where} \quad l \overset{\widehat{\alpha}}{\rightsquigarrow} l' \overset{\text{def}}{\iff} \begin{cases} l = l' & \widehat{\alpha} = \mathsf{id} \\ \exists \sigma, \sigma'.\ ((l, \sigma), (l', \sigma')) \in [\![\widehat{\alpha}]\!]^{\mathsf{id}}_{lo} & \text{otherwise} \end{cases}$$

(Coq: SSafeEx in Starling.Views.Frameworks.LVF.SemJudgements)

The condition $l \overset{\widehat{\alpha}}{\rightsquigarrow} l'$ limits the recursive checks we make on the safety of the rest of a program. After making a step with label $\widehat{\alpha}$, we only consider starting from local states l' that can arise from the semantics of $\widehat{\alpha}$ starting from l.

From the explicit judgement, we define an implicit form that quantifies over l. This judgement shows that a thread's program is safe from all starting local states.

Definition 5.9. The *single-thread semantic judgement* $s_{lo} \Vdash_{lo} \{p\}\ c\ \{q\}$ is:

$$s_{lo} \Vdash_{lo} \{p\}\ c\ \{q\} \overset{\text{def}}{\iff} \forall l.\ s_{lo} \Vdash^{l}_{lo} \{p\}\ c\ \{q\}$$

(Coq: SSafe in Starling.Views.Frameworks.LVF.SemJudgements)

Multi-thread. To reason about the safety of a program with n threads, in which any thread can progress at any time, we use a *multi-thread* semantic judgement. As before, we define a version with an explicit current local state, then quantify over it to make an implicit form. Unlike the single-thread case, we now assume that the views, local states, and commands inside the judgement are in the form of parallel lists of length n.

Definition 5.10. The *explicit multi-thread semantic judgement* $s_{lo} \Vdash^{\mathbf{l}}_{lo\mathsf{M}} \{\mathbf{p}\}\ \mathbf{c}\ \{\mathbf{q}\}$, for signature s_{lo}, local view lists $\mathbf{p}$ and $\mathbf{q}$, and program list $\mathbf{c}$ starting in local state $\mathbf{l}$, where all lists are of the same size, is:

$$s_{lo} \Vdash^{\mathbf{l}}_{lo\mathsf{M}} \{\mathbf{p}\}\ \mathbf{c}\ \{\mathbf{q}\} \overset{\text{def}}{\iff} \begin{pmatrix} (\forall t \in \mathsf{Tid}.\ \mathbf{c}[t] = \texttt{skip}) \\ \implies (\forall t \in \mathsf{Tid}, l'.\ \mathsf{Sig}^{\uparrow}_{lo}(s_{lo}), (\mathsf{id}, \mathbf{l}[t], l') \Vdash \{\mathbf{p}[t](\mathbf{l}[t])\}\{\mathbf{q}[t](l')\}) \end{pmatrix} \wedge \forall t \in \mathsf{Tid}, \widehat{\alpha}, \mathbf{c}'. \begin{pmatrix} \mathbf{c} \xrightarrow{(\widehat{\alpha}, t)}_{m} \mathbf{c}' \implies \\ \exists r.\ \forall l'.\ \mathsf{Sig}^{\uparrow}_{lo}(s_{lo}), (\widehat{\alpha}, l, l') \Vdash \{p(l)\}\{r(l')\} \\ \quad \wedge\ l \overset{\widehat{\alpha}}{\rightsquigarrow} l' \implies s_{lo} \Vdash^{\mathbf{l}[t \mapsto l']}_{lo} \{\mathbf{p}[t \mapsto r]\}\ \mathbf{c}'\ \{\mathbf{q}\} \end{pmatrix}$$

(Coq: MSafeEx in Starling.Views.Frameworks.LVF.SemJudgements)

As before, we form the semantic judgement proper by quantification.

Definition 5.11. The *multi-thread semantic judgement* $s_{lo} \Vdash_{lo\mathsf{M}} \{\mathbf{p}\}\ \mathbf{c}\ \{\mathbf{q}\}$ is:

$$s_{lo} \Vdash_{lo\mathsf{M}} \{\mathbf{p}\}\ \mathbf{c}\ \{\mathbf{q}\} \stackrel{\text{def}}{\iff} \forall \mathbf{l}.\ s_{lo} \Vdash^{\mathbf{l}}_{lo\mathsf{M}} \{\mathbf{p}\}\ \mathbf{c}\ \{\mathbf{q}\}$$

(Coq: MSafe in Starling.Views.Frameworks.LVF.SemJudgements)

In place of a parallel composition rule, we have a separate result that lifts single-threaded safety to multi-threaded safety. This, intuitively, tells us that the semantic judgement on one thread is strong enough to ensure that said thread can operate correctly in any context that is safe under the same definition of views and actions.

Theorem 5.2. If each thread in a multi-thread LVF program satisfies the explicit single-thread semantic judgement, the program satisfies the explicit multi-thread semantic judgement:

$$\frac{\forall \mathsf{t} \in \mathsf{Tid}.\ s_{lo} \Vdash^{\mathbf{l}[\mathsf{t}]}_{lo} \{\mathbf{p}[\mathsf{t}]\}\ \mathbf{c}[\mathsf{t}]\ \{\mathbf{q}[\mathsf{t}]\}}{s_{lo} \Vdash^{\mathbf{l}}_{lo\mathsf{M}} \{\mathbf{p}\}\ \mathbf{c}\ \{\mathbf{q}\}}$$

(Coq: all_SSafeEx_MSafeEx in Starling.Views.Frameworks.LVF.SemJudgements)

Corollary 5.2.1. If each thread in a multi-thread program satisfies the single-thread semantic judgement, the program satisfies the multi-thread semantic judgement. (Coq: all_SSafe_MSafe in Starling.Views.Frameworks.LVF.SemJudgements)

The LVF program logic

As with the CVF [42][Def. 8], the LVF's semantic judgements induce a program logic. Figure 5.2 presents the logic as inference rules over the implicit judgements. As in the CVF, the frame rule (5.2(f)) is a special case of the generalised frame rule (5.2(g)). The atomic rule (5.2(b)) *does not* subsume the skip rule (5.2(a)) as it does not accept $\alpha = \mathrm{id}$.

5.2 The LVF programming language

To use the LVF to prove whole programs, we use the same language as the GLL: the *Views* language, without parallel composition[2]. Instead, we have outer parallel compositions of multiple threads, each running a sequential program but communicating with other threads through shared-state atomic actions.

Structural operational semantics

To give the LVF language a semantics, we can use the same techniques as the *Views* paper [42] and build a small-step structural operational semantics in the form of a single-step labelled transition system and its multi-step transitive closure. We make one big change from *Views*: as parallel composition now happens outside the (per-thread) programs under proof, we provide separate single-thread and multi-thread transition systems.

[2]This is also the language Calcagno *et al.* use to demonstrate abstract separation logic in [41].

$$\frac{s_{lo}, \mathrm{id} \Vdash_{lo} \{p\}\{q\}}{s_{lo} \Vdash_{lo} \{p\}\ \texttt{skip}\ \{q\}}$$

(a) Skip

$$\frac{s_{lo}, \alpha \Vdash_{lo} \{p\}\{q\}}{s_{lo} \Vdash_{lo} \{p\}\ \texttt{<}\alpha\texttt{>}\ \{q\}}$$

(b) Atomic

$$\frac{s_{lo} \Vdash_{lo} \{p\}\ c\ \{r\} \qquad s_{lo} \Vdash_{lo} \{r\}\ c'\ \{q\}}{s_{lo} \Vdash_{lo} \{p\}\ c\texttt{;}c'\ \{q\}}$$

(c) Seq

$$\frac{s_{lo} \Vdash_{lo} \{p\}\ c\ \{q\} \qquad s_{lo} \Vdash_{lo} \{p\}\ c'\ \{q\}}{s_{lo} \Vdash_{lo} \{p\}\ c\texttt{+}c'\ \{q\}}$$

(d) Choice

$$\frac{s_{lo} \Vdash_{lo} \{p\}\ c\ \{p\}}{s_{lo} \Vdash_{lo} \{p\}\ \texttt{(}c\texttt{)*}\ \{p\}}$$

(e) Iter

$$\frac{s_{lo} \Vdash_{lo} \{p\}\ c\ \{q\}}{s_{lo} \Vdash_{lo} \{p \bullet \mathsf{const}(f)\}\ c\ \{q \bullet \mathsf{const}(f)\}}$$

(f) Frame

$$\frac{f \text{ locally f-step preserving} \qquad s_{lo} \Vdash_{lo} \{p\}\ c\ \{q\}}{s_{lo} \Vdash_{lo} \{f(p)\}\ c\ \{f(q)\}}$$

(g) GenFrame

$$\frac{s_{lo}, \mathrm{id} \Vdash_{lo} \{p\}\{p'\} \qquad s_{lo} \Vdash_{lo} \{p'\}\ c\ \{q\}}{s_{lo} \Vdash_{lo} \{p\}\ c\ \{q\}}$$

(h) Cons-p

$$\frac{s_{lo}, \mathrm{id} \Vdash_{lo} \{q'\}\{q\} \qquad s_{lo} \Vdash_{lo} \{p\}\ c\ \{q'\}}{s_{lo} \Vdash_{lo} \{p\}\ c\ \{q\}}$$

(i) Cons-q

Figure 5.2: The LVF logic, with local variables left implicit.

Single-thread transitions. Let us define a labelled transition system that captures the possible steps we can take on a single thread when executing a LVF program. As in *Views*, the labels of the transitions correspond to atomic labels. Recall that, in LVF, these labels capture both shared-state and local-state operations.

Definition 5.12. The *single-thread labelled transition system* $- \xrightarrow{-}_{\mathsf{s}} -$ is the system given by the rules in Figure 5.3. (Coq: Transition in Starling.Views.Frameworks.Common.Language)

This transition system is the same as the *Views* transition system [42, Def. 3], with two differences. First, as there is no $\|$ operator, there are no parallel transitions. Second, there is no single-thread equivalent of the multi-step operational transition relation. The multi-step relation only exists in the multi-thread case, to allow for interleaving of multiple threads[3].

Multi-thread transitions. In an n-thread LVF program, there are n single-thread programs executing at once. As we assume sequential consistency, each transition of the n-thread program is a legal single-thread transition (given by $- \xrightarrow{-}_{\mathsf{s}} -$) in one of the n threads, leaving the other $n - 1$ programs unchanged. We can model n-thread programs as lists[4]

[3] In *Views*, this interleaving is explicit in the labelled transition relation itself.

[4] Modelling as lists helps us fold n-thread preconditions and postconditions into single views later on.

$$\frac{}{\texttt{<}\alpha\texttt{>} \xrightarrow{\alpha}_{\mathsf{s}} \texttt{skip}}$$

(a) Atom

$$\frac{P \xrightarrow{\widehat{\alpha}}_{\mathsf{s}} P'}{P \texttt{ ; } Q \xrightarrow{\widehat{\alpha}}_{\mathsf{s}} P' \texttt{ ; } Q} \qquad \frac{}{\texttt{skip ; } P \xrightarrow{\mathrm{id}}_{\mathsf{s}} P} \qquad \frac{}{\texttt{(}P\texttt{)*} \xrightarrow{\mathrm{id}}_{\mathsf{s}} P \texttt{ ; (}P\texttt{)*}}$$

(b) Seq-step (c) Seq-skip (d) Iter-step

$$\frac{}{\texttt{(}P\texttt{)*} \xrightarrow{\mathrm{id}}_{\mathsf{s}} \texttt{skip}} \qquad \frac{}{P + Q \xrightarrow{\mathrm{id}}_{\mathsf{s}} P} \qquad \frac{}{P + Q \xrightarrow{\mathrm{id}}_{\mathsf{s}} Q}$$

(e) Iter-skip (f) Ndt-left (g) Ndt-right

Figure 5.3: Single-thread labelled transitions.

of length n, fixing Tid as $\{0, \ldots, n-1\}$. To model updates to one thread, we can use a *list override* operator $l[i \mapsto x]$, meaning 'l, but with the value at index i replaced with x'[5].

The multi-thread transition system is straightforward. Every time we can update the t'th thread's program with a transition labelled l, we derive a transition labelled (l, t) that updates the list of n programs with the corresponding list override.

Definition 5.13. The *multi-thread labelled transition system* $- \xrightarrow{(-,-)}_{\mathsf{m}} -$ is the system defined by the rule:

$$\frac{t \in \mathsf{Tid} \qquad \mathbf{c}[t] \xrightarrow{\widehat{\alpha}}_{\mathsf{s}} c'}{\mathbf{c} \xrightarrow{(\widehat{\alpha}, t)}_{\mathsf{m}} \mathbf{c}[t \mapsto c']}$$

Local state as thread-partitioned shared state. Before we can define the multi-step operational transition relation, which gives the semantics of multi-thread LVF programs executing to completion, we need a model for the state of the abstract machine that executes such programs. As we've previously only dealt with the state visible to a single thread (one local state and one shared state), we need a way to encode the existence of n local states, each assigned to its corresponding thread as above.

Like the GLL, the model for machine states is a product between 'true' shared states (as we saw in μStarling), and maps from unique, linear thread identifiers (that is, Tid) to local states. The latter forms a parallel n-list of local sets L.

Definition 5.14. An *LVF machine state* $(\mathbf{l}, \sigma) : \mathrm{list}\ L \times S$ is an abstract representation of the state of a multi-threaded machine executing an n-thread LVF program, consisting of a list of n local states and a single shared state.

[5] See Definition A.5 for a more formal definition.

Multi-step operational transition relation. We can now define a multi-step relation similar to that given for *Views*. This, intuitively, is the reflexive transitive closure of $- \xrightarrow{(-,-)}_{\mathsf{m}} -$, with accounting for the initial and final state of the abstract machine.

Each time we take a step in thread $\mathtt{t}$ with label $\widehat{\alpha}$, we use the local semantic relation (Definition 5.2) to work out all possible resulting values of the shared state and $\mathtt{t}$'s local state. For each resulting machine state — the combination of the shared state, and the initial local state overridden with the new value for $\mathtt{t}$ —, we produce a transition.

Definition 5.15. The *multi-step operational transition relation* $- \xrightarrow{*}_{\mathsf{m}} -$ is the system defined by the rules:

$$\frac{}{\mathbf{c}, (\mathbf{l}, \sigma) \xrightarrow{*}_{\mathsf{m}} \mathbf{c}, (\mathbf{l}, \sigma)} \qquad \frac{\mathbf{c} \xrightarrow{(\widehat{\alpha}, \mathtt{t})}_{\mathsf{m}} \mathbf{c'} \qquad ((\mathbf{l}[\mathtt{t}], \sigma), (\mathtt{l}', \sigma')) \in [\![\widehat{\alpha}]\!]^{\mathsf{id}}_{lo}}{\mathbf{c}, (\mathbf{l}, \sigma) \xrightarrow{*}_{\mathsf{m}} \mathbf{c'}, (\mathbf{l}[\mathtt{t} \mapsto \mathtt{l}'], \sigma')}$$

5.3 Soundness of the LVF

For the LVF to be useful, we must show that it gives us the expected guarantees about programs. Like *Views* (but unlike the GLL), our target is proving safety properties — specifically, of multi-threaded LVF programs. Intuitively, this means that, from any machine state that satisfies a given precondition, any machine state we can reach where the program has terminated (that is, all threads are at `skip`) satisfies a given postcondition.

From local-view lists to single views

To formulate the safety judgement, we must be able to map the multi-threaded program's preconditions and postconditions — local-view lists — into state predicates. As LVF reification functions accept only single, non-local views, we must reduce the former to the latter.

First, we reduce all local views to non-local views. As the views and local states are parallel lists, we can apply each thread's view to the thread's local state in a zipping operation. We call said operation — from lists of local views, and local states, to lists of non-local views — Vlc (view list combine). We define Vlc formally in Definition A.11.

Next, we must combine the lists of non-local views into single views in a way that captures the multi-thread program's sum precondition and postcondition. Recall the *Views* parallel composition rule (from Figure 2.4):

$$\frac{\vdash \{\mathrm{p}_1\}\ \mathrm{c}_1\ \{\mathrm{q}_1\} \qquad \vdash \{\mathrm{p}_2\}\ \mathrm{c}_2\ \{\mathrm{q}_2\}}{\vdash \{\mathrm{p}_1 \bullet \mathrm{p}_2\}\ \mathrm{c}_1 \parallel \mathrm{c}_2\ \{\mathrm{q}_1 \bullet \mathrm{q}_2\}}$$

To safely enter a two-thread composition, we must establish the $\bullet$-join of their preconditions; to leave, we establish their postconditions' $\bullet$-join. The same principle holds for n-thread *LVF* programs. To join together view lists, we use a right fold[58] of $\bullet$ with base case ε; we represent this with a new *iterated view join* operator $\odot$ [6], and formally define it in Definition A.12. As $\bullet$ is commutative and associative, the order of thread views does not

[6] It is not sufficient to satisfy each thread's conditions in isolation: the join may introduce restrictions on how the threads interact and interfere with each other.

matter, and we can rearrange the iterated join of two[7] or more threads into the form $\nu \bullet \odot(\boldsymbol{\nu})$, where ν is an arbitrary thread's view and $\boldsymbol{\nu}$ holds all other threads' views. This means that, when a thread updates its view, we can frame over all other threads' views.

Let us now adapt the CVF's safety set-up for multi-thread LVF programs.

Definition 5.16. A list $\mathbf{c}$ of thread programs is safe with respect to parallel local view lists $\mathbf{p}$ of preconditions and $\mathbf{q}$ of postconditions when:

$$\forall \mathbf{l}, \mathbf{l}', \sigma, \sigma'.\ \sigma \in \lfloor \odot(\mathsf{Vlc}(\mathbf{p}, \mathbf{l})) \rfloor \wedge (\mathbf{c}, \mathbf{l}, \sigma) \xrightarrow{*}_{\mathsf{m}} (\mathtt{skip}, \mathbf{l}', \sigma') \implies \sigma' \in \lfloor \odot(\mathsf{Vlc}(\mathbf{q}, \mathbf{l}')) \rfloor$$

(Coq: ltsound in Starling.Views.Frameworks.LVF.Soundness)

Soundness of the program logic

Showing soundness of the program logic involves investigating each rule in Figure 5.2 in turn. The approach below involves, for most rules, first showing soundness of an explicit version of the rule. We can then derive the implicit rule's soundness by stepping inside the quantification over local states $\mathtt{l}$, applying Corollary 5.14.1 to each LVF action judgement in the hypotheses, and instantiating the hypotheses with $\mathtt{l}$ where necessary.

Skip rule. The semantic judgement over `skip` holds if, and only if, the given precondition entails the given postcondition:

$$\frac{\forall \mathtt{l}'.\ \mathsf{Sig}^{\uparrow}_{lo}(s_{lo}), (\mathsf{id}, \mathtt{l}, \mathtt{l}') \Vdash \{\mathtt{p}(\mathtt{l})\}\{\mathtt{q}(\mathtt{l}')\}}{s_{lo} \Vdash^{\mathtt{l}}_{lo} \{\mathtt{p}\}\ \mathtt{skip}\ \{\mathtt{q}\}}\ \textsc{Skip-ex}$$

Lemma 5.3. Skip-ex is sound in both directions.
(Coq: safe_skip_ex in Starling.Views.Frameworks.LVF.SafetyRules)

Corollary 5.3.1. Skip is sound in both directions.
(Coq: safe_skip in Starling.Views.Frameworks.LVF.SafetyRules)

Atomic rule. The atomic rule lifts action judgements to semantic judgements over programs that perform the respective atomic action, then terminate (as long as the action did not diverge). As with the skip rule, it works in both directions:

$$\frac{\forall \mathtt{l}'.\ \mathsf{Sig}^{\uparrow}_{lo}(s_{lo}), (\alpha, \mathtt{l}, \mathtt{l}') \Vdash \{\mathtt{p}(\mathtt{l})\}\{\mathtt{q}(\mathtt{l}')\}}{s_{lo} \Vdash^{\mathtt{l}}_{lo} \{\mathtt{p}\}\ <\alpha>\ \{\mathtt{q}\}}\ \textsc{Atomic-ex}$$

Lemma 5.4. Atomic-ex is sound in both directions. (Coq: safe_atomic_ex in Starling.Views.Frameworks.LVF.SafetyRules)

Corollary 5.4.1. Atomic is sound in both directions. (Coq: safe_atomic in Starling.Views.Frameworks.LVF.SafetyRules)

[7] One or more threads, if we have a views monoid.

Iteration. There is no sound explicit iteration rule (and so only an implicit rule exists). To see why, suppose our local state is an integer k, and we prove $s_{lo} \Vdash^{5}_{lo} \{p\}$ <k = k+1> $\{p\}$. From this, again starting from 5, can we prove the iteration ($s_{lo} \Vdash^{5}_{lo} \{p\}$ (<k=k+1>)* $\{p\}$)?

We cannot always do this. For example, let the meaning of $p(k)$ be $k < 10$. We can see that the original judgement holds: $k = 5$, $k' = k+1 = 6$, and both are less than 10. However, the iteration can (nondeterministically) expand to an arbitrary number of increments to k, which can result in $k \geqslant 10$. We must show that the original program satisfies its loop invariant p *regardless* of local state — hence, the implicit rule.[8]

Lemma 5.5. ITER is sound. (Coq: safe_loop in Starling.Views.Frameworks.LVF.SafetyRules)

Sequence. The sequence rule has a semi-explicit form: the proof of the first program can name an initial state, but the proof of the second must not. This is because the initial local state of the second program is the final local state of the first, and the LVF has no means to narrow the set of possible final states further than L.

$$\frac{s_{lo} \Vdash^{l}_{lo} \{p\}\ c_1\ \{r\} \qquad s_{lo} \Vdash_{lo} \{r\}\ c_2\ \{q\}}{s_{lo} \Vdash^{l}_{lo} \{p\}\ c_1;c_2\ \{q\}}\ \textsc{Seq-ex}$$

Lemma 5.6. SEQ-EX is sound. (Coq: safe_seq_ex in Starling.Views.Frameworks.LVF.SafetyRules)

Corollary 5.6.1. SEQ is sound. (Coq: safe_seq in Starling.Views.Frameworks.LVF.SafetyRules)

Single-atomic sequence. In the special case where the first program in a sequential composition is a lone atomic, we can consider just the local states that can result from the atomic's semantics as initial states for the second program.

$$\frac{s_{lo} \Vdash^{l}_{lo} \{p\}\ <\alpha>\ \{r\} \qquad \forall l'.\, l \overset{\alpha}{\rightsquigarrow} l' \implies s_{lo} \Vdash^{l'}_{lo} \{r\}\ c\ \{q\}}{s_{lo} \Vdash^{l}_{lo} \{p\}\ <\alpha>;c\ \{q\}}\ \textsc{Atomic-seq}$$

Lemma 5.7. ATOMIC-SEQ is sound.
(Coq: safe_seq_atom_ex in Starling.Views.Frameworks.LVF.SafetyRules)

Nondeterministic choice. The explicit nondeterministic choice rule is straightforward:

$$\frac{s_{lo} \Vdash^{l}_{lo} \{p\}\ c_1\ \{q\} \qquad s_{lo} \Vdash^{l}_{lo} \{p\}\ c_2\ \{q\}}{s_{lo} \Vdash^{l}_{lo} \{p\}\ c_1+c_2\ \{q\}}\ \textsc{Choice-ex}$$

Lemma 5.8. CHOICE-EX is sound. (Coq: safe_ndt_ex in Starling.Views.Frameworks.LVF.SafetyRules)

Corollary 5.8.1. CHOICE is sound. (Coq: safe_ndt in Starling.Views.Frameworks.LVF.SafetyRules)

[8] We would, instead, require just that $\{p\}\ c\ \{p\}$ holds for all local states reachable from the initial state, but this would complicate the logic.

Frame rules. The LVF frame rule, like its CVF equivalent, is a special case of the generalised frame rule. We can, then, use a sketch of that rule (given below) to justify the frame rule.

$$\frac{\mathsf{f}\text{ locally }\mathsf{f}\text{-step preserving} \qquad s_{lo} \Vdash^{\mathsf{l}}_{lo} \{\mathsf{p}\}\ \mathsf{c}\ \{\mathsf{q}\}}{s_{lo} \Vdash^{\mathsf{l}}_{lo} \{\mathsf{f}(\mathsf{p})\}\ \mathsf{c}\ \{\mathsf{f}(\mathsf{q})\}}\ \textsc{GenFrame-ex}$$

Definition 5.20 defines local f-step preservation — this delay in definition occurs as the definition depends on encoding the LVF into the CVF.

Lemma 5.9. GenFrame-ex is sound.

(Coq: safe_genframe_ex in Starling.Views.Frameworks.LVF.SafetyRules)

Corollary 5.9.1. GenFrame is sound.

(Coq: safe_genframe in Starling.Views.Frameworks.LVF.SafetyRules)

We can now prove the frame rule. The rule comes from the $\bullet$-closure property on action judgements, and as such is limited to framing over constant view functions.

$$\frac{s_{lo} \Vdash^{\mathsf{l}}_{lo} \{\mathsf{p}\}\ \mathsf{c}\ \{\mathsf{q}\}}{s_{lo} \Vdash^{\mathsf{l}}_{lo} \{\mathsf{p} \bullet \mathsf{const}(v)\}\ \mathsf{c}\ \{\mathsf{q} \bullet \mathsf{const}(v)\}}\ \textsc{Frame-ex}$$

Lemma 5.10. Frame-ex is sound. (Coq: safe_frame_ex in Starling.Views.Frameworks.LVF.SafetyRules)

Corollary 5.10.1. Frame is sound. (Coq: safe_frame in Starling.Views.Frameworks.LVF.SafetyRules)

Consequence. We give consequence as separate precondition (p) and postcondition (q) rules. This is because we can derive the combined rule from applying both rules in sequence. We start with the p-rule, as it is more straightforward to prove.

In the explicit form of the p-rule, the view entailment that strengthens the precondition need only hold for the initial local state.

$$\frac{\mathsf{Sig}^{\uparrow}_{lo}(s_{lo}), (\mathsf{id}, \mathsf{l}, \mathsf{l}) \Vdash \{\mathsf{p}(\mathsf{l})\}\{\mathsf{p}'(\mathsf{l})\} \qquad s_{lo} \Vdash^{\mathsf{l}}_{lo} \{\mathsf{p}'\}\ \mathsf{c}\ \{\mathsf{q}\}}{s_{lo} \Vdash^{\mathsf{l}}_{lo} \{\mathsf{p}\}\ \mathsf{c}\ \{\mathsf{q}\}}\ \textsc{Cons-p-ex}$$

Lemma 5.11. Cons-p-ex is sound.

(Coq: safe_cons_p_ex in Starling.Views.Frameworks.LVF.SafetyRules)

Corollary 5.11.1. Cons-p is sound. (Coq: safe_cons_p in Starling.Views.Frameworks.LVF.SafetyRules)

The q-rule is weaker, and harder to prove, as it involves the end of a program rather than the beginning. Again, we cannot work out the final local state of c, so the explicit form of the q-rule must consider all local states.

$$\frac{\forall \mathsf{l}'.\ \mathsf{Sig}^{\uparrow}_{lo}(s_{lo}), (\mathsf{id}, \mathsf{l}', \mathsf{l}') \Vdash \{\mathsf{q}'(\mathsf{l}')\}\{\mathsf{q}(\mathsf{l}')\} \qquad s_{lo} \Vdash^{\mathsf{l}}_{lo} \{\mathsf{p}\}\ \mathsf{c}\ \{\mathsf{q}'\}}{s_{lo} \Vdash^{\mathsf{l}}_{lo} \{\mathsf{p}\}\ \mathsf{c}\ \{\mathsf{q}\}}\ \textsc{Cons-q-ex}$$

Lemma 5.12. Cons-q-ex is sound.

(Coq: safe_cons_q_ex in Starling.Views.Frameworks.LVF.SafetyRules)

Corollary 5.12.1. Cons-q is sound. (Coq: safe_cons_q in Starling.Views.Frameworks.LVF.SafetyRules)

Soundness. The main LVF soundness property is as follows:

Theorem 5.13. Let $\mathbf{p}$, $\mathbf{c}$, $\mathbf{q}$ be parallel lists giving the preconditions, single-thread programs, and postconditions of a LVF multi-threaded program. Then, for any s_{lo}, $s_{lo} \Vdash_{lo\mathsf{M}} \{\mathbf{p}\}\ \mathbf{c}\ \{\mathbf{q}\}$ implies the safety of $\mathbf{c}$ with respect to $\mathbf{p}$ and $\mathbf{q}$, using s_{lo}'s reifier and local semantics.
(Coq: msafe_ltsound in Starling.Views.Frameworks.LVF.Soundness)

5.4 Embedding the LVF in the CVF

The previous sections presented the LVF as a separate logic framework from the CVF. To use LVF with Starling (which depends on an axiom-sound CVF instance backing each frontend), we must either retarget large parts of the theory we built in Chapters 3 and 4 atop the LVF, or build a formal link between the LVF and CVF.

We can encode LVF instances as CVF instances, if we use the intuition that local state is a layer of parametrisation over shared-state views and actions. The core idea is to map each LVF axiom to the set of all possible instantiations of that axiom with local pre–and post-states, replacing each label with a triple of label and local states.

Definition 5.17. The *local axiom lift* $\mathsf{Ax}^{\uparrow}_{lo}$ lifts each LVF axiom to a set of CVF axioms by explicitly quantifying over local state and moving it into the atomic action:

$$\mathsf{Ax}^{\uparrow}_{lo} : \mathsf{AHoare}(\mathsf{L} \to V, A) \leftrightarrow \mathsf{AHoare}(V, (A^{\mathsf{id}} \times \mathsf{L} \times \mathsf{L}))$$

$$\mathsf{Ax}^{\uparrow}_{lo} \stackrel{\text{def}}{=} \left\{ \left(\langle p \rangle\ \widehat{\alpha}\ \langle q \rangle \mapsto \langle p(l) \rangle\ (\widehat{\alpha}, l, l')\ \langle q(l') \rangle \right) \;\middle|\; \begin{array}{l} p, q \in \mathsf{L} \to V \\ \widehat{\alpha} \in A^{\mathsf{id}} \\ l, l' \in \mathsf{L} \end{array} \right\}$$

There are some subtleties here. First, id maps not to id, but to the action $\langle \mathsf{id}, l, l' \rangle$. This is because the semantics of id requires that $l' = l$, which we cannot model without having the local states available. Second, one LVF axiom can map to an unbounded set of CVF axioms: this makes this encoding unsuitable as a way to automate LVF proofs. Third, by moving the local post-state l' into the atomic action, we appear to move it into a negative position in the triple. In fact, this copy of the post-state exists just to restrict the shared state transformer that the atomic action produces (see Definition 5.18); the view $q(l')$ carries the post-state's effect on the final view into the positive position of the triple.

Semantics

For the above idea to work, we must give a shared-state semantics to triples $(\widehat{\alpha}, l, l')$ that reflects the local-state semantics of $\widehat{\alpha}$ when moving from local state l to local state l'. Since we quantify over all possible local states, not just those mapped by $\widehat{\alpha}$'s local semantics, we must make sure the encoded semantics diverges properly when it meets invalid local states. We do so through applying a *local semantics lift*, or $\mathsf{Sem}^{\uparrow}_{lo}$, to the LVF semantics.

Definition 5.18. The *local semantics lift* $\mathsf{Sem}^{\uparrow}_{lo}$ lifts a local semantics to a shared-state encoding by moving the local states into the atomic label:

$$\mathsf{Sem}^{\uparrow}_{lo} : (A \to ((L \times S) \leftrightarrow (L \times S))) \to ((A^{\mathsf{id}} \times L \times L) \to S \to \mathbb{P}(S))$$
$$\mathsf{Sem}^{\uparrow}_{lo}([\![-]\!]_{lo})((\alpha, l, l'))(\sigma) \stackrel{\text{def}}{=} \{\, \sigma' \mid ((l, \sigma), (l', \sigma')) \in [\![\alpha]\!]^{\mathsf{id}}_{lo} \,\}$$

(Coq: lsem_erase in Starling.Views.Frameworks.LVF.Signatures)

Signatures

Using the local semantics erase, and the encoding of local state in atomic labels, we can lift local signatures to CVF signatures (as in Definition 2.13).

Definition 5.19. The *local signature lift* $\mathsf{Sig}^{\uparrow}_{lo}$ lifts local signatures to CVF signatures:

$$\mathsf{Sig}^{\uparrow}_{lo} : \mathsf{LSig}(V, A, L, S) \to \mathsf{Sig}(V, (A \times L \times L), S)$$
$$\mathsf{Sig}^{\uparrow}_{lo}((\bullet, \equiv, \lfloor - \rfloor, [\![-]\!]_{lo})) \stackrel{\text{def}}{=} (\bullet, \equiv, \lfloor - \rfloor, \mathsf{Sem}^{\uparrow}_{lo}([\![-]\!]_{lo}))$$

(Coq: lsig_erase in Starling.Views.Frameworks.LVF.Signatures)

Action judgement

We can relate the LVF action judgement over a local signature to the CVF action judgement over its lifting. This mapping introduces an explicit quantification over the local pre–and post-state, treating each as part of the CVF atomic label.

Lemma 5.14. We can relate the LVF and CVF Views–Hoare judgements as follows:

$$s_{lo}, \widehat{\alpha} \Vdash_{\mathsf{LVFH}} \{p\}\{q\} \quad \iff \quad \forall l, l'.\, \mathsf{Sig}^{\uparrow}_{lo}(s_{lo}), (\widehat{\alpha}, l, l') \Vdash_{\mathsf{VFH}} \{p(l)\}\{q(l')\}$$

Appendix B.3 gives a proof.

Corollary 5.14.1. The LVF and CVF action judgements relate as follows:

$$s_{lo}, \widehat{\alpha} \Vdash_{lo} \{p\}\{q\} \quad \iff \quad \forall l, l'.\, \mathsf{Sig}^{\uparrow}_{lo}(s_{lo}), (\widehat{\alpha}, l, l') \Vdash \{p(l)\}\{q(l')\}$$

Proof sketch. Apply Lemma 5.14 to both sides of the judgement, rewriting $(p \bullet \mathsf{const}(v))(l)$ to $p(l) \bullet v$, and similarly for q and l', in the RHS. □

This encoding introduces a potentially unbounded pair of quantifiers in the atomic label, and, therefore, axiom soundness. Proof rule templates targeting the LVF through this encoding, then, must take care to handle these quantifiers in an automation-friendly manner.

Generalised framing

The CVF framing operation $\lambda x.\, x * v$ is an f-*step preserving* function [42, Prop. M]: a function $f : V \to V$ where action judgement closes over applying f to p and q. The CVF frame rule generalises to any such f, and this lets us encode non-trivial framing properties such as that

of rely/guarantee. We can define a similar idea for LVF, which backs the earlier definition of generalised framing. The LVF f-step preservation must consider the same local variables on both sides of the closure, so the definition uses the CVF action-judgement encoding.

Definition 5.20. A function $\mathsf{f} : (\mathsf{L} \to \mathsf{V}) \to (\mathsf{L} \to \mathsf{V})$ is *locally f-step preserving* if:

$$\forall \mathsf{l}, \mathsf{l}' : \mathsf{L},\ \mathsf{p}, \mathsf{q} : \mathsf{L} \to \mathsf{V},\ \widehat{\alpha} : \mathsf{A}^{\mathsf{id}}.$$
$$\mathsf{Sig}^{\uparrow}_{lo}(\mathsf{s}), (\widehat{\alpha}, \mathsf{l}, \mathsf{l}') \Vdash \{\mathsf{p}(\mathsf{l})\}\{\mathsf{q}(\mathsf{l}')\} \implies \mathsf{Sig}^{\uparrow}_{lo}(\mathsf{s}), (\widehat{\alpha}, \mathsf{l}, \mathsf{l}') \Vdash \{\mathsf{f}(\mathsf{p})(\mathsf{l})\}\{\mathsf{f}(\mathsf{q})(\mathsf{l}')\}$$

Instances

We can encode LVF instances in the CVF using the signature and axiom lifts.

Definition 5.21. The *local instance lift* $\mathsf{Inst}^{\uparrow}_{lo}$ lifts LVF instances to views instances:

$$\mathsf{Inst}^{\uparrow}_{lo} : \mathsf{LInst}(\mathsf{V}, \mathsf{A}, \mathsf{L}, \mathsf{S}) \to \mathsf{Inst}(\mathsf{V}, (\mathsf{A} \times \mathsf{L} \times \mathsf{L}), \mathsf{S})$$
$$\mathsf{Inst}^{\uparrow}_{lo}(\mathsf{s}_{lo}, \mathsf{T}_{lo})) \stackrel{\text{def}}{=} \left(\mathsf{Sig}^{\uparrow}_{lo}(\mathsf{s}_{lo}), \left\{ \langle \mathsf{p} \rangle\ \widehat{\alpha}\ \langle \mathsf{q} \rangle \;\middle|\; \begin{array}{l} \exists \langle \mathsf{p}' \rangle\ \widehat{\alpha}'\ \langle \mathsf{q}' \rangle \in \mathsf{T}_{lo}. \\ \quad (\langle \mathsf{p}' \rangle\ \widehat{\alpha}'\ \langle \mathsf{q}' \rangle, \langle \mathsf{p} \rangle\ \widehat{\alpha}\ \langle \mathsf{q} \rangle) \in \mathsf{Ax}^{\uparrow}_{lo} \end{array} \right\} \right)$$

(Coq: instance_lvf_to_cvf in Starling.Views.Frameworks.LVF.Instances)

Lemma 5.15. A CVF encoding of an LVF instance is axiom-sound if, and only if, the original instance was locally axiom sound.

5.5 Local states in backends

Let us now port Starling to the LVF. As we can encode the LVF in the CVF, much of the theory carries over with a small amount of encoding, but we do need to make some changes to both Starling backends and frontends to accommodate local-state reasoning.

This section discusses how to quantify over local state in backends. Not doing so (in other words, quantifying in the frontend) limits us in practice to a finite, enumerable set of local states — the same limit we hit if we encode our LVF axioms as CVF axioms.

Local state in local context

We can add local-state quantification by encoding it into LCtx. Such an encoding is required because local state applies to backend conditions in a different manner from shared state: while we distribute the shared pre-state to the weakest precondition and the shared post-state to the goal, we send both local states to the weakest precondition. Specifically, local pre-state goes to the original precondition, and post-state to the original postcondition.

Let us define LCtx as the triple $\mathsf{L} \times \mathsf{L} \times \mathsf{LCtx}'$, where LCtx' is some other local context set (or $\emptyset$). Here, the first local state represents the pre-state, and the second represents the post-state. We rely on the frontend to use the correct state when expanding each view and action. Defining the local-state encoding like this makes adding local state on top of an existing backend set-up straightforward.

Functions as propositions and relations

If we have backends over shared state only, we want to be able to make them work with local state without changing their expression languages to add native support. Specifically, given the appropriate local state up-front, we should be able to reduce backend expressions over local state to expressions that only mention shared state.

We can model the above encoding by lifting the backend's expressions to functions with local state as their domain. In fact, we can lift expressions over any local context LCtx to a larger local context $\mathsf{LCtx}' \times \mathsf{LCtx}$ by turning them into functions with domain LCtx'. The lifting is pointwise, and similar to how we lift views to functions:

$$\mathrm{true}_{\mathsf{Pr}} \stackrel{\mathrm{def}}{=} \mathsf{const}(\mathrm{true}_{\mathsf{Pr}}) \qquad e_1 \wedge_{\mathsf{Pr}} e_2 \stackrel{\mathrm{def}}{=} \lambda x_l.\, e_1(x_l) \wedge_{\mathsf{Pr}} e_2(x_l)$$

$$[\![e]\!]_{\mathsf{Pr}}(x_g)((x'_l, x_l)) \stackrel{\mathrm{def}}{=} [\![e(x'_l)]\!]_{\mathsf{Pr}}(x_g)(x'_l) \qquad \mathrm{id}_{\mathsf{Rl}} \stackrel{\mathrm{def}}{=} \mathsf{const}(\mathrm{id}_{\mathsf{Rl}})$$

$$\emptyset_{\mathsf{Rl}} \stackrel{\mathrm{def}}{=} \mathsf{const}(\emptyset_{\mathsf{Rl}}) \qquad [\![e]\!]_{\mathsf{Rl}}(x_g)((x'_l, x_l)) \stackrel{\mathrm{def}}{=} [\![e(x'_l)]\!]_{\mathsf{Rl}}(x_g)(x'_l)$$

(Coq: Starling.Backend.Transformers.Function)

Erasing functions in backend conditions

Representing local-state parametrisation as predicate and relation functions lets us reduce sets of functional backend conditions to sets of backend conditions over their codomains in a straightforward manner. We call this reduction the *function erasure*, or FErase.

Definition 5.22. Given a set X of backend verification conditions over functions, a verification condition is in $\mathsf{FErase}(X)$ if, and only if, we can reach it by applying some valid local context to a condition in X:

$$\mathsf{FErase}(X) \stackrel{\mathrm{def}}{=} \{\, \langle\!\langle w(x_l) \rangle\!\rangle\ c(x_l)\ \langle\!\langle g(x_l) \rangle\!\rangle \mid \langle\!\langle w \rangle\!\rangle\ c\ \langle\!\langle g \rangle\!\rangle \in X, x_l \in \mathsf{LCtx} \,\}$$

(Coq: fun_erase in Starling.Backend.Transformers.Function)

Lemma 5.16. A backend condition satisfies the verification-condition Hoare judgement if, and only if, its erasure also satisfies it for all local contexts:

$$(x_g, c) \Vdash_{\mathsf{EVFH}} \{w\}\{g\} \iff \forall x_l.\, (x_g, c(x_l)) \Vdash_{\mathsf{EVFH}} \{w(x_l)\}\{g(x_l)\}$$

(Coq: bvhoare_fun in Starling.Backend.Transformers.Function)

Corollary 5.16.1. A set X of functional backend conditions satisfies the verification-condition Hoare judgement if, and only if, each condition in its function erasure also satisfies it:

$$\begin{pmatrix} \exists x_g.\ \forall \langle\!\langle w \rangle\!\rangle\ c\ \langle\!\langle g \rangle\!\rangle \in X. \\ (x_g, c) \Vdash_{\mathsf{EVFH}} \{w\}\{g\} \end{pmatrix} \iff \begin{pmatrix} \exists x_g.\ \forall \langle\!\langle w \rangle\!\rangle\ c\ \langle\!\langle g \rangle\!\rangle \in \mathsf{FErase}(X). \\ (x_g, c) \Vdash_{\mathsf{EVFH}} \{w\}\{g\} \end{pmatrix}$$

(Coq: bvhoare_fun_erase in Starling.Backend.Transformers.Function)

Since we build the erased set with an existential quantifier over local contexts, the set is neither guaranteed bounded nor guaranteed finite. This poses a problem with solving a functional system by solving its erasure: if we rely on sending each erased condition to the solver individually to get a result for the program, our decision process will not terminate. As a result, we will avoid the function erasure in the frontends we build in Chapter 6.

5.6 $_{\text{lo}}$Starling

Having laid the groundwork for a local-state extension to $_{\mu}$Starling in the previous sections, we now build the extension itself. $_{\text{lo}}$Starling uses the same outline decomposition technique as $_{\mu}$Starling, but adds local state and support for using local state to choose between views, and targets the local backends we built in § 5.5.

Using proof outlines with the LVF

When frontends appeared in § 4.2, they formed the middle of a pipeline from outline decompositions (§ 3.2) to backends. To keep this pipeline when using the LVF, we must make sure that we decompose valid LVF outlines into valid triple sets. As the LVF and program logics are almost identical, the original decomposition rules still work, though we must prove them sound in the new framework. As the LVF logic sits atop the single-thread semantic judgement, each outline will represent the program of one thread: to prove a multi-thread program, we prove multiple outlines in the same context, and use Corollary 5.2.1 to derive a multi-thread result.

A problem with re-using our existing outline decomposition is that, as some valid CVF programs are invalid LVF programs, not all valid CVF outlines are valid LVF outlines. Specifically, we forbid parallel compositions, and the frame rule only works for constant frames. We can solve this with a filtering predicate.

Definition 5.23. An outline is *LVF-compatible* if its assertions are view functions, it has no parallel compositions, and each frame view is equivalent to $\mathsf{const}(x)$ for some x.

We can extract LVF single-thread programs from such outlines. Combined with accessor notation (Definition 3.1), this lets us state the soundness result we need to use the outlines to represent LVF program proofs.

Definition 5.24. The *LVF program extraction* $\mathsf{LVFExtract}(o)$, over LVF compatible o, is:

$$\begin{aligned}
\mathsf{LVFExtract}(\{p\}\ \mathsf{skip}\ \{q\}) &= \texttt{skip} \\
\mathsf{LVFExtract}(\{p\}\ \langle\alpha\rangle\ \{q\}) &= \texttt{<}\alpha\texttt{>} \\
\mathsf{LVFExtract}(\{p\}\ \mathsf{frame}\ v\ \mathsf{in}\ (o)\ \{q\}) &= \mathsf{LVFExtract}(o) \\
\mathsf{LVFExtract}(\{p\}\ (o)^*\ \{q\}) &= \texttt{(}\mathsf{LVFExtract}(o)\texttt{)*} \\
\mathsf{LVFExtract}(\{p\}\ o_1; o_2\ \{q\}) &= \mathsf{LVFExtract}(o_1)\ \texttt{;}\ \mathsf{LVFExtract}(o_2) \\
\mathsf{LVFExtract}(\{p\}\ o_1 + o_2\ \{q\}) &= \mathsf{LVFExtract}(o_1)\ \texttt{+}\ \mathsf{LVFExtract}(o_2)
\end{aligned}$$

Theorem 5.17. For all local-axiom-sound instances $\mathfrak{i}_\mathfrak{l}$, and outlines $\mathfrak{o}$ over $\mathfrak{i}_\mathfrak{l}$'s views, local states and atomic actions, $\mathsf{oflat}(\mathfrak{o}) \subseteq \mathfrak{i}_\mathfrak{l}.\mathsf{T} \implies \mathfrak{i}_\mathfrak{l} \Vdash_{lo} \{\mathfrak{o}.\mathfrak{p}\}\ \mathsf{LVFExtract}(\mathfrak{o})\ \{\mathfrak{o}.\mathfrak{q}\}$.

To show this, we can use similar decomposition reasoning to that we used for normal outlines, now targeting the program logic in § 5.1.

Defining $_{lo}$Starling's frontend

The definition of $_{lo}$Starling's frontend uses the same high-level approach as that of $_{\mu}$Starling in § 4.4, but with some differences:

- atomic Hoare triples now have assertions from $\mathsf{L} \to \mathsf{V}$;
- backend conditions now come from the function-lifted expression languages from § 5.5. To use a $_{\mu}$Starling-compatible solver with $_{lo}$Starling, we must take the function erasure of the generated set, which is not bounded in general;
- syntactic atomic definers now return function-lifted relation expressions (view definers still define shared-state views, and so use normal proposition expressions);
- to make sure id is the identity on local states, it has a different translation.

Fixing identity. In $_{\mu}$Starling, to lift the atomic definer ASDef over labels, we returned $\mathsf{id}_{\mathsf{Rl}}$ for id, and ASDef's definition otherwise. We might try to do the same for $_{lo}$Starling, returning the function-lifted version of $\mathsf{id}_{\mathsf{Rl}}$. Unfortunately, as we defined this as $\mathsf{const}(\mathsf{id}_{\mathsf{Rl}})$, the resulting expression ignores local states and considers only whether the *shared* states are equal. This means that id would behave as havoc over local states.

We could fix this by redefining the function lifting to return $\emptyset_{\mathsf{Rl}}$ when the local state has changed, and $\mathsf{id}_{\mathsf{Rl}}$ otherwise. This would force us to specialise the lifting to work only when using the local context for local states, and would not scale if we added more local context later on. Instead, let us build a new lifting function.

Definition 5.25. Given an syntactic atomic definer ASDef over $(\mathsf{L} \times \mathsf{L}) \to \mathsf{E}_{\mathsf{Rl}}$, $\mathsf{ASDef}^{\mathsf{id}}_{lo}$ is:

$$\mathsf{ASDef}^{\mathsf{id}}_{lo}(\widehat{\alpha})((\mathfrak{l},\mathfrak{l}')) \stackrel{\text{def}}{=} \begin{cases} \mathsf{id}_{\mathsf{Rl}} & \widehat{\alpha} = \mathrm{id} \ \wedge\ \mathfrak{l} = \mathfrak{l}' \\ \emptyset_{\mathsf{Rl}} & \widehat{\alpha} = \mathrm{id} \ \wedge\ \mathfrak{l} \neq \mathfrak{l}' \\ \mathsf{ASDef}(\alpha)((\mathfrak{l},\mathfrak{l}')) & \text{otherwise} \end{cases}$$

(Coq: ls_label_avd in Starling.Logics.LoStarling)

As the built expression depends on whether $\mathfrak{l} = \mathfrak{l}'$, that equality must be decidable.

Backend decomposition. As before, each verification condition represents an atomic Hoare triple and a goal view. The backend decomposition mirrors its $_{\mu}$Starling counterpart:

Definition 5.26. The function $\mathsf{D}^{lo}_{\mathsf{g}}$ translates a local atomic Hoare triple, given a goal view g, into a backend condition:

$$\mathsf{D}^{lo}_{\mathsf{g}} : \mathsf{AHoare}(\mathsf{L} \to \mathsf{V}, \mathsf{A}) \to \mathsf{V} \to \mathsf{VConds}((\mathsf{L} \times \mathsf{L}) \to \mathsf{E}_{\mathsf{Pr}}, (\mathsf{L} \times \mathsf{L}) \to \mathsf{E}_{\mathsf{RI}})$$

$$\mathsf{D}^{lo}_{\mathsf{g}}(\langle \mathsf{p} \rangle\ \widehat{\alpha}\ \langle \mathsf{q} \rangle)(\mathsf{g}) \stackrel{\text{def}}{=}$$

$$\langle\!\langle \lambda(\mathsf{l}, \mathsf{l}').\, \mathsf{sdReify}(\mathsf{d})(\mathsf{p}(\mathsf{l}) \bullet (\mathsf{g} \setminus \mathsf{q}(\mathsf{l}'))) \rangle\!\rangle\ \mathsf{ASDef}^{\mathsf{id}}_{lo}(\widehat{\alpha})\ \langle\!\langle \mathsf{const}(\mathsf{sdDef}(\mathsf{d})(\mathsf{g})) \rangle\!\rangle$$

(Coq: ls_decomp_ni_single in Starling.Logics.LoStarling)

If we take $\mathsf{D}^{lo}_{\mathsf{g}}$ for every defining (shared-state) view g, we can build a bounded set of function-lifted backend conditions.

Definition 5.27. The function $\mathsf{D}^{lo}_{\mathsf{b}}$ translates an atomic Hoare triple into a set of backend conditions that cover both sequential safety and non-interference:

$$\mathsf{D}^{lo}_{\mathsf{b}}(\langle \mathsf{p} \rangle\ \widehat{\alpha}\ \langle \mathsf{q} \rangle) \stackrel{\text{def}}{=}$$

$$\{ \langle\!\langle \lambda(\mathsf{l}, \_).\, \mathsf{sdReify}(\mathsf{d})(\mathsf{p}(\mathsf{l})) \rangle\!\rangle\ \mathsf{ASDef}^{\mathsf{id}}_{lo}(\widehat{\alpha})\ \langle\!\langle \lambda(\_, \mathsf{l}').\, \mathsf{sdReify}(\mathsf{d})(\mathsf{q}(\mathsf{l}')) \rangle\!\rangle \}$$

$$\cup \{ \mathsf{D}^{lo}_{\mathsf{g}}(\langle \mathsf{p} \rangle\ \widehat{\alpha}\ \langle \mathsf{q} \rangle)(\mathsf{g}) \mid \exists e.\, \mathsf{d} = \mathsf{d}_1 +\!\!+ \langle (\mathsf{g}, e) \rangle +\!\!+ \mathsf{d}_2 \}$$

(Coq: ls_decomp in Starling.Logics.LoStarling)

Views instances. In $_{\mu}$Starling, each flattened outline's atomic triples formed the axioms of its underlying views instance. While the $_{lo}$Starling outline flattening gives us valid LVF axioms, and we could directly build a LVF instance over them, the template-based approach only works with CVF axioms and instances. To use it without modification, we can first build CVF instances, then show that they correspond to the LVF instances we would have built.

As before, we can build templates from backend decompositions using fTemp. The $_{lo}$Starling templates use $\mathsf{Ax}^{\uparrow}_{lo}$ (Definition 5.17) to map the incoming triples to CVF axioms.

Definition 5.28. The *$_{lo}$Starling template*, given a global context x_{g} and the appropriate definers, is $\mathsf{fTemp}(\mathsf{D}^{lo}_{\mathsf{b}})(\mathsf{Ax}^{\uparrow}_{lo})(\mathsf{x}_{\mathsf{g}})$.

CVF instances generated using this template are axiom sound, by connection to the defining-views template. Through the link between CVF and LVF instances (Lemma 5.15), this shows that valid $_{lo}$Starling proofs correspond to local-axiom-sound LVF instances.

Lemma 5.18. The $_{lo}$Starling template yields subsets of the defining-views template.

Corollary 5.18.1. The $_{lo}$Starling template produces axiom-sound CVF instances.

Corollary 5.18.2. The $_{lo}$Starling template produces local-axiom-sound LVF instances.

(The Coq development does not have direct mechanisations of the above results, but does have results for the simplified form over views monoids, in which the backend decomposition omits the local safety verification condition: (Coq: ls_template_ni in Starling.Logics.LoStarling), (Coq: ls_strengthens_defining_ni in Starling.Logics.LoStarling).)

As with μStarling, we can build outline rules by combining the flattening, frontend, and backend. These rules entail the LVF single-thread semantic judgement on their respective outlines, by the *Views* logic; the definition of flattening (§ 3.2); and frontend soundness.

5.7 Atomicity of local actions

The LVF assumes that all primitive actions are atomic (and sequentially consistent): this assumption is unrealistic for actions that only modify local state. Atomic, sequentially consistent actions need expensive memory fences, making them unpalatable for operations that are not visible to other threads. In practice, the default semantics of primitive actions in languages such as C is non-atomic, as the assumption is that most state is local.

This section explores some options for reasoning about non-atomic local actions in the LVF. These results do not need a specific action set or semantics, but do make some high-level assumptions. This makes the guarantees available weak, but generalisable.

Local actions and view preservation

Let us formally define what it means for actions[9] to be *local*. Local actions have no effect on shared state. We also forbid local actions from depending on shared-state observations: if a local action maps l to l' when the state is σ, it must do so when the state is σ', too. In practice, programs can load shared-state dependencies into local state using atomic actions.

Definition 5.29. An atomic action α is *local* if, and only if:

$$\forall l, l' : L,\ \sigma, \sigma' : S.\ ((l, \sigma), (l', \sigma')) \in [\![\alpha]\!]_{lo} \implies \sigma = \sigma' \wedge (\forall \sigma'' : S,\ ((l, \sigma''), (l', \sigma'')) \in [\![\alpha]\!]_{lo})$$

(Coq: is_local_cmd in Starling.Views.Frameworks.LVF.Atomicity)

Local actions preserve any shared-state view held before the action executes. We can capture this in a series of inference rules:

$$\frac{\alpha \text{ local}}{\mathsf{Sig}^{\uparrow}_{lo}(s_{lo}), (\alpha, l, l') \Vdash_{\mathsf{VFH}} \{v\}\{v\}}\ \textsc{LPreserve-vh}$$

$$\frac{\alpha \text{ local}}{\mathsf{Sig}^{\uparrow}_{lo}(s_{lo}), (\alpha, l, l') \Vdash \{p(l)\}\{\mathsf{const}(p(l))(l')\}}\ \textsc{LPreserve-act}$$

$$\frac{\alpha \text{ local}}{s_{lo} \Vdash^{l}_{lo} \{p\}\ <\alpha>\ \{\mathsf{const}(p(l))\}}\ \textsc{LPreserve-ex}$$

Lemma 5.19. LPreserve-vh is sound.

(Coq: local_cmd_view_stability_hoare in Starling.Views.Frameworks.LVF.Atomicity)

We can validate the other rules using LPreserve-vh, Atomic-ex, and the structure of action judgements.

[9] id is inherently local — it has no effect on state whatsoever — so we only consider actions.

Corollary 5.19.1. LPRESERVE-ACT is sound.

(Coq: local_cmd_view_stability in Starling.Views.Frameworks.LVF.Atomicity)

Corollary 5.19.2. LPRESERVE-EX is sound.

Atomic compositions

Adding native non-atomic local-state actions to the LVF would complicate it. Instead, we observe that, in many programming models, non-atomic local actions are sequential compositions of atomic local actions. For example, we can rewrite the action `x = y++` as `x = y; y++`: these two actions are more likely to be atomic in a given model than the original.

Definition 5.30. An action α *composes* actions α_1 and α_2 (denoted $\alpha \sim (\alpha_1; \alpha_2)$) if:

$$\alpha \sim (\alpha_1; \alpha_2) \overset{\text{def}}{\iff} \forall l_1, l_3 : L,\ s_1, s_3 : S.$$
$$((l_1, s_1), (l_3, s_3)) \in [\![\alpha]\!]_{lo} \iff \begin{pmatrix} \exists l_2 : L,\ s_2 : S.\ ((l_1, s_1), (l_2, s_2)) \in [\![\alpha_1]\!]_{lo} \\ \qquad \wedge\ ((l_2, s_2), (l_3, s_3)) \in [\![\alpha_2]\!]_{lo} \end{pmatrix}$$

(Coq: atomic_comp in Starling.Views.Frameworks.LVF.Atomicity)

Atomic expansion rules

When an action is an atomic composition of a local action and another action (local or not), we can in certain cases use proofs over that action to infer proofs over sequential compositions of the other two. The inference rules, and the restrictions they place on their actions, differ depending on the local action's position in the composition. The most straightforward case is when the local action comes first or both actions are local:

$$\frac{\alpha \sim (\alpha_1; \alpha_2) \qquad \alpha_1 \text{ local} \qquad s_{lo} \Vdash^{l}_{lo} \{p\}\ \texttt{<}\alpha\texttt{>}\ \{q\}}{s_{lo} \Vdash^{l}_{lo} \{p\}\ \texttt{<}\alpha_1\texttt{>;<}\alpha_2\texttt{>}\ \{q\}}\ \textsc{Latom-l-ex}$$

Lemma 5.20. LATOM-L-EX is sound.

(Coq: local_pre_cmd_ex in Starling.Views.Frameworks.LVF.Atomicity)

When the local action comes after a non-local action, we must make stronger assumptions; we need to predict the local state after the action given the local state before the action. As such, we must be able to model the action as a deterministic, total function $f : L \to L$.

Definition 5.31. An action α is *deterministic*, denoted $\det(\alpha)$, if it maps a local and shared pre-state to at most one corresponding pair of local and shared post-state:

$$\det(\alpha) \overset{\text{def}}{\iff} \forall l, l', l'' : L,\ \sigma, \sigma', \sigma'' : S.\ ((l, \sigma), (l', \sigma')) \in [\![\alpha]\!]_{lo} \wedge ((l, \sigma), (l'', \sigma'')) \in [\![\alpha]\!]_{lo} \implies l' = l'' \wedge \sigma' = \sigma''$$

(Coq: is_deterministic in Starling.Views.Frameworks.LVF.Atomicity)

Definition 5.32. A function pair (f, g) *models* an action α, denoted $\mathsf{mdl}(\alpha, f, g)$, if α maps each pair of local and shared state to the projection of the local state through f and shared state through g:

$$\mathsf{mdl}(\alpha, f, g) \overset{\text{def}}{\iff} \forall l : L,\ \sigma : S.\ ((l, \sigma), (f(l), g(\sigma))) \in [\![\alpha]\!]_{lo}$$

(Coq: is_fun in Starling.Views.Frameworks.LVF.Atomicity)

When an action is both deterministic and modelled by a function pair, every state pair (l, σ) maps to $(f(l), g(\sigma))$, without divergence or nondeterminism. This lets us predict exactly which local state we need for the intermediate view.

Lemma 5.21. If (f, id), for some f, models a deterministic action, that action is local. (Coq: is_det_fun_local in Starling.Views.Frameworks.LVF.Atomicity)

With these components, we can introduce an inference rule:

$$\frac{\alpha \sim (\alpha_1; \alpha_2) \qquad \mathsf{det}(\alpha_2) \qquad \mathsf{mdl}(\alpha_2, f, \mathrm{id}) \qquad s_{lo} \Vdash^{l}_{lo} \{p\}\ <\alpha>\ \{q\}}{s_{lo} \Vdash^{l}_{lo} \{p\}\ <\alpha_1>;<\alpha_2>\ \{q\}}\ \textsc{Latom-r-ex}$$

Lemma 5.22. LATOM-R-EX is sound.
(Coq: local_post_cmd_ex in Starling.Views.Frameworks.LVF.Atomicity)

5.8 Summary

This chapter introduced the *Local Views Framework* (LVF), a modified version of the CVF that adds native support for thread-local state. This framework adapts ideas from Khyzha *et al.*'s general linearisability logic, recasting them in Views's safety-properties setting. We can embed the core of the LVF in the CVF, letting us re-use large parts of meta-theory.

The chapter then used the LVF to produce $_{\text{lo}}$Starling. This frontend showed that we can, in theory, extend logics like $_{\mu}$Starling with local-state reasoning. Using $_{\text{lo}}$Starling in practice, though, is difficult. As it returns backend conditions over arbitrary functions from local state, we must either use a solver that can accept any such function, or use function erasure to generate a potentially unbounded set of conditions over shared state. Both options limit the usefulness of the approach, ruling out combinations of backend and local state model.

Another approach is to limit the set of local-state functions we can take as assertions. If we choose a set where we can express each function's body in the backend expression languages, we can expand and eliminate the functions as we build the backend conditions. This gives us a bounded set of conditions that quantify over local state, but no longer contain uninterpreted functions. We take this approach in the next chapter, building specialised frontends that balance expressivity with automation-friendliness.

Chapter 6

A Practical Starling Frontend

While $_{\text{lo}}$Starling adds sound local-state reasoning to $_{\mu}$Starling, it loses the guarantee that a bounded proof outline maps to a bounded set of backend verification conditions. This makes it unsuitable for the form of automation we aim to support.

To move forwards, we can restrict local-state parametrisation to forms that preserve boundedness. To start, let us allow two forms. First, local observations can *guard* views: the view only participates in the assertion when the local guard is true. Second, views can take expressions over local state as parameters. A third form — *iterated views*, a form of counter abstraction — appears at the end. Starling$_{\text{tool}}$ uses iterated views, but there is not yet a full formalisation of them.

The frontends seen so far ($_{\mu}$Starling and $_{\text{lo}}$Starling) are liberal in the views semigroups, backends, and other parameters they accept. While this makes them adaptable, it means that there is no guaranteed uniformity between proofs. The lack of structure in the frontend parameters also limits the frontend-level optimisations and extensions we can make.

This chapter poses a frontend, $_{\text{g}}$Starling, which moves towards a practical automatic proof system by constraining frontend parameters and local-state parametrisation. In specific:

- unlike the previous frontends, $_{\text{g}}$Starling assumes that we can express the shared state model as a total map from variables to values, and that we can parametrise predicate and relation expressions by the pre–and post-states of those variables;
- the new frontend specialises $_{\text{lo}}$Starling's view functions into forms that we can translate into bounded condition sets. View functions now use *Concurrent Abstract Predicates* [46] style *view atoms*: indivisible tokens parametrised by local-state expressions. The above restrictions on local-state parametrisation now apply;
- assertions are now syntactic *view expressions* that map indirectly to atom-based views. We can refine this expression language without changing the underlying view model;
- the $_{\text{g}}$Starling frontend uses a richer form of syntactic definer, based on a general idea of view patterns that match against view expressions.

Some of the changes just refine $_{\text{lo}}$Starling; others, such as pattern-based definers, need deep logic changes. As such, we would need to prove $_{\text{g}}$Starling's soundness afresh; doing so formally is, for now, left to future work.

6.1 Practical views semigroups

Our new frontend will need stronger laws over views algebras than the existing classes give us. These laws form a new class, which this dissertation calls *separating views semigroups*.

The class of separating views semigroups has a large set of requirements, so showing that the class is inhabited is important. We show that the natural numbers form a separating views semigroup. While $\mathbb{N}$ is unlikely to be a practical views algebra on its own, this result lets us derive a more useful algebra: multisets, the free commutative monoid. We use multisets as the core views algebra in the logic underlying $\text{Starling}_{\text{tool}}$.

Separating views semigroups

Though our existing views algebra classes covers a large amount of operations and laws — sufficient enough to build ${}_{\mu}$Starling and ${}_{\text{lo}}$Starling —, the developments we make in this chapter need even more structure from views algebras. We will need:

- the converse of the adjoint law of Definition 3.8;
- cancellativity, but strengthened to $\sqsubseteq$: when we have two ordered views with a common part, we can remove that part and still preserve order;
- distribution of $\setminus$ over $\bullet$: subtracting c from $a \bullet b$ is the same as subtracting c from a, then subtracting any remainder from b.

These properties do not form a cohesive unit, unlike the other classes of algebra, but similar properties appear in various treatments of separation algebras. Cancellativity, in the standard mathematical sense, is an assumed property of Calcagno *et al.*'s separation algebras [41]. Cao *et al.*'s version of separation logic [59] assumes that separating implication (in our system, flipped $\setminus$) is a right adjoint in both directions. (The last property is less relatable to other formulations, but is similar in spirit to Cao *et al.*'s *magic wand as frame* rules.) As such, we refer to views semigroups with the above properties as *separating*.

Definition 6.1. A *separating views semigroup* $(V, \bullet, \setminus, \sqsubseteq, \equiv)$ is a subtractive views semigroup that obeys the following extra laws:

$$
\begin{aligned}
(a \setminus b) \sqsubseteq c \quad \implies \quad & a \sqsubseteq b \bullet c && (\sqsubseteq\text{-residual-backwards}) \\
& a \sqsubseteq (a \bullet b) \setminus b && (\sqsubseteq\text{-cancellativity}) \\
& (a \bullet b) \setminus c \equiv (a \setminus c) \bullet (b \setminus (c \setminus a)) && (\setminus\text{-distribution})
\end{aligned}
$$

(Coq: FullView in Starling.Views.Classes)

While the above cancellativity property looks different from the intuitive description we gave above, the two forms are actually equivalent.

Lemma 6.1. $\forall a, b.\ a \sqsubseteq (a \bullet b) \setminus b \iff \forall a, b, c.\ (a \bullet c \sqsubseteq b \bullet c) \implies (a \sqsubseteq b)$.

This new class adds various properties that were strikingly absent from the subtractive class. By rearranging the backwards adjoint property, we get a maximality result about $\setminus$:

Lemma 6.2. $\forall a, b.\ a \sqsubseteq b \bullet (a \setminus b)$. (Coq: inc_sub_dot in Starling.Views.Classes)

Lemma 6.3. Signatures over separating semigroups are adjoint compatible (Definition 3.9). (Coq: adjoint_compat_FullView in Starling.ProofRules)

Natural numbers as a separating views monoid

Natural numbers form a separating views monoid. While such a monoid is of little use by itself, we can later use it to define a monoid over multisets. Each result below has a corresponding proof in Appendix B.2 and in the Coq mechanisation (Coq: Starling.Views.Instances.Nat).

For natural numbers, we just define equivalence as $=$ (Leibniz equality), which is trivially reflexive, commutative, and associative.

Lemma 6.4. $(\mathbb{N}, =)$ is a setoid.

Addition ($+$) is our chosen join operator for naturals. Addition is associative, commutative, and has compatibility with $=$, as needed.

Lemma 6.5. $(\mathbb{N}, +, =)$ is a views semigroup.

As the unit of $+$, 0 has the right property ($0 + x = x$) to be a views monoid unit.

Lemma 6.6. $(\mathbb{N}, 0, +, =)$ is a views monoid.

We can define $\sqsubseteq$ on naturals as $\leqslant$. This gives us the expected relationship with $=$, and also makes 0 the least element of our monoid.

Lemma 6.7. $(\mathbb{N}, 0, +, \leqslant, =)$ is an ordered views semigroup.

When defining subtraction over naturals, we must be careful: the $-$ operator does not give us natural numbers when its right operand exceeds its left ($5 - 6 = -1$). To be a legal subtractive views semigroup, the subtraction operator we choose must close over $\mathbb{N}$, so we instead use *truncated subtraction* $\dot{-}$ (also known as *monus*)[1].

Truncated subtraction has arithmetically unusual properties. For instance, $(a + b) - c = (a - c) + b$, but $(a + b) \dot{-} c = (a \dot{-} c) + (b \dot{-} (c \dot{-} a))$. This suggests that $\mathbb{N}$ might be a separating views semigroup, and, indeed, we can show that it is.

Lemma 6.8. $(\mathbb{N}, +, \dot{-}, \leqslant, =)$ is an subtractive, separating views semigroup.

Multisets as a separating views monoid

When discussing how to prove *Peterson* (Figure 3.2), we used multisets of program locations as our views algebra. Multisets are the free commutative monoid, and we can show that they also form a subtractive, ordered (decidably so, if finite) separating views monoid. As such, they form a flexible, if minimalist, views algebra.

Let us define a multiset $m \in \text{bag}\ T$ as a total function from each item in a carrier set T to the number of times it occurs. Any item not in the multiset maps to 0. (For completeness, we give a formal definition in Definition A.3).

[1] For completeness, Definition A.4 gives a formal definition.

This representation has a straightforward argument for being a separating views monoid: we can function-lift (Lemma 5.1) the $\mathbb{N}$ algebra[2]. The pointwise lift $\lambda x, y.\, \forall t : T.\, x(t) = y(t)$ relates multisets x and y provided that they contain each item in T the same number of times. Let us denote this form of multiset equivalence using the operator $\equiv_m$.

Lemma 6.9. For all T, $(\text{bag } T, \equiv_m)$ is a setoid.

Lifting $+$ gives us an operator producing a multiset with every item appearing as many times as the sum of its appearances in both parents — the multiset sum $\uplus_m$.

Lemma 6.10. For all T, $(\text{bag } T, \uplus_m, \equiv_m)$ is a views semigroup.

Lifting 0 gives us a *multiset unit* $\emptyset_m$. This multiset has every item in T exactly zero times.

Lemma 6.11. For all T, $(\text{bag } T, \emptyset_m, \uplus_m, \equiv_m)$ is a views monoid.

Lifting $\leqslant$ gives us a *multiset subset* operator $\subseteq_m$. For multisets x and y over T, $x \subseteq_m y$ if, and only if, each element in T appears no more often in x than in y.

Lemma 6.12. For all T, $(\text{bag } T, \uplus_m, \subseteq_m, \equiv_m)$ is an ordered views semigroup.

Lifting $\dot{-}$ gives us an operator we call *multiset minus*, or $\setminus_m$. The multiset $m_1 \setminus_m m_2$ maps every item in T to the $\mathbb{N}$-closed subtraction of its cardinality in m_1 by its cardinality in m_2. Intuitively, we take items away from m_1 until either none remain or the amount we have taken is equal to the amount present in m_2.

Lemma 6.13. For all T, $(\text{bag } T, \uplus_m, \setminus_m, \subseteq_m, \equiv_m)$ is a subtractive, separating semigroup.

Multiset finiteness and ordering decidability

It is not possible to decide $\subseteq_m$ or $\equiv_m$ in general. This is because, for any multiset m, the set $\{ x \in T \mid 0 < m(x) \}$ of elements present in m is bounded only by T, which is itself unbounded. This set is precisely that which a decision procedure must enumerate, and so *must* be bounded. The fact that $m(T)$ may be ∞ also poses issues for arithmetic-based decision procedures.

We need a decidable $\subseteq_m$ to build ${}_g$Starling-based tooling[3]. As a result, the rest of this dissertation makes implicit assumptions that multiset views have a finite set of present elements, and that the multiplicity of each element is also finite. In practice, multiset views track bounded quantities of bounded varieties of assertions, and these assumptions hold.

A refinement of multisets-as-functions that enforces these finiteness assumptions is to model multisets as lists, ignoring element order. (This is how $\text{Starling}_{\text{tool}}$, and parts of the Coq mechanisation, model multisets.) We can then decide inclusion by (say) checking whether the subtraction is empty ($(x \setminus y) = \langle\rangle$), and equivalence by checking in both directions.

[2]The Coq development does not do this, but does show that its instances are compatible with the ones we would have built from doing so.

[3]See also Appendix A.2.

6.2 View expressions

As we saw in § 4.2, the assertion algebra of Starling outlines need not be the view algebra of its justification. This lets us convert from LVF views to CVF views as in $_{\text{lo}}$Starling, but also lets us refine the assertion algebra to be more abstract and syntactic. We can then:

- embed syntactic sugar and complex structures into syntactic views, without affecting the underlying semigroup;
- make changes to the logic's underlying semigroup without changing the syntax of the high-level assertions;
- abstract away more of the underlying Starling logical machinery, providing a more user-friendly interface.

We can explore some of these ideas using *view expressions*: syntactic tree-representations[4] of applications of view operations ($\bullet$, $\backslash$, and ε) over a set Atom of primitive view elements, or *atoms*. Each atom itself maps to a concrete view.

Definition 6.2. The set VExpr(Atom) of *view expressions*, where Atom is an atom set, is the set of all productions of the grammar below:

$\langle vexpr \rangle$ ::=	'1'	unit
	'(@' $\langle atom \rangle$ ')'	atom
	'(•' $\langle vexpr \rangle$ $\langle vexpr \rangle$ ')'	join
	'(\' $\langle vexpr \rangle$ $\langle vexpr \rangle$ ')'	part

We can adapt view expressions to non-subtractive views monoids that provided that we delete the *part* production. We can support non-monoidal semigroups if we remove *unit*. For simplicity we only consider the subtractive monoidal case in this section; we give limited support for other cases in the Coq development.

Atom sets, languages, and interpretation

The choice of atom set defines the interface between consumers of a view-expression proof outline and the underlying reasoning system. Atom sets can be high-level, representing abstract ideas like 'the lock is locked', or low-level, representing concrete facts such as 'thread A is at program counter 1'. Atom sets need not map to indivisible information fragments; the sets this dissertation covers do. This flexibility comes from atom sets' loose coupling to views (which, in turn, loosely couple to state observations).

Example. The line numbers in our *Peterson* assertions (Figure 3.2) form an atom set Atom $=$ $\{$ A1, A2, A3, A4, B1, B2, B3, B4 $\}$. Using this set directly, valid view expressions include:

$$1, \qquad (\bullet\ 1\ 1), \qquad (\bullet\ (\bullet\ (@\text{A1})\ (@\text{B4}))\ (\backslash\ (@A3)\ 1))$$

[4] Though view expressions are S-expressions here, this is not required; Chapter 7 uses a more C-like notation.

Abstract predicates. For consistency across proofs, the atom sets that this and further chapters use derive from Dinsdale-Young *et al.*'s *concurrent abstract predicates* [46]. In these sets, each atom is, or contains, a pair $(\mathtt{t}, \mathfrak{a})$: $\mathtt{t}$ is a *tag* representing an abstract shared-state observation[5]; $\mathfrak{a}$ is an *argument list* of local-state expressions parametrising the observation.

Definition 6.3. An *abstract predicate*, in Starling, is a member of the set:

$$\mathsf{APred}(\mathsf{Tag})(\mathsf{E}_{\mathsf{Val}}) \stackrel{\text{def}}{=} (\mathsf{Tag} \times \text{list } \mathsf{E}_{\mathsf{Val}})$$

where Tag is some tag set. (Coq: LooseAPred in Starling.Frontend.APred.Core)

Abstract predicates appear below as S-expressions of the form $(t\ \ \mathfrak{a}0\ \ \mathfrak{a}1\ \ \ldots)$. For example, the various *Peterson* atoms, having no parameters, become $(A1)$, $(A2)$, and so on.

Interpreting atoms. We must be able to interpret atom sets in terms of a low-level views semigroup for them to have practical use. As atom sets can have multiple interpretations and target multiple semigroups, we can port proof outlines over high-level atom sets across reasoning systems. We call a pair of atom set and one such interpretation an *atom language*.

Definition 6.4. An *atom language* is a triple $(\mathsf{Atom}, V, \mathtt{r})$ of atom set, underlying view set, and interpretation function (of type $\mathsf{Atom} \to V$).
(Coq: AtomLanguage in Starling.Views.Expr.AtomLanguage)

A possible atom language for *Peterson* is $(\mathsf{APred}(\mathsf{Atom})(\emptyset), \mathsf{bag}\ \mathsf{Atom}, \lambda(\mathbf{x}).\ \lbag x \rbag)$.

Interpreting view expressions. To lift $\mathtt{r}$ over view expressions, we define a new function.

Definition 6.5. The *view expression interpretation* veinterp, parametrised over an atom language $(\mathsf{Atom}, V, \mathtt{r})$, is:

$$\begin{aligned}
\mathsf{veinterp} : (\mathsf{Atom} \to V) &\to \mathsf{VExpr}(\mathsf{Atom}) \to V \\
\mathsf{veinterp}(\mathtt{r})(\mathit{1}) &= \varepsilon \\
\mathsf{veinterp}(\mathtt{r})((@\mathfrak{a})) &= \mathtt{r}(\mathfrak{a}) \\
\mathsf{veinterp}(\mathtt{r})((\bullet\ \ \mathfrak{a}\ \ \mathtt{b})) &= \mathsf{veinterp}(\mathtt{r})(\mathfrak{a}) \bullet \mathsf{veinterp}(\mathtt{r})(\mathtt{b}) \\
\mathsf{veinterp}(\mathtt{r})((\backslash\ \ \mathfrak{a}\ \ \mathtt{b})) &= \mathsf{veinterp}(\mathtt{r})(\mathfrak{a}) \setminus \mathsf{veinterp}(\mathtt{r})(\mathtt{b})
\end{aligned}$$

(Coq: interpret_sm in Starling.Views.Expr.Type)

View expressions as views semigroups

Views expressions form views semigroups by projection into the underlying semigroup. The projection resembles a flipped version of the functional lift of Lemma 5.1:

[5] Unlike normal abstract predicates, tags in Starling abstract predicates need not be strings; later on, we use composite types, such as lists, as tags.

Lemma 6.14. If V forms a views algebra, we can derive a views algebra on $X \to V$, for any X, by projection through $\mathsf{veinterp}(r)$:

$$x \bullet y \stackrel{\text{def}}{=} (\bullet\ x\ y)$$
$$x \setminus y \stackrel{\text{def}}{=} (\setminus\ x\ y)$$
$$\varepsilon \stackrel{\text{def}}{=} \mathit{1}$$
$$x \equiv y \stackrel{\text{def}}{=} \mathsf{veinterp}(r)(x) \equiv \mathsf{veinterp}(r)(y)$$
$$x \sqsubseteq y \stackrel{\text{def}}{=} \mathsf{veinterp}(r)(x) \sqsubseteq \mathsf{veinterp}(r)(y)$$

(Coq: Starling.Views.Transformers.Function)

We can derive the algebra laws by unfolding the view interpretations then applying the same laws on the underlying algebra. Formal proofs exist in Coq and Appendix B.2.

Replacing atoms in view expressions

The main advantage of view expressions over concrete views is that they give us a regular structure over which we can perform atom transformations. This helps us to add syntactic abstractions on top of the underlying views algebra. The free-form structure of view expressions means that we can substitute both atoms and arbitrary view expression trees for atoms: schemes for both appear below.

Definition 6.6. The *view bind function* vbind maps a function f over every atom in a view expression. The function f must return view expressions over some atom set; this need not be the original atom set.

$$\mathsf{vbind} : (\mathsf{Atom} \to \mathsf{VExpr}(\mathsf{Atom}')) \to \mathsf{VExpr}(\mathsf{Atom}) \to \mathsf{VExpr}(\mathsf{Atom}')$$
$$\mathsf{vbind}(f)(\mathit{1}) \stackrel{\text{def}}{=} \mathit{1}$$
$$\mathsf{vbind}(f)((@a)) \stackrel{\text{def}}{=} f(a)$$
$$\mathsf{vbind}(f)((\bullet\ x\ y)) \stackrel{\text{def}}{=} (\bullet\ \mathsf{vbind}(f)(x)\ \mathsf{vbind}(f)(f))$$
$$\mathsf{vbind}(f)((\setminus\ x\ y)) \stackrel{\text{def}}{=} (\setminus\ \mathsf{vbind}(f)(x)\ \mathsf{vbind}(f)(f))$$

(Coq: vbind in Starling.Views.Expr.Instances)

The *view map* vmap behaves as vbind, but takes a function that returns single atoms:

$$\mathsf{vmap} : (\mathsf{Atom} \to \mathsf{Atom}') \to \mathsf{VExpr}(\mathsf{Atom}) \to \mathsf{VExpr}(\mathsf{Atom}')$$
$$\mathsf{vmap}(f) \stackrel{\text{def}}{=} \mathsf{vbind}((\lambda a.\ (@a)) \circ f)$$

(Coq: vmap in Starling.Views.Expr.Instances)

The function vbind makes view expressions an example of the functional programming abstraction known as a *monad*; similarly, vmap makes view expressions a *functor* [60].

Lemma 6.15. (V, vmap) is a functor: that is, vmap obeys the functor laws [60][4.2]:

$$\mathsf{vmap}(\mathrm{id}) = \mathrm{id} \qquad \mathsf{vmap}(f \circ g) = \mathsf{vmap}(f) \circ \mathsf{vmap}(g)$$

Lemma 6.16. $(V, \lambda x.\ (@x), \mathsf{vbind})$ is a monad with (V, vmap) as the underlying functor: that is, vbind obeys the monad laws [60][4.5]:

$$\begin{aligned}
\mathsf{vbind}(\quad & k)(\quad (@a)) &&= k(a) \\
\mathsf{vbind}(\lambda x.\ (@x))(\quad & m) &&= m \\
\mathsf{vbind}(\quad & k)(\mathsf{vbind}(h)(m)) &&= \mathsf{vbind}(\lambda x.\ \mathsf{vbind}(k)(h(x)))(m) \\
\mathsf{vmap}(\quad & f)(\quad m) &&= \mathsf{vbind}((\lambda x.\ (@x)) \circ f)(m)
\end{aligned}$$

Let us adopt the Haskell notation for functor mapping and monadic binding: let $\mathsf{fmap}(f)(x)$ refer to any mapping operation that obeys the functor laws (for example, $\mathsf{vbind}(f)(x)$), and $x \gg\!= f$ refer to any binding operation that obeys the monad laws (for example, $\mathsf{vbind}(f)(x)$). Similarly, $\mathsf{return}(x)$ can stand for any valid monadic return (in the above case, $\lambda x.\ (@x)$).

Equivalence-preserving transformations. Using view expressions as a functor or monad preserves equivalence provided that the given function does. As we define equivalence in terms of expressions' interpretations, this extends to functions that change the atom language but preserve the underlying views set.

Lemma 6.17. Given the atom languages (Atom, V, f) and (Atom', V, g), and a function $h : \mathsf{Atom} \to \mathsf{VExpr}(\mathsf{Atom}')$, binding preserves equivalence over V provided that h does:

$$\forall a : \mathsf{Atom}.\ f(a) \equiv \mathsf{veinterp}(g)(h(a)) \quad \iff \quad \forall m : \mathsf{VExpr}(\mathsf{Atom}).\ m \equiv m \gg\!= h$$

Corollary 6.17.1. Given the atom languages (Atom, V, f) and (Atom', V, g), and a function $h : \mathsf{Atom} \to \mathsf{Atom}'$, mapping preserves equivalence over V provided that h does:

$$\forall a : \mathsf{Atom}.\ f(a) \equiv g(h(a)) \quad \iff \quad \forall m : \mathsf{VExpr}(\mathsf{Atom}).\ m \equiv \mathsf{fmap}(h)(m)$$

These results let us perform 'syntactic sugar' transformations where we reduce a complex atom set to a simpler atom set while preserving the underlying views.

Normalising view expressions

View expressions, as symbolic trees of operations on atoms, do not map bijectively to equivalence classes of underlying views. In fact, we can represent the same view as infinitely many expressions of varying complexity. For example, `(\ a (• b c))` and `(\ (\ a b) c)` represent equivalent views in a subtractive, separating semigroup. This complicates matching view expressions against definitions later on.

Ideally, we would reduce view expressions to a normal form where each view equivalence class has a unique expression. The associative, commutative nature of views makes this hard: for example, should the conjunction of `a` and `b` be `(• a b)` or `(• b a)`? We would need to make strong assumptions about the atom and view sets, such as the existence of total orderings between atoms, to reach a single normal form.

We can make view expressions more regular, if not fully normalised, by arranging them into the form `(• (@a) (• ... (• (@z) 1)))`. This form resembles an S-expression *cons list*; as a result, we call such expressions *list-normalised*. We can manipulate such expressions using recursive procedures similar to those used on cons lists in Lisp-style languages.

Definition 6.7. A view expression is *list-normalised* provided that it is $\mathit{1}$, or is $(\bullet\ (@\mathfrak{a})\ x)$ where x is list-normalised. We denote list-normalised expressions with the suffix 'l' in type signatures. Let $(\odot\ \mathfrak{a}\ \mathfrak{b}\ \ldots\ z)$ be notational shorthand for $(\bullet\ \mathfrak{a}\ (\bullet\ \mathfrak{b}\ \ldots(\bullet\ z\ \mathit{1})))$.

List normalisation algorithm. To reduce view expressions to list-normalised form, we must eliminate all part operations. We do so by symbolically computing the subtraction in terms of the atom language. This means we cannot provide a fully generic list-normalising algorithm; we instead build an example algorithm and revisit it as we change our atom language.

Our example list-normalisation algorithm, lnorm, works over atom languages where subtraction occurs atom-by-atom with each subtraction being 'all-or-nothing'. The set of program locations we used for Peterson's algorithm is one such language; the languages we use in ${}_{\mathrm{g}}$Starling and Starling are not. We will therefore need to refine lnorm as we proceed.

To allow this refinement, we define lnorm in four stages. Each stage makes more assumptions about the atom language; we need only swap out stages with assumptions that no longer hold. The stages have the following names and type signatures:

$$
\begin{array}{rll}
\mathsf{lnorm} : \mathsf{VExpr}(\mathsf{Atom}) & & \to\ \mathsf{VExpr}(\mathsf{Atom})_{\mathrm{l}} \\
\mathsf{lnorm}_{\mathrm{B}} : \mathsf{VExpr}(\mathsf{Atom})_{\mathrm{l}} & \to \mathsf{VExpr}(\mathsf{Atom})_{\mathrm{l}} & \to\ \mathsf{VExpr}(\mathsf{Atom})_{\mathrm{l}} \\
\mathsf{lnorm}_{\mathrm{A}} : \mathsf{VExpr}(\mathsf{Atom})_{\mathrm{l}} & \to \mathsf{Atom} & \to\ \mathsf{VExpr}(\mathsf{Atom})_{\mathrm{l}} \\
\mathsf{lnorm}_{\mathrm{P}} : \mathsf{Atom} & \to \mathsf{Atom} & \to (\mathsf{Atom} \cup \{\bot\}) \times (\mathsf{Atom} \cup \{\bot\}))
\end{array}
$$

Subtracting atoms from atoms. $\mathsf{lnorm}_{\mathrm{P}}(m)(s)$ computes subtraction of an atom s from an atom m. It returns a pair (m', s') of optional atoms: m' represents any remainder after the subtraction; s' represents the atom to subtract from any forthcoming atom. Here, we assume an atom language where two atoms cancel-out provided that they are equal.

$$
\mathsf{lnorm}_{\mathrm{P}}(m)(s) \stackrel{\text{def}}{=} \begin{cases} (\bot, \bot) & m = s \\ (m, s) & m \neq s \end{cases}
$$

This definition may come as an anti-climax given $\mathsf{lnorm}_{\mathrm{P}}$'s signature. This is deliberate: it lets us swap $\mathsf{lnorm}_{\mathrm{P}}$ with a more complex definition if the atom language needs one.

Subtracting atoms from expressions. We can lift $\mathsf{lnorm}_{\mathrm{P}}$ (or any function with the same signature and properties) to subtract atoms from list-normalised expressions.

$$
\begin{array}{rrcl}
 & \mathsf{lnorm}_{\mathrm{A}}(\mathit{1})(s) & \stackrel{\text{def}}{=} & 1 \\
 & \mathsf{lnorm}_{\mathrm{A}}((\bullet\ (@\mathfrak{a})\ x))(s) & \stackrel{\text{def}}{=} & \mathsf{recur}(\mathsf{lnorm}_{\mathrm{P}}(\mathfrak{a}, s))(x) \\
\text{where} & \mathsf{recur}((\bot, s'))(x) & \stackrel{\text{def}}{=} & \mathsf{lnorm}_{\mathrm{A}}(x)(s') \\
 & \mathsf{recur}((\mathfrak{a}', s'))(x) & \stackrel{\text{def}}{=} & (\bullet\ (@\mathfrak{a}')\ \mathsf{lnorm}_{\mathrm{A}}(x)(s'))
\end{array}
$$

The function $\mathsf{lnorm}_{\mathrm{A}}$ returns list-normalised expressions by construction.

Subtracting expressions from expressions. The function lnorm_B lifts lnorm_A to accept subtrahends that are full expressions, using a rule that holds for all separating views semigroups:

Lemma 6.18. $a \setminus (b \bullet c) \equiv (a \setminus b) \setminus c$. (Coq: sub_by_dot in Starling.Views.Classes)

We can define lnorm_B as a recursion on the subtrahend.

$$\mathsf{lnorm}_B(m)(\mathbf{1}) \stackrel{\text{def}}{=} m$$
$$\mathsf{lnorm}_B(m)((\bullet\ (@a)\ s')) \stackrel{\text{def}}{=} \mathsf{lnorm}_B(\mathsf{lnorm}_A(m)(a))(s')$$

The finished algorithm. We now define lnorm. We start with a view expression v and no assumptions about its structure; our first goal is to reduce v to either a list-normalised expression or a parting of two list-normalised expressions.

As we can use lnorm recursively to reduce sub-expressions, most of its complexity arises in the handling of joins and parts. Applying $\bullet$ directly to two list-normalised expressions does not give a list-normalised result in general. Instead, we build an auxiliary function to append one list-normalised expression to another.

Definition 6.8. The *list-normalised append* function lapp is:

$$\mathsf{lapp} : \mathsf{VExpr}(\mathsf{Atom})_l \to \mathsf{VExpr}(\mathsf{Atom})_l \to \mathsf{VExpr}(\mathsf{Atom})_l$$
$$\mathsf{lapp}(\mathbf{1})(y) \stackrel{\text{def}}{=} y \qquad \mathsf{lapp}((\bullet\ (@a)\ x))(y) \stackrel{\text{def}}{=} (\bullet\ (@a)\ \mathsf{lapp}(x)(y))$$

(Coq: lapp in Starling.Views.Expr.List)

The last stage arranges the most primitive expressions into list-normal form, then replaces joins with lapp and parts with lnorm_B:

$$\mathsf{lnorm}(\mathbf{1}) \stackrel{\text{def}}{=} \mathbf{1}$$
$$\mathsf{lnorm}((@a)) \stackrel{\text{def}}{=} (\odot\ (@a))$$
$$\mathsf{lnorm}((\bullet\ x\ y)) \stackrel{\text{def}}{=} \mathsf{lapp}(\mathsf{lnorm}(x))(\mathsf{lnorm}(y))$$
$$\mathsf{lnorm}((\setminus\ x\ y)) \stackrel{\text{def}}{=} \mathsf{lnorm}_B(\mathsf{lnorm}(x))(\mathsf{lnorm}(y))$$

As the two base cases produce list-normalised, and the two inductive cases preserve list normalisation, the result of lnorm is list-normalised.

Summary

This section introduced view expressions as an abstract notation for views. View expressions let us manipulate views in regular ways. We can, for example, apply a reduction across the atoms in an expression, and use the equivalence lemmas to reason about the semantics preservation of such operations. View expressions abstract over the underlying view algebra in a way that preserves said algebra's laws and operations.

One issue with views expressions is that their free-form nature complicates matching against patterns. To remedy this, we gave an example algorithm for partially normalising

views expressions. The result — list-normalised form — is not strictly a normal form, but is easier to work with. We modify the algorithm as we introduce increasingly elaborate atom schemes in later chapters, but the general structure persists.

The next section discusses an abstract notation for predicate expressions. This gives us the same advantages as view expressions: regularity across proofs and backends; functor and monad operations; and a straightforward approach to substitution and syntactic sugar.

6.3 Structured propositions

While § 4.1 dismissed the idea of a single backend interface across all solvers, a level of uniformity is useful in practice. If we abstract over the exact syntax of each solver's expressions, we can make our proofs portable between solvers (to a degree). In addition, most solvers support a common set of theories: for example, variable assignment, linear integer arithmetic, and propositional logic. By assuming these features (and, so, solvers that supports them), we can make the logic and tooling easier to use, and more efficient, when applying them.

This section introduces *structured propositions*: an abstract syntax for proposition expressions. Structured propositions represent propositions over a set of abstract *values* in much the same way that view expressions range over abstract atoms. While these values can be primitive, later sections refine them to be expressions over shared and local variables.

The structured proposition language aims to capture the common operations of the solvers we can use with $\text{Starling}_{\text{tool}}$. It includes the operations of the proposition expression class; to support the advanced frontends that later sections build, it also depends on the operations of several new, expanded expression classes, which we encounter below.

To support the variety of solver operations that are not captured in an expression class, structured propositions have support for custom operators over both propositions and values. We can assume certain distributivity laws over those operations, but nothing more. Sometimes, proofs may depend on features of a backend that are unavailable through custom operators; to support this, structured propositions allow *symbols*: arbitrary pieces of value-parametrised syntax that the backend can expand into concrete propositions.

This section does not construct an equivalent structured abstraction over relation expressions. Instead, we can model relations as two-state structured propositions.

Extending proposition expressions

This section extends the proposition expression class in three stages. First, it extends proposition expressions to support falsehood, implication, and negation; these let us move guards from atoms into the backend conditions. The next two stages further extend proposition expressions to support equality reasoning over some abstract *value set*; we can then fill that set parameter with a new class of *value expressions*, parametrised over variables.

Implying proposition expressions. The proposition expression class in Definition 4.4 is intentionally minimal, requiring only truth and conjunction. Let us extend the class to support some features that $_{\text{g}}$Starling needs.

Definition 6.9. An *implying proposition expression* language is a proposition expression language with expression $\text{false}_{\mathsf{Pr}} : \mathsf{E}_{\mathsf{Pr}}$ and operation $\Rightarrow_{\mathsf{Pr}} : \mathsf{E}_{\mathsf{Pr}} \to \mathsf{E}_{\mathsf{Pr}} \to \mathsf{E}_{\mathsf{Pr}}$ where:

$$[\![\text{false}_{\mathsf{Pr}}]\!]_{\mathsf{Pr}}(x_g)(x_l) = \emptyset \qquad \text{(false-empty)}$$

$$[\![(e_1 \Rightarrow_{\mathsf{Pr}} e_2) \wedge_{\mathsf{Pr}} e_1]\!]_{\mathsf{Pr}}(x_g)(x_l) \subseteq [\![e_2]\!]_{\mathsf{Pr}}(x_g)(x_l) \qquad \text{(modus ponens)}$$

$$\forall s.\, ([\![e_1]\!]_{\mathsf{Pr}}(x_g)(x_l)(s) \implies [\![e_2]\!]_{\mathsf{Pr}}(x_g)(x_l)(s)) \implies [\![e_1 \Rightarrow_{\mathsf{Pr}} e_2]\!]_{\mathsf{Pr}}(x_g)(x_l)(s) \qquad \text{(conditional proof)}$$

We can derive negation from these operations in a similar way to systems like Coq:

Definition 6.10. On implying proposition expression languages, the *negation operation* $\neg_{\mathsf{Pr}}$ is a derived operation with the following definition:

$$\neg_{\mathsf{Pr}} : \mathsf{E}_{\mathsf{Pr}} \to \mathsf{E}_{\mathsf{Pr}} \qquad \neg_{\mathsf{Pr}}(x) \stackrel{\text{def}}{=} x \Rightarrow_{\mathsf{Pr}} \text{false}_{\mathsf{Pr}}$$

Alphabetised proposition expressions. We can extend proposition expressions to carry variables from some alphabet. The requirements on such an alphabet are loose: we need only assume that we can compare two variables for equality (forming a proposition expression), and that we can traverse a proposition expression to rewrite all variables inside equality comparisons. Let us first focus on shared state only; a later section considers local variables.

While the previous classes of proposition expression targeted any valid state set, *alphabetised proposition expressions* assume that the state set contains partial[6] functions from some abstract alphabet T to some concrete value set Val. Alphabetised proposition expression languages assume a particular Val, but *not* a particular T; they form a second-order type where each instantiation for a given T forms an implying proposition expression language.

Definition 6.11. An *alphabetised proposition expression* language $\mathsf{E}_{\mathsf{VcPr}}(\mathsf{Val})$, for a value set Val, is a family $\forall \mathsf{T}.\, \mathsf{E}_{\mathsf{Pr}}(\mathsf{T})$ of implying proposition expression languages such that:

- the state model used for $[\![-]\!]_{\mathsf{Pr}}$ is the set of partial functions $\mathsf{T} \rightharpoonup \mathsf{Val}$;
- there exists an operation $=_{\mathsf{Pr}} : \mathsf{T} \to \mathsf{T} \to \mathsf{E}_{\mathsf{Pr}}(\mathsf{T})$, for all T, that represents an equality comparison between values in some *value set* T;
- E_{Pr} is a legal functor.

The following laws must also hold:

$$\sigma \in [\![x =_{\mathsf{Pr}} y]\!]_{\mathsf{Pr}}(x_g)(x_l) \iff x \in \text{dom}\ \sigma \wedge y \in \text{dom}\ \sigma \wedge \sigma(x) \equiv \sigma(y)$$

$$\sigma \in [\![\mathsf{fmap}(f)(x)]\!]_{\mathsf{Pr}}(x_g)(x_l) \iff (\sigma \circ f) \in [\![x]\!]_{\mathsf{Pr}}(x_g)(x_l)$$

[6] Partiality lets us model the possibility of ill-formed members of T; for instance, T could be a set of pointer expressions that may be ill-typed, reference illegal locations, and so on.

Finally, the functor instance must satisfy the following distributivity equivalences:

$$\begin{aligned}
\mathsf{fmap}(f)(\mathrm{true}_{\mathsf{Pr}}) &\equiv \mathrm{true}_{\mathsf{Pr}} \\
\mathsf{fmap}(f)(\mathrm{false}_{\mathsf{Pr}}) &\equiv \mathrm{false}_{\mathsf{Pr}} \\
\mathsf{fmap}(f)(x \wedge_{\mathsf{Pr}} y) &\equiv \mathsf{fmap}(f)(x) \wedge_{\mathsf{Pr}} \mathsf{fmap}(f)(y) \\
\mathsf{fmap}(f)(x \Rightarrow_{\mathsf{Pr}} y) &\equiv \mathsf{fmap}(f)(x) \Rightarrow_{\mathsf{Pr}} \mathsf{fmap}(f)(y) \\
\mathsf{fmap}(f)(x =_{\mathsf{Pr}} y) &\equiv f(x) =_{\mathsf{Pr}} f(y)
\end{aligned}$$

(Coq: EqPredEx in Starling.Backend.Alpha.Classes)

Values in Val need not be expressible in the backend theory. This means that we can, for example, model a heap as a 'heap variable', despite heaps being intractable to express.

Value expressions

While $\mathsf{E}_{\mathsf{VcPr}}$ lets us reference values (or variables) in propositions, it gives us no obvious way to embed constants or complex expressions. We can capture these with another class of expressions: *value expressions*. Like alphabetised proposition expressions, value expressions have an interpretation supplied by the backend solver; this time, the interpretation accepts *total* variable-to-value functions, and returns values from that function's codomain.

Value expressions form a second-order type over alphabets, as do alphabetised proposition expressions. This time, alphabets are sets of valid variable names in the backend theory.

Definition 6.12. A *variable set* Var, ranged over by x, contains all possible denotations for variables in the shared-state model.

The variable set is likely to be infinite; it contains every single possible variable identifier the backend can understand. The alphabet we choose as the domain for the value expressions (and, therefore, the domain of the state model) will be a finite subset[7] of Val. Let Σ range over any such alphabets formed in this way.

Definition 6.13. A *value expression* language $\forall \mathsf{Var}.\ \mathsf{E}_{\mathsf{Val}}(\mathsf{Val})(\mathsf{Var})$ is a second-order type where, for all Var:

- we have an interpretation $[\![-]\!]_{\mathsf{Val}}$ with type $\mathsf{E}_{\mathsf{Val}}(\mathsf{Var})x_{\mathsf{g}} \to x_{\mathsf{l}} \to (\mathsf{Var} \to \mathsf{Val}) \twoheadrightarrow \mathsf{Val}$;
- we have a *bottom* expression $\bot$, symbolising inconsistency;
- $\mathsf{E}_{\mathsf{Val}}$ is a legal functor and monad over Var, with fmap corresponding to variable rewriting and $\gg\!=$ corresponding to variable substitution; the monadic return, which we call var, corresponds to a variable reference expression.

[7] The Coq development represents alphabets using dependent subtypes of Var, in which lists model finite subsets. Whenever set operations over alphabets appear, the Coq equivalent uses similar *list* operations.

The following laws must also hold:

$$\begin{aligned} [\![\mathsf{fmap}(f)(x)]\!]_{\mathsf{Val}}(x_g)(x_l)(\sigma) &\equiv [\![x]\!]_{\mathsf{Val}}(x_g)(x_l)(\sigma \circ f) \\ \sigma \equiv \sigma' &\implies [\![x]\!]_{\mathsf{Val}}(x_g)(x_l)(\sigma) \\ [\![\mathsf{var}]\!]_{\mathsf{Val}}(x)(x_g)(x_l)(\sigma) &\equiv \sigma(x) \end{aligned}$$

(Coq: ValEx in Starling.Backend.Alpha.Classes)

Inside alphabetised expressions. Consider inserting value expressions into alphabetised proposition expressions. Doing so directly, by instantiating $\mathsf{E}_{\mathsf{VcPr}}(\mathsf{E}_{\mathsf{Val}}(\Sigma))$, works, but gives us a state model over $\mathsf{E}_{\mathsf{Val}}(\Sigma) \nrightarrow \mathsf{Val}$. As $[\![-]\!]_{\mathsf{Val}}$ has type $\mathsf{E}_{\mathsf{Val}}(\mathsf{Var})x_g \to x_l \to (\mathsf{Var} \to \mathsf{Val}) \nrightarrow \mathsf{Val}$, we can regain the $\mathsf{Var} \to \mathsf{Val}$ state model, but must build a new proposition language to do so.

Definition 6.14. The *chained proposition language* over an alphabetised proposition language $(\forall \mathsf{T}.\, \mathsf{E}_{\mathsf{VcPr}}(\mathsf{Val})(\mathsf{T}))$ and value expression language $(\forall \Sigma.\, \mathsf{E}_{\mathsf{Val}}(\mathsf{Val})(\Sigma))$ is the language formed by the set $(\forall \Sigma.\, \mathsf{E}_{\mathsf{VcPr}}(\mathsf{Val})(\mathsf{E}_{\mathsf{Val}}(\mathsf{Val})(\Sigma)))$ and the interpretation:

$$[\![x]\!]^{\text{chained}}_{\mathsf{Pr}}(x_g)(x_l)(\sigma) \stackrel{\text{def}}{=} [\![x]\!]_{\mathsf{Pr}}(x_g)(x_l)(\lambda e_v.\, [\![e_v]\!]_{\mathsf{Val}}(x_g)(x_l)(\sigma))$$

Local variables

To use our new expression classes in local-state logics such as ${}_g$Starling, we must handle local variables correctly. Doing so needs a few changes from the above set-up.

Expressible values. A soundness argument for ${}_g$Starling will rely on being able to erase local variables in abstract predicates by substituting their final values. As these final values will eventually form part of shared-state propositions over value expressions, they must be expressible in the value expression language. Instead of building a new type (or subtype of Val) for these values, we can model them as value expressions with no variables[8].

Definition 6.15. The *expressible value set* of a language $\mathsf{E}_{\mathsf{Val}}(\mathsf{Val})$ is the set $\mathsf{E}_{\mathsf{Val}}(\mathsf{Val})(\emptyset)$.

We can lift expressible values into expressions over any variable set Var: formally, we can do this with $\mathsf{fmap}(\emptyset)$ (using $\emptyset$ as an uninhabited function $\emptyset \to \mathsf{Var}$). We can also use $[\![-]\!]_{\mathsf{Val}}$ to lift expressible values into ordinary values; we can use this to lift local states $\Sigma_{lo} \to \mathsf{E}_{\mathsf{Val}}(\mathsf{Val})(\emptyset)$ into the form $\Sigma_{lo} \to \mathsf{Val}$.

Combining states. The variable-function approach to modelling states gives us a straightforward way of modelling the combination and separation of states, provided that the alphabets are disjoint. Removing state is straightforward: we can use a function over Σ_1 in any situation where we need one over $\Sigma_1 \setminus \Sigma_2$ by dropping all mappings of variables in Σ_2.

Combining state needs more work, but is also tractable. Suppose we have a state function s_1, over a variable set Σ_1, and need to frame onto another function s_2 over Σ_2. Provided that

[8] Technically, this is an over-approximation; it permits complex expressions such as $12 + (42/3.14)$.

s_1 and s_2 produce equivalent results where their domains overlap[9], the *state join* $\mathsf{SJoin}(s_1, s_2)$ models the resulting composed state.

Definition 6.16. The *state join* $\mathsf{SJoin} : (\Sigma_1 \to \mathsf{Val}) \to (\Sigma_2 \to \mathsf{Val}) \to (\Sigma_1 \cup \Sigma_2 \to \mathsf{Val})$ is:

$$\mathsf{SJoin}(\sigma_1)(\sigma_2)(x) \stackrel{\text{def}}{=} \begin{cases} \sigma_1(x) & x \in \Sigma_1 \\ \sigma_2(x) & x \in \Sigma_2 \end{cases}$$

To join a shared state $\sigma : \Sigma_s \to \mathsf{Val}$ and a local state $l : \Sigma_{lo} \to \mathsf{E}_{\mathsf{Val}}(\mathsf{Val})(\emptyset)$, we can use $\mathsf{SJoin}(\sigma)([\![-]\!]_{\mathsf{Val}} \circ l)$. We can then pass the combined state to interpretation functions.

Marked proposition expressions as relation expressions

We now have a class of proposition expressions that let us traverse variables (in expression equalities, at least) and substitute new variables, or arbitrary expressions, for each.

It would be useful to have similar functionality for relation expressions. To do so, we could build a class of value-parametrised relational expressions with broadly the same operations and requirements. Instead, to avoid duplication between the two types of expressions, we repurpose proposition expressions as relation expressions. While this set-up is not suitable for all backends, backends that receive input as single propositions to begin with (such as Z3) work well with it. This unification of proposition and relation expression languages takes inspiration from the *Unifying Theories of Programming* [61, Def. 2.0.1].

Variable marking. To use one-state proposition expressions as two-state relation expressions, we can treat a pair of pre–and post-states as one single state. As our states are maps from a variable set Var, the result is effectively a map from variable and position in time to value. We can use a convention similar to the UTP [61, §1.1], marking each variable with its time position (pre-state or post-state) and using primes to denote the post-state.

Definition 6.17. Given a variable set Var, the *marked variable set* $\mathsf{Var}^{\mathsf{M}}$ is the set of pairs $(\{\mathsf{Pre}, \mathsf{Post}\} \times \mathsf{Var})$. Given an alphabet $\Sigma \subseteq \mathsf{Var}$, the *marked alphabet* Σ^{M} is the corresponding subset of $\mathsf{Var}^{\mathsf{M}}$.

Where it is unambiguous to do so, let any primed identifier $v' : \mathsf{Var}^{\mathsf{M}}$ stand for (Post, v), and any unprimed identifier $v : \mathsf{Var}^{\mathsf{M}}$ stand for (Pre, v).

Assuming that the variable alphabet does not change mid-program, we can build the combined state function as follows:

$$\sigma^{\mathsf{M}} = \{((\mathsf{Pre}, v), \sigma(v)) \mid v \in \operatorname{dom} \sigma\} \cup \{((\mathsf{Post}, v), \sigma'(v)) \mid v \in \operatorname{dom} \sigma'\}$$

Relational identity of proposition expressions. Using proposition expressions as relation expressions requires us to implement $\mathsf{id}_{\mathsf{Rl}}$ and $\emptyset_{\mathsf{Rl}}$. The empty relation is straightforward to encode as a proposition in implying proposition languages: it is just $\mathsf{false}_{\mathsf{Pr}}$. (Completing the lattice, $\mathsf{true}_{\mathsf{Pr}}$ relates any pre-state to any post-state and so encodes *havoc*.)

[9] The easiest way to enforce this is by requiring $\Sigma_1 \cap \Sigma_2 = \emptyset$.

For $\mathrm{id}_{\mathsf{Rl}}$, we need a proposition that is true provided that the post-state σ' equals the pre-state σ. We cannot assert $\sigma = \sigma'$ directly in the proposition language, but *can* emit equality checks for each variable in dom σ. If we assume the domain does not change, these equalities cover the whole state. This gives us a recursive definition for $\mathrm{id}_{\mathsf{Rl}}$:

$$\mathrm{id}^{\emptyset}_{\mathsf{Rl}} \stackrel{\text{def}}{=} \mathrm{true}_{\mathsf{Pr}} \qquad \mathrm{id}^{\{\,a\,\}\uplus\Sigma^{\mathsf{M}}}_{\mathsf{Rl}} \stackrel{\text{def}}{=} \mathrm{id}^{\Sigma^{\mathsf{M}}}_{\mathsf{Rl}} \wedge_{\mathsf{Pr}} (a =_{\mathsf{Pr}} a')$$

(Coq: gen_frame in Starling.Backend.Alpha.PredAsRel)

Framing. To define an atomic or local action using proposition-as-relation expressions, we must make sure that it preserves all of the local and shared state it does not affect. Doing so manually scales poorly: if we add new variables, we must change all of the action definitions to add preservation. Instead, we want a function that *frames* proposition-as-relation expressions over Σ_1^{M}-expressions to preserve all variables in Σ_2^{M}.

A framing function is valid if it converts a proposition-as-relation expression over Σ_1^{M} to one over a larger alphabet $\Sigma_1^{\mathsf{M}} \uplus \Sigma_2^{\mathsf{M}}$ such that any verification condition holding over the original expression still holds after conversion. At the same time, any valid precondition p over Σ_2 must also be a valid postcondition. In other words:

$$c \Vdash_{\mathsf{EVFH}} \{w\}\{g\} \implies \mathsf{Frame}(c) \Vdash_{\mathsf{EVFH}} \{w \wedge_{\mathsf{Pr}} p\}\{g \wedge_{\mathsf{Pr}} p\}$$

Unlike normal proposition expressions, we cannot just lift the expression to the wider variable domain. The resulting expression would behave as $\mathrm{true}_{\mathsf{Pr}}$ (full non-determinism) for the framed variables. We can instead take the expression, compute $\mathrm{id}_{\mathsf{Rl}}$ over the new variables, lift each to the union of their domains, and conjoin them.

Definition 6.18. The *relational frame* rframe adds equalities to a proposition-as-relation expression to expand its variable domain, behaving as the identity on the new variables:

$$\mathsf{rframe}(\Sigma_2^{\mathsf{M}} \subseteq \mathsf{Var}^{\mathsf{M}}) \;:\; \mathrm{E}^{\Sigma_1^{\mathsf{M}}}_{\mathsf{Pr}} \to \mathrm{E}^{\Sigma_1^{\mathsf{M}}\uplus\Sigma_2^{\mathsf{M}}}_{\mathsf{Pr}} \qquad \mathsf{rframe}(\Sigma_2^{\mathsf{M}})(e) \stackrel{\text{def}}{=} e \wedge^{\Sigma_1^{\mathsf{M}}\uplus\Sigma_2^{\mathsf{M}}}_{\mathsf{Pr}} \mathrm{id}^{\Sigma_2^{\mathsf{M}}}_{\mathsf{Rl}}$$

(Coq: add_frame in Starling.Backend.Alpha.PredAsRel)

Theorem 6.19. rframe is a valid framing function, according to the definition above. (Coq: framing_preserves_bvhoare in Starling.Backend.Alpha.PredAsRel)

Non-determinism. When describing atomic actions as two-state propositions, we can encode non-determinism as ambiguity in the post-state. For example, a command that *may* fetch-and-increment but *may instead* decrement may look like $(x' = y \wedge (y' = y - 1 \vee y' = y + 1))$.

We can model 'havoc' — full non-determinism over a variable v's post-state, and so the loss of any information about its value — by omitting v' from the relation expression. As with the UTP, we can model *abort* — non-determinism over a full shared state — with the Boolean expression true. Similarly, we can model *miracle* — the hypothetical command that satisfies any specification asked of it — with false.

The grammar of structured propositions

This dissertation gives structured propositions a S-expression-based grammar based on SMT-LIB's core syntax [62]. To let us extend structured propositions, the grammar provides extension points. These permit custom unary and binary operators on both propositions and values, as well as *symbols*: a generic method for quoting pieces of the backend's own proposition syntax, parametrised over values (or value expressions).

While symbols are not the focus here, they become important for proving heap-based algorithms in § 8.1. Symbols are the only part of a structured proposition that may depend on the solver's global context; this becomes useful later on.

Definition 6.19. The *structured propositions set* SPred(Val), where Val is a value set (or value expression set), is the set of all productions of the grammar below:

⟨SPred⟩	::= `true` | `false`	literals
	| (`and` ⟨SPred⟩ ⟨SPred⟩)	conjunction
	| (`=>` ⟨SPred⟩ ⟨SPred⟩)	implication
	| (`=` ⟨Val⟩ ⟨Val⟩)	equality
	| (`cvbop` ⟨*cvbop*⟩ ⟨Val⟩ ⟨Val⟩)	custom value binary
	| (`cebop` ⟨*cebop*⟩ ⟨SPred⟩ ⟨SPred⟩)	custom proposition binary
	| (`cvmop` ⟨*cvmop*⟩ ⟨Val⟩)	custom value unary
	| (`cemop` ⟨*cemop*⟩ ⟨SPred⟩)	custom proposition unary
	| (`sym` ⟨*symbol*⟩ ⟨Val⟩*)	symbols

In examples where we assume a particular theory, and therefore a particular set of custom operators, we can elide the `c…op` prefix for concision.

As a functor. We can map over the values in a structured proposition. Unlike proposition languages in general, where we only assume that fmap reaches values inside equalities, fmap on structured propositions is guaranteed to reach all values in the expression (except any values embedded within a symbol's syntax).

Lemma 6.20. SPred(Val) forms a legal functor with the following fmap:

$$
\begin{aligned}
\mathsf{fmap}(f)(&\quad\mathsf{true}) &&= \mathsf{true}\\
\mathsf{fmap}(f)(&\quad\mathsf{false}) &&= \mathsf{false}\\
\mathsf{fmap}(f)(&\quad(\mathbf{and}\ x\ y)) &&= (\mathbf{and}\ \ (\mathsf{fmap}(f)(x))\ \ (\mathsf{fmap}(f)(y)))\\
\mathsf{fmap}(f)(&\quad(\mathbf{\Rightarrow}\ x\ y)) &&= (\mathbf{\Rightarrow}\ \ (\mathsf{fmap}(f)(x))\ \ (\mathsf{fmap}(f)(y)))\\
\mathsf{fmap}(f)(&\quad(\mathbf{cvbop}\ o\ x\ y)) &&= (\mathbf{cvbop}\ o\ \ (f(x))\ \ (f(y)))\\
\mathsf{fmap}(f)(&\quad(\mathbf{cebop}\ o\ x\ y)) &&= (\mathbf{cebop}\ o\ \ (\mathsf{fmap}(f)(x))\ \ (\mathsf{fmap}(f)(y)))\\
\mathsf{fmap}(f)(&\quad(\mathbf{cvmop}\ o\ x)) &&= (\mathbf{cvmop}\ o\ \ (f(x)))\\
\mathsf{fmap}(f)(&\quad(\mathbf{cemop}\ o\ x)) &&= (\mathbf{cemop}\ o\ \ (\mathsf{fmap}(f)(x)))\\
\mathsf{fmap}(f)((&\mathbf{sym}\ s\ x_1\ \ldots\ x_n)) &&= (\mathbf{sym}\ s\ \ (f(x_1))\ldots\ \ (f(x_n)))
\end{aligned}
$$

As proposition expressions. Structured propositions have productions for $\text{true}_{\mathsf{Pr}}$ (true), $\text{false}_{\mathsf{Pr}}$ (false), $\wedge_{\mathsf{Pr}}$ ((**and** - -)), $\Rightarrow_{\mathsf{Pr}}$ ((⇒ - -)), and $=_{\mathsf{Pr}}$ ((= - -)). If structured propositions abstract over a member of one of our previously-defined language classes, they inhabit that class too.

Encoding heap reasoning

The discussion of Definition 6.11 claimed that the state-variables approach would support heap-based reasoning. Let us expand on this claim.

Heap values and heap variables. To encode a heap, we can assume that Val contains a potentially infinite set of *heap values*. The exact nature of these values depends on the backend. We can then set aside a variable denotation in Σ to reference the heap; the discussion below refers to this variable as heap. As backend results always imply those of the verification-condition Hoare judgement, which quantifies over all valid state functions that satisfy the weakest-precondition and command, valid heap backends must quantify over all possible assignments for heap, and so all heaps.

Heap equality and framing. As heap appears in Σ, the backend theory must allow us to express the comparison (= heap heap), and the implementation of id_{RI} for propositions-as-relations generates the proposition (= heap heap′). At first, this seems intractable, as it asks us to compare heaps for equality. In practice, so long as we only allow one heap variable to exist, this comparison can only take one of three dischargeable forms:

- reflexivity (= heap heap), which we can model as true without further heap inspection;
- comparison (= heap x) with some other variable x; since only one heap variable exists, the comparison is ill-typed, and we can model it as false;
- framing (= heap heap′).

The translation of the last case depends on the backend solver. If the backend heap is a discrete variable with explicit framing, we can emit the equality as normal. Where heap framing is implicit (at each atomic action, the absence of any heap operation is equivalent to the preservation of the heap) we can model the equality specially, for example as true or an empty command. As framing is the only case where a comparison between two heap variables with lexically distinct variable names is valid, detecting this case is straightforward.

Rationale. This work models the heap as a special variable both for historical reasons (the first versions of Starling worked only with discrete variables) and because parts of the theory are more straightforward if we assume discrete variables throughout. Adding the heap as a first-class theory concept remains future work.

This approach works, framing clumsiness aside, for several reasons. We make few assumptions about how variables fit into proposition expressions, and can easily work around them for heaps. We assume that heap accesses are atomic and sequentially consistent; each

verification condition can assume exclusive access over the heap. By assuming only one heap variable exists, any appearance of heap equalities in practice has a well-defined meaning.

6.4 Guarded views

To build an automation-friendly LVF logic, we must restrict how local variables affect the choice of shared state view (as we saw in § 5.6). We must forbid local variables from influencing the choice of which defining views appear inside a view — unless they do so in ways we can offload to the backend solver. The *abstract predicate* atom scheme (Definition 6.3), along with the value expressions in § 6.3, lets us embed local observations into views as parameters to the underlying definitions.

This section proposes an additional way to embed local-state reasoning. *Guarded views* are view assertions containing atoms annotated with a predicate over the local state. The predicates form guard conditions over their atoms: when the predicate is false, the atom logically disappears from the view and no longer participates in reification or definition. By lifting the guards into implications in the underlying theory, and propagating them across reification, we can delay the evaluation of which atoms appear in each assertion until the backend solver can quantify over the local variables on which the guards depend.

Guard expressions

We can model guards as proposition expressions targeting L as the state set. Guard expressions cannot use any global or local context beside the current local state. The meta-theory justification of proofs over guarded views assumes that we can reduce a guarded view expression to its underlying views monoid; this means we must be able to decide the truth of each guard.

Definition 6.20. A *guard expression language* E_{Gd} is a proposition expression set with both context sets fixed to the singleton set $\{\,\mathtt{tt}\,\}$, and the property:

$$\forall e : \mathsf{E}_{\mathsf{Gd}},\ \mathsf{l} : \mathsf{L}.\ [\![e]\!]_{\mathsf{Pr}}(\mathtt{tt})(\mathtt{tt})(\mathsf{l}) \ \vee\ \neg[\![e]\!]_{\mathsf{Pr}}(\mathtt{tt})(\mathtt{tt})(\mathsf{l})$$

The set of guard expressions is, typically, a subset of the normal proposition expression set. The guard-compatible subset must only reference local state, and be independent of any global or local context. When using structural proposition expressions, as in ${}_{\mathsf{g}}$Starling, we can guarantee this by forbidding shared-variable references and symbols.

Guarded atoms

Guard expressions attach to individual atoms. This lets us enable or disable the atoms depending on local-state observations.

Definition 6.21. For all Atom, the *guarded atom set* $\mathsf{Atom}^{\mathsf{G}}$ is the set $(\mathsf{E}_{\mathsf{Gd}} \times \mathsf{Atom})$.

If we can expand unguarded atoms into views, we can expand guarded atoms into the same view algebra — provided that the view set is $\mathsf{L} \to V$ for some V. This captures the guards' local-state dependency. (We assume that the original atom language depends on local state for two reasons: first, we can use const to lift languages that do not; second, languages such as abstract predicates already have such a dependency.)

Definition 6.22. The *guarded interpretation* function intG wraps an atom language interpreter to handle a guard:

$$\mathsf{intG} : (\mathsf{Atom} \to \mathsf{L} \to V) \to \mathsf{Atom}^{\mathsf{G}} \to \mathsf{L} \to V$$

$$\mathsf{intG}(\mathsf{r})(\mathsf{g}, \mathsf{a})(\mathsf{l}) \stackrel{\text{def}}{=} \begin{cases} \mathsf{r}(\mathsf{a}) & [\![\mathsf{g}]\!]_{\mathsf{Pr}}(\mathsf{tt})(\mathsf{tt})(\mathsf{l}) \\ \varepsilon & \neg[\![\mathsf{g}]\!]_{\mathsf{Pr}}(\mathsf{tt})(\mathsf{tt})(\mathsf{l}) \end{cases}$$

Lemma 6.21. If $(\mathsf{Atom}, (\mathsf{L} \to V), \mathsf{r})$ is an atom language, so is $(\mathsf{Atom}^{\mathsf{G}}, (\mathsf{L} \to V), \mathsf{intG}(\mathsf{r}))$.

Guarded abstract predicates. We can apply atom guarding to the abstract-predicate atom scheme (Definition 6.3) to produce *guarded abstract predicates* (GAPs). Let us denote GAP sets as $\mathsf{APred}(\mathsf{Tag})(\mathsf{E}_{\mathsf{Val}})^{\mathsf{G}}$, and give GAPs the following S-expression grammar:

$\langle \mathsf{APred}(\mathsf{Tag})(\mathsf{E}_{\mathsf{Val}})^{\mathsf{G}} \rangle$ ::= (⟨*tag*⟩ ⟨*arg*⟩*) | (→ ⟨*expr*⟩ ⟨*tag*⟩ ⟨*arg*⟩*)

A guardless S-expression is syntactic sugar for one with the guard true.

We assume the existence of a functor instance over GAPs, where fmap distributes over the expressions in both argument and guard positions.

Example. We can now try to write an idiomatic assertion set for Peterson's algorithm using view expressions over guarded abstract predicates[10].

The first task is build a tag set that represents high-level assertions about the algorithm's state. We can do so by mapping each state from the finite-state automaton we gave earlier (Figure 3.3) to a tag. This gives us the tags *flagDown*, *flagUp*, *waiting*, and *holdLock*. The two threads will share these tags, but we can disambiguate by parametrising the abstract predicates with thread identifiers. For this example, let A and B be legal values corresponding to the thread of the same name. Then, (*flagDown* A) corresponds to thread A having its flag down — A1 and A4 in the original line-number system. We can annotate thread A as follows:

A1	(**@**(*flagDown* A))	`Q1 := true`
A2	(**@**(*flagUp* A))	`TURN := 1`
A3	(**@**(*waiting* A))	`wait until !Q2 or TURN=2`
	(Critical Section)	
A4	(**@**(*holdLock* A))	`Q1 := false`

In practice, we will likely not implement A3 as a single atomic action. A more realistic implementation is to loop over locally fetching the turn counter and other thread's flag, testing them, and proceeding to the critical section if the condition is met:

[10] We do not convert the algorithm itself into a valid LVF outline yet; doing so is unnecessary at this stage.

```
A3a      (@                                  (waiting  A))   |  q := Q2;
A3b  (• (@(→              q                   waiting  A))   |  t := TURN;
         (@(→         (not q)                  holdLock A)))  |
A3c  (• (@(→    (and      q     (= t 1))       waiting  A))   |  if q and t=1
         (@(→     (or (not q)   (= t 2))      holdLock A)))  |  then goto A3a;
 A4      (@                                  (holdLock A))   |  ...
```

The guarded atoms show the increase in information after each read operation. This progression highlights an important property of Peterson's algorithm: once thread A is waiting for the lock, it acquires the lock as soon as B releases it (by dropping its flag or by setting TURN to 2), and keeps it until it exits the critical section. As soon as we see one of those conditions, we infer (@*holdLock*)A. Regardless of whether the conditions remain true, the predicate is stable, and we can keep the lock.

Guarded view equivalence and inclusion

Lemma 6.14 states that view expressions lift their underlying views algebra. This lets us use operators like ε, $\bullet$, and $\setminus$ at the view expression level; these operators work as expected, and distribute properly down to the underlying algebraic level.

This lifting also defines $\equiv$ and $\sqsubseteq$. These lifted operators must be used with caution on guarded view expressions; the underlying algebra is over functions from local state to atom multisets, so the relations only hold over the expressions where they hold for the underlying multisets over *all* possible local states. This is neither decidable nor guaranteed to give the expected result at the multiset level.

We can also define *structural* equivalence and inclusion on list-normalised guarded view expressions. A view expression v_1 structurally includes a view expression v_2 if every atom provided that no atom appears in v_2 more times than it appears in v_1; structural equivalence is inclusion in both directions. (In other words, we treat view expressions as multisets of atoms, and use the usual views algebra for multisets). These structural definitions form the core of the pattern matching algorithms we introduce in § 6.5.

Adapting lnorm for guarded atoms

Suppose we now want to build a set of verification conditions for one of the atomic actions in Peterson's algorithm: for example,

$$\{(@(\mathit{waiting}\ \mathrm{A}))\}\ \mathrm{q}\ :=\ \mathrm{Q2};\ \{(@(\rightarrow \mathrm{q}\ \mathit{flagUp}\ \mathrm{A}))\}$$

If our frontend is based on the adjoint proof rule (gStarling is), then we will eventually consider part expressions over guarded atoms: for example,

$$\forall \mathrm{x} \in \{\mathrm{A}, \mathrm{B}\}.\ (\setminus\ (@(\mathit{flagDown}\ \mathrm{x}))\ (@(\rightarrow \mathrm{q}\ \mathit{flagUp}\ \mathrm{A})))$$

To make the matching process more straightforward, we can list-normalise such expressions. While the above example is trivial (the atoms have different tags and thus cannot cancel out) we must be careful in general; the assumption we made when building lnorm

— the atom set has equality that we can decide while list-normalising — does not hold for guarded atoms. For example, consider this equality on guarded abstract predicates[11]:

$$(\rightarrow(x > 5)\ \texttt{atom}\ y) \stackrel{?}{=} (\rightarrow(z < 6)\ \texttt{atom}\ 3)$$

This decision depends on the values of x, y, and z; these are not known until we reach the backend solver! To list-normalise guarded view expressions, we must replace lnorm_P with a function that computes atom-wise subtraction *without* deciding equality itself. We do so by embedding the decision into the guards of the returned atoms.

Equality guards. Let us assume an *equality guard* function $\mathsf{eqG} : \mathsf{Atom} \to \mathsf{Atom} \to \mathsf{E}_{\mathsf{Gd}}$. This function returns a guard that evaluates to true provided that the two provided atoms, less their current guards, are equal (formally: $[\![\mathsf{eqG}(x)(y)]\!]_{\mathsf{Pr}} \iff r(x) \equiv r(y)$). If tag equality is decidable, we can define abstract predicate equality guarding as follows:

$$\mathsf{eqG}_{\mathsf{ap}}((s_1 a_1\ \ldots\ z_1))((s_2 a_2\ \ldots\ z_2)) \stackrel{\text{def}}{=} \begin{cases} \mathsf{false}_{\mathsf{Pr}} & s_1 \neq s_2 \\ a_1 =_{\mathsf{Pr}} a_2 \wedge_{\mathsf{Pr}} \cdots \wedge_{\mathsf{Pr}} z_1 =_{\mathsf{Pr}} z_2 & \text{otherwise} \end{cases}$$

In the above example, the equality guard would be $y =_{\mathsf{Pr}} 3$.

Remainder guards. We can use the equality and atom guards to work out the guards of the atoms we return. Let $(\backslash\ (@(\rightarrow g_m\ m))\ (@(\rightarrow g_s\ s)))$ be the subtraction in question. Then, the remainder guards, and their intuitive justifications, are:

$$g'_m \stackrel{\text{def}}{=} \underset{\substack{\textbf{if}\\ m\text{ was active}}}{g_m} \wedge \neg(\underset{\substack{\textbf{and either}\\ s\text{ was inactive}}}{g_s} \wedge \underset{\substack{\textbf{or}\\ \text{atoms do not match}}}{\mathsf{eqG}(m)(s)})$$

$$g'_s \stackrel{\text{def}}{=} \underset{\substack{\textbf{if}\\ s\text{ was active}}}{g_s} \wedge \neg(\underset{\substack{\textbf{and either}\\ m\text{ was inactive}}}{g_m} \wedge \underset{\substack{\textbf{or}\\ \text{atoms do not match}}}{\mathsf{eqG}(m)(s)})$$

In our `atom` example, g'_m becomes $x > 5 \wedge \neg(z < 6 \wedge y = 3)$, and g'_s becomes $z < 6 \wedge \neg(x > 5 \wedge y = 3)$.

Replacing lnorm_P. The remainder guard definitions lead towards a new version of lnorm_P. This new function, lnorm_P^G, returns guarded atoms instead of $\mathsf{Atom} \cup \{\bot\}$.

$$\mathsf{lnorm}_P^G : \mathsf{Atom}^G \to \mathsf{Atom}^G \to (\mathsf{Atom}^G \times \mathsf{Atom}^G)$$

$$\mathsf{lnorm}_P^G((g_m, m))((g_s, s)) \stackrel{\text{def}}{=} \big((g'_m, m), (g'_s, s)\big)$$

Lemma 6.22. If we apply the remainder guards to their respective atoms, we get the appropriate remainder atoms to perform a single step of lnorm_A:

$$(\backslash\ (\bullet\ (@(\rightarrow g_m\ m))\ r)\ (@(\rightarrow g_s\ s))) \quad\equiv\quad (\bullet\ (@(\rightarrow g'_m\ m))\ (\backslash\ r\ (@(\rightarrow g'_s\ s))))$$

[11] While we use GAPs in the examples, these adaptations work with any guarded atom set.

Replacing lnorm_A**.** We must alter lnorm_A slightly to reflect the new return type of lnorm_P^G.

$$\mathsf{lnorm}_A^G(\mathit{1})(s) \stackrel{\text{def}}{=} \mathit{1}$$
$$\mathsf{lnorm}_A^G((\bullet\ (@a)\ x))(s) \stackrel{\text{def}}{=} \mathsf{recur}^G(\mathsf{lnorm}_P^G(a, s))(x)$$
$$\text{where}\quad \mathsf{recur}^G((a', s'))(x) \stackrel{\text{def}}{=} (\bullet\ (@a')\ \mathsf{lnorm}_A^G(x)(s'))$$

The only change we make to lnorm_B and lnorm is to substitute lnorm_A^G for lnorm_A. We call the resulting functions lnorm_B^G and lnorm^G respectively, but omit their definition.

6.5 Guarded syntactic definers

The syntactic definers we used to define views in $_\mu$Starling and $_{lo}$Starling directly map view fragments to proposition expressions. This approach is unsuitable, without adaptation, for guarded abstract predicates; using it, we would be unable to abstract over the arguments supplied to a guarded abstract predicate, and our definers would need a separate definition for every possible argument combination.

We would also need to work out whether one guarded abstract predicate expression is included in another. We must be able to evaluate each guard at definition expansion time or build a decidable ordering over guards; neither restriction is appropriate.

We can instead adapt syntactic definers to permit a looser relationship between views, definitions, and the resulting proposition expressions.

The ticket lock

As *Peterson* has a highly regular atom set and verbose definitions, a new example — Algorithm 1, the *ticket lock* popularised by Mellor-Crummey and Scott [63][12] — better demonstrates guarded syntactic definers. The lock provides mutual exclusion for an arbitrary number of threads using a queuing system based on taking integer tickets, and waiting for a monotonically increasing 'now serving' counter to reach the taken number.

Specification. The specification we will try to prove is that given for *Peterson* in § 3.3: that there is some abstract Lock resource that can have at most one instance (which persists across a thread's critical section). In Algorithm 1, the critical section is the period between the end of a call to Lock and the beginning of a call to Unlock inclusive. A Lock exists when $s < n$ (since, if $s = n$, the ticket being served has not yet been taken).

Defining the definers

This section outlines the structure of guarded syntactic definers. As an example, it builds a guarded syntactic definer for the ticket lock.

[12] As Starling does not consider temporal properties, we need not model exponential backoff.

Algorithm 1 Mellor-Crummey and Scott's ticket lock

```
s : ℕ shared                                   ▷ ticket currently being served
n : ℕ shared                                   ▷ next ticket

procedure LOCK
    t : ℕ                                      ▷ local storage for ticket
    c : ℕ                                      ▷ local storage for current view of s
    ⟨t := n;  n := n + 1⟩                      ▷ take a ticket
    repeat
        ⟨c := s⟩                               ▷ wait until our ticket is being served
    until c = t
end procedure

procedure UNLOCK
    ⟨s := s + 1⟩                               ▷ serve the next ticket
end procedure
```

Predicate prototypes. The first part of a guarded syntactic definer is a system of *predicate prototypes*. These describe which tags, and argument counts, yield valid abstract predicates.

> **Definition 6.23.** A *predicate prototype function* $\mathsf{PProto}(\mathsf{Tag})$ is a function $\mathsf{Tag} \to \mathbb{N}$, mapping tags to the number of arguments they expect.

Each thread using the ticket lock can be in one of three states: idle, waiting in the queue (holding a ticket), and holding the lock. There is no limit on the number of threads using the lock, and each thread uses the shared state in the same way, so we need not encode thread IDs into the abstract predicates. Idle threads have no non-invariant information about the shared state, as we see in the final definer, so we need only two tags for the proof: *tick*, which represents queuing, and *lock*, which represents locking. We parametrise *tick* by the number of the ticket the thread is holding. This gives us the prototype list $[(\mathit{tick}, 1), (\mathit{lock}, 0)]$.

Patterns. In § 4.3, we matched definitions to views by directly deciding view inclusion. As definitions now abstract over the arguments of their defined atoms, we need a pattern language and pattern matcher that can handle such definitions.

If all atoms are well-formed according to a prototype list, then tags hold all of the information we need about an atom inside a pattern. We can, then, model patterns in guarded syntactic definers as tag lists.

Definitions. Each definition in a guarded syntactic definer is a pair of a tag list and a structured predicate *template*. The template contains value expressions parametrised over both shared variables and the parameters of each abstract predicate the pattern represents. Let us model parameter indices as $\mathbb{N}$: 0 is the first parameter of the first abstract predicate, and indexing proceeds left-to-right across all parameters and predicates[13].

[13] This allows out-of-bounds indexing, but we can just map invalid references to $\bot$.

```
([          ], (>= n s)          )   (we cannot reallocate expired tickets)
([tick      ], (> n P0)          )   (we cannot reallocate queuing tickets)
([lock      ], (not (= n s))     )   (we cannot reallocate lock-holding tickets)
([lock, tick], (not (= s P0))    )   (tickets are given up when taking the lock)
([tick , tick], (not (= P0 P1))) )   (tickets are unique)
([lock, lock], false             )   (mutual exclusion)
```

Figure 6.1: A guarded syntactic definer for the Mellor-Crummey/Scott ticket lock.

Definition 6.24. The *definition set* over a prototype set P and shared alphabet Σ is:

$$\mathsf{GDefn}(\mathsf{P})(\Sigma) \stackrel{\text{def}}{=} (\mathrm{dom}\ \mathsf{P} \times \mathbb{N})$$

A *guarded syntactic definer* over P and Σ is a list of $\mathsf{GDefn}(\mathsf{P})(\Sigma)$.

Figure 6.1 is an example definer for the ticket lock, in which P_n denotes the nth parameter.

The pattern matching algorithm

To match a (list-normalised) view expression against a pattern, we can calculate all partial permutations of that view's atoms such that each atom's tag matches the tag at the same position in the pattern. Each permutation becomes a distinct instantiation of the definition; if at least one such permutation exists, the view matches the pattern.

For example, the view $(\odot\ (@(\mathit{tick}\ 1))\ (@(\mathit{tick}\ 2)))$ matches $[\mathit{tick}, \mathit{tick}]$ in two ways: either we map the first tag to the 1 atom and the second to the 2 atom, or we reverse the order. In this case (but not in general), both matches yield equivalent predicates.

This section develops a pattern matching algorithm in stages, with pseudocode.

Matching a single tag. In the first stage, we traverse a view until we either run out of atoms or find one with the required tag. When we find a match, we record its position in the view and continue traversing; once we reach the end of the view, we have the positions of all matches for that tag. We will need to 'un-traverse' the skipped atoms for use in later matches, so the algorithm uses a *zipper* [64] data structure to hold each position:

$$\mathsf{Zipper} \stackrel{\text{def}}{=} (\ \underbrace{\mathsf{VExpr}(\mathsf{APred}(\mathsf{Tag})(\mathsf{E}_{\mathsf{Val}})^{\mathsf{G}})_{\mathfrak{l}}}_{\substack{\text{view traversed so far}\\ \text{(in reverse order)}\\ \downarrow}} \times \underbrace{\mathsf{VExpr}(\mathsf{APred}(\mathsf{Tag})(\mathsf{E}_{\mathsf{Val}})^{\mathsf{G}})_{\mathfrak{l}}}_{\substack{\text{view left to traverse}\\ \text{(in normal order)}\\ \downarrow}}\ )$$

(Coq: Starling.Utils.Zipper)[14]

[14]The Coq development represents the views inside the zipper as atom lists.

The first-stage algorithm accepts views-to-traverse as zippers. The view starts on the right-hand side of the zipper; as we check each atom, it moves to the left.

Algorithm 2 First stage of guarded pattern matcher

```
function MatchTag(t : Tag, v : Zipper)
    p : list Zipper := []
    while ∃a, v_l, v_r. v = (v_l, (• (@a) v_r)) do
        if a has tag t then
            p := p ++[v]
        end if
        v := ((• (@a) v_l), v_r)
    end while
    return p
end function
```

Processing the match positions. Next, for each position in the resulting list, we split the view, removing the matched atom. This gives us two results per match: the actual atom we matched against, and the rest of the view to use when matching further tags.

Let us now assume each position is a zipper whose second view starts with the matched atom; we can then remove the atom and rewind the view traversal using cons and uncons operations. Assume also that we have some view expression v_p that contains the subview match for the sub-pattern preceding tag t. By appending each matched atom onto this expression in turn, we get the list of subview matches up to and including t. This operation is exponential, as each choice between two matches for t doubles the number of overall matches, but the length of the patterns is usually too small to cause issues.

Algorithm 3 Second stage of guarded pattern matcher

```
function ProcessTagMatch(p : list Zipper, v_p : VExpr(APred(Tag)(E_Val)^G)_l)
    m : list (VExpr(APred(Tag)(E_Val)^G)_l × Zipper) := []
    for all p in p by reference do
        assert ∃a, p_l, p_r. p = (p_l, (• (@a) p_r))
        p := (p_l, p_r)                          ▷ remove matched atom
        while ∃a', p'_r. p_r = (• (@a') p'_r) do
            p := ((• (@a') p_l), p_r)            ▷ rewind zipper
        end while
        m := m ++[((• (@a) v_p), p)]             ▷ add match and remaining view to results
    end for
    return m
end function
```

Matching an entire pattern. To match a pattern, we can recursively apply the previous stages for each tag in the pattern. As the second stage appends matched atoms to the front of the view, we match the tags in reverse order[15].

[15]The mechanisation implements this loop using a right fold.

Algorithm 4 Third stage of guarded pattern matcher

```
function MatchPattern(v : VExpr(APred(Tag)(E_Val)^G)_l, t : list Tag)
    m : list (VExpr(APred(Tag)(E_Val)^G)_l × Zipper) := [(1, (1, v))]   ▷ empty pattern matches unit
    for all t in t reversed do
        m' : list (VExpr(APred(Tag)(E_Val)^G)_l × Zipper := []
        for all (v_p, v_n) in m do
            p := MatchTag(t, v_n)
            m' := m' ++ ProcessTagMatch(p, v_p)
        end for
        m := m'                                   ▷ use new matches for next tag
    end for
    r : list VExpr(APred(Tag)(E_Val)^G)_l := []
    for all (v_p, v_n) in m do                    ▷ remove zippers
        r := r ++[v_p]
    end for
    return r
end function
```

Final steps. The pattern-matching algorithm returns a list of view expressions over guarded abstract predicates. For the match to hold, all of the matched abstract predicates must be switched on; in other words, all of the guards must be true. We can, therefore, replace the atom-level guards with a single guard across the whole view expression. We can do so in the obvious way: extracting and conjoining each atom guard.

Theorem 6.23. The pattern-matching algorithm is *sound*:

- each result contains exactly as many atoms as the pattern has tags;
  (Coq: g_pattern_matches_merged_length in Starling.Frontend.APred.GMatch)
- each atom's tag corresponds to the tag in the same position in the pattern;
  (Coq: g_pattern_matches_tags in Starling.Frontend.APred.GMatch)
- each result is a subview of the original input view.
  (Coq: g_pattern_matches_subview in Starling.Frontend.APred.GMatch)

Defining and reifying views using guarded syntactic definers

The next step towards using guarded syntactic definers is to consider how to build schemes for defining and reifying views using them. These schemes depend on the ability to instantiate the definitions of pattern matches.

Instantiation. Once we have a pattern match, we can instantiate its definition. This involves replacing each positional index used in the definition with the corresponding argument in the match. As the indices count upwards across the match's atoms, we must resolve each index to the right argument of the right atom. To make this easier, we can *flatten* the match into a single atom, combining the tags and arguments of the original atoms.

Definition 6.25. The function vflat flattens list-normalised view expressions over abstract predicates into single abstract predicates, where the tag is the list of each atom's tag, and the argument list the concatenation of the atoms' argument lists:

$$\mathsf{vflat} : \mathsf{VExpr}(\mathsf{APred}(\mathsf{Tag})(\mathsf{E_{Val}})^{\mathsf{G}})_{\mathsf{l}} \to \mathsf{APred}(\mathsf{list}\ \mathsf{Tag})(\mathsf{E_{Val}})^{\mathsf{G}}$$

$$\mathsf{vflat}(v) \stackrel{\text{def}}{=} \mathsf{vflat}'(v)(\texttt{([]})$$

$$\text{where} \quad \mathsf{vflat}'(\quad\quad\quad\quad\quad\quad 1)(\quad\quad a) \stackrel{\text{def}}{=} a$$

$$\mathsf{vflat}'(\quad (\bullet\ (@(t\ \ e))\ v))(\quad (t\ \ e)) \stackrel{\text{def}}{=} \mathsf{vflat}'(v)((t +\!\!+ [t]\ \ e +\!\!+ [e]))$$

Flattened abstract predicates are valid over a prototype set P when each tag in their tag list appears in P and the argument count is the sum of each tag's argument count in P. More formally, we can derive a new prototype set for flattened predicates:

$$P^F \stackrel{\text{def}}{=} \{\, ([t_1, t_2, \ldots, t_n], \Sigma\{P(t_1), P(t_2), \ldots, P(t_n)\}) \mid t_1, t_2, \ldots, t_n \in \mathrm{dom}\ P \,\}$$

Definition 6.26. The *instantiation* $\mathsf{ginst}(d)(a)$ of a definition d against a flattened guarded abstract predicate a is the substitution of each argument in a for the corresponding reference in d's expression, guarded by implication over a's guard:

$$\mathsf{ginst} \ : \ \mathsf{GDefn}(P)(\Sigma) \to \mathsf{VExpr}(\mathsf{APred}(\mathsf{Tag})(\mathsf{E_{Val}})^{\mathsf{G}})_{\mathsf{l}}$$

$$\mathsf{ginst}((p, e))((\Rightarrow\ g\ t\ a_1\ \ldots\ a_n)) \stackrel{\text{def}}{=} (\Rightarrow\ g\ \ \mathsf{fmap}(\lambda v.\, v \gg\!= \mathsf{ginst}_p)(e))$$

$$\text{where} \quad \mathsf{ginst}_p(n \in \mathbb{N}) \stackrel{\text{def}}{=} \begin{cases} a_n & \text{in bounds} \\ \bot & \text{otherwise} \end{cases}$$

$$\mathsf{ginst}_p(v \in \Sigma) \stackrel{\text{def}}{=} \mathsf{return}(v)$$

(Coq: inst_gmatch in Starling.Frontend.APred.GDefiner)

We can reduce a match for the pattern [*tick*, *tick*] to an instantiated definition as follows:

$$\begin{array}{ccccccc}
(\odot & (@ & (tick & 27 &)) & (@ & (tick\ \ 53\ )) \ ) \\
 & & & \big| & \text{flattening} & & \big| \\
 & (@ & (tick\ \ tick) & 27 & & & 53\ ) \\
 & & & \big| & \text{apply to definition} & & \big| \\
 & & (\Rightarrow\ \ (= & P_0 & & & P_1\ ) \ \mathsf{false}) \\
 & & & \downarrow & \text{instantiation} & & \downarrow \\
 & & (\Rightarrow\ \ (= & 27 & & & 53\ ) \ \mathsf{false})
\end{array}$$

Lemma 6.24. vflat maps P-valid abstract predicates to P^F-valid abstract predicates.

Reification. Reification on guarded syntactic definers is similar to that of syntactic definers (Definition 4.11): we traverse the definer to find definitions with view-matching patterns. As there can be multiple (or zero) such matches, we must conjoin the resulting instantiations.

Definition 6.27. The function gsdReify reifies views with a guarded syntactic definer:

$$\mathsf{gsdReify} \ : \ \mathsf{listGDefn}(\mathsf{P})(\Sigma) \to \mathsf{VExpr}(\mathsf{APred}(\mathsf{Tag})(\mathsf{E_{Val}})^{\mathsf{G}})_{\mathfrak{l}} \to \mathsf{SPred}(\mathsf{E_{Val}}(\Sigma))$$
$$\mathsf{gsdReify}(\mathbf{d})(v) \stackrel{\text{def}}{=} \bigwedge_{\mathsf{Pr}} \{\, \mathsf{ginst}((\mathbf{t}, e))(\mathfrak{m}) \mid (\mathbf{t}, e) \in \mathbf{d} \wedge \mathfrak{m} \in \mathsf{matchPattern}(v)(\mathbf{t}) \,\}$$

Definition. Views need not have a single definition in guarded syntactic definers. Definition, like reification, involves pattern matching and instantiation. To convert reification into definition, we must ensure all of the matches we instantiate are not just sub-expressions of the view to define, but are equivalent (as discussed in § 6.4).

Lemma 6.25. If a pattern match has the same number of guarded abstract predicates as the matched view expression, the two expressions are equivalent.

Definition 6.28. The function gsdDef defines views with a guarded syntactic definer:

$$\mathsf{gsdDef} \ : \ \mathsf{listGDefn}(\mathsf{P})(\Sigma) \to \mathsf{VExpr}(\mathsf{APred}(\mathsf{Tag})(\mathsf{E_{Val}})^{\mathsf{G}})_{\mathfrak{l}} \to \mathsf{SPred}(\mathsf{E_{Val}}(\Sigma))$$
$$\mathsf{gsdDef}(\mathbf{d})(v) \stackrel{\text{def}}{=} \bigwedge_{\mathsf{Pr}} \left\{ \mathsf{ginst}((\mathbf{t}, e))(\mathfrak{m}) \;\middle|\; \begin{array}{l} (\mathbf{t}, e) \in \mathbf{d} \\ \wedge\ \mathfrak{m} \in \mathsf{matchPattern}(v)(\mathbf{t}) \\ \wedge\ |\mathfrak{m}| = |\mathbf{t}| \end{array} \right\}$$

Because of the issues we saw in § 6.4, showing that gsdReify has the right relationship with gsdDef is challenging, and left to a formal soundness proof.

6.6 The $_{g}$Starling frontend

This section discusses the $_{g}$Starling frontend, and the work done on its soundness proof.

Assumptions. The $_{g}$Starling frontend fixes many parameters left open in previous frontends, and makes more assumptions about the frontend's usage, such that:

- the views algebra for outline assertions is $\mathsf{VExpr}(\mathsf{APred}(\mathsf{Tag})(\mathsf{E_{Val}})^{\mathsf{G}})$, the set of (non-normalised) guarded views expressions;
- the front-facing guard and predicate languages is $\mathsf{SPred}(\mathsf{E_{Val}})$, the set of structured propositions. We forbid symbols in guards, making each guard context-independent;
- the outline is LVF-compatible (Definition 5.23);
- all variables mentioned in the proof belong to a single set Var with two disjoint alphabets: Σ_{lo} for local variables, and Σ_{s} for shared variables;
- local states map variables to expressible values (Definition 6.15). In practice, this is not a strong requirement; it mainly forbids cases like modelling heaps as local variables.

Parameters. As with $_{\mu}$Starling and $_{lo}$Starling, each $_{g}$Starling proof depends on several parameters describing the specific view algebra and command language that the proof uses. The above requirements limit $_{g}$Starling's parameter interface to the following:

- the aforementioned variable set Var with disjoint alphabets Σ_{lo} and Σ_s;
- an atomic action set A; with semantics function $[\![-]\!] : A \to \mathsf{SPred}(\Sigma_{lo} \uplus \Sigma_s)$;
- a tag set Tag with prototype function $P : \mathsf{PProto}(\mathsf{Tag})$;
- a definer $d : \mathsf{list}\ \mathsf{GDefn}(P)(\Sigma_s)$.

From atomic triples to proof terms

As usual, the frontend is a relation from outline triples $\langle p \rangle\ c\ \langle q \rangle$ to backend conditions $\langle\!\langle w \rangle\!\rangle\ c\ \langle\!\langle g \rangle\!\rangle$. Like previous frontends (§ 4.4 and § 5.6), g is one syntactic definition of a goal view g', and w the syntactic reification of a weakest-precondition $w' = (p \bullet (g' \setminus q))$; unlike before, we now use gsdReify and gsdDef instead of the functions from § 4.3.

This section discusses how to derive g' and w', and so derive the high-level *proof terms* we use to generate backend conditions. To demonstrate how this process works, we use the following triple, which corresponds to the wait loop in the ticket lock:

$$\langle (\odot\ (@(\mathit{tick}\ t))) \rangle\ c \leftarrow s\ \langle (\odot\ (@(\to(=\ c'\ t')\ \mathit{lock}))\ (@(\to(\neq\ c'\ t')\ \mathit{tick}\ t'))) \rangle$$

Deriving g'. Previously, the definer domain contained full views which we could use as g'. As definitions are now over view *templates*, we must instantiate those templates instead.

While the arguments of abstract predicates inside p and q are expressions over the local state of the current thread, the arguments of g''s abstract predicates must represent all possible instantiations of said predicates, and therefore must quantify over all possible arguments. To have the solver perform the specific quantification for the goal view in use, we must map each argument to a fresh *goal variable*.

Handling goal variables raises several difficulties. Unlike local and shared variables, goal variables (being parameters to stable assumptions about the shared state) implicitly frame over atomic actions, and therefore have only one state. Like each local state, but unlike the shared states, they appear in both w and g.

The number of goal variables in use depends on the goal. If we modelled this dependency accurately, the alphabet of each term would be different. This complicates formalisation, especially in the Coq mechanisation where the variable domain is part of the type of terms. Instead, we can loosen the domain to $\mathbb{N}$, as with parameters in definitions. The solver need not instantiate the spurious goal variables; it can, for example, pull them to $\bot$.

Instantiating a goal from a tag list involves distributing the goal variables in ascending order, from left to right, according to each tag's defined argument count. The resulting atoms then form a list-normalised view expression.

Definition 6.29. The function instGoal instantiates a pattern as a goal view, mapping each parameter to a fresh goal variable:

$$\mathsf{instGoal} : \mathsf{PProto} \to \mathsf{listTag} \to \mathsf{VExpr}(\mathsf{APred}(\mathsf{Tag})(\mathbb{N}))_\mathfrak{l}$$

$$\mathsf{instGoal}(P)(\mathbf{t}) \stackrel{\text{def}}{=} \mathsf{recur}(P)(\mathbf{t})(0)$$

where

$$\mathsf{recur}(P)([])(n) \stackrel{\text{def}}{=} \mathit{1}$$

$$\mathsf{recur}(P)([t] +\!\!+ \mathbf{t})(n) \stackrel{\text{def}}{=} (\bullet\ (@(t\ \ G_n\ \ \ldots\ \ G_{n+P(t)}))\ \ \mathsf{recur}(P)(\mathbf{t})(n + P(t)))$$

Example. The function above instantiates the ticket lock goals as follows:

$$[\ \ ] \longrightarrow \mathit{1}$$
$$[\mathit{tick}\ \ ] \longrightarrow (\odot\ (@(\mathit{tick}\ \ G_0)))$$
$$[\mathit{lock}\ \ ] \longrightarrow (\odot\ (@(\mathit{lock})))$$
$$[\mathit{lock}, \mathit{tick}] \longrightarrow (\odot\ (@(\mathit{lock}))\ \ (@(\mathit{tick}\ \ G_0)))$$
$$[\mathit{tick}, \mathit{tick}] \longrightarrow (\odot\ (@(\mathit{tick}\ \ G_0))\ \ (@(\mathit{tick}\ \ G_1)))$$
$$[\mathit{lock}, \mathit{lock}] \longrightarrow (\odot\ (@(\mathit{lock}))\ \ (@(\mathit{lock})))$$

We can use these instantiated goals in terms. We implicitly append the guard true to each goal atom; we need not quantify over all possible guards as we already consider every defining combination of atoms. For example, the goals $\mathit{1}$, $(\odot\ (@(\mathit{lock})))$, $(\odot\ (@(\mathit{tick}\ \ G_0)))$, and $(\odot\ (@(\mathit{lock}))\ \ (@(\mathit{tick}\ \ G_0)))$ together model the situation where an environment thread holds the view: $(\odot\ (@(\to b_1\ \ \mathit{tick}\ \ x))\ \ (@(\to b_2\ \ \mathit{lock})))$.

Deriving w'. As in previous frontends, w' is $p \bullet (g' \setminus q)$. One difference is that w' now contains references to both goal and local variables. As with § 5.6, we must assign the right local states to the right parts of w'.

To demonstrate this, and further transformations, let us focus on three of the ticket lock goal views. $g' = (\odot\ (@(\mathit{lock})))$ checks that the lock acquisition itself is sound. $g' = (\odot\ (@(\mathit{tick}\ \ G_0)))$ checks that we can safely retain the ticket if we did not acquire the lock. Finally, $g' = (\odot\ (@(\mathit{lock}))\ \ (@(\mathit{lock})))$ models mutual exclusion.

In our example, the non-normalised w' for $g' = (\odot\ (@(\mathit{lock})))$ is:

$$(\bullet\ (@(\mathit{tick}\ \ t))\ \ (\backslash\ (\odot\ (@(\mathit{lock})))$$
$$(\odot\ (@(\to(=\ c'\ \ t'))\ \ \mathit{lock}))\ \ (@(\to(\neq\ c'\ \ t'))\ \ \mathit{tick}\ \ t))))$$

We get the other two versions of w' by substituting the appropriate goal.

Building the proof terms. Each proof term combines w', c, and some representation of the goal. While we could use g' here, this is wasteful: we synthesise a view over g''s underlying definition g_d, then break it down again. The resulting proposition expression will be the conjunction of all exact pattern matches for $g' - g_d$ and any other definitions over the same tags; we can get the same result by building a separate proof term for each g_d directly, and using the outer quantification over proof terms to reconstruct the conjunction.

Definition 6.30. A ${}_{\mathsf{g}}$Starling *proof term* is a triple $(w', \widehat{\alpha}, \mathsf{g}_\mathsf{d})$, where $\widehat{\alpha}$ is an atomic label; g_d is a definition from the frontend's definer; and w' is a view expression $(\bullet\ \mathsf{p}\ (\backslash\ \mathsf{g}'\ \mathsf{q}))$, where g_d is one of the definitions of g'.

Building verification conditions

Once we have a proof term $(w', \mathsf{c}, \mathsf{g}_\mathsf{d})$, we can generate its underlying backend verification condition by reification and semantics analysis.

Command. To get the relation expression c from an atomic label $\widehat{\alpha}$, where $\widehat{\alpha} \neq \mathrm{id}$, we just apply $[\![-]\!]$ directly. For our example, we can assume that $\mathsf{c} := \mathsf{s}$ has the expected semantics:

$$[\![\mathsf{c} := \mathsf{s}]\!] \;=\; \mathsf{rframe}((=\ \mathsf{c}'\ \mathsf{s})) \;=\; (\textbf{and}\ (=\ \mathsf{c}'\ \mathsf{s})\ (=\ \mathsf{s}'\ \mathsf{s})\ (=\ \mathsf{t}'\ \mathsf{t})\ (=\ \mathsf{n}'\ \mathsf{n}))$$

Goal variables (for example, G_0) are single-state, and need no framing.

As in ${}_{\mathsf{lo}}$Starling, the case where $\widehat{\alpha} = \mathrm{id}$ needs care; the returned expression must act as identity on both local and shared state. Unlike Definition 5.25, we need not compare local states in the frontend. Instead, we just emit framing equalities:

$$[\![\mathrm{id}]\!] \;=\; \mathrm{id}^{\{\mathsf{c},\mathsf{s},\mathsf{t},\mathsf{n}\}}_{\mathsf{Rl}} \;=\; (\textbf{and}\ (=\ \mathsf{c}'\ \mathsf{c})\ (=\ \mathsf{s}'\ \mathsf{s})\ (=\ \mathsf{t}'\ \mathsf{t})\ (=\ \mathsf{n}'\ \mathsf{n}))$$

List normalisation. The guarded syntactic definition and reification functions accept list-normalised view expressions. While g' is list-normalised by construction, w' is not; our next step, therefore, is to list-normalise w'. We can do so with $\mathsf{rmpart}^{\mathsf{G}}$ from § 6.4.

In our three examples, the sole non-trivial part of this normalisation is subtraction. For each, we subtract first $(\textbf{@}(\textit{lock}))$, then $(\textbf{@}(\textbf{→}(\neq\ \mathsf{c}'\ \mathsf{t}')\ \textit{tick}\ \mathsf{t}))$, from g':

$$(\odot\ (\textbf{@}(\textit{lock}))) \longrightarrow (\odot\ (\textbf{@}(\textbf{→}(\neq\ \mathsf{c}'\ \mathsf{t}')\ \textit{lock})))$$
$$(\odot\ (\textbf{@}(\textit{tick}\ \mathsf{G}_0))) \longrightarrow (\odot\ (\textbf{@}(\textbf{→}(\textbf{not}\ (\textbf{and}\ (\neq\ \mathsf{c}'\ \mathsf{t}')\ (=\ \mathsf{G}_0\ \mathsf{t})))\ \textit{tick}\ \mathsf{G}_0)))$$
$$(\odot\ (\textbf{@}(\textit{lock}))\ (\textbf{@}(\textit{lock}))) \longrightarrow (\odot\ (\textbf{@}(\textbf{→}(\neq\ \mathsf{c}'\ \mathsf{t}')\ \textit{lock}))\ (\textbf{@}(\textit{lock})))$$

To get w', we conjoin these results with the precondition $(\textbf{@}(\textit{tick}\ \mathsf{t}))$ in a list-normalised manner. For example, when $\mathsf{g}' = (\odot\ (\textbf{@}(\textit{lock})))$,

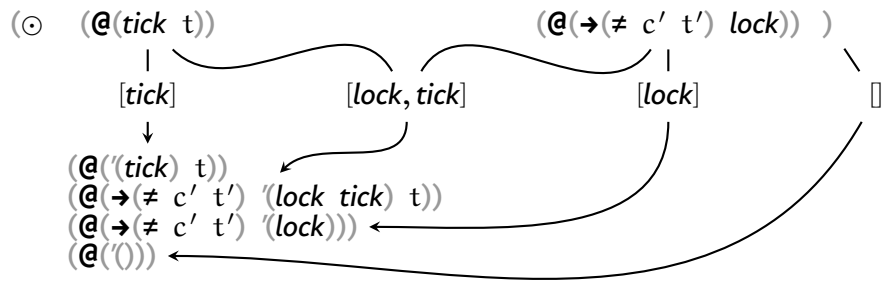
$$w' = (\odot\ (\textbf{@}(\textit{tick}\ \mathsf{t}))\ (\textbf{@}(\textbf{→}(\neq\ \mathsf{c}'\ \mathsf{t}')\ \textit{lock})))$$

Below, we concentrate on this particular pair of w' and g'.

Reification and definition. The next step is to define g' and reify w' against the definer from § 6.5. Atoms in g', by construction, only have true guards; this simplifies the former. In our example, just one pattern matches g' with the same number of atoms ($[\textit{lock}]$), and so its definition contains one instantiation. Recall that, when we match against guarded views, we lift and conjoin the guards, then flatten the view into one atom. For g', we have just one atom with a tautological guard, so the result of this match processing is exactly g':

$$\mathsf{gsdDef}(\mathsf{g}') \;=\; \mathsf{ginst}(([\textit{lock}], (\neq\ \mathsf{n}'\ \mathsf{s}')))((\bullet\ (\textbf{@}(\textit{lock}))\ \textit{1})) \;=\; (\neq\ \mathsf{n}'\ \mathsf{s}')$$

When reifying w', four patterns match:



Note that the explicit guard on *lock* carries over to *both* matches involving it[16]. Finally, we instantiate and conjoin the definitions of the pattern matches:

$$
\begin{aligned}
\mathsf{gsdReify}(w') \quad = \quad & (\texttt{>}\ n\ t) \longleftarrow [\mathit{tick}] \\
\wedge \quad & (\texttt{=>}\ (\neq\ c'\ t)\ (\neq\ n\ s)) \leftarrow [\mathit{lock}, \mathit{tick}] \\
\wedge \quad & (\texttt{=>}\ (\neq\ c'\ t)\ (\neq\ s\ t)) \leftarrow [\mathit{lock}] \\
\wedge \quad & (\texttt{>=}\ n\ s) \longleftarrow []
\end{aligned}
$$

Putting w', c', and g' together, we get the final condition:

$$
\begin{aligned}
&\langle\!\langle (\textbf{and}\ (\texttt{>}\ n\ t)\ (\texttt{=>}\ (\neq\ c'\ t)\ (\neq\ n\ s))(\texttt{=>}\ (\neq\ c'\ t)\ (\neq\ s\ t))\ (\texttt{>=}\ n\ s)) \rangle\!\rangle && (w') \\
&\qquad (\textbf{and}\ (=\ c'\ s)\ (=\ s'\ s)\ (=\ t'\ t)\ (=\ n'\ n)) && (c') \\
&\qquad\qquad \langle\!\langle (\neq\ n'\ s') \rangle\!\rangle && (g')
\end{aligned}
$$

We can dry-run this condition by case analysis on whether $c' = t$ (and, transitively, $s = t$). If so, then $n > t$ implies $n > s$, and therefore $n' > s'$. If not, the guarded parts of the weakest-precondition guarantee that $n \neq s$, and therefore $n' \neq s'$.

Towards a soundness argument for GStarling

There is not, yet, a formal soundness argument for GStarling. To achieve this, we could use the standard frontend argument (Figure 4.1), using fTemp (Definition 4.10) to build a template with the necessary properties. Earlier sections discussed the translation from GStarling atomic Hoare triples to verification conditions via proof terms, so the remaining burden lies in deriving a sound *LVF* instance[17].

Relating definition and reification. As in LoStarling, we must show that reifying a view v entails the conjunction of the definitions of each subview u. For GStarling, conjoining every pattern match of v should be equivalent to conjoining every *exact* pattern match of every u.

[16] Strictly speaking, the guard for ⟨(*lock tick*) is the conjunction of those for *lock* and *tick*.
[17] Again, Lemma 5.15 helps us carry out our soundness proof on the *CVF* encoding.

Lowering $_{g}$Starling for LVF compatibility. To target the LVF, we must be able to lower our assertions and definers to view functions $L \to V$. This was straightforward in $_{lo}$Starling, as the assertions were already view functions. In $_{g}$Starling, this is no longer the case — the local dependency comes indirectly through argument vectors and guards — so we must produce a scheme for evaluating and removing the guards, and substitute the arguments' (expressible) values for their expressions, that we can express in the form of view functions. At the same time, we must also handle the arguments of goal views.

The 'lowered' form of $_{g}$Starling maps outline views to functions that, given mappings from local and goal variables to expressible values, perform the evaluations and substitutions needed to produce shared-state-only views. These view functions have the type:

$$
\begin{aligned}
&(\Sigma_{lo} \to \mathsf{E}_{\mathsf{Val}}(\mathsf{Val})(\emptyset)) && \text{(pre-state)}\\
\to\ &(\Sigma_{lo} \to \mathsf{E}_{\mathsf{Val}}(\mathsf{Val})(\emptyset)) && \text{(post-state)}\\
\to\ &(\mathbb{N} \to \mathsf{E}_{\mathsf{Val}}(\mathsf{Val})(\emptyset)) && \text{(goal-state)}\\
\to\ &\mathsf{bag}\ \mathsf{APred}(\mathsf{Tag})(\mathsf{E}_{\mathsf{Val}}(\mathsf{Val})(\emptyset)) && \text{(erased view)}
\end{aligned}
$$

Note the existence of a new *goal state* parameter; we use this to deal with the assignment of view parameters in goal views, and describe it in detail later on.

By treating the states as part of the local context of the backend, we can hopefully use the same techniques we saw in the $_{lo}$Starling soundness proof. As we rearrange both the views and backend conditions to push through the soundness argument, we must show that the sets of states permitted by the original and lowered schemes equate. We must also show that any pattern matches over normal $_{g}$Starling correspond to matches over lowered $_{g}$Starling.

Goal states. In $_{lo}$Starling's meta-theory, the solver quantification over local states becomes a pair (l, l') inside the backend's LCtx. For $_{g}$Starling, we can keep this model, but need a third piece of local context: a *goal state*, modelling solver quantifications over goal parameters.

> **Definition 6.31.** A *goal state* $l_g : \mathbb{N} \to \mathsf{E}_{\mathsf{Val}}(\mathsf{Val})(\emptyset)$ maps goal variables to expressible values.

When we lower a view built using $\mathsf{instGoal}$, we substitute $l_g(n)$ for each G_n in the view.

Erasing variables in arguments. Erasure depends on the distribution, through functorial and monadic operations, of argument-erasing functions. Since different situations contain different combinations of local-pre, local-post, and goal variables, we define a family of functions $\mathsf{eraseArg}X$, where X is some combination of P (pre), Q (post), and G (goal). Here, we formally define $\mathsf{eraseArg}PQG$; the other definitions are trivial alterations.

Definition 6.32. The function eraseArgPQG uses local and goal states, bundled together as a local context triple, to lower a view argument into an expressible value:

$$\begin{aligned}&\mathsf{eraseArgPQG}\ :\\ &\quad(\Sigma_{lo}\to \mathsf{E}_{\mathsf{Val}}(\mathsf{Val})(\emptyset))\to(\Sigma_{lo}\to \mathsf{E}_{\mathsf{Val}}(\mathsf{Val})(\emptyset))\to(\mathbb{N}\to \mathsf{E}_{\mathsf{Val}}(\mathsf{Val})(\emptyset))\\ &\qquad\qquad\to(\Sigma_{lo}^{\mathsf{M}}\uplus\mathbb{N})\to \mathsf{E}_{\mathsf{Val}}(\mathsf{Val})(\emptyset)\end{aligned}$$

$$\mathsf{eraseArgPQG}(l)(l')(l_g)(v) \stackrel{\text{def}}{=} l(v) \qquad \text{(local pre-state)}$$

$$\mathsf{eraseArgPQG}(l)(l')(l_g)(v') \stackrel{\text{def}}{=} l'(v') \qquad \text{(local post-state)}$$

$$\mathsf{eraseArgPQG}(l)(l')(l_g)(n) \stackrel{\text{def}}{=} l_g(n) \qquad \text{(goal-state)}$$

We also assume that, for each eraseArgX, we have a lifting eraseArgXS that can map over shared variables, ignoring them.

To apply eraseArgX to a value expression e, we use $e \gg\!= \mathsf{eraseArgX}$; by chaining this with the instance of fmap over GAPs, we can apply it over whole atoms.

Suppose we apply this erasure to the example w' from earlier. In this example, t, c', and t' are local variables that must be erased:

$$\begin{array}{llllllllll}(\odot & (@(\mathit{tick} & t &) & (@(\to & (\neq & c' & t' &) & \mathit{lock})) &)\\ & & \downarrow & & & & & \downarrow & & & \\ (\odot & (@(\mathit{tick} & l(t) &) & (@(\to & (\neq & l'(c) & l'(t) &) & \mathit{lock})) &)\end{array}$$

Any goal variables would also have been erased; for example, G_0 would become $l_g(0)$.

We can interpret the lowered expression as a multiset of unguarded abstract predicates, according to Definitions 6.5 and 6.22. Doing so finally evaluates, and eliminates, the guards. For instance, if we suppose that $l(t) = 1$, $l'(c) = 2$, and $l'(t) = 3$, we can interpret our example as $\wr(\mathbf{tick}\ 1), (\mathbf{lock})\int$.

Shadowing. At the frontend level, goal views represent a quantification over all possible instantiations of a given template. This means that each goal view represents potentially infinitely many LVF-level defining views. As our soundness argument requires us to show that each axiom in $_{\mathsf{g}}$Starling is also inside the corresponding defining-views instance, we must show the reverse: *at least* one goal view covers each defining view. To show this, we show that wherever we have a view v that is an exact pattern match for a pattern t — and therefore corresponds structurally to a goal view —, we can decompose that view into said goal g^v and a goal state l_g^v. If we lower g^v with l_g^v, each original argument re-appears in its original position[18] — and so, l_g^v *shadows* v.

Definers. In lowered $_{\mathsf{g}}$Starling, definers operate on erased views, and ignore guards; in normal $_{\mathsf{g}}$Starling, they operate on the parametric forms, and respect guards. To show that

[18] Modulo possible permutation of the atoms inside the view.

the lowered form has the right relationship with the normal form, we must show that a meaningful mapping exists between the two definer forms. While formally doing so remains further work at this stage, we outline some of the subtleties here.

First, we must show that erasure distributes across pattern matches. Ignoring guards for now, this shows that, if the input to a lowered definer matches the input to a normal definer, then so do their respective outputs.

Lemma 6.26. For all views v, and functions f over atoms, if f preserves tags, then the list of pattern matches on $\mathsf{fmap}(\mathsf{f})(v)$ is equal to mapping $\mathsf{fmap}(\mathsf{f})$ over all matches on v.

Intuitively, this property holds because the matcher *only* considers tags in decisions.

Second, we must deal with the absence of guards in the lowered version. The only views with guards other than true appear in the weakest-precondition position. Finally, to show that ${}_{\mathsf{g}}$Starling's approach yields subsets of the defining-views rule, we would need to adapt guarded syntactic definers into the semantic definer format given in Definition 3.10.

6.7 Iterated views and other extensions

While ${}_{\mathsf{g}}$Starling is an expressive frontend that can handle proofs of real-world algorithms such as the ticket lock, it still has restrictions that complicate proof of other classes of fine-grained concurrent program. In this section, we sketch extensions to ${}_{\mathsf{g}}$Starling that relax these restrictions while maintaining the overall structure and soundness argument we have built. These extensions are available in $\text{Starling}_{\text{tool}}$, and outlined in previous publications [2], though we leave formal and mechanised soundness arguments to future work.

Iterated views

One of the main restrictions is the requirement that patterns contain a finite, bounded, known quantity of each tag; this prevents us from writing proofs for many algorithms that involve the transfer of an unbounded number of resources. *Iterated views* are an extension to the guarded syntactic definers from § 6.5 that allows tags in patterns to match an indefinite number of atoms, and definition predicates to use the number of matches as a parameter.

Iterated tags. The first change we make to ${}_{\mathsf{g}}$Starling is to add a second class of tag: the set Tag^* of *iterated tags*. Iterated tags differ from normal tags in several ways:

- patterns either contain zero or more non-iterated tags, or exactly one iterated tag;
- iterated tags contain an implicit extra parameter, which binds during pattern-matching to the number of times the iterated tag appears in the match;
- definitions over iterated tags have additional restrictions, which we discuss below.

Adapting rmpart **for guarded-iterated atoms.** When extending atom subtraction to guarded atoms, we needed to deal with the issue of not knowing whether or not an atom was present

(specifically, the truth value of its guard) in advance of sending verification conditions to the backend solver. Guarded-iterated atoms complicate subtraction further: we now have two axes on which the presence and quantity of atoms can vary. As a result, subtraction for guarded-iterated atoms needs significant adaptation from the guarded case.

Let us assume that the subtrahend always has a known iterator k. This is a reasonable assumption for our frontends, as the only atoms in this position are those from a proof-outline postcondition, and we can syntactically restrict these to have known iterators.

When the other atom also has a known iterator, we can use the same subtraction procedure as before. If not, we must statically model removal from a quantity n known only at solve-time — which may be smaller than the amount we intend to remove! As before, we do so by moving decisions into the guards. In the case that $k = 1$, we have:

$$
\begin{aligned}
&(\backslash\ (@(\rightarrow g_m\ a^n))\ (@(\rightarrow g_s\ b))) = \\
&\qquad (\backslash\ (\bullet\ (@(\rightarrow(\textbf{and}\ g_m\ g_s\ (=\ a\ b)\ (>\ n\ 1))\ a^{(-\ n\ 1)}))\\
&\qquad\qquad (@(\rightarrow(\textbf{and}\ g_m\ (\textbf{not}\ (\textbf{and}\ g_s\ (=\ a\ b))))\ a^n)))\\
&\qquad\qquad (@(\rightarrow(\textbf{and}\ g_s\ (\textbf{not}\ (\textbf{and}\ g_m\ (=\ a\ b)\ (>\ n\ 0))))\ a)))
\end{aligned}
$$

We can reduce other cases to this by rewriting subtractions of iterator k into k subtractions of iterator 1. If we assume a statically known iterator, this rewrite is bounded, if inefficient.

Adapting pattern matching. Recall that the usual scheme for reifying a view v is to collect the definitions of all defining $u \sqsubseteq v$; in guarded syntactic definers, this translates to collecting all definitions with patterns that match v. This works, and is automation-friendly, as each pattern maps to defining views in a bounded manner. We can construct a set of goal views that shadow all possible defining views, and the number of pattern matches in a view that lead to unique defining views is bounded and finite.

Unrestricted iterated view patterns do not have these properties. Consider, for example, the definition $[\mathit{bad}^*] \to (=\ x\ P_*)$. If x is a shared variable, this pattern expands to a different observation about the shared state each time we add another copy of `bad` into the view. To expand this pattern into a goal view, we must consider all possible n, and so the goal's iterator becomes a universally quantified variable v in the same manner as other goal arguments. In turn, this causes the weakest precondition of any term over the goal to contain v copies of it, where the exact value of v is opaque until we reach the backend solver.

Suppose we consider all possible ways in which bad^* matches v copies of an atom with tag `bad`. The most obvious match is against all v copies, giving us the instantiated definition $(=\ x\ v)$. However, we can also match against $v - 1$ copies for $(=\ x\ (-\ v\ 1))$, $v - 2$ copies for $(=\ x\ (-\ v\ 2))$, and so on. In fact, all views satisfy the pattern when $n = 0$, causing $(=\ x\ 0)$ to become a system invariant! As we do not know the final value of v when expanding the matches, the expansion of each of the v matches into predicates is undecidable, and impossible to automate. To resolve this problem, we restrict definitions such that:

- we need never consider matches where the iterator is 0 (so, above, we never match for $n = 0$); we call this restriction *base downclosure*;

- we need only consider the maximal match for iterated atoms (so, above, we only match for $n = \nu$); we call this restriction *inductive downclosure*.

Definition 6.33. An iterated definition $\mathtt{P}(n)$ satisfies *base downclosure* when the definition of the empty pattern implies $\mathtt{P}(0)$.

The definition of ***bad*** only satisfies base downclosure if $(= \ x \ 0)$ is invariant — in which case the presence of any ***bad*** atoms leads to an inconsistent state.

Definition 6.34. An iterated definition $\mathtt{P}(n)$ satisfies *inductive downclosure* when:

$$\forall n : \mathbb{N}, \mathtt{P}(n+1) \implies \mathtt{P}(n)$$

Our `bad` definition fails this requirement outright: $(= \ x \ (+ \ n \ 1))$ does not imply $(= \ x \ n)$.

Matching guarded-iterated abstract predicates. Pattern-matching in the presence of guarded-iterated abstract predicates is complicated, as iterated view patterns can match any number of atoms in the view being reified. While downclosure means that, in practice, we need only consider the largest possible match, subtleties remain.

When we have two (or more) atoms that match a pattern, but have different guards, we must consider all possible combinations of those atoms, conjoining the guards and summing the iterators. With downclosure, this ensures that we get the right final iterator regardless of the truth value of the atoms' guards. For example, when matching $[A^*]$ against $(\odot \ (@(\to \mathtt{B}_1 \ A^i)) \ (@(\to \mathtt{B}_2 \ A^j)))$, we first match against the two instances of A individually, then also match against the combination to get $(@(\to (\textbf{and} \ \mathtt{B}_1 \ \mathtt{B}_2) \ A^{i+j}))$.

Iterated patterns can also match combinations of atoms when parameter equality can make them equal. When we match $[A^*]$ against $(\odot \ (@(\to \mathtt{B}_1 \ A^i \ y)) \ (@(\to \mathtt{B}_2 \ A^j \ z)))$, we both match against each side of the join individually *and* the combination — adding an equality guard over x and y —, which gives us $(@(\to (\textbf{and} \ \mathtt{B}_1 \ \mathtt{B}_2 \ (= \ y \ z)) \ A^{i+j} \ y))$.

Local assertions

Our LVF-based set-up has no first-class support for assertions on local state. To reason about the local state (even if reasoning *only* about local state), we must wrap the assertion in a view. This is unwieldy and wasteful: we must consider such wrapped assertions as being susceptible to interference from other threads.

Fully introducing first-class local assertions into ${}_g$Starling would require invasive changes to the LVF, and these would likely cause it to diverge heavily from its *Views* base. Instead, let us consider a lightweight encoding that simplifies the task of writing proofs over local assertions while not (yet) addressing the efficiency issues.

Local-lift atoms. To encode local assertions into ${}_g$Starling-style logics without changing the underlying views algebra, we can introduce a single-parameter atom ***local*** with an attached definition that resolves directly to that parameter (for example, $([\textit{local}], \mathtt{P}_0)$). This has the expected semantics so long as there are no other definitions over ***local***.

This encoding works only if we can express proposition expressions in the value expression language. We can do so in $\mathrm{C}_{\mathrm{view}}$(§ 7.1), and so $\mathrm{Starling}_{\mathrm{tool}}$ supports local lifts.

Zero and false views

Our views algebras are (primarily) monoids, with a unit view representing baseline knowledge about the shared state. Various reasoning systems, such as the UTP and some separation logic models, contain a *zero* assertion; in Starling, a zero view would represent an unsatisfiable observation of the shared state. Such a view would be useful for asserting unreachable states, error conditions, and other inconsistencies.

Algebraic zero. Suppose we extend our views algebras with a zero in the semigroup sense: an element 0 such that, for all a, $\bullet\, 0 \equiv 0 \bullet a \equiv 0$. To establish 0 as the view holding the most restrictions on shared state, analogous to ε's role as the least-restrictive valid view, we can also require that $a \sqsubseteq 0$. This set-up, while compatible with our existing algebra classes, leads to a meaningless algebra in which 0 (and, by ordering, every view) is equivalent to ε:

$$\varepsilon \equiv \varepsilon \qquad \text{(reflexivity)}$$
$$\rightarrow(\varepsilon \setminus 0) \equiv \varepsilon \qquad \text{(subtraction on } \varepsilon \text{ idempotent)}$$
$$\rightarrow\varepsilon \equiv (\varepsilon \bullet 0) \qquad \text{(adjoint)}$$
$$\rightarrow\varepsilon \equiv 0 \qquad (0)$$

This problem relates to that of division by zero in $\mathbb{N}$, both intuitively and directly through the views algebra in § 6.1. We could take the analogy further by making $\setminus$ partial (and $a \setminus 0$ undefined), but this would complicate our algebra classes, and we do not do so here.

False observations. To capture most of the advantages of a zero view without reworking our views algebras, we can encode failure as a local observation of false. We can then use the local-assertion encoding we introduced above. Strictly speaking, the existence of such a view signifies that its holding thread has entered an impossible local state; the view also does not obey either of the algebraic properties we suggested above. In practice, the encoding suffices for several use cases of an zero atom: for example, marking certain control-flow paths in a proof as theoretically unreachable, and then checking that this is indeed the case.

6.8 Summary

This chapter introduced ${}_{\mathrm{g}}$Starling, a Starling frontend based on guarded abstract predicates. Like ${}_{\mathrm{lo}}$Starling, ${}_{\mathrm{g}}$Starling supports local-state parametrisation of shared-state observations. Unlike ${}_{\mathrm{lo}}$Starling, it does so in a more structured manner that preserves the bounded enumerability of defining views. This, along with other changes such as view expressions and a pattern-matching view definer, makes it more suitable for practical use.

Work on ${}_{\mathrm{g}}$Starling is ongoing. A soundness proof, as well as a formalised version of the extensions the preceding section discussed, remain as further work (see § 9.2).

Chapter 7

Automating Proofs with $\text{Starling}_{\text{tool}}$

So far, we've used Starling as a framework for building *theoretically* automatable program logics. This chapter shows the *practical* suitability of Starling for automated reasoning by building a tool, $\text{Starling}_{\text{tool}}$. This tool uses a variant of ${}_{\text{g}}\text{Starling}$ as its underlying theory and accepts proof outlines in a C-like language we call C_{view}.

At time of writing, source code for $\text{Starling}_{\text{tool}}$, as well as examples (including those in § 8.1), is available at `https://github.com/MattWindsor91/starling-tool`.

7.1 The C_{view} language

C_{view} is $\text{Starling}_{\text{tool}}$'s input language. It takes syntactic cues from C and derivatives, while remaining compatible with the LVF language. We discuss C_{view} as it appears in the tool; see § 9.2 for future development ideas, and Appendix A.6 for a BNF sketch of the C_{view} grammar.

Semantics. C_{view} has a semantics in terms of compilation to the *Views* language with local state. This means that we can apply Starling on the compiled program and quickly get correctness results over the original program, so long as we trust the compilation process. A standalone operational semantics remains future work.

> **Definition 7.1.** The notation $[\![-]\!]_{\text{c}}$ denotes compilation from C_{view} to the *Views* language.

Running example. This section uses a C_{view} version of the ticket lock (Algorithm 1) as an example. This example does not exercise some parts of C_{view}, such as iterated views; § 8.1 gives further examples (including a version of *Peterson*) that cover more of C_{view}.

Types

As an extension to ${}_{\text{g}}\text{Starling}$, C_{view} has a type system — a near-subset of that of C99, but with changes to better support interfacing with external solvers. This lets C_{view} tools catch various mistakes in algorithm specification that the logic would otherwise map to $\bot$ or false. It also helps tools interface with solvers, like Z3, that have many-sorted underlying theories.

Primitive types. C_{view} supports the C99 **`int`** and **`bool`** types, with one difference: we define **`int`** as arbitrary-precision, with range $[-\infty, \infty]$, rather than being fixed to an architecture-defined size. This set-up mirrors the definition of integers in solvers such as Z3.

Typedefs. C_{view} lets users define new types from existing primitive types[1] with **`typedef`**. Unlike C, C_{view} treats the new types as distinct *subtypes* of the old type. It uses rules similar to the treatment of *defined types* and constants in Go [65]: unifying two different subtypes fails (regardless of their original type), as does unifying a subtype with its original type; but subtypes *can* accept constants and expressions with no specific type.

Arrays. C_{view} also has array types. While array indices are always **`int`**s, array elements can be of any type, including nested array types. Arrays can optionally contain a size bound.

Variables

Non-goal variables must be declared before use. The syntax follows that of C, except that we explicitly mark variable declarations as either **`thread`** (thread-local) or **`shared`**.

The ticket lock's variables translate to C_{view} as follows:

```
shared int s,  // ticket currently being served
           n;  // next ticket
thread int t,  // local storage for ticket
           c;  // local storage for current view of s
```

Thread variable declarations may also appear in methods as statements (with the syntax above) or parameters (less the **`thread`** prefix). In both cases, the semantics is the same as if the variables were declared at the top-level (with appropriate scoping and freshening).

Expressions

C_{view}'s expression language is almost exactly that of C; see Appendix A.6 for differences.

For direct backend-theory access, C_{view} expressions support $_{g}$Starling's *symbols*. C_{view} symbols are variable-interpolated pieces of uninterpreted backend syntax, and look like this:

```
%{(>= [| n |] [| s |])}
```

The above example encodes the invariant from Figure 6.1 by escaping from $\text{Starling}_{\text{tool}}$ syntax into the SMT-LIB language understood by Z3 [2]

Assertions

Shared-state assertions in C_{view} proofs take the form of view expressions over guarded iterated abstract predicates. The C_{view} view expression syntax differs from that used in the previous chapters, more closely resembling traditional separation logic: **`emp`** represents 1, $*$ represents $\bullet$, and view atoms have a syntax similar to C function calls.

[1] Future work may allow transitive **`typedef`**s and **`typedef`**s of arrays.

[2] $\text{Starling}_{\text{tool}}$ does not yet support this particular use of symbols, but we consider it to be future work.

Prototyping atoms. We must prototype C_{view} atoms before use. While $_g$Starling prototypes only map tags to parameter counts, C_{view} prototypes also carry parameter types. Prototypes have similar syntax to those of C functions, but allow empty parameter lists `()` to be dropped; for example, we can prototype the ticket lock's atoms as `view Lock, Tick(int t);`.

We can translate each C_{view} prototype to its guarded-iterated equivalent by counting the number of parameters; extracting the tag (checking for the `iter` keyword, which denotes an iterated tag); and building the appropriate mapping.

Defining atoms. To add a definition to C_{view}'s equivalent of $_g$Starling's guarded syntactic definer, we use a `constraint` statement. Instead of using positional parameter references such as P_0, we bind each parameter to a unique identifier in the pattern, then place those identifiers into scope in the proposition expression. We can also define patterns as `?`—this relates to inference, and we discuss it later on. The ticket lock's definer translates as follows:

```
constraint emp                 -> n >= s;
constraint Tick(t)             -> n >  t;
constraint Lock                -> n != s;
constraint Tick(a) * Tick(b)   -> a != b;
constraint Lock    * Tick(t)   -> s != t;
constraint Lock    * Lock      -> false;
```

To map normal (non-`?`) definitions to guarded syntactic definers, we substitute a positional parameter reference for each named parameter in the definition; then, we extract the list of tags from the pattern, replacing `emp` with $[]$.

$$[\![\texttt{constraint emp ->}P]\!]_c \stackrel{\text{def}}{=} ([], [\![P]\!]_c)$$

$$[\![\texttt{constraint } a(x_0,\ldots,x_i) * b(x_i+1,\ldots,x_j) * \ldots * z(x_k,\ldots,x_n) \texttt{ ->}P(x_0,\ldots,x_n)]\!]_c \stackrel{\text{def}}{=} ([a,\ldots,z], [\![P(P_0,\ldots,P_n)]\!]_c)$$

Local lifting. In § 6.7, we discussed how to encode local observations in $_g$Starling by lifting them into a local atom. C_{view} supports embedding local predicates e into assertions using the syntax `local{`e`}`. To be able to lower this syntax later on, we insert the following code:

```
view _local(bool e); // where '_local' is some fresh identifier
constraint _local(e) -> e;
```

Local lifting, in turn, gives us a way to embed the false atoms we discussed in § 6.7.

Using atoms. As before, we use these atoms in view expressions. Compared to $_g$Starling, C_{view} has a richer syntax for view expressions, though all extensions are in the form of syntactic sugar that we lower down to $_g$Starling-compatible constructs.

We translate view expressions (except `?`, which we discuss later) as follows:

$$
\begin{aligned}
[\![v]\!]_c &\stackrel{\text{def}}{=} \mathsf{lower}(\mathbf{true})(v) \\
\mathsf{lower}(g)(\mathbf{emp}) &\stackrel{\text{def}}{=} \mathit{1} \\
\mathsf{lower}(g)(x * y) &\stackrel{\text{def}}{=} (\bullet\ \mathsf{lower}(g)(x)\ \ \mathsf{lower}(g)(y)) \\
\mathsf{lower}(g)(\mathbf{if}\ e\ \{\ v\ \}) &\stackrel{\text{def}}{=} \mathsf{lower}(g)(\mathbf{if}\ e\ \{\ v\ \}\ \mathbf{else}\ \{\ \mathbf{emp}\ \}) \\
\mathsf{lower}(g)(\mathbf{if}\ e\ \{\ x\ \}\ \mathbf{else}\ \{\ y\ \}) &\stackrel{\text{def}}{=} (\bullet\ \mathsf{lower}((\mathbf{and}\ g\ [\![e]\!]_c))(x) \\
&\qquad\qquad \mathsf{lower}((\mathbf{and}\ g\ (\mathbf{not}\ [\![e]\!]_c)))(y)) \\
\mathsf{lower}(g)(\mathbf{false}) &\stackrel{\text{def}}{=} \mathsf{lower}(g)(\mathtt{local}\{\ \mathbf{false}\ \}) \\
\mathsf{lower}(g)(\mathtt{local}\{\ e\ \}) &\stackrel{\text{def}}{=} \mathsf{lower}(g)(\mathtt{\_local}(e)) \\
\mathsf{lower}(g)(f(x_1, x_2, \ldots, x_n)) &\stackrel{\text{def}}{=} (\mathbf{@}(\mathbf{\rightarrow} g\ f\ [\![x_1]\!]_c\ [\![x_2]\!]_c\ \ldots\ [\![x_n]\!]_c))
\end{aligned}
$$

Action language

We now discuss C_{view}'s primitive actions. These actions represent the high-level set A; for each, we give an informal definition of $[\![-]\!]$ in terms of structured predicates (Definition 6.19).

Atomic and non-atomic actions. Anything between a pair of triangles—`<|` and `|>`—forms a single atomic action. Inside such actions, both **`shared`** and **`thread`** variables are in scope. C_{view} permits a limited set of control-flow constructs (see below) inside atomic actions, which translate into logical operators in the two-state proposition encoding.

Anything outside of triangles is non-atomic. Primitive commands encode into LVF actions, but the atomicity rules from § 5.7 apply where possible. Control flows encode into LVF control flows, with some desugaring. In non-local actions, only **`thread`** variables are in scope; non-atomic actions on **`shared`** variables would break the correspondence to the LVF.

Assignment. We use a subset of C's assignment syntax. Expressions on the left of assignments (*lvalues*) must be variables or array subscripts thereof. Right sides of assignments (*rvalues*) may be expressions, or lvalues followed by `++` (fetch-and-increment) or `--` (fetch-and-decrement). The semantics, using rframe (Definition 6.18) to represent framing, is:

$$
\begin{aligned}
[\![\mathtt{l = r}]\!] &\stackrel{\text{def}}{=} \mathsf{rframe}((\mathbf{and}\ (\mathtt{=}\ \mathtt{l'}\ \mathtt{r})\ (\mathtt{=}\ \mathtt{r'}\ \mathtt{r}))) \\
[\![\mathtt{l = r{+}{+}}]\!] &\stackrel{\text{def}}{=} \mathsf{rframe}((\mathbf{and}\ (\mathtt{=}\ \mathtt{l'}\ \mathtt{r})\ (\mathtt{=}\ \mathtt{r'}\ (\mathtt{+}\ \mathtt{r}\ 1)))) \\
[\![\mathtt{l = r{-}{-}}]\!] &\stackrel{\text{def}}{=} \mathsf{rframe}((\mathbf{and}\ (\mathtt{=}\ \mathtt{l'}\ \mathtt{r})\ (\mathtt{=}\ \mathtt{r'}\ (\mathtt{-}\ \mathtt{r}\ 1))))
\end{aligned}
$$

Increment and decrement. We can also use the above modifiers on an lvalue directly. The semantics is that of the fetch-and-modify actions, but without assignment:

$$
v{+}{+} \stackrel{\text{def}}{=} \mathsf{rframe}((\mathtt{=}\ v'\ (\mathtt{+}\ v\ 1))) \quad \text{etc.}
$$

Assume and assert. Some proof scripts must make *assumptions* that a condition holds before proceeding, or *assertions* that abort the program if a condition *does not* hold. C_{view}'s **assume** and **assert** commands permit this. Encoding assumption is straightforward:

$$[\![\textbf{assume}\ \ P]\!] \stackrel{\text{def}}{=} \quad \mathsf{rframe}((\Rightarrow\ (\textbf{not}\ P)\ \emptyset_{\mathsf{RI}}))$$

We encode assertion in two steps. First, we insert the following variable and invariant:

```
shared bool _ok; constraint emp -> _ok; // && previous invariant
```

Here, _ok is similar to the UTP [61, Def. 3.0.1] ok variable; it witnesses that the program has not crashed. With these additions, we can define assertion:

$$[\![\textbf{assert}\ P]\!] \stackrel{\text{def}}{=} \quad \mathsf{rframe}((\textbf{and}\ (\Rightarrow\ (\textbf{not}\ P)\ (=\ \_ok'\ \mathsf{false}))\ (\Rightarrow\ P\ (=\ \_ok'\ \_ok))))$$

As this encoding modifies shared variables, we cannot permit it outside atomic actions.

Compare-and-swap. C_{view} also contains primitives for compare-and-swap. The expressions correspond to the destination, the variable containing the expected value, and the value-to-set respectively. The expected-value variable always receives the original destination value; this resembles the semantics of the x86 `CMPXCHG` instruction [66].

Errors. We may want to prove that a program branch is unreachable. While we could use an opening `{| false |}` assertion, or `<| assert false; |>`, C_{view} provides an explicit `<| error; |>` action. This has the same semantics as assert-false, but is clearer in intent.

Non-determinism. C_{view} has two constructs for statement-level non-determinism: **havoc** v, which non-deterministically sets v to any value in its domain; and ... ('miracle'), which stands for any set of statements that obey its proof obligation.

The **havoc** command is semantically equivalent to $\mathsf{id}_{\mathsf{RI}}$ for all variables except v and v', where it is equivalent to true. We allow **havoc** v wherever v is in scope: as a result, havoc on **thread** variables can appear in non-atomic actions. This effect is entirely encodable in structured predicates, and so we can class each **havoc** as a member of A.

The semantics of miracle is harder to reconcile with *Views*. Informally, $\{P\}\ \ldots\ \{Q\}$ represents a gap into which we can insert some program provided that it obeys P, Q, and the usual non-interference properties[3]. We can see C_{view} outlines containing miracles as incomplete proofs to be filled in later, rather than valid proofs in their own right.

Still, we can simulate miracle in the *Views* language if we have access to P and Q while doing so. Let $+S$ be the iteration of the choice operator $+$ over all programs in S. Then:

$$\ldots(P, Q) \approx +\{\, C \mid \vdash \{P\}\ C\ \{Q\}\}$$

Control flows

We now outline the control flow constructs in C_{view}, and how they map to the LVF language.

[3] As we are investigating safety properties only, divergence (**assume false**) is one such program, so the validity of a miracle *does not* guarantee that the subsequent outline is reachable.

Sequential composition. Unlike *Views*, sequential composition is not an explicit operator in C_{view}. Instead, like C, we implicitly sequentially-compose statements, using semicolons to terminate statements with ambiguous endings.

Choice. Instead of the *Views* $+$ operator, C_{view} has C-style **if** statements. The syntax is the same as that of Go: a modified version of C's syntax, permitting only brace-delimited blocks as branches, and allowing the brackets around conditions to be elided.

The special $*$ condition represents a straight non-deterministic choice.

$$\begin{aligned} [\![\texttt{if * } \{X\}]\!]_c &\stackrel{\text{def}}{=} [\![\texttt{if * } \{X\} \texttt{ else } \{\}]\!]_c \\ [\![\texttt{if * } \{X\} \texttt{ else } \{Y\}]\!]_c &\stackrel{\text{def}}{=} ([\![X]\!]_c) + ([\![Y]\!]_c) \end{aligned}$$

Statement-level choices with condition expressions reduce to *Views* non-deterministic choices guarded by assumptions. This follows the observation we made in Figure 2.2.

$$\begin{aligned} [\![\texttt{if } P \{X\}]\!]_c &\stackrel{\text{def}}{=} [\![\texttt{if } P \{X\} \texttt{ else } \{\}]\!]_c \\ [\![\texttt{if } P \{X\} \texttt{ else } \{Y\}]\!]_c &\stackrel{\text{def}}{=} [\![\texttt{if * } \{ \texttt{ assume } P\texttt{; } X\} \texttt{ else } \{ \texttt{ assume } \neg P\texttt{; } Y\}]\!]_c \end{aligned}$$

We allow **if** inside atomic actions, with the following two-state predicate semantics:

$$\begin{aligned} [\![\texttt{if } P \{X\}]\!] &\stackrel{\text{def}}{=} [\![\texttt{if } P \{X\} \texttt{ else } \{ \texttt{ id; } \}]\!] \\ [\![\texttt{if } P \{X\} \texttt{ else } \{Y\}]\!] &\stackrel{\text{def}}{=} (\texttt{and } (\Rightarrow P\ [\![X]\!])\ (\Rightarrow (\texttt{not } P)\ [\![Y]\!])) \\ [\![\texttt{if * } \{X\}]\!] &\stackrel{\text{def}}{=} [\![\texttt{if * } \{X\} \texttt{ else } \{ \texttt{ id; } \}]\!] \\ [\![\texttt{if * } \{X\} \texttt{ else } \{Y\}]\!] &\stackrel{\text{def}}{=} (\texttt{or } [\![X]\!]\ [\![Y]\!]) \end{aligned}$$

Iteration. C_{view} has two C-style loop constructs: **while**, which conditionally executes a block zero or more times, and **do while**, which guarantees at least one execution. Like **if**, loop bodies must be braced blocks; unlike **if**, atomic actions cannot contain loops.

Semantically, both loops reduce to the *Views* iteration, adding **assume** guards to enforce the truth or falsehood of the loop condition.

$$\begin{aligned} [\![\texttt{do } \{B\} \texttt{ while } P\texttt{;}]\!]_c &\stackrel{\text{def}}{=} [\![B\texttt{; while } P \{B\}]\!]_c \\ [\![\texttt{while } P \{B\}]\!]_c &\stackrel{\text{def}}{=} ([\![\texttt{assume } P\texttt{; } B]\!]_c)^*; [\![\texttt{assume } \neg P]\!]_c \end{aligned}$$

Methods. We organise blocks of C_{view} code, at the top level, into one or more **method**s. These resemble C functions, but can (presently) neither return a value nor return early. We can consider methods, or sequential invocations thereof, as being the programs that each thread is running in an instance of the LVF multi-thread set-up (Definition 5.13).

Inference

We allow a ? operator to replace certain C_{view} elements. This lets us use the inference abilities of a suitable backend to find stable views and definitions to complete the proof.

In constraints. When ? appears in a **constraint** clause, it asks the backend to infer a stable definition. While its semantics depends on the backend in question, typically we lower ? to a symbol invoking an uninterpreted function over the shared variables and pattern parameters. For example, suppose we wanted to infer the following ticket lock constraint:

```
constraint Tick(ta) * Tick(tb) -> ?;
```

This constraint might expand to a symbol similar to the following:

```
constraint Tick(ta) * Tick(tb) ->
  %{infer_Tick_Tick([| s |], [| n |], [| ta |], [| tb |])};
```

Constraint search. As a convenience, C_{view} has a top-level directive for constraint search. A **search** n**;** directive expands to a series of **constraint** X `->` **?;** directives, one for each pattern X of size $0 \leqslant |X| \leqslant n$ with no existing definition.

In assertions. We can infer whole view assertions by using the `{| ? |}` form. This is syntactic sugar: to reduce it, we expand it to a fresh atom, parametrised by all **thread** variables in scope. While we synthesise a prototype for the new atom, we do not generate any constraints for it; we instead rely on the existence of a **search** directive.

Building whole proof outlines

We can now construct C_{view} proof outlines. For convenience, and to better support C-style syntax, Starling$_{\text{tool}}$ does not expect assertions surrounding every statement and control-flow construct as in Figure 3.1. Instead, we require one assertion at each block start, block end, and sequence point between statements. When a required view assertion is missing, the backend supports inference, and we have a **search** directive, we insert an implicit `{| ? |}`.

To demonstrate, Listing 7.1 gives a valid proof outline for the ticket lock. This proof focuses on mutual exclusion, and assumes that the lock is statically allocated into shared variables[4]. In § 8.1, we discuss how Starling$_{\text{tool}}$ verifies the proof.

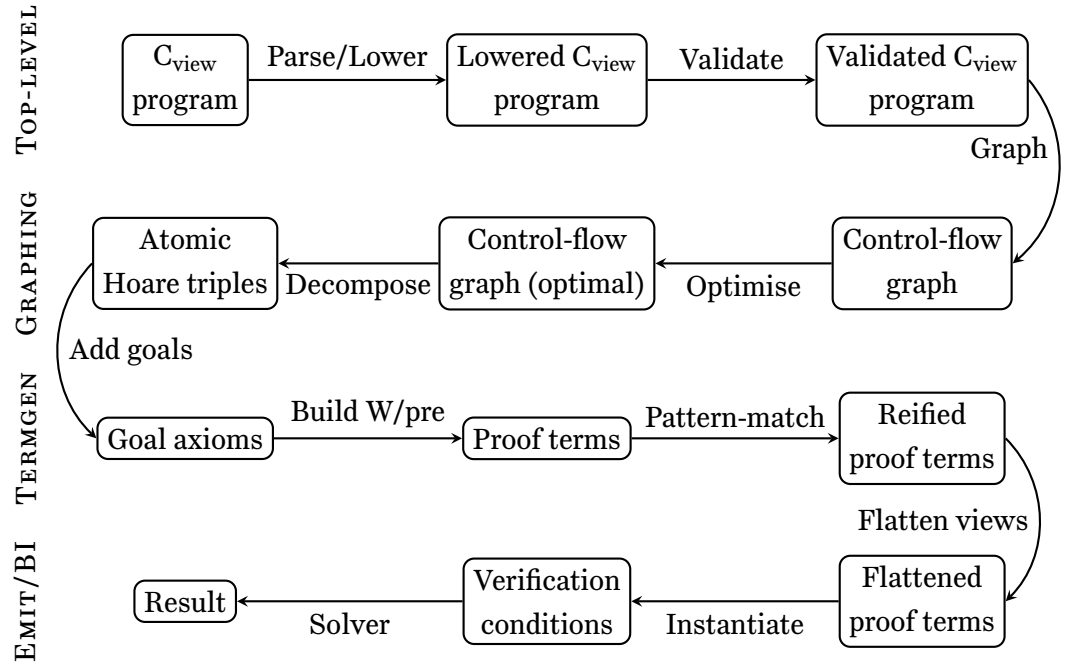
7.2 The Starling$_{\text{tool}}$ frontend

We now discuss how Starling$_{\text{tool}}$ converts C_{view} outlines into verification conditions. We can view Starling$_{\text{tool}}$ as a pipeline, roughly organised into four main stages—the C_{view} top-level; graph analysis; term generation; and the emitter and backend interface:

[4] We give a heap-allocated version of the ticket lock proof in Starling$_{\text{tool}}$'s examples directory.

Listing 7.1: C_{view} proof of mutual exclusion for the Mellor-Crummey/Scott ticket lock.

```
shared int s, n;
method lock() {
  {| emp |}
    thread int t, c; <| t = n++; |>
  {| Tick(t) |}
    do {
      {| Tick(t) |}
        <| c = s; |>
      {| if c == t { Lock } else { Tick(t) } |}
    } while c != t;
  {| Lock |}
}
method unlock() { {| Lock |} <| s++; |> {| emp |} }
view Tick(int t), Lock;
constraint emp                  -> n >= s;
constraint Tick(t)              -> n > t;
constraint Lock                 -> n != s;
constraint Tick(a)  *  Tick(b)  -> a != b;
constraint Lock     *  Tick(t)  -> s != t;
constraint Lock     *  Lock     -> false;
```



We now briefly discuss each main stage, and any major subtleties in its interpretation of the gStarling frontend. For each, we link to the relevant parts of Starling$_{\text{tool}}$'s F♯ source code.

C_{view} top-level

(Code: Starling.Lang.Parser) (Code: Starling.Lang.Collator) (Code: Starling.Lang.Modeller)

Starling$_{\text{tool}}$ first parses the outline, checks it for issues, and assigns types to expressions.

Parsing. The first step is to parse the outline. We also *collate* each top-level entity into separate sets: one for view atoms; one for variable declarations; and so on.

To simplify the rest of the tool, we lower as much syntactic sugar as possible at this point. For example, we synthesise the `_ok` variable and `_local` view from in the last section, and lower local and false view expressions to use the latter. We also insert `{| ? |}` wherever a view assertion is expected but missing; reduce C_{view} view assertions to a form closer to view expressions (with guarded atoms rather than selections); expand `{| ? |}` to fresh views; and expand **`search`** directives to constraints; amongst other lowerings.

Validation. At this stage, Starling$_{\text{tool}}$ also checks for issues in the outline, such as references to variables that are out of scope (including uses of **`shared`** variables in non-atomic code); references to missing view atoms; and iterated constraints on non-iterated atoms. Doing so at this level gives users assurance that their proofs are free from several classes of human error, and makes C_{view} act more like a practical programming language.

Type checking. Type checking is orthogonal to the rest of the verification process, so we perform it in advance. Starling$_{\text{tool}}$'s goal is to assign a full type record (including subtype information) to every expression and lvalue in the outline. The main subtlety here is that Starling$_{\text{tool}}$ respects the subtyping rules in § 7.1 during type checking.

Aside: microcode

(Code: Starling.Core.Command)

While performing lowering and validation, Starling$_{\text{tool}}$ reduces C_{view}'s command language to a smaller set of primitive commands; this simplifies the later translation to two-state predicates. By analogy with modern CPUs, we call this small language *microcode*. Microcode consists of the following commands:

`(← l v)`	(assignment)
`(↚? l)`	(havoc/nondeterministic assignment)
`(assume b)`	(assumption)
`(if b x y)`	(if-then-else)
`(% s)`	(symbol)
`(seq a b ... z)`	(sequence)

For example, `a = b++` lowers to `(seq (← a b) (← b (+ b 1)))`; `CAS(d, t, s)` becomes:

```
(if (= d t) (seq (← d s) (← t t)) (seq (← d d) (← t d)))
```

Graph analysis

(Code: Starling.Lang.Grapher)

Once we have a validated, typed, and simplified outline, we decompose it into a set of atomic Hoare triples. While we already have a scheme for decomposition, it targets the LVF

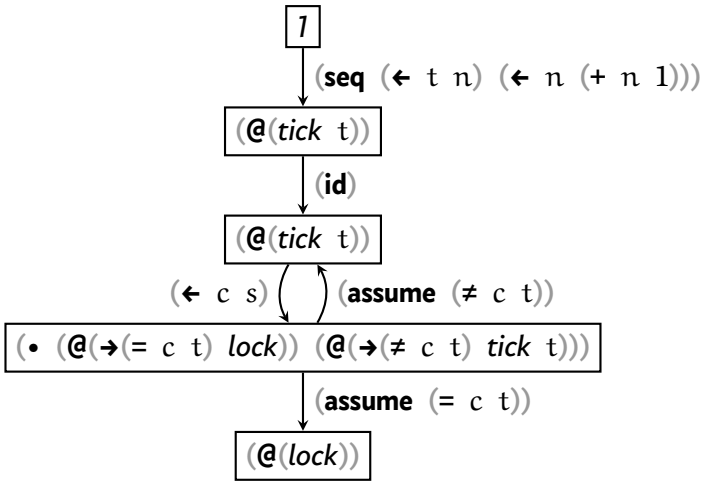


Figure 7.1: Unoptimised CFG of the ticket lock's `lock` method.

outline language; we would need to first convert the outline from C_{view} to LVF. Doing so would waste time and throw away program structure; such structure is useful for optimisations, and giving feedback to the user when proofs fail.

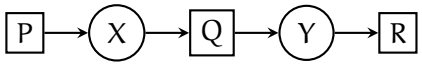
Instead, we translate method outlines into an intermediate *control-flow graph* (CFG) representation. Each graph node is a sequence point, holding a view assertion. Each edge is an LVF label (a primitive command or id-transition). We then optimise the CFG to reduce the number of generated verification conditions while entailing the original proof.

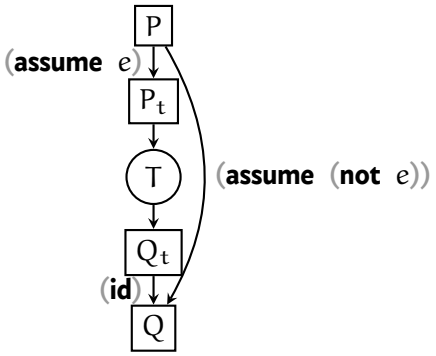
From outlines to CFGs. Starling$_{\text{tool}}$ translates outline methods into CFGs by recursively analysing each control-flow construct and instantiating corresponding graph fragments. Each CFG should decompose into the same triple set we get from lowering the outline to the LVF and applying the LVF decomposition; as such, each fragment closely resembles its construct's lowering in § 7.1. Table 7.1 gives translations for some of C_{view}'s control flows; we omit the non-deterministic forms, as they are broadly similar. As an example, Figure 7.1 shows the CFG corresponding to the ticket lock's `lock` method.

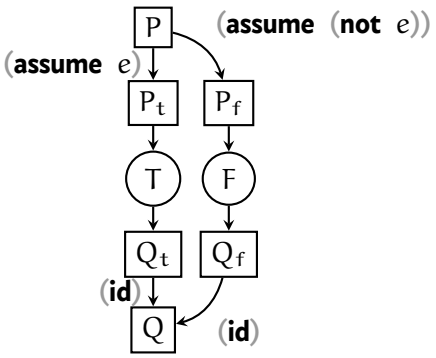
As the CFG transformation pieces together graph fragments without any further analysis, the resulting graphs have a large degree of redundancy. We can see this in the `lock` method: the second transition in the graph is $\langle(\texttt{@}(\textit{tick}\ \texttt{t}))\rangle\ \texttt{id}\ \langle(\texttt{@}(\textit{tick}\ \texttt{t}))\rangle$, which is trivial. Before we proceed, we need to limit the amount of redundancy in the CFG, and, therefore, the amount of redundant verification conditions we send to the backend solver.

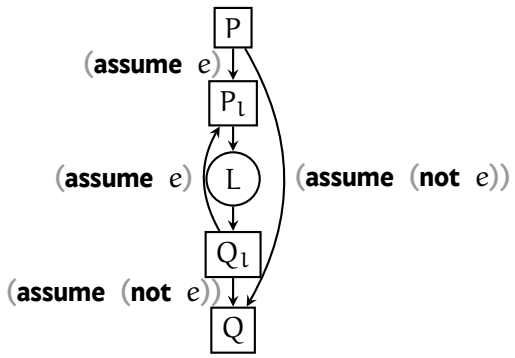
Graph optimisations. Once we have a CFG, we can perform optimising transformations on it. These transformations range in severity: some preserve the semantics of the original program; some sacrifice the program semantics but preserve the relationship between the output verification conditions and the program's proof; and some — disabled by default — weaken the proof by eliminating advisory views.

Figure 7.2 gives the optimised form of the `lock` CFG. While there are clear semantic differences, we see later that the final proofs are equivalent.

{| P |} X {| Q |} Y {| R |}	

{| P |}
if *e* {
{| P_t |} T {| Q_t |}
}
{| Q |}
	

{| P |}
if *e* {
{| P_t |} T {| Q_t |}
} **else** {
{| P_f |} F {| Q_f |}
}
{| Q |}
	

{| P |}
while *e* {
{| P_l |} L {| Q_l |}
}
{| Q |}
	

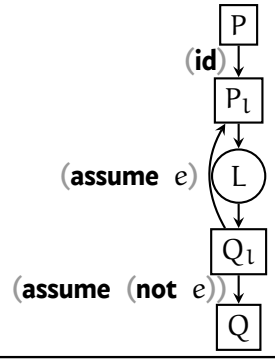
{| P |} **do** { {| P_l |} L {| Q_l |} } **while** *e*; {| Q |}	

Table 7.1: A selection of C_{view} control flows, and corresponding control-flow graphs.

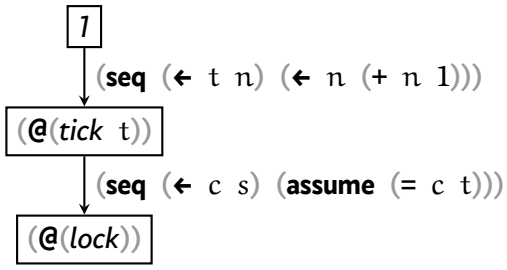


Figure 7.2: Optimised CFG of the ticket lock's `lock` method.

From CFGs to atomic Hoare triples. The structure of Starling$_{\text{tool}}$'s CFGs makes reduction to a set of atomic Hoare triples straightforward: we just take every edge c, its source node p, and its target node q, and return $\langle p\rangle\ c\ \langle q\rangle$. The CFG in Figure 7.2, for example, reduces to:

$$\langle 1\rangle\ (\textbf{seq}\ (\leftarrow\ t\ n)\ (\leftarrow\ n\ (+\ n\ 1)))\ \langle(\textbf{@}(\textit{tick}\ t))\rangle$$
$$\langle(\textbf{@}(\textit{tick}\ t))\rangle\ (\textbf{seq}\ (\leftarrow\ c\ s)\ (\textbf{assume}\ (=\ c\ t)))\ \langle(\textbf{@}(\textit{lock}))\rangle$$

By looking at the same reduction for the unoptimised CFG, we can check that the optimisations have not weakened the method's proof obligation. For `lock()`, these are:

$$\langle 1\rangle\ (\textbf{seq}\ (\leftarrow\ t\ n)\ (\leftarrow\ n\ (+\ n\ 1)))\ \langle(\textbf{@}(\textit{tick}\ t))\rangle$$
$$\langle(\textbf{@}(\textit{tick}\ t))\rangle\ (\textbf{id})\ \langle(\textbf{@}(\textit{tick}\ t))\rangle$$
$$\langle(\bullet\ (\textbf{@}(\rightarrow(=\ c\ t)\ \textit{lock}))\ (\textbf{@}(\rightarrow(\neq\ c\ t)\ \textit{tick}\ t)))\rangle\ (\textbf{assume}\ (\neq\ c\ t))\ \langle(\textbf{@}(\textit{tick}\ t))\rangle$$
$$\langle(\bullet\ (\textbf{@}(\rightarrow(=\ c\ t)\ \textit{lock}))\ (\textbf{@}(\rightarrow(\neq\ c\ t)\ \textit{tick}\ t)))\rangle\ (\textbf{assume}\ (=\ c\ t))\ \langle(\textbf{@}(\textit{lock}))\rangle$$
$$\langle(\textbf{@}(\textit{tick}\ t))\rangle\ (\leftarrow\ c\ s)\ \langle(\bullet\ (\textbf{@}(\rightarrow(=\ c\ t)\ \textit{lock}))\ (\textbf{@}(\rightarrow(\neq\ c\ t)\ \textit{tick}\ t)))\rangle$$

We see that: the first triple is the same as its optimised counterpart; the second triple is trivial; the next two triples are straightforward entailments once we process the assumptions; and the last triple maps either onto the second optimised triple or a relatively-straightforward entailment over the stability of $(\textbf{tick}\ t)$ — depending on how we case-split $(=\ c\ t)$.

Term generation

(Code: goalAdd in Starling.Core.Axiom) (Code: Starling.Core.TermGen)

Next, Starling$_{\text{tool}}$ generates high-level proof terms, analogous to those in Definition 6.30. We first take the Cartesian product of atomic triples and **`constraint`** patterns, instantiating each pattern into a view expression over goal variables (as in $_{\text{g}}$Starling). Internally, we call the resulting $(\langle p\rangle\ c\ \langle q\rangle, g)$ pairs *goal axioms*.

To turn goal axioms into $\langle w\rangle\ c\ \langle g\rangle$ terms, we build the weakest-precondition $w = p \bullet (g \setminus q)$. Instead of building w *then* list-normalising it as in $_{\text{g}}$Starling, we build a normalised version directly by appending p to a variant of rmpart over g and q. The C_{view} syntax guarantees that q has constant iterators, so we avoid non-termination problems.

Throughout this process, we try to keep as much information about the original goal and triple as possible. This helps us provide meaningful feedback when a term fails.

Emitting: microcode expansion

(Code: Starling.Semantics)

We now expand microcode into two-state predicates. To do so, we must handle three subtleties: arrays, **seq** clauses, and framing over variables not mentioned in the microcode.

Arrays. Array assignments only modify part of a variable. This causes problems when framing, so we lift them into whole-variable assignments. We lift normal assignments (of the form (← a[i] v)) to the list override (← a a[i ↦ v]), and lift multi-dimension array assignments recursively, from outer subscript inwards.

Starling$_{\text{tool}}$'s treatment of array havocs is incomplete; the havoc (↚ a[i]) propagates outwards to become (↚ a). This is an over-approximation of the original command's non-determinism, meaning that Starling$_{\text{tool}}$ may reject some terms where the postcondition depends on at least part of the array avoiding havoc. In practice, we can encode such situations as (**seq** (↚ x)(← a[i] x)), where x is a fresh local variable[5].

Sequential composition. Sequential compositions inside actions need care, as intermediate states must not be observable outside of the action. To expand sequential compositions, we can either symbolically compute their effect (eliminating intermediate states entirely), or introduce fresh, solver-quantified variables to hold intermediate results; for example:

Original	Symbolic computation	Intermediate variables
(**seq** (← y x) (← x (+ x 1)) (← y (+ y x)))	(**and** (= x′ (+ x 1)) (= y′ (+ x (+ x 1))))	(**and** (= y_0 x) (= x′ (+ x 1)) (= y′ (+ y_0 x′)))

While symbolic computation leads to smaller verification conditions, it requires heavy-weight analysis on each command—analysis that gets complicated when we add conditionals and variable havoc. The solvers we target, on the other hand, cope acceptably well with encodings using intermediate variables, and so we choose that approach for Starling$_{\text{tool}}$.

Intermediate variable generation interacts subtly with havoc. When an (↚ x) action sequences before another action using x, we generate (but *do not* assign to) a variable to represent whichever value x non-deterministically assumes after havoc. If nothing assigns to x afterwards, this variable is just x′; else, we make an intermediate variable. For example, `<| havoc x; havoc y; y = y+x; |>` becomes (= y′ (+ y_0 x′)).

Framing. While microcode actions only refer to the specific variables they affect, two-state predicates must bind all variables that are not the subject of a (↚ —) clause. To handle this discrepancy, we apply a version of `rframe` to each translated microcode action, introducing an (= x′ x) constraint for each x neither assigned nor subjected to havoc.

[5] Future versions of Starling$_{\text{tool}}$ may apply this encoding automatically.

Framing does not occur over intermediate variables. This is because such variables correspond to intermediate assignments; if an intermediate variable does not exist for a given x, there are no such assignments, and generating a framing constraint on x' will suffice.

Emitting: view expansion

(Code: Starling.Reifier) (Code: Starling.Flattener) (Code: Starling.Instantiate)

As well as expanding microcode into two-state predicates, we expand views into one-state predicates. This process closely follows the analogous part of the $_{\text{g}}$Starling frontend.

Pattern matching. First, we expand the weakest precondition into the set of all pattern matches against the `constraint` system. This expansion happens in the *reifier*, thus named because of its correspondence to the (syntactic) reification stage in $_{\text{g}}$Starling.

Since the `constraint` system translates to a guarded (iterated) syntactic definer, the pattern-matching stage corresponds to the syntactic reification we sketched earlier.

View flattening. After pattern matching, the weakest precondition is a multiset of views; the goal remains a single view. To simplify the final conversion to predicates, we flatten these views into single atoms. This process resembles the $_{\text{g}}$Starling flattening step; one difference is that we route shared states to term views (pre-state to weakest precondition, post-state to goal) by appending the right shared variables onto each atom's arguments list.

View instantiation. In the final step, we instantiate the constraint predicate for each flattened atom in both views. As each atom carries every variable—shared variable or argument—used in the constraint predicate, this is a straightforward substitution step.

Emitting: interfacing with solvers

Once we have fully-instantiated verification conditions, we can prepare them for consumption by a solver. This includes checking for anything in the outgoing conditions that cannot be expressed in the given solver; translating the conditions to the solver's native format; and calling the solver. We discuss solver-specific interfacing considerations in the case studies.

Boolean simplification. As well as interfacing with a solver to check verification conditions, we use a combination of Z3 and bespoke symbolic analysis to perform basic Boolean simplification throughout Starling$_{\text{tool}}$. As Z3 is optimised for checking Boolean expressions over the domains we use in Starling$_{\text{tool}}$, we use it to check whether Boolean expressions are tautological, contradictory, or reducible to simpler expressions.

7.3 Summary

This chapter introduced Starling$_{\text{tool}}$, an automated concurrency verifier based on $_{\text{g}}$Starling, and its input language, C$_{\text{view}}$. It discussed how C$_{\text{view}}$ maps to the LVF language, and how Starling$_{\text{tool}}$ performs the functions of a $_{\text{g}}$Starling frontend.

Chapter 8

Case Studies and Validation

We have Starling, a method for building automatable concurrent program logics; $_{\mathrm{g}}$Starling, one such logic; and Starling$_{\mathrm{tool}}$, which implements a derivative of it. While Starling$_{\mathrm{tool}}$'s existence suggests that Starling can produce tooling of some form, we have not yet seen evidence that said tooling (or, indeed, the meta-theory behind it) is *useful*. Can Starling$_{\mathrm{tool}}$ prove real-world concurrent algorithms correct, and disprove algorithms that are broken for interesting reasons? Can it give us insights into why concurrent algorithms work and fail?

We use a variety of approaches to answer these questions. In § 8.1, we show that Starling$_{\mathrm{tool}}$ can prove and disprove real-world algorithms using a selection of *case studies*. For reproducibility, we distribute the case studies with Starling$_{\mathrm{tool}}$. As evidence for the tool's correctness, we use unit and regression testing; we discuss these approaches in § 8.2.

We also may ask whether the Starling meta-theory we produced in the previous chapters is consistent, and whether the tool implements $_{\mathrm{g}}$Starling-style reasoning correctly. While we do not have a formal mechanisation of all of Starling, or Starling$_{\mathrm{tool}}$, we have mechanised parts of each. § 8.3 outlines the mechanisation and its usefulness as a validation tool.

8.1 Case studies

So far, we have seen one C$_{\mathrm{view}}$ proof: the ticket lock (Listing 7.1). In this section, we explore other algorithms for which our tool can prove useful properties. We give full proofs for each, as well as examples of code and specification bugs that Starling$_{\mathrm{tool}}$ can detect. We discuss which solvers we can use for each proof, and subtleties in how proofs map to solver input.

Table 8.1 summarises the case studies we give in this section.

Verifying the ticket lock proof

(Code: Examples/Pass/ticketLock.cvf)

This section outlines how Starling$_{\mathrm{tool}}$ verifies the ticket lock proof in Listing 7.1. As the proof contains neither inference requests nor heap accesses, we can use Starling$_{\mathrm{tool}}$'s Z3 backend to carry out the proof.

Starling$_{\mathrm{tool}}$ sends each verification condition to Z3, using its .NET API[1], as a separate neg-

[1] A more portable, but less efficient, manner would be to send conditions to Z3 as *SMT-LIB* input.

	Z3		GRASShopper	HSF
Study	Pass	Fail	Pass	Pass
Ticket lock	Listing 7.1	Listings 8.9 and 8.10		Listing 8.8
Peterson	Listing 8.1	Listing 8.11		
Circular buffer	Listing 8.2			
ARC	Listing 8.3		Listing 8.5	
CLH lock			Listing 8.7	

Table 8.1: Case study matrix, mapping case studies below to solvers used and whether the proofs pass or fail. Each proof entry refers to its discussion in the dissertation, which, in turn, refers to the source in the Starling$_{\text{tool}}$ repository.

ated Hoare judgement ($w \wedge c \wedge \neg g$). A proof succeeds when all such clauses are unsatisfiable; this double negation represents a search for counter-examples to each condition.

Consider this goal axiom, which comes from the optimised ticket lock CFG:

$$(\langle(\textbf{@}(\textit{Tick}\ \mathtt{t}))\rangle\ (\textbf{seq}\ (\leftarrow\ \mathtt{c}\ \mathtt{s})\ (\textbf{assume}\ (=\ \mathtt{c}\ \mathtt{t})))\ \langle(\textbf{@}(\textit{Lock}))\rangle\ ,\quad (\textbf{@}(\textit{Lock}))\,)$$

The resulting proof term, in structured predicate form, is:

$$\langle\!\langle(\textbf{and}\ (>=\ \mathtt{n}\ \mathtt{s})\ (>\ \mathtt{n}\ \mathtt{t}))\rangle\!\rangle\ (\textbf{and}\ (=\ \mathtt{c}'\ \mathtt{s})\ (=\ \mathtt{c}'\ \mathtt{t}))\ \langle\!\langle(\neq\ \mathtt{n}'\ \mathtt{s}')\rangle\!\rangle$$

This proof term is smaller than, for example, the proof term we generated in § 6.5 through rote application of $_{\text{g}}$Starling. This reflects the fact that Starling$_{\text{tool}}$ performs expression optimisation (both at Boolean and view level), and its approach to framing lets us eliminate post-state variables when no assignment or havoc occurs to them.

The proof term generates a Z3 query similar to the following SMT-LIB input:

```
(and (and (>= VnBEFORE VsBEFORE) (> VnBEFORE VtBEFORE))
     (and (= VcAFTER VsBEFORE) (= VcAFTER VtBEFORE))
     (= VnBEFORE VsBEFORE))
```

Peterson's algorithm

(Code: Examples/Pass/petersonArray.cvf)

Having used Peterson's algorithm as a source of running examples in previous chapters, let us now build a C$_{\text{view}}$ version of the algorithm and prove that it obeys mutual exclusion.

Proof outline. There now follows a C$_{\text{view}}$ proof outline for *Peterson*. This outline corresponds to the sketches of § 6.4, but uses C$_{\text{view}}$'s higher-level syntax.

We start by modelling the algorithm's variables. While the original sketch used a two-element type to model thread IDs, we opt for `int`; 0 represents thread A, and 1 thread B. (We can encode the assumption that IDs are always 0 or 1 in our `constraint` table later on.) We then model the flags as an ID-indexed Boolean array. This lets us reduce the difference between the two threads to the ID itself, which we can pass as a method parameter.

Listing 8.1: Peterson's algorithm: C_{view} proof using Z3

```
shared bool[2] flag;  // Whether each flag is seeking the lock
shared int     turn;  // Used to give the other thread priority
thread bool    oFlag; // The thread's view of its opponent's flag
thread int     oTurn; // The thread's view of the current turn
```

The view atoms we use correspond to the tag set we gave for Peterson's algorithm in § 6.4 (and therefore to the per-thread states we identified in Figure 3.3):

```
view FlagDown (int i), FlagUp (int i), Waiting (int i), Lock (int i);
```

The `lock` and `unlock` methods follow. There are some changes over the previous sketch, reflecting the encoding of thread IDs as integers. Here, `turn` stores the ID of the thread that *does not* have turn priority (with appropriately altered turn conditions). To read another thread's flag, we can use modular arithmetic to find that thread's ID.

```
method lock(int i /* thread id */) {
  {| FlagDown(i) |}
    <| flag[i] = true; |>
  {| FlagUp(i) |}
    <| turn = i; |>
  {| Waiting(i) |}
    do {
      {| Waiting(i) |}
        <| oFlag = flag[(i + 1) % 2]; |>
      {| if oFlag { Waiting(i) } else { Lock(i) } |}
        <| oTurn = turn; |>
      {| if oFlag && oTurn == i { Waiting(i) } else { Lock(i) } |}
    } while oFlag && (oTurn == i);
  {| Lock(i) |}
}
method unlock(int i) {
  {| Lock(i) |} <| flag[i] = false; |> {| FlagDown(i) |}
}
```

Constraints. The constraints start with the invariant: `turn` must be a valid thread ID.

```
constraint emp -> 0 <= turn <= 1;
```

The goal of *Peterson*, as with the ticket lock, is mutual exclusion. We can re-use the ticket lock's mutual-exclusion constraint, with changes to account for thread IDs:

```
constraint Lock(me) * Lock(you) -> false;
```

In Peterson's algorithm, the individual thread automaton states (and, by extension, the atoms) do not correspond directly to the concrete state of the shared variables. In fact, *FlagUp*, *Waiting*, and *Lock* all have the same definition:

```
26 constraint FlagDown (t) -> flag[t] == false && (0 <= t <= 1);
27 constraint FlagUp   (t) -> flag[t] == true  && (0 <= t <= 1);
28 constraint Waiting  (t) -> flag[t] == true  && (0 <= t <= 1);
29 constraint Lock     (t) -> flag[t] == true  && (0 <= t <= 1);
```

The difference between them relates to constraints on the allowed *pairs* of thread states. For Peterson's algorithm to work, the situation in which two threads simultaneously contest the lock must resolve to a stable situation in which only one thread holds the lock. This property comes from the fact that both threads atomically flip the `turn` variable before they begin waiting — this imposes an ordering on the threads where the last thread to do so must wait. We enforce this ordering with a two-atom constraint on *Waiting* and *Lock*:

```
30 constraint Lock(me) * Waiting(you) -> me != you && turn == you;
```

The additional `me != you` relates to the fact that each thread can only be in one automaton state at a given time. To enforce this, we constrain each pair of states such that the thread identifiers must differ; this gives us the following extra constraints:

```
31 constraint FlagDown (me) * FlagDown (you) -> me != you;
32 constraint FlagUp   (me) * FlagUp   (you) -> me != you;
33 constraint FlagUp   (me) * Waiting  (you) -> me != you;
34 constraint FlagUp   (me) * Lock     (you) -> me != you;
35 constraint Waiting  (me) * Waiting  (you) -> me != you;
```

Other proofs. Peterson's 1981 article gave an informal correctness argument in prose, including a paragraph on mutual exclusion [19]. The argument resembles ours; it also focuses on the way in which the turn variable breaks ties between threads, therefore eliminating states in which mutual exclusion fails.

Dijkstra gave a more formal Owicki-Gries-style proof in 1981 [67]. This proof is similar to ours[2], but uses auxiliary program-counter variables to achieve the two-thread assertions that we represent natively in gStarling with two-atom constraints.

These two proofs are more human-readable than ours — their main arguments are prosaic, and Dijkstra's annotations are flat logical propositions. This informality makes the proofs hard to verify automatically; in addition, the auxiliary variables used in Dijkstra's treatment blur the boundaries between proof and algorithm.

Circular buffer

(Code: Examples/Pass/circular.cvf)

So far, we have only considered mutual exclusion algorithms. To validate the approach over other algorithm types, let us consider Algorithm 5: a toy *circular buffer* implementation.

Circular buffers let us send an unbounded data stream from one producer to one consumer in constant space, with the only interruptions to data flow being buffer underruns or overruns. Their use cases include transferring audio data from a decoder thread to a playback thread.

[2]Excluding the fact that Dijkstra's presentation uses guarded commands rather than C-like control flows.

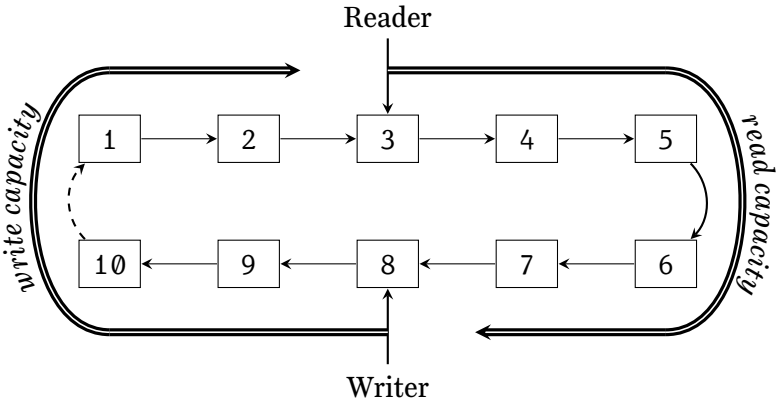


Figure 8.1: Circular buffer with writer and reader spaced evenly apart; both capacities are 5.

In the circular buffer, both producer and consumer maintain separate pointers into the same memory space. As the producer produces data, it writes the data at its pointer and advances it, as the consumer consumes data, it reads from, and advances, *its* pointer. When pointers reach the end of the memory space, they cycle back to its start: hence, they make circular paths across the buffer. The two pointers divide the buffer into disjoint regions as seen in Figure 8.1: the span from the read pointer to the write pointer holds data ready to consume, and the span from write to read is free space the producer can use.

Specification. The ideal specification for the circular buffer is that it refines a fixed-capacity, immutable queue with support for batch reading and writing as well as erasing ('flushing'). Let us call the number of items in the queue the *write capacity*, and the number of extra items for which the queue has space at any given time is the *read capacity*. Any writes to the queue must not exceed the write capacity, and any reads must not exceed the read capacity. Flushing can only happen if both reader and writer agree.

This specification leads to the properties that:

- the read and write capacities are always non-negative and sum to the queue capacity;
- once an item is written, it does not change until it is read;
- if the writer writes n items, then, eventually (and perhaps over multiple reads), the reader will read those n items in first-in-first-out order;
- after a flush (in which both reader and writer must synchronise), there exist no elements in the queue (read capacity is 0).

The level of expressivity in $\text{Starling}_{\text{tool}}$ limits the amount of this specification that we can prove. There is no known way to express the first-in-first-out property, or immutability (the nature of the circular buffer involves repeatedly overwriting unused data, and expressing in views which data is known to be 'used' is not straightforward). As such, the case study below proves a somewhat weaker specification in which we show the remaining properties, as well as a small step towards showing immutability: that there is no overlap between the regions of the circular buffer accessed by the reader and writer.

Algorithm 5 Circular buffer

```
b  : array 1...100 of ℕ shared        ▷ buffer
r  : ℕ shared                         ▷ read capacity
w  : ℕ shared                         ▷ write capacity
p  : ℕ local                          ▷ current position of thread in buffer
l  : array 1...100 of ℕ local         ▷ local buffer

procedure WRITE(n)
    x : ℕ                             ▷ storage for estimated write capacity
    d : ℕ                             ▷ amount written so far
    ⟨x := w⟩
    for d := 0...MIN(x, n) do         ▷ (exclusive)
        ⟨b[p] := l[d]⟩
        p := (p + 1) mod 100          ▷ advance pointer with wrap-around
    end for
    ⟨r := r + d; w := w − d⟩          ▷ update capacities
end procedure

procedure READ(n)
    x : ℕ                             ▷ storage for estimated read capacity
    d : ℕ                             ▷ amount read so far
    ⟨x := r⟩
    for d := 0...MIN(x, n) do         ▷ (exclusive)
        ⟨l[d] := b[p]⟩
        p := (p + 1) mod 100          ▷ advance pointer with wrap-around
    end for
    ⟨r := r − d; w := w + d⟩          ▷ update capacities
end procedure
```

Proof outline. We now consider a C_{view} proof of Algorithm 5. For simplicity, the outline keeps as close to the pseudocode as possible. As a result, it has some features, such as representing read and write capacity as separate variables kept in lock-step, that do not necessarily correspond to real-world circular buffer implementations.

The variable declarations are exactly the same as their equivalents in the pseudocode:

Listing 8.2: Circular buffer: C_{view} proof using Z3 (full version at Appendix A.7)

```
1 shared int[100] circ_buf;
2 shared int      r_capacity, w_capacity;
3 thread int      position;
4 thread int[100] local_buf;
```

The writer-thread outline follows. Throughout, we track the writer's position in the buffer and its *estimate* of the write capacity in the *Writer* atom. Local assertions ensure the amount to be written (`c`) and the amount already written (`wrote`) remain in-bounds.

```
5 method write(int c) {
6   {| Writer(position, 0) * local{0 <= c <= 100} |}
7     thread int wc; <| wc = w_capacity; |>
8   {| Writer(position, wc) * local{0 <= c <= 100} |}
```

```
9     if wc < c {
10      {| Writer(position, wc) * local{0 <= c <= 100} |}
11        c = wc;
12      {| Writer(position, wc) * local{0 <= c <= wc} |}
13    }
14  {| Writer(position, wc) * local{0 <= c <= wc} |}
15    thread int wrote; wrote = 0;
16  {|  Writer(position, wc) * local{0 <= wrote <= c <= wc} |}
17    while wrote < c {
18      {|  Writer(position, wc) * local{0 <= wrote < c <= wc} |}
19        <| circ_buf[position] = local_buf[wrote]; |>
20        position = (position + 1) % 100;
21        wrote++;
22      {|  Writer(position, wc) * local{0 < wrote <= c <= wc} |}
23    }
24  {| Writer(position, wc) * local{0 <= wrote && wrote <= wc} |}
25    <|  w_capacity = w_capacity - wrote;
26        r_capacity  = r_capacity + wrote; |>
27  {| Writer(position, wc - wrote) |}
28 }
```

The `read` method is, for the most part, a mirror image of `write`. As such, it does not appear here; Appendix A.7 gives the full proof script.

To empty the buffer, we just mark the whole buffer as writable. In this implementation, we do so by atomically resetting the write capacities. To flush safely, we need both threads to synchronise; we can model this by collecting both atoms in one assertion.

```
53 method flush(int p1, int p2) {
54   {| Reader(p1, 0) * Writer(p2, 0) |}
55     <| r_capacity = 0; w_capacity = 100; |>
56   {| Reader(p1, 0) * Writer(p2, 100) |}
57 }
```

Flushing expects both threads to have capacity estimates of 0. We can show that each thread can safely forget its estimate like so (eliding a similar method for the reader):

```
58 method forget_wcap(int c) {
59   {| Writer(position, c) |} ; {| Writer(position, 0) |}
60 }
```

Constraints. The invariant constraint ensures that the read and write capacities sum to 100, the buffer's storage capacity. It also requires the capacities to be non-negative (to prevent, say, situations where one capacity is 101, but the other is -1).

```
64 constraint emp -> 0 <= w_capacity && 0 <= r_capacity
65                 && w_capacity + r_capacity == 100;
```

The writer thread's constraint ensures that: its buffer position must be in-bounds; its capacity estimate must be non-negative; and the estimate must be pessimistic with regards to the actual write capacity at all times. This pessimism lets the reader grow the write capacity in parallel with the writer. As in *Peterson*, another constraint prevents two threads from assuming the writer role.

```
66 view Writer(int position, int cap_estimate);
67 constraint Writer(position, cap_estimate) —>
68   0 <= position < 100 && 0 <= cap_estimate <= w_capacity;
69 constraint Writer(xp, xc) * Writer(yp, yc) —> false;
```

The reader thread's constraints mirror those of the writer, so we do not discuss them here.

Atomic reference counter: static

(Code: Examples/Pass/arc.cvf)

Guarded views work well on proofs over protocols between fixed numbers of threads, with pre-arranged thread roles and limited resource movement. When proofs involve the transfer of an unbounded number of counting permissions, we need iterated views.

The *atomic reference counter* (ARC) is a small, but realistic, algorithm that relies on such transfers. The ARC counts references to a shared resource (using atomic actions to ensure the counter's accuracy in the face of multi-thread reference acquisition and releasing), freeing the resource once no references remain. A sophisticated ARC forms a major plank of the *Rust* language's concurrency support [68][16.3]; Algorithm 6 gives an idealised version.

Algorithm 6 Idealised atomic reference counter, due to Dreyer [69].

```
function ArcClone(x : Arc)
    ⟨x.count := (x.count) + 1⟩                          ▷ add reference
    return x                          ▷ all threads reference the same counter and data
end function

procedure ArcPrint(x : Arc)
    Print(x.data)
end procedure

function ArcDrop(x : Arc)
    c : ℕ                                    ▷ storage for current counter value
    ⟨c := x.count; x.count := (x.count) − 1⟩                ▷ remove reference
    if c = 1 then                                          ▷ no references remain
        Free(x)
    end if
end function
```

This case study concerns a version of the ARC that only uses shared variables: while this does not reflect usual ARC usage, it simplifies the proof, and lets us use Z3. A further study later on concerns a variant that allocates the ARC in a heap; that study targets GRASShopper.

Specification. The ARC specification this case study uses concerns the memory safety of reference counting (specifically, a property we can call *no-use-after-free*):

> Given a reference counter over a potentially unavailable resource Data in which each reference is a copy of a resource Ref, Data is unavailable ('freed') only if precisely zero copies of Ref exist.

As long as a thread is aware that a Ref exists (for example, by owning one itself), it can rely on the availability of Data (and so, if a reference guards all uses of the data, there can be 'no use after free'). In Algorithm 6, count tracks the number of Refs in existence.

The specification does not guarantee that the data is freed *if* no more references exist. This is not a safety property, but rather one of resource efficiency.

Proof outline. As usual, the outline starts listing the shared variables. While the integer `count` directly represents the reference count, the outline abstracts away the actual data being governed by the ARC. Instead, it uses a flag, `free`, as an abstract representation of whether or not the data has been deallocated by the ARC.

Listing 8.3: Atomic reference counter: C_{view} proof using Z3 (static allocation)

```
1 shared int count; shared bool free;
```

The proof uses two atoms. The *iterated* atom *Arc* represents *one* reference to the reference-counted resource. A thread may hold as many *Arc*s as needed, and transfer them to other threads. The regular atom *CountWas* over c asserts that, at one time, the reference count was c; we use this to reason about whether we can safely free the resource.

```
2 view iter Arc; view CountWas(int c);
```

The first method models ArcClone. As C_{view} does not support `returns`, and this form of the ARC does not heap-allocate the counter, the method omits the second statement.

```
3 method clone() { {| Arc |} <| count++; |> {| Arc * Arc |} }
```

The next method models ArcDrop up to freeing the ARC. We cannot model freeing directly, so we instead simulate it by raising the `free` flag. Between fetch-and-incrementing the counter and (potentially) freeing the ARC, a *CountWas* atom holds the fetched value.

```
4  method drop() {
5    {| Arc |}
6      thread int c; <| c = count--; |>
7    {| CountWas(c) |}
8      if c == 1 { {| CountWas(1) |} <| free = true; |> {| emp |} }
9    {| emp |}
10 }
```

As we have neither data nor a way to print it, we model ArcPrint in a stylised manner. To simulate the act of reading the data for printing, we test whether the `free` flag has been set to true, and assert that it has not; we also assert that the test preserves our *Arc* atom.

```
11 method print() {
12   {| Arc |} thread bool f; <| f = free; |> {| local{ !f } * Arc |}
13 }
```

Constraints. We first constrain *Arc*. Holding n copies of *Arc* tells us that *at least* n references are active[3], so our constraints reflect this. As we are proving no-use-after-free, we also assert that the presence of even one *Arc* means that the data has not yet been freed.

```
14 constraint iter[n] Arc -> n > 0 => (free == false && n <= count);
```

For *CountWas*, we cannot assert any inequalities between the observed and current values of `count`. This is because any thread holding an *Arc* can `clone` it (increasing `count`) or `drop` it (decreasing `count`). Instead, we note that once we observe a count of 1, we must have just relinquished the last reference; if so, the count is 0 but the data is still present. We further constrain *CountWas* to assert that only one thread can be in this position.

```
15 constraint CountWas(c) -> c == 1 => (free == false && count == 0);
16 constraint CountWas(m) * CountWas(n) -> (m != 1) || (n != 1);
```

This example demonstrates that constraints need not (and sometimes *cannot*) fully and directly describe the atom's abstract meaning in terms of the program's concrete state.

Verification. While we again rely on Z3 to verify the static ARC, this time we must deal with iterated views. To demonstrate, we focus on the following goal axiom of `clone` (line 3):

$$(\langle(@(Arc))\rangle\ (\leftarrow\ \texttt{count}\ (+\ \texttt{count}\ 1))\ \langle(\bullet\ (@(Arc))\ (@(Arc)))\rangle\ ,\quad (@(Arc^{n})))$$

Converting this goal axiom into a verification condition is less straightforward than in non-iterated cases. We must calculate $(\setminus\ (@(Arc^{n}))\ (@(Arc^{2})))$, a subtraction[4] of a known number of atoms from a universally-quantified number[5]. This subtraction results in a guarded view, $(\rightarrow(>\ n\ 2)\ Arc^{(-\ n\ 2)})$; this models the fact that the subtraction only leaves behind some *Arc* atoms when that number is positive. We now have this proof term:

$$\left\langle(\bullet\ (@(Arc))\ (@(\rightarrow(>\ n\ 2)\ Arc^{(-\ n\ 2)})))\right\rangle\ (\leftarrow\ \texttt{count}\ (+\ \texttt{count}\ 1))\ \langle(@(Arc^{n}))\rangle$$

We can now apply the command semantics and reification. The semantics is straightforward:

$$[\![(\leftarrow\ \texttt{count}\ (+\ \texttt{count}\ 1))]\!] = (\textbf{and}\ (=\ \mathsf{count}'\ (+\ \mathsf{count}\ 1))\ (=\ \mathsf{free}'\ \mathsf{free})\ (=\ \mathsf{c}'\ \mathsf{c}))$$

The reification is subtle, for the reasons we discussed in § 6.7. While we need only match against the pattern $[Arc^{*}]$, and can use downclosure to put bounds on the number of matches we consider, we must match *all* potential combinations of the *Arc* atom. This includes both $(@(Arc))$ and $(@(\rightarrow(>\ n\ 2)\ Arc^{(-\ n\ 2)}))$ as expected, but *also* the result of matching the two atoms together, which simplifies to $(@(\rightarrow(>\ n\ 2)\ Arc^{(-\ n\ 1)}))$. Inductive downclosure means that this merged case subsumes the $Arc^{(-\ n\ 2)}$ case, so we can eliminate it.

[3] We cannot stably assert that exactly n references exist; this, for example, would violate the frame rule.
[4] Technically, at time of writing, $\text{Starling}_{\text{tool}}$ would subtract each of the two atoms separately.
[5] We assume that n is disjoint from all other variables.

As with the ticket lock, we finish by expanding the definitions and sending the negation of the judgement to Z3. Viewed as a SMT-LIB query, the result looks like this:

```
(and (= count′ (+ count 1)) (= free′ free) (= c′ c)
     (=> (> n 2) (=> (> (- n 1) 0) (and (not free) (<= (- n 1) count))))
     (not (=> (> n 0) (and (not free′) (<= n count′)))))
```

Atomic reference counter: heap-based

(Code: Examples/PassGH/arc.cvf)

We now modify the ARC to allocate the counter structure on a shared heap. This is more realistic, but requires us to target a new backend solver and make changes to our proof.

GRASShopper. For heap-based programs, we can use *GRASShopper* [55]. GRASShopper implements a decision procedure for *GRASS*, a reachability logic over pointer graphs highly similar to Reynoldsian separation logic.

GRASShopper's model is based on sets of heap locations and reachability properties over sets. By building predicates over the pointer paths between set elements, we can assert the existence and correctness of heap data structures. The syntax below defines a GRASShopper predicate asserting that the location set `Footprint` contains a list with head `x` and tail `y`:

```
predicate list_segment(Footprint: Set<Node>, x: Node, y: Node) {
  acc(Footprint) &*&
  Footprint = {z: Node :: Btwn(next, x, z, y)}
}
```

First, `acc(Footprint)` asserts that each node in `Footprint` exists in the heaplet that the predicate can access. The `Btwn(next,x,z,y)` predicate asserts that we can reach `z` by starting at `x` and following the `next` pointers until `y`; each such `z` thus belongs to a list between `x` and `y`. The comprehension `{z: Node :: Btwn(next, x, z, y)}` tells us that each node in `Footprint` is on said list.

Adapting the proof. The first changes we make to the proof set up several types and pragmata that the GRASShopper backend expects. First, in a separate file (let us assume the filename `arc-module.spl`), we define the shape of an ARC structure (or *node*):

Listing 8.4: Auxiliary GRASShopper module `arc-module.spl`

```
1 struct ArcNode { var count : Int; var val : Int; }
```

On the C_{view} side, `ArcNode` is a subtype of *int*. This reflects the fact that, at the $_{g}$Starling level, ARC nodes are opaque pointers. Another `int` subtype handles the fact that GRASShopper uses the type `Int` (capital 'I') for integers.

Listing 8.5: Atomic reference counter: C_{view} proof using GRASShopper (dynamic allocation)

```
1 typedef int Int; typedef int ArcNode;
```

Next follows a set of **pragma** statements: key-value pairs that provide instructions to the backend. Here, they tell the backend: where to find the file with the `ArcNode` definitions; that the set of all ARC nodes is `ArcFoot`; and that `ArcFoot` is a set of `ArcNodes`.

```
pragma grasshopper_include         {arc-module.spl};
pragma grasshopper_footprint       {ArcFoot};
pragma grasshopper_footprint_sort  {Set<ArcNode>};
```

The view atoms remain broadly the same as their static counterparts, except that they now take an extra parameter: the pointer to the `ArcNode` that the atom concerns.

```
view iter Arc (ArcNode x); view CountWas (ArcNode x, Int c);
```

A new `init` method models the heap allocation of ARC nodes. It uses symbols to access GRASShopper primitives for memory allocation and variable update. Let us assume that the ARC node is exclusively owned by the initialising thread throughout the method; treating the method body as an atomic action models this.

```
method init(ArcNode ret) {
  {| emp |}
    <| ret = %{ new ArcNode }; %{ [|ret|].count := 1 }; |>
  {| Arc(ret) |}
}
```

Cloning a dynamic ARC is the same as cloning a static ARC, except that we delegate the increment action to GRASShopper through a symbol.

```
method clone(ArcNode x) {
  {| Arc(x) |}
    <| %{ [|x|].count := [|x|].count + 1 }; |>
  {| Arc(x) * Arc(x) |}
}
```

As the proof targets a backend that ensures that heap accesses target properly-allocated objects (that is, objects inside the footprint set), we can model `print` more accurately than before. `print` now does so by fetching the ARC node's value into a thread variable.

```
method print(ArcNode x) {
  {| Arc(x) |} thread int p; <| p = %{ [|x|].val }; |> {| Arc(x) |}
}
```

The model of `drop` is largely unchanged. Instead of simulating freeing the ARC node with a flag, the model now uses GRASShopper's language to `free` the node for real.

```
19 method drop(ArcNode x) {
20   {| Arc(x) |}
21     thread int c;
22     <| c = %{ [|x|].count }; %{ [|x|].count := [|x|].count − 1 }; |>
23   {| CountWas(x, c) |}
24     if c == 1 {
25       {| CountWas(x, 1) |} <| %{ free([|x|]) }; |> {| emp |}
26     }
27   {| emp |}
28 }
```

The constraint set receives some minor changes, too. First, each single-atom constraint now refers to the ARC node pointed to by the atom. Second, the free-status of node x now corresponds to its membership in `ArcFoot`. Third, the final constraint relaxes such that it need only hold if both *CountWas* atoms refer to the same node.

```
29 constraint iter[n] Arc(x) −>
30    n > 0 => %{ [|x|] in ArcFoot && [|n|] <= [|x|].count };
31 constraint CountWas(x, c) −>
32    c == 1 => %{ [|x|] in ArcFoot && [|x|].count == 0 };
33 constraint CountWas(x, m) * CountWas(y, n) −>
34    x == y => ((m != 1) || (n != 1));
```

Verification. GRASShopper accepts sequential programs in a C-like language; each program consists of procedures with requires–ensures specifications. Unlike Z3, where we translated verification conditions into universally-quantified predicates and asked the solver to try to falsify each separately, we now model each condition as one procedure in a program.

Most of the pipeline for producing GRASShopper proofs is similar to the SMT case. (Indeed, $\text{Starling}_{\text{tool}}$ first tries to discharge as many verification conditions as possible using Z3, substituting true and false for each symbol to produce a sound but incomplete approximation.) The presence of a heap model causes some differences. Suppose we try to model the allocated-ARC equivalent of our previous working example:

$$\langle (@(\mathit{Arc}\ x)) \rangle\ (\leftarrow\ \texttt{count}\ (+\ \texttt{count}\ 1))\ \langle (\bullet\ (@(\mathit{Arc}\ x))\ (@(\mathit{Arc}\ x))) \rangle$$

Given a context of $(\bullet\ (@(\mathit{Arc}\ x))\ (@(\mathit{Arc}\ x)))$ (the same x as in the local state of the thread), our translation would give the following in pseudo-SMT format:

```
(and (sym x.count := x.count + 1;)
     (sym x in ArcFoot && 1 <= x.count)
     (=> (> n 2) (and (sym x in ArcFoot) (<= (- n 1) (sym x.count))))
     (not (=> (> n 0) (and (sym x in ArcFoot) (<= n (sym x.count))))))
```

We cannot discharge this term using SMT, but can convert it into a GRASShopper procedure. The command becomes the procedure body, and the left- and right-hand sides of

the proof rule body become **requires** and **ensures** clauses. Both of these quantify over a *footprint set* representing the whole heap – in the ARC this is the `ArcFoot` set. This allows predicates to state conjunctive constraints over a single shared heap. Arguments to the procedure stand for input and output variables. With this translation, the above becomes:

```
procedure Example (n: Int, x: ArcNode)
requires exists ArcFoot:Set<ArcNode> :: (
         acc(ArcFoot) &*&
         ((x in ArcFoot && 1 <= x.count) &&
          (n <= 2 || (x in ArcFoot && n <= x.count))) )
ensures exists ArcFoot:Set<ArcNode> :: (
        acc(ArcFoot) &*&
        (n <= 0 || (x in ArcFoot && n <= x.count)) )
{ x.count := x.count + 1; }
```

Whenever we must model variable mutation, we can declare fresh GRASShopper variables in the procedure body, and connect them to the input and output variables by assertion.

CLH queue lock

GRASShopper's dynamic-data-structure support lets us target more complex algorithms than the ARC. Here, we verify the *CLH* lock (due to Craig [70] and Landin and Hagersten [71]).

In the CLH lock, each thread owns a single node in a linked queue. To contend for the lock, a thread raises a flag on the node, atomically adds it to the back of the queue, then waits on its predecessor. To release the lock, the thread lowers the node's flag; once a thread's node's predecessor is released, the thread can take the lock.

When a thread finishes releasing the lock, it must still own a node — else it cannot re-acquire the lock later on. The thread cannot immediately re-use its last node, as another thread may still be observing it; instead, the thread takes ownership of its *predecessor*. Threads always exclusively hold *some* node, but the node varies over time.

To make sure the lock queue is always in a valid state, the lock starts with one unlocked *sentinel* node. In addition, each thread allocates an initial node when it joins the lock system.

Let us once again design a proof by viewing the algorithm as a series of interacting finite-state automata. The automaton, which we give in Figure 8.2, has the same abstract shape of that of Peterson's algorithm (Figure 3.3). This time, each automaton corresponds to a queue node (not a thread), and we give the states a different concrete meaning. The differences between the states are subtle, so Figure 8.3 gives diagrams for each.

Specification. The specification that this case study explores is that used for previous lock algorithms (§ 3.3). Like the ticket lock, the CLH lock proof has separate `lock` and `unlock` methods, and a critical section is any span between their respective calls. A thread acquires an abstract Lock when its node matches the shape shown in the rightmost diagram in Figure 8.3.

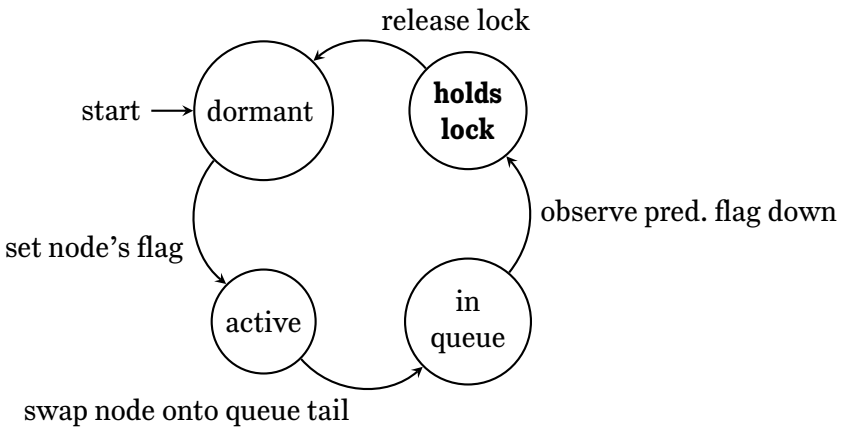


Figure 8.2: The finite-state automaton underlying a single node in the CLH lock.

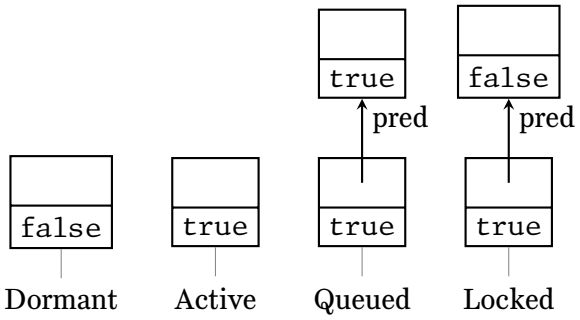


Figure 8.3: The four states of a CLH lock node, as box diagrams.

Auxiliary definitions. As before, a separate GRASShopper file defines CLH lock queue nodes. Each node physically contains the flag that signals whether the node's owner is contesting the lock. To make the proof easier, a ghost-state pointer tracks the predecessor node.

Listing 8.6: Auxiliary GRASShopper module `clh-module.spl`.

```
1 struct Node { var lock: Bool; var pred: Node; }
```

Proof. Like the dynamic ARC proof, the C_{view} script starts with type definitions and pragmata that define the node type, footprint, and footprint sort.

Listing 8.7: CLH lock: C_{view} proof using GRASShopper

```
1 typedef int Node;
2 pragma grasshopper_include         {clh-module.spl};
3 pragma grasshopper_footprint       {Foot};
4 pragma grasshopper_footprint_sort  {Set<Node>};
```

Let us track the node queue's tail pointer at the C_{view} level with a shared variable. The head of the queue appears as ghost state; the combination of tail node, head node, and predecessor pointers lets us model the CLH queue as a linked list in GRASShopper, despite this structure being entirely implicit in the actual algorithm.

```
5 shared Node tail, Node head; // ('head' is ghost code)
```

Each thread tracks the node it holds and, once the node is queued, its predecessor.

```
thread Node mynode, mypred;
```

The tracked nodes mainly serve as arguments to view atoms representing each node's automaton state. By associating the states with nodes, we need not assign thread IDs.

```
view Dormant (Node node),              Active  (Node node),
     Queued  (Node node, Node pred), Lock    (Node node, Node pred);
```

The CLH lock functions now follow. These use C_{view} to model assignments and other actions, and GRASShopper symbols for things C_{view} cannot express.

```
method lock() {
  {| Dormant(mynode) |}
    <| %{ [|mynode|].lock := true }; |>
  {| Active(mynode) |}
    <|  mypred = tail; tail = mynode;
        %{[|tail|].pred := [|mypred|]}; |>
  {| Queued(mynode, mypred) |}
    thread bool test;
    do {
      {| Queued(mynode, mypred) |}
        <| test = %{ [|mypred|].lock }; |>
      {| if test { Queued(mynode, mypred) }
         else    { Lock(mynode, mypred) } |}
    } while test;
  {| Lock(mynode, mypred) |}
```

In the `unlock` method, the subtle ownership transfer — where a thread abandons its node and unilaterally acquires its predecessor — becomes a straightforward substitution.

```
method unlock() {
  {| Lock(mynode, mypred) |}
    <|  %{ [|mynode|].lock := false }; %{ [|mynode|].pred := null };
        head = mynode; |> // last two assignments are ghost code
  {| Dormant(mypred) |}
     mynode = mypred; // ownership transfer
  {| Dormant(mynode) |}
}
```

As usual, we prove mutual exclusion, which we formulate as the usual constraint:

```
constraint Lock   (a, ap) * Lock    (b, bp) -> false;
```

We must also assert that each node, regardless of state, is held exclusively by one thread:

```
constraint Queued (a, ap) * Queued  (b, bp) -> a != b;
constraint Dormant (a)    * Dormant (b)     -> a != b;
constraint Active  (a)    * Active  (b)     -> a != b;
```

So far, the CLH lock proof has been straightforward. This is because most of the proof complexity manifests in the invariant and single-atom definitions, through reachability predicates across the lock nodes. The invariant specifies several properties:

- both ends of the queue must be inside the footprint, and there must be a chain of predecessor pointers from the tail to the head;
- the head (the last-unlocked node, or the sentinel if none exist) has a lowered flag;
- all nodes with a predecessor have a raised flag; all nodes without one are either outside the queue, or are the head.

These properties, together, define the lock queue as a chain of nodes seeking the lock, with precisely one unlocked end node at any time.

```
36 constraint emp -> %{
37       [|head|] in Foot && [|tail|] in Foot
38    && Reach(pred, [|tail|], [|head|])
39    && ![|head|].lock
40    && (forall x : Node :: (x in Foot && x.pred != null) ==> x.lock)
41    && (forall x : Node ::
42         (x in Foot && Reach(pred, [|tail|], x) && !x.lock)
43         ==> x == [|head|]) };
```

We now define the node states. Dormant and active nodes exist in the footprint, but are not on the queue; we enforce this by stating that the nodes have no predecessors, and are not the head node. The two states differ only by the status of the lock flag.

```
44 constraint Dormant(node) -> %{
45       [|node|] in Foot
46    && [|node|] != [|head|] && [|node|].pred == null
47    && [|node|].lock == false };
48 constraint Active(node) -> %{
49       [|node|] in Foot
50    && [|node|] != [|head|] && [|node|].pred == null
51    && [|node|].lock == true };
```

If a node is enqueued, it must be somewhere on the path of predecessor pointers from the tail to the head, and must have a raised lock flag. Since we track the predecessor node in the *Queued* atom, we must also assert that it is, indeed, the node's predecessor. The *Locked* atom is definitionally identical to the *Queued* atom with one exception: instead of asserting that the lock is raised, we instead assert that the node is directly behind the head.

```
52 constraint Queued(node, pred) -> %{
53       [|node|] in Foot && [|pred|] in Foot
54    && [|node|].pred == [|pred|]
55    && [|node|].lock
56    && Btwn(pred, [|tail|], [|node|], [|head|]) };
```

	Number of lines				Number of terms		Time (s)		
	Input			Output					
Study	C_{view}	Aux.	Proof	GH	Gen.	SMT-elim	Tool	Z3	GH
Ticket lock	44	-	14	-	18	18	1.31	0.05	-
Peterson	94	-	27	-	72	72	1.51	0.17	-
Circular	122	-	36	-	138	138	1.87	0.28	-
ARC static	48	-	17	-	40	40	1.42	0.06	-
ARC dynamic	57	13	32	850	20	5	1.42	0.01	1.98
CLH	124	10	58	1407	50	21	1.38	0.02	4.79

Table 8.2: Metrics for each valid case study mentioned so far. From left: lines of input in C_{view} script and auxiliary GRASShopper files; estimated lines of which are proof annotations; lines of GRASShopper output; number of proof terms generated; number of terms discharged by Z3 (remainder sent to GRASShopper); and time spent in $Starling_{tool}$, Z3, and GRASShopper.

```
57 constraint Locked(node, pred) -> %{
58       [|node|] in Foot && [|pred|] in Foot
59    && [|node|].pred == [|pred|]
60    && Btwn(pred, [|tail|], [|node|], [|head|])
61    && [|pred|] == [|head|] };
```

Measurements

Table 8.2 collects various measurements of $Starling_{tool}$, Z3, and GRASShopper's performance on the (passing) case studies we discussed above.

Case studies with inference

So far, we have not seen any usage of $Starling_{tool}$'s inference support. There follows some brief examples of inference in C_{view} proofs. Each example uses the ticket lock.

Inferring predicate definitions. Starling proofs often depend on many small **constraint**s whose relation to the constrained atoms' human intuition is loose. Delegating such constraints' discovery to a constraint solver lets us focus on mapping the high-level intuition onto the proof by writing atom definitions and assertions.

The ticket lock has several such constraints, so we alter its proof to *infer* definitions for each (except mutual exclusion, since this is the property we want to prove):

Listing 8.8: Ticket lock: modified constraint set for use with HSF

```
17 constraint emp                -> ?;
18 constraint Tick(t)            -> ?;
19 constraint Lock               -> ?;
20 constraint Tick(a)  *  Tick(b) -> ?;
21 constraint Lock     *  Tick(t) -> ?;
22 constraint Lock     *  Lock    -> false;
```

The constraints between lines 17 and 21 amount to a search for constraints on all views of size 0 through 2, so we can replace them with a `search 2;` statement.

Neither backend seen so far supports C_{view}'s inference features; instead, we can target HSF. As HSF is a constraint solver over a relaxed variant of Horn clauses, we send its input as one self-contained system. First, we translate each definite constraint to HSF's Datalog-like syntax, modelling it as a pair of implications in both directions (effectively an if-and-only-if). For example, $Starling_{tool}$'s translation of the constraint over `Lock * Lock` is:

```
false :- v_lock_lock(Vserving, Vticket).
v_lock_lock(Vserving, Vticket) :- false.
```

Unlike the other backends, we must give HSF an initial state from which it can begin inference, and show that this state satisfies the invariant. In the current version of $Starling_{tool}$, we assume that this state is that where each variable is set to 0:

```
emp(Vserving, Vticket) :- Vserving = 0, Vticket = 0.
```

We can then translate the verification conditions themselves. For example:

```
emp(VservingBEFORE, VticketBEFORE + 1) :-
    emp(VservingBEFORE, VticketBEFORE),
    VtAFTER = VticketBEFORE.
```

Inferring view assertions. While inferring the `constraint`s shifts some proof effort from proof author to computer, the onus is still on the author to provide view assertions at each sequence point in the algorithm. While these assertions often line up with underlying state automata or other structures inside the algorithm, sometimes this is not obviously the case.

Caveats. $Starling_{tool}$'s HSF backend only supports a fragment of $Starling_{tool}$; for example, Boolean and array variables are unsupported (though Boolean expressions are allowed in conditions and constraints). We could improve this situation by performing more encoding and transformation at the $Starling_{tool}$ end; for example, we already manually model Boolean variables in HSF-bound proofs as integers constrained to the set $\{0, 1\}$, with Boolean operations modelled as integer equivalents. We leave this to future work.

Checking $Starling_{tool}$'s response to induced proof errors

This section shows the type of proof problems that $Starling_{tool}$ can detect by investigating two deliberately-induced mistakes in C_{view} proofs. Both of these studies use Z3 as the backend[6].

Ticket lock, specification. Proof failures can indicate problems with the specification (the combination of assertions and constraints). With well-known algorithms such as the ticket lock, this is likely to be the source of any such failures we encounter.

To demonstrate how $Starling_{tool}$ copes with such failures, we return to the valid lock proof in Listing 7.1, but sabotage the `constraint` set as follows:

[6]For more failing-proof examples, see the `Fail` and `FailGH` directories in the tool's examples set.

Listing 8.9: Ticket lock: induced specification failure

```
18 constraint emp -> n != s;
```

This specification mistake allows the lock to hand out tickets with values *below* that of the ticket being served, and forbids it from handing out a ticket that immediately acquires the lock. Trying to prove the lock now gives us three failures:

```
lock_C000_000 fail:
  Could not prove action: ASSIGN t, n; ASSIGN n, (+ n 1)
  under weakest precondition: the invariant
  establishes: the invariant
lock_C000_002 fail:
  Could not prove action: ASSIGN t, n; ASSIGN n, (+ n 1)
  under weakest precondition: Lock()
  establishes: Lock()
unlock_C000_000 fail:
  Could not prove action: ASSIGN s, (+ s 1)
  under weakest precondition: Lock()
  establishes: the invariant
```

The first two failures arise from the case where $n = s - 1$; the fetch-and-increment gives us $n' = s'$, violating the constraint. This situation cannot arise in a valid ticket lock, and so the broken constraint fails to forbid an invalid situation. The third failure arises the case where $s = n - 1$; the increment again gives us $n' = s'$, but the fact that no locks can exist afterwards makes this situation well-formed. Here, the constraint forbids a valid situation.

Over-constrained and under-constrained specifications, therefore, result in similar Z3-level failures. This is a side-effect of the weakest-precondition approach $_{g}$Starling takes to generating proof terms. We leave the generation of more sophisticated proof failures, including the ability to distinguish between these classes of proof error, as future work.

Ticket lock, control flow. Starling$_{\text{tool}}$ failures can also come from errors in the algorithm itself. We do not have an example of a known-broken algorithm to explore, but we can manufacture bugs in existing code and check whether Starling$_{\text{tool}}$ notices.

Accidentally inverting the conditional in a **`while`** loop can produce significant errors that are then hard to track down. Consider one such inversion in the body of the ticket lock:

Listing 8.10: Ticket lock: induced code failure

```
10 /* ... */ } while c == t;
```

This one-character change produces many failures, which Figure 8.4 collects. These actions correspond to the program-logic entailments needed when moving from the end of the **`while`** loop to the top of the loop and out of the loop respectively. Many of these failures correspond to the fact that the precondition of these entailments now asserts `Tick` and `Lock` in the opposite order from those needed to establish the entailments. Changing this precondition would fix many of these issues, but introduce others.

```
lock_C002_001 fail:
  Could not prove action: ASSUME (= c t)
  under weakest precondition:
    ((= (before c) (before t)) -> Lock())
    ((not (= (before c) (before t))) -> Tick((before t)))
    ((not (= (goal 29 t) (after t))) -> Tick((goal 29 t)))
  establishes: Tick((goal 29 t))
lock_C002_004 fail:
  Could not prove action: ASSUME (= c t)
  under weakest precondition:
    ((or
       (= (goal 36 ta) (after t))
       (not (= (goal 36 tb) (after t)))
    ) -> Tick((goal 36 tb)))
    ((= (before c) (before t)) -> Lock())
    ((not (= (before c) (before t))) -> Tick((before t)))
    ((not (= (goal 36 ta) (after t))) -> Tick((goal 36 ta)))
  establishes:
    Tick((goal 36 ta))
    Tick((goal 36 tb))
lock_C003_002 fail:
  Could not prove action: ASSUME (not (= c t))
  under weakest precondition:
    ((= (before c) (before t)) -> Lock())
    ((not (= (before c) (before t))) -> Tick((before t)))
  establishes: Lock()
lock_C003_003 fail:
  Could not prove action: ASSUME (not (= c t))
  under weakest precondition:
    Tick((goal 19 t))
    ((= (before c) (before t)) -> Lock())
    ((not (= (before c) (before t))) -> Tick((before t)))
  establishes:
    Lock()
    Tick((goal 19 t))
lock_C003_005 fail:
  Could not prove action: ASSUME (not (= c t))
  under weakest precondition:
    Lock()
    ((= (before c) (before t)) -> Lock())
    ((not (= (before c) (before t))) -> Tick((before t)))
  establishes:
    Lock()
    Lock()
```

Figure 8.4: Failures caused by the code change in Listing 8.10.

Peterson, atomic action. We can observe the importance of the `<| turn = i; |>` action on Line 10 of Peterson's algorithm by considering the following erroneous change:

Listing 8.11: Peterson's algorithm: induced code failure

```
9   {| FlagUp(i) |}
10    <| turn = (i + 1) % 2; |> // accidentally prioritise THIS thread
11  {| Waiting(i) |}
```

If we run Starling$_{\text{tool}}$ with this mistake present, we get the output:

```
Could not prove action: ASSIGN turn, (% (+ i 1) 2)
under weakest precondition:
  FlagUp((before i))
  Lock((goal 129 me))
  ((not (= (goal 129 you) (after i))) -> Waiting((goal 129 you)))
establishes:
  Lock((goal 129 me))
  Waiting((goal 129 you))
```

As expected, the broken turn action failed. Looking at the goal definition, we see that we cannot establish `me != you && turn == you` in the post-state. We then see that, when `you` is the current thread `i`, the weakest precondition states that `me != i`, and so we must set the turn to `i`. Instead, we set it to the other thread — precisely the inserted bug.

A broken mutual exclusion algorithm. In the 1982 edition of *Principles of Concurrent Programming* [1], Ben-Ari gives a step-by-step construction of a mutual exclusion algorithm by refining a broken attempt into a safe and fair implementation (a two-thread form of Dekker's algorithm). Most of the intermediate attempts fail for liveness reasons and not safety reasons, and so Starling$_{\text{tool}}$ cannot express their failures. As the second attempt (of which Listing 8.12 is a C$_{\text{view}}$ translation) *does* fail to uphold mutual exclusion, it serves as a useful example.

Listing 8.12: Attempted proof of a flawed mutual exclusion algorithm [1]

```
1  shared bool[2] flag;
2
3  method lock(int tid) {
4    {| Idle(tid) |}
5      thread bool other_flag;
6      do {
7        {| Idle(tid) |}
8          <| other_flag = flag[(tid + 1)% 2]; |>
9        {| if other_flag { Idle(tid) } else { Checked(tid) } |}
10     } while other_flag;
11   {| Checked(tid) |} // `checked' the other thread is not locking
12     <| flag[tid] = true; |>
13   {| Lock(tid) |}
14 }
```

```
15
16 method unlock(int tid) {
17   {| Lock(tid) |} <| flag[tid] = false; |> {| Idle(tid) |}
18 }
```

The algorithm is broken: both threads can pass through the loop at the beginning of `lock` before either raises its `flag`. In theory, no assignment of views to constraints will make the proof go through; while it is impossible to show that this is the case in this study, we can still explore a couple of possibilities.

Let us start with a system that maps each of the three views to the state of `flag`, asserts mutual exclusion, and forbids the holding of multiple views with the same thread ID. At this stage, there is no difference between `Checked` and `Idle`.

```
19 view Idle(int tid), Checked(int tid), Lock(int tid);
20 constraint Idle(tid)     -> (tid == 0 || tid == 1) && !flag[tid];
21 constraint Checked(tid) -> (tid == 0 || tid == 1) && !flag[tid];
22 constraint Lock(tid)     -> (tid == 0 || tid == 1) && flag[tid];
23 constraint Idle(x)       *  Idle(y)     -> x != y;
24 constraint Idle(x)       *  Checked(y)  -> x != y;
25 constraint Idle(x)       *  Lock(y)     -> x != y;
26 constraint Checked(x)    *  Checked(y)  -> x != y;
27 constraint Checked(x)    *  Lock(y)     -> x != y;
28 constraint Lock(x)       *  Lock(y)     -> false;
```

This proof fails as follows:

```
Could not prove action: ASSIGN (select 13_1_tid flag), true
under weakest precondition:
  Checked((before 13_1_tid))
  ((or
    (= (goal 30 x) (after 13_1_tid))
    (not (= (goal 30 y) (after 13_1_tid)))
  ) -> Lock((goal 30 y)))
  ((not (= (goal 30 x) (after 13_1_tid))) -> Lock((goal 30 x)))
establishes:
  Lock((goal 30 x))
  Lock((goal 30 y))
```

The failure, while somewhat verbose, shows that raising `flag[tid]` does not establish mutual exclusion. we can see that the weakest-precondition view may contain a copy of `Lock` with a thread ID other than `tid`, and the goal is `Lock(x) * Lock(y)`.

One issue with the constraint system above is that it does not capture the intuition that, after acquiring `Checked`, we should have stable knowledge that the opposite thread has *not* raised its flag. A first try to encode this could involve the following change:

```
21 constraint Checked(tid) -> (tid == 0 || tid == 1)
22                          && !flag[tid] && !flag[(tid + 1)%2];
```

This change fixes the mutual exclusion failure, but opens a new failure:

```
Could not prove action: ASSIGN (select 8_1_tid flag), true
under weakest precondition:
  Checked((before 8_1_tid))
  Checked((goal 26 tid))
establishes: Checked((goal 26 tid))
```

This failure tells us that the new constraint is not stable. The specific situation shown corresponds to both threads checking each other (resulting in two instances of `Checked`), then one thread trying to raise its flag, immediately violating the other thread's `Checked`.

A possible solution is to rule out two threads being in `Checked` at the same time, which we can model by changing another constraint:

```
27 constraint Checked(x) * Checked(y) -> false;
```

Again, this replaces the previous failure with a new one:

```
Could not prove action:
  ASSIGN 11_9_other_flag, (select (% (+ 8_1_tid 1) 2) flag); ASSUME (not 11_9_other_wants)
under weakest precondition:
  Idle((before 8_1_tid))
  ((or
    (= (goal 66 x) (after 8_1_tid))
    (not (= (goal 66 y) (after 8_1_tid)))
  ) -> Checked((goal 66 y)))
  ((not (= (goal 66 x) (after 8_1_tid))) -> Checked((goal 66 x)))
establishes:
  Checked((goal 66 x))
  Checked((goal 66 y))
```

Like the first error, this represents a failure to achieve mutual exclusion (over `Checked`). This failure echoes Ben-Ari's explanation that the problem with this algorithm is that its critical sections begin immediately on threads exiting the check loop, not after raising `flag`.

8.2 Testing Starling$_{\text{tool}}$

Starling$_{\text{tool}}$ is, in many ways, a typical software engineering project, and so standard testing-based software validation methods apply. Though such methods cannot show the absence of bugs, only their presence [15], they prove to be useful for that purpose.

Case-study regression tests

When making changes to Starling$_{\text{tool}}$, we can cause changes (intentional or otherwise) to the tool's output. These changes can range from cosmetic (changes in output format, condition naming and order, etc.) to erroneous (non-termination, crashing), and even subtly unsound (false negatives and positives). To help us detect these cases, and make sure they only belong to the first class, we can use *regression tests*.

We can use our case studies as regression tests. To do so, the tool repository keeps a list of expected results (the names of each failing verification condition) for each Z3 and GRASShopper case study[7]. When executing the tests, a script checks each Z3 and GRASShopper case study in the tool directory. If the case study is not in the results list, the test fails; this helps us detect accidental omissions of test data.

Unit tests

While regression tests help us detect errors that manifest in $\text{Starling}_{\text{tool}}$'s output, they do not directly flag a particular part of the tool as the cause. For finer-grained validation, the $\text{Starling}_{\text{tool}}$ codebase contains *unit tests*. These tests check $\text{Starling}_{\text{tool}}$'s behaviour on a function-by-function basis, providing simplified sample input.

Some parts of $\text{Starling}_{\text{tool}}$ are unsuitable for comprehensive unit testing. This, together with the complexity of building useful unit tests, means that $\text{Starling}_{\text{tool}}$'s unit tests cover a smaller range of the code than the regression tests. Where unit tests do cover the code, though, we can detect errors faster and increase our confidence about the tool's correctness.

8.3 Mechanisation

While testing gives us confidence that the tool correctly implements the logic, it does not help us validate the logic itself; at the same time, formal reasoning about the tool's algorithms can significantly improve our validity argument. As a result, there exists a Coq mechanisation of core results from the Starling theory, as well as some algorithms used in $\text{Starling}_{\text{tool}}$. Certain dissertation definitions and results contain notes like this one:

(Coq: sub_by_dot in Starling.Views.Classes)

These link definitions with parts of the Coq mechanisation; the above example states that `Starling.Views.Classes.sub_by_dot` mechanises some (attached) definition[8]. The location of such notes gives an idea of which parts of Starling have successful mechanisations.

At time of writing, the mechanisation is available as Coq scripts at `https://gitlab.com/MattWindsor91/starling-coq`.

Mechanisation as validation

We mechanise the Starling theory to gain confidence in its soundness. The mechanisation, therefore, complements the tool (which witnesses Starling's automatability and practical usefulness). We describe ways in which the mechanisation validates our theory below.

Traceability to the Views Framework. By encoding the Views Framework in Coq (or by directly targeting its existing Coq mechanisation), we can prove that the Starling framework produces sound program logics by applying the Views mechanisation as a sub-theorem.

[7] HSF studies are not automatically regression-tested, as they can be slow to process.

[8] In digital copies of the dissertation, the file name serves as a link to an on-line copy of the mechanisation.

Local views. The mechanisation's treatment of the LVF is the main source of confidence we have in its soundness. For the most part, we created the LVF by trying to push $_{\text{lo}}$Starling through the normal *Views* framework in Coq, and solving the arising problems through a combination of insightful discussion[9] and interactive Coq sessions.

Termination of algorithms. Recursive programs written in Coq's programming language must carry termination proofs [72][7]. Therefore, by mechanising Starling$_{\text{tool}}$ algorithms and decision procedures in Coq, we can prove their termination.

For example, an old version of Starling$_{\text{tool}}$'s strategy for list-normalising guarded iterated abstract predicates[10] tried to subtract one atom from another without considering the value of their iterators. It did so by recursively subtracting any resulting remainders in a way that diverged in some situations. While our case studies and unit tests did not exercise those cases, Coq rejected an attempt to mechanise the strategy for reasons of non-termination — revealing the bug. Starling$_{\text{tool}}$ now uses a more conservative process that makes more assumptions about the iterators, but terminates correctly and works on our case studies.

Mechanisation as a proof tool

We can instantiate the mechanisation's backend interface directly using Coq's own predicate types (`Prop` for constructive logic, and `bool` for classical logic) and corresponding decision procedures based on the verification-condition Hoare judgement. This way, we can use the mechanised frontends, views frameworks, and their soundness relationships to carry out Starling proofs inside Coq.

(Coq: Starling.Backend.Instances.Prop) (Coq: Starling.Backend.Instances.Bool)

As the Starling method targets fully automated solvers, direct interactive proof of Starling verification conditions is intractable. This is because the method skews towards generating large amounts of small conditions.

8.4 Related work

This section discusses *existing* tools and languages for computer-assisted verification. It aims to compare and contrast the Starling approach, highlight relative strengths and weaknesses, and motivate potential future work.

SmallfootRG

SmallfootRG [24][Cp6] is a tool for verifying programs against the *RGSep* logic (see § 2.3). It has a similar purpose to Starling$_{\text{tool}}$: validation for a new reasoning system. Though RGSep captures interference and separation in a different and more elaborate way than $_{\text{g}}$Starling, we draw inspiration from parts of SmallfootRG's design.

[9] See the collaboration history for details.

[10] See also: `https://github.com/MattWindsor91/starling-tool/commit/faa7559`.

SmallfootRG asks the user to give a requires–ensures specification at the method level, with further annotations if necessary (including annotations on each atomic action that modifies state). Our approach — expecting a Hoare-style proof outline with view-level annotations at each sequence point — can demand more user annotation at the method level, but removes the need for explicit invariant and rely-condition annotation [2].

SmallfootRG symbolically executes programs to verify their specifications. Starling's use of solvers to quantify over states when verifying atomic triples has a similar flavour, but at a finer grain: each triple and goal gives rise to a symbolic execution in the solver, with Starling's focus being on building sets of easily-discharged verification conditions.

SmallfootRG's language, while self-admittedly toy, is a good balance between the mathematical rigour of RGSep's native GPPL and the practicality of a 'real' programming language. C_{view} aims for a similar effect, and re-uses some of SmallfootRG's control-flow syntax.

Caper

Caper [47] is a tool for automated verification of *CAP* proofs. As such, it exposes the typical CAP logical machinery (shared memory, guard algebras, and so on) to users. In contrast with $\text{Starling}_{\text{tool}}$, certain logical actions, like determining where to open shared regions, need interactive user input. While Caper uses Z3 as a backend, it adds a bespoke heap solver.

Though Caper and $\text{Starling}_{\text{tool}}$ have a shared heritage in CAP, the shape of their input is quite different. As an example, Listing 8.13 gives Caper's version of the ticket lock.

If we compare this to Listing 7.1, we see a more CAP-style proof process: the ticket lock forms a shared region `TLock`; the tickets themselves form a counting algebra `TICKET` of abstract guards; and the tickets both guard an interference transition system **`actions`** and participate in the concrete **`interpretation`** of the lock region. In C_{view}, all of these concepts map either to views or methods.

Comparing our work against Caper is insightful for several reasons. First, it shows the difference between a proof-outline based approach such as ours, and a requires-ensures-invariants approach; we argue that, while our approach can impose a larger burden for proof authors, it maps well (through outline decomposition) to the Starling workflow and neatly reflects the typical style of on-paper Concurrent Separation Logic-style proofs.

Second, we see that Caper, in targeting an existing logic designed for rich on-paper reasoning, brings expressivity — Caper proofs, for example, can reason about functional properties, such as whether stack push operations correctly insert the right element —, as well as modularity. It also imports a lot of conceptual overhead (shared regions, guard algebras, transitively closed action transition systems) and hard-to-automate logical machinery. In contrast, $\text{Starling}_{\text{tool}}$ exposes a more minimalistic proof environment designed specifically for automation, but struggles with functional properties and has no modularity support.

Threader

Threader [51] supports automatic Owicki-Gries and rely/guarantee reasoning over C. It implements both reasoning systems using recursive abstraction refinement and Horn-clause

Listing 8.13: Ticket lock: CAPER proof (via `https://github.com/caper-tool/caper/blob/1e88663/examples/iterative/TicketLock.t`)

```
region TLock(r,x) {
  guards #TICKET;
  interpretation {
    n : x |-> m &*& (x + 1) |-> n &*& r@TICKET{ k | k >= m }
      &*& m >= n;
 }
 actions { n < m | TICKET{ k | n <= k, k < m } : n ~> m; }
}
function makeLock()
  requires true; ensures TLock(r,ret,_);
{ v := alloc(2); [v + 0] := 0; [v + 1] := 0; return v; }
function acquire(x)
  requires TLock(r,x,_); ensures TLock(r,x,n) &*& r@TICKET(n);
{
  do { t := [x + 0]; b := CAS(x + 0, t, t + 1); }
  invariant TLock(r,x,ni) &*&
    (b = 0 ? true : r@TICKET(t) &*& t >= ni);
  while (b = 0);
  do { v := [x + 1]; }
  invariant TLock(r,x,ni) &*& r@TICKET(t) &*& t >= ni &*& ni >= v;
  while (v < t);
}
function release(x)
  requires TLock(r,x,n) &*& r@TICKET(n); ensures TLock(r,x,_);
{ v := [x + 1]; [x + 1] := v + 1; }
```

solving; this set-up allows it to infer invariants. This use of Horn clauses relates indirectly to our use of Gupta and Rybalchenko's later HSF project as a backend.

Like our set-up, Threader allows for non-thread-modular proofs, and its input is similar to real-world concurrent C. One major difference is that it assumes a fixed number of threads, each corresponding directly to a C function; our approach makes no such assumption as we never explicitly apply the LVF parallel rule. Another is that assertions in Threader take the form of free-form C `assert(expr);` statements over shared variables; this is more lightweight than our view-outline approach, and closer to how C programmers typically record such assertions, but limits the shape of expressible assertions.

Threader only supports discrete variables, as with $\text{Starling}_{\text{tool}}$ when combined with Z3 or HSF. When using $\text{Starling}_{\text{tool}}$ with GRASShopper, we can go beyond this limitation.

To compare and contrast Threader proofs against C_{view}, Listing 8.14 replicates its example proof for *Peterson*. The assertions in the critical section indirectly witness mutual exclusion: both threads entering their section at the same time may invalidate one of the assertions.

Other work

This section identifies several more loosely-related projects.

Listing 8.14: Peterson's algorithm: Threader proof (via `https://www.model.in.tum.de/~popeea/research/threader.html`)

```
int turn;
int x;
int flag1 = 0, flag2 = 0;
void thr1() {
  flag1 = 1;
  turn = 1;
  do {} while (flag2==1 && turn==1);
  // begin: critical section
  x = 0;
  assert(x<=0);
  // end: critical section
  flag1 = 0;
}
void thr2() {
  flag2 = 1;
  turn = 0;
  do {} while (flag1==1 && turn==0);
  // begin: critical section
  x = 1;
  assert (x>=1);
  // end: critical section
  flag2 = 0;
}
```

Boogie and its frontends. Some verification tools are not directly user-facing, but instead accept an *intermediate* language between high-level conditions and low-level solver input; verification tool builders can then target that language to reduce their workload.

Microsoft's *Boogie* [49] is one such system, designed for verifiers targeting imperative and object-oriented languages. Its frontends include the *Dafny* [73] sequential programming language, which can produce compiled programs targeting the .NET platform [74].

Boogie may be a good fit for $\text{Starling}_{\text{tool}}$ in the future. Targeting GRASShopper showed that there are parallels between the shape of verification conditions Starling logics produce and imperative sequential programs with requires-ensures specifications. As this parallel was not evident when $\text{Starling}_{\text{tool}}$'s development started — and, in many cases, the verification conditions fit directly into Z3's domain —, this remains as future work.

VeriFast. Instead of creating a new verification language, we can introduce formal reasoning to an existing one. A characteristic example is *VeriFast* [75, 76], a program verifier for C and Java. VeriFast verifies requires–ensures-style comment assertions over variables, arrays, heaps, fractional permissions, and *pthreads*-style multi-threading.

As with CAPER, comparing VeriFast with our approach reveals the effect of trade-offs. Certainly, VeriFast is a polished and highly expressive system that can verify properties of real-world C and Java code. However, we argue that this comes at the expense of high cognitive and annotation burden: VeriFast proofs contain many moving parts, such as lemma

functions and predicate opening and closing, that gStarling-style reasoning omits. It is also unclear as to how one can use VeriFast to reason about atomic-action concurrency: the focus appears to be on coarser synchronisation techniques such as mutexes.

This contrasts with C_{view} (which takes just enough cues from existing languages to be familiar). Targeting real-world languages, and capturing their semantics in terms of LVF programs, is by no means straightforward, and left to future work.

Theorem provers as verification languages. Another approach concerns building a domain-specific algorithm specification language in a theorem prover (usually Coq), then building a reasoning system on top using the prover's vernacular as a framework. *FCSL* [53], which targets fine-grained concurrency, and *Bedrock* [77], which targets reasoning about programs at the assembly level of abstraction, typify this approach. Another project, *Verifiable C* [57, 78], provides a Coq-embedded separation logic for C; this represents a richer (but interactive and sequential) reasoning system than gStarling, but also inspires parts of our mechanisation.

Though these approaches need more user input than ours, using theorem provers has advantages. These include expressivity; access to the prover's existing result and tactic library; and a smaller code-base that we must trust before accepting the resulting proofs.

8.5 Summary

This chapter discussed the validation of the gStarling logic and its derived tool ($Starling_{tool}$). It considered three approaches: a set of C_{view} case studies based on existing algorithms; a set of conventional unit tests integrated into $Starling_{tool}$; and a partial mechanisation of gStarling and its underlying Starling framework in Coq.

These results give us confidence that the Starling approach can verify properties of small, but realistic, concurrent algorithms. The approach fits well when said algorithms form a set of thread-local finite state automata, and when we can describe any inter-thread protocols as constraints on which combinations of thread states can occur simultaneously.

There remains more work to be done. The case studies consider small algorithms mostly concerning mutual-exclusion; we need more results on $Starling_{tool}$'s scalability to larger programs. The Coq mechanisation is partial: we need more results on iterated views, and a practical means to use the mechanisation as a stand-alone program logic. Code coverage in unit tests can be improved. This said, $Starling_{tool}$ (and, to an extent, gStarling) is a proof of concept for the Starling approach, and we can expect a large pool of arising future work.

Chapter 9

Conclusions and Further Work

> What the world needs are not more proofs of ten-line concurrent algorithms. The world needs some way of getting Bank of America to be able to eliminate those 95% of their crashes — some tool, some method, maybe some way of teaching programmers how to use the techniques that we already have, but some way of getting these proof methods out into the real world. I strongly advise people to knock on the doors of Bank of America and say. 'Hey, can we help you?'
>
> Leslie Lamport, in 1985 [13]

9.1 Conclusions

This dissertation covered:

- how to use *Views*'s axiom soundness to build templates for building *Views* axiomatisations, and, from them, instances customised to the proofs they underpin;
- how to use the *defining-views* template to reduce atomic Hoare triples to finite sets of verification conditions which we can discharge using sequential Hoare-logic tactics;
- Starling: a scheme for reducing *Views* proof outlines to such conditions, using a *frontend* that implements defining-views and a *backend* that implements Hoare reasoning;
- $_{\mu}$Starling, $_{\text{lo}}$Starling, and $_{\text{g}}$Starling: frontends that implement increasingly elaborate forms of defining-views, adding features such as guarded views and local state;
- the *Local Views Framework*, a layer on top of *Views* that allows for said local reasoning;
- a sketched extension to $_{\text{g}}$Starling adding *iterated views*, which parametrise the definition of a view atom on the number of copies of that atom held by a given thread;
- $\text{Starling}_{\text{tool}}$, a tool based on $_{\text{g}}$Starling, and C_{view}, the C-like proof language it accepts.

Though Starling's part-formalisation, both here and in the Coq developments, is voluminous in areas, the core ideas — defining-views, its implementation in $\text{Starling}_{\text{tool}}$, and its

informal justification against *Views* — are straightforward. The first stages of $\text{Starling}_{\text{tool}}$'s development form evidence that prototyping minimal tools atop Starling, then using them as a base for more complex tooling, works well.

Every part of the approach expands on *Views*, either directly or through Khyzha *et al.*'s work on generalised linearisability logics. *Views*'s focus on small, well-defined parameters that connect together to form sound program logics shaped Starling in many ways. While the most obvious example of this is the way in which we derived our axiomatisation templates, we also, for example, used *Views*'s focus on semigroups and monoids as inspiration for tying Starling's requirements on assertion languages used in proofs to a tightly-defined series of algebraic classes, providing a large amount of flexibility and future expandability.

While this dissertation did not fully explore the generality of the approach, it structured the approach in a modular, layered manner, expressing dependencies as minimal algebraic laws where possible. It explored several different ways to build Starling frontends, from the basic $_{\mu}$Starling to the more heavyweight $_{g}$Starling, as well as the frontend informally implemented by our tooling. By doing so, the dissertation intends to give evidence of Starling's flexibility and expandability, while keeping $\text{Starling}_{\text{tool}}$ as the main deliverable.

The approach can capture both Owicki-Gries reasoning and, through GRASShopper-style backends, an approximation of concurrent separation logic. This dissertation showed that our approach produces tooling that can prove interesting properties of real-world algorithms: mutual exclusion of Linux-style ticket locks and implementations of Peterson's algorithm; memory safety of Rust-style atomic reference counters; and so on.

9.2 Further work

This section highlights possible avenues for further work. It begins with the most promising future directions: modularity and inference. It then gives an outline of other possible work directions, grouped by the part of our work they extend.

Modularity

This dissertation gives no modularity results for Starling or its derived logics. This is a weakness in comparison to CAP (and CAPER), where modularity is one of the main features.

A modularity story would improve the practical usefulness of C_{view} and $\text{Starling}_{\text{tool}}$. While we can prove properties of small components of concurrent systems, such as locks and reference counters, trying to prove algorithms that use those components without modularity requires either textually inlining proofs (resulting in poor scaling) or leaving gaps in the proof (resulting in an unconvincing correctness argument). For instance, while we explore the ticket lock *implementations* in § 8.1, the CAPER paper [47] gives proofs of both implementations *and* clients, where the latter just relies on the abstract specification of the former.

Disjoint state and views. Suppose that we just considered connecting fully disjoint proofs, where there is no sharing of views or state. This would give us a limited form of modularity where we can join proofs of separate systems into single proofs without further proof-work.

We expect that this form of modularity would just amount to applying *Views*'s existing framing properties across the proofs, and be trivial to show.

Disjoint state but shared views. A more realistic target is proof joining where, while the proofs' shared states remain disjoint, we allow abstract information to cross between the proofs in the form of views. In the simplest case, the information moving across would just be view atoms; such a scheme can capture situations where we need to thread a permission to use an external module through a long function.

To support work-flows such as lock clients where we depend on the external lock guaranteeing mutual exclusion over the client's resource, we would need to be able to import *constraints* from the external module. In a disjoint-state scenario, these constraints would need to have no shared-state dependencies. By importing a subset of the original constraints, we can build a CAP [46]-style modularity story where we abstract over the meaning of the external views before using them around calls into the external code.

When sharing views, we must ensure that clients of an external view cannot use that view in ways that violate the external module's constraints. If we imported a *Lock* view from a lock proof but none of its constraints, the triple $\langle(\textbf{@}\mathit{Lock})\rangle$ id $\langle(\bullet\ (\textbf{@}\mathit{Lock})\ (\textbf{@}\mathit{Lock}))\rangle$ may be valid in the client but would violate mutual exclusion in the lock.

Clients generally do not know which constraints exist in the external module. As a result, we hypothesise that the set of safe operations a client can do with external views is to forget them, or receive and consume them in calls into the external module. We can loosen this restriction by constraining the possible external constraints on a shared view: for instance, by specifying that the imported constraint set is precisely the set in the external module. Finding a balance between abstraction and expressivity in this area remains future work.

Overlapping state. An ideal modularity system would permit the sharing of state as well as views. We would then need to handle the interface of possible shared-state interference between modules as well as the possibility of proof violation by incorrect use of shared views. This may take the form of a rely/guarantee-style model: by adding new actions to the proof summarising the set of possible interactions in a foreign module, we can approximate a rely; by importing views from a foreign module's proof, we can approximate a guarantee.

Inference

While Starling$_{\text{tool}}$ has rudimentary support for definition inference using HSF, we can improve Starling$_{\text{tool}}$'s inference story in a variety of ways.

Initial states. Starling$_{\text{tool}}$'s HSF-based inference set-up requires that, if we set every shared variable to 0, we obey **emp**. This makes using inference in situations where 0 is not a valid starting value for one or more variables cumbersome, but should be straightforward to fix.

Stabilising definitions. SmallfootRG [24] can automatically strengthen unstable definitions to stabilise them. We may be able to support a similar tactic in HSF/Starling$_{\text{tool}}$ by mapping

such definitions to their atoms in one direction ($\Longrightarrow$) rather than two ($\Longleftrightarrow$). As view definitions appear in both positive and negative positions, this form of inference is not guaranteed to be monotone, and we may also need to consider the ability to *weaken* definitions.

Improving assertion-level inference. The '`?`' syntax for assertion-level inference always creates a fresh view, which is not always an efficient solution. Sometimes, we know that the view we need is some combination of the *existing* view atoms, but do not necessarily know what the combination is — inference need just choose the correct set of atoms to put in the gap. This form of inference is different from that implemented by $\text{Starling}_{\text{tool}}$/HSF, as it involves objects — atoms — that exist at a higher level than HSF natively understands.

A straightforward first approach to inferring combinations of existing atoms may be to try heuristics such as inserting **`emp`**, or the previous assertion, or a bounded iterative deepening search of atom combinations. These approaches would not need specific backend support, but would be incomplete (for example, we cannot exhaustively consider all combinations of atoms, or even all mappings of local-state parameters in a single atom), and it is unclear how they would perform in practice. More sophisticated forms of inference may be possible.

In other cases, we know that the inferred assertion contains some existing atoms, and just want the inferrer to strengthen those atoms into a valid assertion. We may model this explicitly in the outline as `SomeView * ?`, at which point the future work mainly just concerns $\text{Starling}_{\text{tool}}$ infrastructure. A more sophisticated form of inference may be to do so automatically on failing proof triples in an attempt to correct the proof.

On Starling and its frontends

This section outlines smaller-scale work avenues over the dissertation's theory contributions.

Sequential consistency and weak memory. This dissertation's contributions assume that atomic operations are *sequentially consistent*; as such, atomic writes propagate immediately to future reads. Multi-core systems rarely have native sequentially consistent atomic actions. Instead, their memory models are *weak*: writes can be reordered, buffered [6, §7.1], and otherwise tampered with on their way to corresponding reads. However, we can regain sequential consistency (at the cost of performance) by inserting fence instructions.

Much research in concurrent separation logics has involved weak memory models [79, 80, 81]. As such, the extension of this dissertation's contributions to follow suit would be a natural further work avenue. It is unclear whether weak memory models would fit in the CVF/LVF language semantics, where atomic writes do indeed propagate immediately to the shared-state model; if an encoding is not possible, it is unclear how to construct a *Views*-style program logic over a weak-memory semantics.

Parallel composition. The LVF adds local-state reasoning by Khyzha *et al.*'s approach of treating programs as an outer parallel composition of threads with specific identifiers [56]. To regain inner parallelism in the LVF and C_{view} languages, we would need to track the splitting and merging of local states)when such compositions begin and end) in their semantics.

Having a concurrency model based on Dijkstra-style **`cobegin–coend`** parallelism means we cannot feasibly express fork/join concurrency. Logics such as *deny-guarantee reasoning* [82] support such constructs; the *Views* logic does not, requiring us to extend *Views* with such support or loosen Starling's dependency on *Views*.

Linearisability. Starling has no native linearisability support. This mainly results from its basis in *Views*'s Floyd/Hoare-style model, and its aim of proving incremental safety properties. Indeed, the LVF results from taking work intended for linearisability proof (the GLL [56]) and paring it down to fit the existing CVF model.

An automation-friendly framework for linearisability proof, based on re-targeting Starling to the GLL, would be an interesting future direction. It is unclear how we can automate the GLL action judgement; much of our work in Chapter 3 comes from the observation that the CVF judgement lines up well with Owicki–Gries-style reasoning, and there is not yet a similar form of intuition available for the GLL. Also, the GLL judgement is more complex than the CVF one, involving tracking of linearisation points.

Soundness proofs. While the Coq development has soundness arguments for $_{\mu}$Starling and $_{\text{lo}}$Starling, it has no arguments for $_{\text{g}}$Starling or its extensions. It also has no results for Starling$_{\text{tool}}$'s frontend. These omissions threaten the validation argument in Chapter 8.

The main reason for the lack of $_{\text{g}}$Starling soundness argument is time constraint; at time of writing, a soundness proof in Coq has been started, but needs much work. Issues that have made work on the soundness proof slow and challenging include, on the mechanisation side: problems with typeclass inference (the means by which the Coq mechanisation structures use of algebra classes) that mean that automatic resolution fails to terminate; Coq's separation of propositions and computable types into distinct universes, which has led to universe mismatches as parts of the meta-theory disagree on whether, say, views can be propositions; and other design decisions taken early in the mechanisation that have not paid off.

The attempted soundness proof relies on the same logical pathways as the $_{\text{lo}}$Starling argument, but correspondence between $_{\text{g}}$Starling and the LVF is more subtle than that of $_{\text{lo}}$Starling, making re-use of the same machinery challenging. This involves the need to erase local and goal variables, and show correspondences between $_{\text{g}}$Starling's pattern-based reifier and $_{\text{lo}}$Starling's more direct reifier. These steps have been harder to justify than expected.

As well as the further work of finishing the soundness proofs, a general work avenue lies in finding a more compositional soundness scheme for Starling instances. The current CVF-based scheme does not fully respect the boundaries between outline decomposition, frontend, and backend, with tight coupling between the three. The encoding of local-variable frontends into the soundness argument requires some mapping from the CVF to the LVF, which adds confusing indirection into the proof; work to recast this would be useful.

Completeness proofs. None of the frontends have completeness proofs. This is less important than soundness (the case studies, targeting a variant of $_{\text{g}}$Starling, serve as evidence that the approach can express a variety of proofs), but is still a gap in the Starling formalisation.

One possible tactic is to encode Owicki-Gries into the frontends; as de Roever *et al.* give a completeness proof for Owicki-Gries [25], this would witness the frontends' completeness.

View partitioning. In theory, it should be possible to split a view definer into partitions based on whether the definitions of certain views refer to disjoint parts of state, and treat each partition as a separate sub-semigroup in a views semigroup product. This would help make defining-view proofs more incremental, as we can prove each subgroup separately.

Refinement. As in the UTP [61], propositional expressions form a lattice: true is the most permissive proposition, false is the least, and $X \leqslant Y$ if each state satisfying X also satisfies Y. We can form a similar lattice over relation expressions, using $\subseteq$ over their sets of pairs. This lattice model gives us a refinement rule across verification conditions:

$$\frac{\langle\!\langle w \rangle\!\rangle \ c \ \langle\!\langle g \rangle\!\rangle \qquad w \leqslant w \qquad c \leqslant c \qquad g' \leqslant g}{\langle\!\langle w' \rangle\!\rangle \ c' \ \langle\!\langle g' \rangle\!\rangle}$$

Since both w and g contain the same view in opposite positions, we cannot easily refine views at the atomic Hoare triple level. Atomic-action refinement may work, though, letting users write proofs on abstract actions and programs and get free proofs over concrete refinements thereof. This may help with modularity: if we weaken the specification of atomic actions from an external proof into vague 'guarantees' of interference, we can reduce the client proof's coupling to that external proof's details.

On Starling$_{\text{tool}}$, and towards adoption

The quote at the chapter head was, and is, a powerful motivating call for practical, adoptable concurrency verification methods. That the tool shown in this dissertation proves small concurrent algorithms and *not* the safety of industrial-strength systems is, of course, a strong limiting factor on its immediate usefulness. While the status of the tool as a proof of a more abstract concept (the usefulness of Starling) mitigates this issue, there remains much work to be done to address Lamport's concerns in 1985, let alone those of practitioners in 2019.

The future work paths below concern the tool itself. A general theme is to improve the usefulness of Starling$_{\text{tool}}$ in a real-world setting, and drive adoption of the tool: this leads us to consider targeting real-world programming languages and improving error reporting. Improvements in modularity and inference would also move Starling$_{\text{tool}}$ towards practicality: modularity would make it easier to verify large-scale systems along module boundaries, and inference would reduce the workload of Starling$_{\text{tool}}$ users when writing proofs.

Implementation language support. While C$_{\text{view}}$ (a thin layer on the Starling theory) suffices for verifying algorithms, basing Starling$_{\text{tool}}$ on a 'real' language such as Ada or C would help in verifying implementations (to use Lamport's example, the code-base of Bank of America). This would bring Starling$_{\text{tool}}$ closer to systems like Threader [51] and VeriFast [75]. Doing so would add complexity, and require a way to encode assertions in the target language.

Code extraction. Another way to verify *implementations* with $\text{Starling}_{\text{tool}}$ would be to support extraction of program code from proofs: either by $\text{Starling}_{\text{tool}}$ working as a compiler for C_{view} to machine code or an intermediate form such as LLVM IR (or, as with Dafny [74], .NET), or by extraction into a language such as C or Rust. While extracting code from a C_{view} proof would be straightforward, showing that the extraction preserves semantics is not, especially with languages such as C where the semantics is often ill-defined.

Error reporting. $\text{Starling}_{\text{tool}}$'s proof-error reporting is at the verification-condition level, and exposes the underlying defining-views rule. This reporting can be difficult to map back to the original proof, as the views in each verification condition combine both proof assertions and external contexts. This is problematic when the proof failure concerns the sequential safety of the failing command — such errors should be more straightforward to debug and fix than concurrency failures, but $\text{Starling}_{\text{tool}}$ treats the two classes identically.

We can add a step to $\text{Starling}_{\text{tool}}$'s error handling that checks the sequential-safety verification condition ($\langle p \rangle\ c\ \langle q \rangle$) and reports resulting failures distinctly. This effectively changes the monoidal defining-views rule to the semigroup form, and so forms a compatible (but slower) proof rule. As such, we can limit it to running on failing goal axioms.

Other backends. Adding more backends to $\text{Starling}_{\text{tool}}$ would provide more evidence that the idea of separating backend from frontend and outline flattener in the Starling framework is a useful engineering decision. Possible backends that could be targeted with minimal changes to the rest of $\text{Starling}_{\text{tool}}$ include Boogie [49].

While Starling broadly assumes that backends target some theory of sequential safety, it may be possible — with appropriate changes to the framework meta-theory, and perhaps the frontend — to achieve stronger guarantees by targeting other forms of solver. We could, for instance, target bounded model checkers to sacrifice soundness for ease of automation.

One exotic possibility would be to investigate the encoding of Starling proofs in a process algebra such as CSP, using tools such as *FDR* [83] as a backend. Such systems support refinement checking over semantic models that can express liveness properties as well as safety properties (for instance, failures–divergences refinement), which would address a significant weakness of the safety-only $\text{Starling}_{\text{tool}}$ system. It is, however, unclear how such a system would reconcile with both the Starling framework, which skews towards Floyd/Hoare-style reasoning over safety properties, and the CVF and LVF, which only guarantee soundness for such forms of reasoning.

Multi-backend proofs. While $\text{Starling}_{\text{tool}}$/GRASShopper discharges under-approximations of verification conditions using Z3 where possible to reduce the workload sent to the more heavyweight solver, it does not support the general combination of solvers in a proof. This stops us from, for example, reasoning about a proof's heap in GRASShopper while applying inference to its shared-variable components using HSF, or using HSF's inference at the same time as Z3's arrays and rich type support. Finding a way to partition proofs into regions that different solvers can discharge, or allowing solvers to collaborate, is future work.

Further case studies

While chapter 8 gives examples of Starling$_{\text{tool}}$'s use, we can improve the case for Starling and Starling$_{\text{tool}}$ by verifying more examples. We can give more examples outside of the realm of mutual exclusion, and of tricky concurrency protocols.

Lock clients. CAPER's case studies include toy lock clients, which use externally-proven locks to enforce mutual exclusion. With a practical modularity story (see above), proofs of such clients should become possible in Starling$_{\text{tool}}$, and would be a natural way to exercise our approach. Such proofs would also witness the ability to prove properties of realistically-sized concurrent programs, more so if we base the lock clients on real-world algorithms.

Time-stamped stack. An early goal of Starling$_{\text{tool}}$ was to prove properties (ideally linearisability) of racy lock-free concurrent data structures such as the *time-stamped stack* [84]. This stack delays the total ordering of pushed items to the point of popping by assigning each item a time-stamp on push, maintaining each stamped item in a set of thread-specific single-producer pools, and comparing time-stamps on pop. This gives good performance, but needs a non-trivial linearisability argument that cannot use syntactic linearisation points.

While initial work occurred to build memory safety proofs of the single-producer pool, a Starling$_{\text{tool}}$ port remains as future work, in part because of time constraints. A linearisability proof remains ambitious: we would need to extend Starling$_{\text{tool}}$ to support such proofs (see above), and tackle the specific difficulties of showing time-stamped stack linearisability.

Appendix A

Additional Definitions

This appendix contains extra definitions, derivations, and other related items. These items are, generally, either too large to give in the main text, or are specific phrasings or developments of well-known mathematical ideas given for completeness.

A.1 Trivial definitions

Definition A.1. A *setoid* $(A, \equiv)$ is a set A with an equivalence relation $\equiv$.
(Coq: Setoid in Coq.Classes.SetoidClass)

Definition A.2 (Constant function)**.** $\mathsf{const} \stackrel{\text{def}}{=} \lambda x.\, \lambda y.\, x$.

Definition A.3. A *multiset* $m : \text{bag } T$, over some element type T, is a function $T \to \mathbb{N}$ from items in type T to the number of times they occur (their *multiplicity*).
(Coq: multiset in Coq.Sets.Multiset)

Definition A.4 (Truncated subtraction)**.** $\forall x, y : \mathbb{N}.\, x \dot{-} y \stackrel{\text{def}}{=} \mathsf{max}(0, x - y)$.

Definition A.5. Given a list l of length n, we define the *list override* $l[i \mapsto x]$ as:

$$l[i \mapsto x] \stackrel{\text{def}}{=} \begin{cases} \langle l[0], \ldots, l[i-1] \rangle +\!\!+ \langle x \rangle +\!\!+ \langle l[i+1], \ldots, l[n] \rangle & i \text{ in bounds} \\ l & \text{otherwise} \end{cases}$$

In the case of thread lists, i is in bounds if, and only if, $i \in \mathsf{Tid}$.
(Coq: list_override in Starling.Utils.List.Override)

A.2 Backend interfaces

Compositional backends

We do not, in general, assume any relationship between the success of Solve on two verification condition sets X and Y, and the success of Solve on $X \cup Y$. This is because adding

or removing verification conditions could change the requirements on the solver context. These changes are not guaranteed to lead to a valid proof. For example, in a constraint solver backend, adding or removing verification conditions will add or remove constraints in the constraint system, which could make the problem underspecified or unsatisfiable.

While this lack of compositionality does not stop us from proving proof outlines as closed systems, it makes it hard to apply the laws of the *Views* logic to them. To fix this, we add a class of *compositional* backends, whose use will permit us to apply the logic rules as normal.

Definition A.6. A *compositional backend* is a tuple $\langle \mathsf{E_{Pr}}, \mathsf{E_{Rl}}, \mathsf{S}, \mathsf{Solve}, \mathsf{GCtx} \rangle$, where said tuple is a backend and, for all X and Y, $\mathsf{Solve}(X \cup Y) \iff \mathsf{Solve}(X) \wedge \mathsf{Solve}(Y)$.

Decidability

To automate $_{\mu}$Starling, we must show that certain predicates, such as view inclusion, are decidable. Specifically, for a predicate $P(x)$, we show that some decision process exists that always terminates with a result of either true ($P(x)$ holds) or false ($P(x)$ does not hold). This distinction, which mirrors the way decidability works in Coq and other constructive logics, helps stop us from accidentally building a logic we cannot implement as a tool.

In § 4.3, we pointed out the need for decidable forms of ordering and equivalence over V. For any two views u and v, we must be able to compute that either $u \sqsubseteq v$ or $\neg(u \sqsubseteq v)$, and likewise for $\equiv$. We formalise this as a new class of views semigroup.

Definition A.7. An algebra $(V, \bullet, \sqsubseteq, \equiv, \mathsf{Inc})$ is a *decidably ordered views semigroup* if $(V, \bullet, \sqsubseteq, \equiv)$ is an ordered views semigroup, and Inc is a function $V \to V \to \mathbb{B}$ where:

$$\forall x, y.\, \mathsf{Inc}(x)(y) = \mathsf{true} \iff x \sqsubseteq y$$

(Coq: DecOrderedViewsSemigroup in Starling.Views.Classes)

If we have decidable order, we have decidable equivalence: as $x \equiv y \iff x \sqsubseteq y \wedge y \sqsubseteq x$, we can define the decidable witness for equivalence as $\lambda x.\, \lambda y.\, (\mathsf{Inc}(x, y) \wedge \mathsf{Inc}(y, x)) = \mathsf{true}$.

Solver functions

Definition 4.8 does not require the solver to be able to distinguish between failed proofs and proofs where correctness cannot be determined. We can model the ability of solvers to do so using a *solver function*.

Definition A.8. A *solver function* $\mathsf{SolveF} : \mathbb{P}(\mathsf{VConds}(\mathsf{E_{Pr}}, \mathsf{E_{Rl}})) \nrightarrow \mathbb{B}$ decides whether a configuration exists in the backend solver such that the given set of verification conditions is correct. For all $P \in \mathrm{dom}\ \mathsf{SolveF}$, SolveF must obey the following rules:

$$\mathsf{SolveF}(P) \implies \exists x_g.\, \forall \langle\!\langle w \rangle\!\rangle\ c\ \langle\!\langle g \rangle\!\rangle \in P. \quad (x_g, c) \Vdash_{\mathsf{EVFH}} \{w\}\{g\}$$
$$\neg\mathsf{SolveF}(P) \implies \forall x_g.\, \exists \langle\!\langle w \rangle\!\rangle\ c\ \langle\!\langle g \rangle\!\rangle \in P.\, \neg((x_g, c) \Vdash_{\mathsf{EVFH}} \{w\}\{g\})$$

To use solver functions as solver predicates, let $\mathsf{Solve}(V) = V \in \mathrm{dom}\ \mathsf{SolveF} \wedge \mathsf{SolveF}(V)$.

A.3 Syntactic definers

List syntactic definers

We can implement syntactic definers as finite lists of individual pairs of view and associated proposition expression.

Definition A.9. A *list syntactic definer* $\mathrm{d} \in \mathsf{Defn}(V, \mathsf{E}_{\mathsf{Pr}})$ is a finite sequence of pairs (v, e), where $v : V$ and $e : \mathsf{E}_{\mathsf{Pr}}$.

Syntactic definers can assign multiple definitions to a view, and can give different definitions for views that are equivalent but not equal. This means we cannot define the definition of a view as the result of finding the first matching definition pair. We can instead use a similar process to the syntactic reification, but matching on equivalence rather than inclusion.

Definition A.10. The *syntactic definer definition* $\mathsf{sdDef} : \mathsf{Defn}(V, \mathsf{E}_{\mathsf{Pr}}) \to V \to \mathsf{E}_{\mathsf{Pr}}$ is the function defined by the recursion:

$$\mathsf{sdDef}([])(v) = \mathrm{true}_{\mathsf{Pr}}$$

$$\mathsf{sdDef}([(u, e)] +\!\!+ \mathrm{d})(v) = \begin{cases} e \wedge_{\mathsf{Pr}} \mathsf{sdDef}(\mathrm{d})(v) & u \equiv v \\ \mathsf{sdDef}(\mathrm{d})(v) & \text{otherwise} \end{cases}$$

(Coq: sd_syn_define_expr in Starling.Frontend.SynDefiner)

Given a definer d, we can lift sdDef to a definer function, per Definition 3.10:

$$\lambda v. \begin{cases} \mathsf{sdDef}(\mathrm{d})(v) & \exists (u, e), \mathrm{d}_1, \mathrm{d}_2 . v \equiv u \wedge \mathrm{d} = \mathrm{d}_1 +\!\!+ \langle (u, e) \rangle +\!\!+ \mathrm{d}_2 \\ \text{undefined} & \text{otherwise} \end{cases}$$

The definer and reifier operations differ only by the condition on which u matches v, so we can write them in terms of a single collector function:

$$\mathsf{sdCollect}(\bullet)(\langle\rangle)(v) = \mathrm{true}_{\mathsf{Pr}}$$

$$\mathsf{sdCollect}(\bullet)(\langle (u, e) \rangle +\!\!+ \mathrm{d})(v) = \begin{cases} e \wedge_{\mathsf{Pr}} \mathsf{sdCollect}(\mathrm{f})(\mathrm{d})(v) & u \bullet v \\ \mathsf{sdCollect}(\mathrm{f})(\mathrm{d})(v) & \text{otherwise} \end{cases}$$

$$\mathsf{sdDef} = \mathsf{sdCollect}(\equiv)$$

$$\mathsf{sdReify} = \mathsf{sdCollect}(\sqsubseteq)$$

A.4 Derivation of outline flattening

Atomic actions. Atomic actions decompose into themselves, as follows:

$$\mathsf{oflat}(\{\mathrm{p}\}\ \langle \mathrm{c} \rangle\ \{\mathrm{q}\}) = \{\, \langle \mathrm{p} \rangle\ \mathrm{c}\ \langle \mathrm{q} \rangle \,\}$$

Skip. Skip commands form the axiom $\{p\}\ \texttt{skip}\ \{p\}$ in the *Views* program logic. Thus, to prove $\{p\}\ \texttt{skip}\ \{q\}$ for arbitrary q, we apply consequence[1] to show that p entails q:

$$\mathsf{oflat}(\{p\}\ \texttt{skip}\ \{q\}) = \{\, \langle p \rangle\ \mathrm{id}\ \langle q \rangle \,\}$$

Since skip introduces a consequence into our decomposition without changing the program semantics, we can use it to insert arbitrary consequence steps into our proof.

Iteration. Recall that iterations correspond to the following proof rule:

$$\frac{\{p\}\ c\ \{p\}}{\{p\}\ c^*\ \{p\}}$$

Suppose we want to prove the following outline, for arbitrary p, p', q', and q:

$$\{p\}\ (\{p'\}\ c\ \{q'\})^*\ \{q\}$$

One way to do so is as follows:

$$\dfrac{\langle p \rangle\ \mathrm{id}\ \langle p' \rangle \in T \qquad \dfrac{\dfrac{\dfrac{\vdash \{p'\}\ c\ \{q'\} \qquad \langle q' \rangle\ \mathrm{id}\ \langle p' \rangle \in T}{\vdash \{p'\}\ c\ \{p'\}}\ \text{RC}}{\vdash \{p'\}\ c^*\ \{p'\}}\ \text{It.} \qquad \langle p' \rangle\ \mathrm{id}\ \langle q \rangle \in T}{\vdash \{p'\}\ c^*\ \{q\}}\ \text{RC}}{\vdash \{p\}\ c^*\ \{q\}}\ \text{LC}$$

This gives us the following decomposition:

$$\mathsf{oflat}(\{p\}\ (\{p'\}\ c\ \{q'\})^*\ \{q\}) = \mathsf{oflat}(\{p'\}\ c\ \{q'\}) \cup \{\, \langle p \rangle\ \mathrm{id}\ \langle p' \rangle, \langle p' \rangle\ \mathrm{id}\ \langle q \rangle, \langle q' \rangle\ \mathrm{id}\ \langle p' \rangle \,\}$$

Intuitively, the consequence applications correspond to entering the loop, exiting the loop, and iterating on the loop.

Sequential composition. Recall the proof rule for sequential composition:

$$\frac{\{p\}\ c1\ \{r\} \qquad \{r\}\ c2\ \{q\}}{\{p\}\ c1;\ c2\ \{q\}}$$

Suppose we aim to prove the following outline, for arbitrary p, p', r, s, q', and q:

$$\{p\}\ (\{p'\}\ c\ \{r\};\{s\}\ d\ \{q'\})\ \{q\}$$

We can do so as follows[2]:

$$\dfrac{\langle p \rangle\ \mathrm{id}\ \langle p' \rangle \in T \qquad \dfrac{\dfrac{\vdash \{p'\}\ c\ \{r\} \qquad \dfrac{\langle r \rangle\ \mathrm{id}\ \langle s \rangle \in T \qquad \vdash \{s\}\ d\ \{q'\}}{\vdash \{r\}\ d\ \{q'\}}}{\vdash \{p'\}\ c;d\ \{q'\}} \qquad \langle q' \rangle\ \mathrm{id}\ \langle q \rangle \in T}{\vdash \{p'\}\ c;d\ \{q\}}}{\vdash \{p\}\ c;d\ \{q\}}$$

This gives us the following decomposition:

$$\begin{aligned}\mathsf{oflat}(\{p\}\ (\{p'\}\ c\ \{r\};\{s\}\ d\ \{q'\})\ \{q\}) = \quad & \mathsf{oflat}(\{p'\}\ c\ \{r\}) \\ \cup\ & \mathsf{oflat}(\{s\}\ d\ \{q'\}) \\ \cup\ & \{\, \langle p \rangle\ \mathrm{id}\ \langle p' \rangle, \langle r \rangle\ \mathrm{id}\ \langle s \rangle, \langle q' \rangle\ \mathrm{id}\ \langle q \rangle \,\}\end{aligned}$$

[1] Whether we apply left or right consequence makes no difference.

[2] We can, instead, apply right-consequence to $\{p'\}\ c\ \{r\}$ — but we still get $\langle r \rangle\ \mathrm{id}\ \langle s \rangle$.

Parallel composition. Recall the proof rule for parallel composition:

$$\frac{\{p_1\}\ c1\ \{q_1\} \qquad \{p_2\}\ c2\ \{q_2\}}{\{p_1 \bullet p_2\}\ c1\ \ ||\ \ c2\ \{q_1 \bullet q_2\}}$$

Consider the following outline, for arbitrary p, p_1, p_2, q, q_1, and q_2:

$$\{p\}\ (\{p_1\}\ c\ \{q_1\} \| \{p_2\}\ d\ \{q_2\})\ \{q\}$$

The proof tree for this outline is as follows:

$$\dfrac{\dfrac{\vdash \{p_1\}\ c\ \{q_1\} \qquad \vdash \{p_2\}\ d\ \{q_2\}}{\vdash \{p_1 \bullet p_2\}\ c\ \ ||\ \ d\ \{q_1 \bullet q_2\}} \qquad \langle q_1 \bullet q_2\rangle\ \mathrm{id}\ \langle q\rangle \in T}{\dfrac{\langle p\rangle\ \mathrm{id}\ \langle p_1 \bullet p_2\rangle \in T \qquad \vdash \{p_1 \bullet p_2\}\ c\ \ ||\ \ d\ \{q\}}{\vdash \{p\}\ c\ \ ||\ \ d\ \{q\}}}$$

This gives us the following decomposition:

$$\begin{aligned}\mathsf{oflat}(\{p\}\ (\{p_1\}\ c\ \{q_1\} \| \{p_2\}\ d\ \{q_2\})\ \{q\}) = \quad & \mathsf{oflat}(\{p_1\}\ c\ \{q_1\}) \\ \cup\ & \mathsf{oflat}(\{p_2\}\ d\ \{q_2\}) \\ \cup\ & \{\langle p\rangle\ \mathrm{id}\ \langle p_1 \bullet p_2\rangle, \langle q_1 \bullet q_2\rangle\ \mathrm{id}\ \langle q\rangle\}\end{aligned}$$

Unlike the earlier control flows, the decomposition for parallel composition mentions views — $p_1 \bullet p_2$ and $q_1 \bullet q_2$ — that are not expressed inside the proof outline.

Nondeterministic choice. Recall the proof rule for nondeterministic choice:

$$\frac{\{p\}\ c1\ \{q\} \qquad \{p\}\ c2\ \{q\}}{\{p\}\ c1\ +\ c2\ \{q\}}$$

Consider the following outline, for arbitrary p, p_1, p_2, q, q_1, and q_2:

$$\{p\}\ (\{p_1\}\ c\ \{q_1\} + \{p_2\}\ d\ \{q_2\})\ \{q\}$$

One proof tree (which we split for space reasons) for this outline is as follows:

$$\dfrac{\dfrac{\langle p\rangle\ \mathrm{id}\ \langle p_1\rangle \in T \qquad \vdash \{p_1\}\ c\ \{q_1\}}{\vdash \{p\}\ c\ \{q_1\}} \qquad \langle q_1\rangle\ \mathrm{id}\ \langle q\rangle \in T}{\dfrac{\vdash \{p\}\ c\ \{q\}}{\text{(A)}}}$$

$$\dfrac{\dfrac{\langle p\rangle\ \mathrm{id}\ \langle p_2\rangle \in T \qquad \vdash \{p_2\}\ d\ \{q_2\}}{\vdash \{p\}\ c\ \{q_2\}} \qquad \langle q_2\rangle\ \mathrm{id}\ \langle q\rangle \in T}{\dfrac{\vdash \{p\}\ d\ \{q\}}{\text{(B)}}}$$

$$\frac{\text{(A)} \qquad \text{(B)}}{\vdash \{p\}\ c\ +\ d\ \{q\}}$$

This gives us the final decomposition:

$$\begin{aligned}\mathsf{oflat}(\{p\}\ (\{p_1\}\ c\ \{q_1\} + \{p_2\}\ d\ \{q_2\})\ \{q\}) = \quad & \mathsf{oflat}(\{p_1\}\ c\ \{q_1\}) \\ \cup\ & \{\langle p\rangle\ \mathrm{id}\ \langle p_1\rangle, \langle q_1\rangle\ \mathrm{id}\ \langle q\rangle\} \\ \cup\ & \mathsf{oflat}(\{p_2\}\ d\ \{q_2\}) \\ \cup\ & \{\langle p\rangle\ \mathrm{id}\ \langle p_2\rangle, \langle q_2\rangle\ \mathrm{id}\ \langle q\rangle\}\end{aligned}$$

A.5 Operations on view lists

View list combine

Definition A.11. Given a local view list $\boldsymbol{\nu}$ and a parallel local state list $\mathbf{l}$, the *view list combine* $\mathsf{Vlc}(\boldsymbol{\nu}, \mathbf{l})$ is the recursion:

$$\mathsf{Vlc}(\langle\rangle, \langle\rangle) = \langle\rangle \qquad \mathsf{Vlc}(\langle \nu \rangle +\!\!+ \boldsymbol{\nu}', \langle \mathrm{l} \rangle +\!\!+ \mathbf{l}') = \langle \nu(\mathrm{l}) \rangle +\!\!+ \mathsf{Vlc}(\boldsymbol{\nu}', \mathbf{l}')$$

Iterated view join

Definition A.12. The *iterated view join* $\odot : \mathrm{list} V \to V$, where V is the carrier of a views semigroup with join $\bullet$ and optional unit ε, is the recursion:

$$\odot(\langle\rangle) = \varepsilon \qquad \odot(\langle \nu \rangle) = \nu \qquad \odot(\langle \nu \rangle +\!\!+ \boldsymbol{\nu}) = \nu \bullet \odot(\boldsymbol{\nu})$$

The iterated view join is undefined on the empty list if V is not carrier of a views monoid.

A.6 $\mathsf{C}_{\mathsf{view}}$

This section contains more information about $\mathsf{C}_{\mathsf{view}}$ (§ 7.1).

Expressions

$\mathsf{C}_{\mathsf{view}}$ makes no syntactic distinction between predicate and value expressions; all valid **bool** expressions, over appropriate variable sets, are valid proposition expressions.

$\mathsf{C}_{\mathsf{view}}$ uses C's expression language, with some changes. C operators that produce side effects — ++, −−, =, and so on — are either statements in $\mathsf{C}_{\mathsf{view}}$ or omitted entirely. This change prevents various classes of errors, for example confusion between = and ==.

$\mathsf{C}_{\mathsf{view}}$ forbids Boolean expressions in arithmetic positions[3]. This means that chaining of relational expressions, such as `x < y <= z`, no longer has its C meaning of evaluating `x < y`, casting it to an integer, and comparing the result against `z`. This, along with the lack of side-effects in expressions, frees us to interpret such chains in their usual mathematical sense: `x < y <= z` is syntactic sugar for `x < y && y <= z`, and so on.

$\mathsf{C}_{\mathsf{view}}$ makes minor changes to the operator set. Table A.1 summarises the operators $\mathsf{C}_{\mathsf{view}}$ understands in expression position (that is, excluding constant-level operators such as signs, and special-purpose operators such as assignment and increment/decrement suffixes). Informally, the operators have the same semantics as their C equivalents (the new operator `x => y` being equivalent to `!x || y`). The formal semantics depends on the backend theory: $\mathsf{Starling}_{\mathsf{tool}}$ maps operators to their equivalent in the solver's language.

Operator precedence

Table A.1 gives a precedence table for the operators in $\mathsf{C}_{\mathsf{view}}$.

[3] In C, **true** behaves as `1`, and **false** `0`, for arithmetic operations.

Group	Arity	Fixity	Assoc.	Input types	Output types	Members
subscripts	2	mixed[4]	left	any[5]	any	`[]`
negation	1	prefix	—	Boolean	Boolean	`!`
multiplicative	2	infix	left	arithmetic	arithmetic	`* / %`
additive	2	infix	left	arithmetic	arithmetic	`+ −`
relational	2	infix	left	arithmetic	Boolean	`< <= >= >`
equality	2	infix	left	any	Boolean	`!= ==`
implication	2	infix	left	Boolean	Boolean	`=>`
conjunction	2	infix	left	Boolean	Boolean	`&&`
disjunction	2	infix	left	Boolean	Boolean	`||`

Table A.1: Operators in C_{view}, and their syntactic properties. Rows denote precedence levels.

Grammar sketch

Below is a sketch of the C_{view} grammar. As the expression language is that of C (aside from the differences mentioned above), the sketch defers to existing C expression grammars [85].

⟨*typedef*⟩ ::= `typedef` ⟨*prim-type*⟩ ⟨*identifier*⟩ `;`

⟨*prim-type*⟩ ::= `int` | `bool`

⟨*type-lhs*⟩ ::= ⟨*prim-type*⟩ | ⟨*identifier*⟩

⟨*array-subs*⟩ ::= `[` [⟨*integer*⟩] `]` [⟨*array-subs*⟩]

⟨*type*⟩ ::= ⟨*type-lhs*⟩ [⟨*array-subs*⟩]

⟨*scope*⟩ ::= `thread` | `shared`

⟨*id-list*⟩ ::= ⟨*identifier*⟩ [`,` ⟨*id-list*⟩]

⟨*var-decl*⟩ ::= ⟨*scope*⟩ ⟨*type*⟩ ⟨*id-list*⟩ `;`

⟨*interpolate*⟩ ::= `[|` ⟨*expression*⟩ `|]`

⟨*sym-word*⟩ ::= ⟨*non-whitespace-string*⟩ | ⟨*interpolate*⟩

⟨*sym-words*⟩ ::= ⟨*sym-word*⟩ [⟨*sym-words*⟩]

⟨*symbol*⟩ ::= `%{` [⟨*sym-words*⟩] `}`

⟨*params*⟩ ::= ⟨*type*⟩ ⟨*identifier*⟩ [`,` ⟨*params*⟩]

⟨*proto*⟩ ::= ⟨*identifier*⟩ [`(` [⟨*params*⟩] `)`]

⟨*protos*⟩ ::= ⟨*proto*⟩ [`,` ⟨*protos*⟩]

⟨*atom-proto*⟩ ::= `view` ⟨*proto-list*⟩ `;` | `view iter` ⟨*func-proto*⟩ `;`

⟨*assign*⟩ ::= ⟨*expr*⟩ = ⟨*expr*⟩ ⟨*var-modifier*⟩`?` `;`

⟨*var-modifier*⟩ ::= `++` | `--`

⟨*binding-list*⟩ ::= ⟨*identifier*⟩ [`,` ⟨*binding-list*⟩]

⟨*atom-pattern*⟩ ::= ⟨*identifier*⟩ [`(` [⟨*binding-list*⟩] `)`]

⟨*view-pattern*⟩ ::= ⟨*atom-pattern*⟩ [`*` ⟨*view-pattern*⟩]

⟨*pattern*⟩ ::= `emp` | `iter` ⟨*atom-pattern*⟩ | ⟨*view-pattern*⟩

⟨*definition*⟩ ::= ⟨*expression*⟩ | `?`

⟨*constraint*⟩ ::= `constraint` ⟨*pattern*⟩ `->` ⟨*definition*⟩ `;`

⟨*expr-list*⟩ ::= ⟨*expression*⟩ [`,` ⟨*expr-list*⟩]

⟨*atom*⟩ ::= ⟨*identifier*⟩ `(` [⟨*expr-list*⟩] `)`

⟨*view-ite*⟩ ::= `if` ⟨*expression*⟩ `{` ⟨*view-product*⟩ `}` [`else` `{` ⟨*view-product*⟩ `}`]

⟨*view-elem*⟩ ::= ⟨*atom*⟩ | `false` | `local` `{` ⟨*expression*⟩ `}` | ⟨*view-local*⟩ | ⟨*view-ite*⟩

⟨*view-product*⟩ ::= `emp` | ⟨*view-elem*⟩ | ⟨*view-product*⟩ `*` ⟨*view-product*⟩

⟨*view-expr*⟩ ::= `?` | ⟨*view-product*⟩

⟨*view-assert*⟩ ::= `{|` ⟨*view-expr*⟩ `|}`

⟨*var-mod*⟩ ::= ⟨*expr*⟩ ⟨*var-modifier*⟩ `;`

⟨*floyd*⟩ := `assume` ⟨*expression*⟩ `;` | `assert` ⟨*expression*⟩ `;`

⟨*cas*⟩ ::= `CAS` `(` ⟨*expression*⟩ `,` ⟨*expression*⟩ `,` ⟨*expression*⟩ `)` `;`

⟨*condition*⟩ ::= ⟨*expression*⟩ | `*`

⟨*selection*⟩ ::= `if` ⟨*condition*⟩ ⟨*block*⟩ [`else` ⟨*block*⟩]

⟨*iteration*⟩ := `while` ⟨*expression*⟩ ⟨*block*⟩
| `do` ⟨*block*⟩ `while` ⟨*expression*⟩ `;`

⟨*statement*⟩ ::= ⟨*iteration*⟩ | ⟨*selection*⟩ | ⟨*block*⟩ | `<|` ⟨*statement*⟩`*` `|>`
| ⟨*action*⟩ | ⟨*miracle*⟩ | ⟨*view-assert*⟩ | ⟨*var-decl*⟩

⟨*block*⟩ ::= `{` ⟨*statement*⟩`*` `}`

⟨*method*⟩ ::= `method` ⟨*proto*⟩ ⟨*block*⟩

⟨*search*⟩ ::= `search` ⟨*positive-integer*⟩ `;`

⟨*pragma*⟩ ::= `pragma` ⟨*identifier*⟩ `{` ⟨*anything-except-close-brace*⟩ `}` `;`

⟨*top*⟩ ::= ⟨*var-decl*⟩ | ⟨*method*⟩ | ⟨*view-proto*⟩ | ⟨*constraint*⟩ | ⟨*search*⟩ | ⟨*pragma*⟩

⟨*cview*⟩ ::= ⟨*top*⟩ [⟨*cview*⟩]

A.7 Full Circular Buffer

This section contains a more complete version of the circular buffer from § 8.1.

```
1  shared int[100] circ_buf;
2  shared int      r_capacity, w_capacity;
3  thread int      position;
4  thread int[100] local_buf;
5  method write(int c) {
6    {| Writer(position, 0) * local{0 <= c <= 100} |}
7      thread int wc; <| wc = w_capacity; |>
8    {| Writer(position, wc) * local{0 <= c <= 100} |}
9      if wc < c {
10       {| Writer(position, wc) * local{0 <= c <= 100} |}
11         c = wc;
12       {| Writer(position, wc) * local{0 <= c <= wc} |}
13     }
14   {| Writer(position, wc) * local{0 <= c <= wc} |}
15     thread int wrote; wrote = 0;
16   {|  Writer(position, wc) * local{0 <= wrote <= c <= wc} |}
17     while wrote < c {
18       {|  Writer(position, wc) * local{0 <= wrote < c <= wc} |}
19         <| circ_buf[position] = local_buf[wrote]; |>
20         position = (position + 1) % 100;
21         wrote++;
22       {|  Writer(position, wc) * local{0 < wrote <= c <= wc} |}
23     }
24   {| Writer(position, wc) * local{0 <= wrote && wrote <= wc} |}
25     <|  w_capacity  = w_capacity - wrote;
26         r_capacity  = r_capacity + wrote; |>
27   {| Writer(position, wc - wrote) |}
28 }
29 method read(int c) {
30   {| Reader(position, 0) * local{0 <= c <= 100} |}
31     thread int rc; <| rc = r_capacity; |>
32   {| Reader(position, rc) * local{0 <= c <= 100} |}
33     if rc < c {
34       {| Reader(position, rc) * local{0 <= c <= 100} |}
35         c = rc;
36       {| Reader(position, rc) * local{0 <= c <= rc} |}
37     }
38   {| Reader(position, rc) * local{0 <= c <= rc} |}
39     thread int read; read = 0;
40   {|  Reader(position, rc) * local{0 <= read <= c <= rc} |}
```

```
41    while (read < c) {
42      {|  Reader(position, rc) * local{0 <= read < c <= rc} |}
43        <| local_buf[read] = circ_buf[position]; |>
44        position = (position + 1) % 100;
45        read++;
46      {|  Reader(position, rc) * local{0 < read <= c <= rc} |}
47    }
48   {| Reader(position, rc) * local{0 <= read <= rc} |}
49    <|  r_capacity  = r_capacity − read;
50        w_capacity  = w_capacity + read; |>
51   {| Reader(position, rc − read) |}
52 }
53 method flush(int p1, int p2) {
54   {| Reader(p1, 0) * Writer(p2, 0) |}
55    <| r_capacity = 0; w_capacity = 100; |>
56   {| Reader(p1, 0) * Writer(p2, 100) |}
57 }
58 method forget_wcap(int c) {
59   {| Writer(position, c) |} ; {| Writer(position, 0) |}
60 }
61 method forget_rcap(int c) {
62   {| Reader(position, c) |} ; {| Reader(position, 0) |}
63 }
64 constraint emp −>  0 <= w_capacity && 0 <= r_capacity
65                 &&  w_capacity + r_capacity == 100;
66 view Writer(int position, int cap_estimate);
67 constraint Writer(position, cap_estimate) −>
68   0 <= position < 100 && 0 <= cap_estimate <= w_capacity;
69 constraint Writer(xp, xc) * Writer(yp, yc) −> false;
70 view Reader(int pointer, int cap_estimate);
71 constraint Reader(pointer, cap_estimate) −>
72   0 <= pointer < 100 && 0 <= cap_estimate <= r_capacity;
73 constraint Reader(xp, xc) * Reader(yp, yc) −> false;
```

Appendix B

Proofs

This appendix contains human-readable proofs for some of the theorems and lemmas posed in the main body. Where these proofs complement a Coq-mechanised proof, we give a reference to the Coq development in the main body, alongside the reference to this appendix.

B.1 The free views instance

Theorem 3.2: maximality of the free views instance

Proof. Backwards proof: strengthen $\mathsf{i.T} \subseteq \mathsf{finst}(\mathsf{i}).\mathsf{T}$ to axiom soundness, which we assume:

$$\mathsf{i.T} \subseteq \mathsf{finst}(\mathsf{i}).\mathsf{T}$$

$$\mathsf{i.T} \subseteq \left\{ \langle p \rangle\ c\ \langle q \rangle \;\middle|\; \begin{array}{c} [\![c]\!]^*_P(\lfloor p \rfloor) \subseteq \lfloor q \rfloor \\ \wedge\, \forall v \in V.\ [\![c]\!]^*_P(\lfloor p \bullet v \rfloor) \subseteq \lfloor q \bullet v \rfloor \end{array} \right\}$$

(expand RHS)

$$\forall a.\ a \in \mathsf{i.T} \Rightarrow a \in \left\{ \langle p \rangle\ c\ \langle q \rangle \;\middle|\; \begin{array}{c} [\![c]\!]^*_P(\lfloor p \rfloor) \subseteq \lfloor q \rfloor \\ \wedge\, \forall v \in V.\ [\![c]\!]^*_P(\lfloor p \bullet v \rfloor) \subseteq \lfloor q \bullet v \rfloor \end{array} \right\}$$

(unfold set inclusion)

$$\forall \langle p \rangle\ c\ \langle q \rangle .\, \langle p \rangle\ c\ \langle q \rangle \in \mathsf{i.T} \Rightarrow [\![c]\!]^*_P(\lfloor p \rfloor) \subseteq \lfloor q \rfloor \wedge \forall v \in V.\ [\![c]\!]^*_P(\lfloor p \bullet v \rfloor) \subseteq \lfloor q \bullet v \rfloor$$

(unfold set membership)

$$\forall \langle p \rangle\ c\ \langle q \rangle .\quad [\![c]\!]^*_P(\lfloor p \rfloor) \subseteq \lfloor q \rfloor \wedge \forall v \in V.\ [\![c]\!]^*_P(\lfloor p \bullet v \rfloor) \subseteq \lfloor q \bullet v \rfloor$$

(weaken precondition)

□

B.2 Views algebras

This section contains proofs that various constructs (multisets, views expressions, and so on) are instances of particular views algebras (semigroups, monoids, and so on).

Function lifts

These proofs correspond to the results in Lemma 5.1. In each case, we assume that the lifted algebra has the appropriate properties.

Setoid

Proof. Per Definition A.1, we must show that the lifted $\equiv$ ($\lambda x, y.\, \forall z : T.\, x \equiv y$) is an equivalence. We do so by proving reflexivity, symmetry, and transitivity:

Reflexivity Unfold to $\forall x : T.\, f(x) \equiv f(x)$; apply reflexivity of $\equiv$.

Symmetry Unfold to $(\forall x : T.\, f(x) \equiv g(x)) \implies (\forall x : T.\, g(x) \equiv f(x))$; apply symmetry of $\equiv$.

Transitivity Backwards proof from RHS to LHS.

$$
\begin{aligned}
& \forall x : T \quad .\, f(x) \equiv h(x) && && && \\
\longleftarrow & \forall x : T \quad .\, f(x) \equiv g(x) && \wedge && g(x) \equiv h(x) && \text{(Transitivity of } \equiv) \\
\longleftarrow & (\forall x : T \quad .\, f(x) \equiv g(x)) && \wedge && (\forall x : T.\, g(x) \equiv h(x)) && \text{(Generalise quantification)}
\end{aligned}
$$

□

Semigroup

Proof. By Definition 2.4, prove commutativity and associativity of lifted $\bullet$ over lifted $\equiv$, and compatibility of lifted $\bullet$ with lifted $\equiv$.

Commutativity To prove: $\forall f, g : T \to V.\, (f \bullet g) \equiv (g \bullet f)$; first, unfold and β-reduce fully:

$$
\begin{aligned}
(f \bullet g) &\equiv (g \bullet f) \\
\forall x : T.\, (f \bullet g)(x) &= (g \bullet f)(x) \\
\forall x : T.\, (\lambda y.\, f(y) + g(y))(x) &= (\lambda y.\, g(y) + f(y))(x) \\
\forall x : T.\, f(x) + g(x) &= g(x) + f(x)
\end{aligned}
$$

Then, prove by commutativity of $+$ over $\mathbb{N}$.

Associativity To prove: $\forall f, g, h : T \to V.\, (f \bullet (g \bullet h)) \equiv ((f \bullet g) \bullet h)$; first, unfold fully:

$$
\begin{aligned}
& & (f \bullet (g \bullet h)) &\equiv ((f \bullet g) \bullet h) \\
\forall x : T. & & (f \bullet (g \bullet h))(x) &\equiv ((f \bullet g) \bullet h)(x) \\
\forall x : T. & & (\lambda y.\, f(y) \bullet (g \bullet h)(y))(x) &\equiv (\lambda y.\, (f \bullet g)(y) \bullet h(y))(x) \\
\forall x : T. & & f(x) \bullet (g \bullet h)(x) &\equiv (f \bullet g)(x) \bullet h(x) \\
\forall x : T. & & f(x) \bullet (\lambda y.\, g(y) \bullet h(y))(x) &\equiv (\lambda y.\, f(y) \bullet g(y))(x) \bullet h(x) \\
\forall x : T. & & f(x) \bullet (g(x) \bullet h(x)) &\equiv (f(x) \bullet g(x)) \bullet h(x)
\end{aligned}
$$

Then, prove by associativity of the underlying $\bullet$.

Compatibility To prove: $\forall f, g, h : T \to V.\ f \equiv g \implies (f \bullet h) \equiv (g \bullet h)$. Backwards proof from RHS to LHS:

$$
\begin{aligned}
& (f \bullet h) \equiv (g \bullet h) \\
\longleftarrow & \forall x : T.\ (f \bullet h)(x) \equiv (g \bullet h)(x) && \text{(Unfold)} \\
\longleftarrow & \forall x : T.\ (\lambda y.\ f(y) \bullet h(y))(x) \equiv (\lambda y.\ (g(y) \bullet h(y))(x) && \text{(Unfold)} \\
\longleftarrow & \forall x : T.\ f(x) \bullet h(x) \equiv g(x) \bullet h(x) && (\beta\text{-reduction}) \\
\longleftarrow & \forall x : T.\ f(x) \equiv g(x) && \text{(Compatibility of underlying algebra)} \\
\longleftarrow & f \equiv g && \text{(Fold)}
\end{aligned}
$$

□

Monoid

Proof. By Definition 2.5), we must show that ε is a unit: $\forall f : T \to V.\ \varepsilon \equiv f \equiv f$. To prove this, unfold and β-reduce fully:

$$
\begin{aligned}
\varepsilon \bullet f &\equiv f \\
\forall x : T.\ (\varepsilon \bullet f)(x) &\equiv f(x) \\
\forall x : T.\ (\lambda y.\ \varepsilon(y) \bullet f(y))(x) &\equiv f(x) \\
\forall x : T.\ \varepsilon(x) \bullet f(x) &\equiv f(x) \\
\forall x : T.\ (\lambda y.\ \varepsilon)(x) \bullet f(x) &\equiv f(x) \\
\forall x : T.\ \varepsilon \bullet f(x) &\equiv f(x)
\end{aligned}
$$

Then, appeal to the underlying views monoid. □

Ordered views semigroup

This proof relies on the following lemmas:

Lemma B.1 (Lifted $\equiv$–$\sqsubseteq$)**.** $\forall f, g : T \to V.\ f \equiv g \implies f \sqsubseteq g$.

Proof. Forwards from LHS to RHS.

$$
\begin{aligned}
& f \equiv g \\
= & \forall x : T.\ f(x) \equiv g(x) && \text{(Unfold)} \\
\to & \forall x : T.\ f(x) \leqslant g(x) && \text{(underlying semigroup)} \\
= & f \sqsubseteq g && \text{(Fold)}
\end{aligned}
$$

□

Lemma B.2 (Lifted ε is least element)**.** $\forall f.\ \varepsilon \sqsubseteq f$.

Proof. Unfold and β-reduce fully:

$$\begin{aligned}
\varepsilon &\sqsubseteq f \\
\forall x : T.\ \varepsilon(x) &\sqsubseteq f(x) \\
\forall x : T.\ (\lambda y.\ \varepsilon)(x) &\sqsubseteq f(x) \\
\forall x : T.\ \varepsilon &\sqsubseteq f(x)
\end{aligned}$$

Then, appeal to the underlying semigroup. □

Lemma B.3 (Lifted $\equiv$-$\sqsubseteq$-congruence)**.** $\forall f, f', g, g' : T \to V.\ (f \equiv f') \wedge (g \equiv g') \wedge (f \sqsubseteq g) \implies (f' \sqsubseteq g')$

Proof. Forwards from LHS to RHS.

$$\begin{aligned}
& (f \equiv f') \wedge (g \equiv g') \wedge (f \sqsubseteq g) && \\
\to\ & (f' \equiv f) \wedge (g \equiv g') \wedge (f \sqsubseteq g) && \text{(Symmetry of } \sqsubseteq\text{)} \\
\to\ & (f' \sqsubseteq f) \wedge (g \sqsubseteq g') \wedge (f \sqsubseteq g) && \text{(Lemma B.1)} \\
=\ & (f' \sqsubseteq f) \wedge (f \sqsubseteq g) \wedge (g \sqsubseteq g') && \text{(Rearrange)} \\
\to\ & f' \sqsubseteq g' && \text{(Transitivity)}
\end{aligned}$$

□

We can now prove that functions are ordered views semigroups.

Proof. Per Definition 3.7, we must prove that $\sqsubseteq$ is a pre-order (reflexive and transitive); we must also prove the $\equiv$-$\sqsubseteq$ and $\sqsubseteq$-$\equiv$ laws, and that $\bullet$ is increasing and inflationary over $\sqsubseteq$.

Reflexivity Unfold to $\forall x : T.\ m(x) = m(x)$; then, by reflexivity of $\mathbb{N}$-ordering.

Transitivity Backwards proof from RHS to LHS.

$$\begin{aligned}
& f \sqsubseteq h && \\
\longleftarrow\ & \forall x : T.\ f(x) \leqslant h(x) && \text{(Unfold)} \\
\longleftarrow\ & \forall x : T.\ f(x) \leqslant g(x) \wedge g(x) \leqslant h(x) && \text{(Transitivity of } \mathbb{N}\text{-ordering)} \\
\longleftarrow\ & (\forall x : T.\ f(x) \leqslant g(x)) \wedge (\forall x : T.\ g(x) \leqslant h(x)) && \text{(Generalise quantification)} \\
\longleftarrow\ & f \sqsubseteq g \wedge f \sqsubseteq h && \text{(Fold)}
\end{aligned}$$

The $\equiv$-$\sqsubseteq$ law See Lemma B.1.

The $\sqsubseteq$-$\equiv$ law To prove: $\forall f, g : T \to V.\ f \sqsubseteq g \wedge g \sqsubseteq f \implies f \equiv g$. Forwards from LHS to RHS.

$$\begin{aligned}
& f \sqsubseteq g \wedge g \sqsubseteq f && \\
\longrightarrow\ & (\forall x : T.\ f(x) \leqslant g(x)) \wedge (\forall x : T.\ g(x) \leqslant f(x)) && \text{(Unfold)} \\
\longrightarrow\ & \forall x : T.\ f(x) \leqslant g(x) \wedge g(x) \leqslant f(x) && \text{(Unify quantifiers)} \\
\longrightarrow\ & \forall x : T.\ f(x) = g(x) && \text{(Antisymmetry of } \leqslant \text{ over } \mathbb{N}\text{)} \\
\longrightarrow\ & f \equiv g && \text{(Fold)}
\end{aligned}$$

Increasing To prove: $\forall f, g, h : T \to V.\ f \sqsubseteq g \implies (f \bullet h) \sqsubseteq (g \bullet h)$. Forward proof from LHS to RHS.

$$
\begin{aligned}
& f \sqsubseteq g \\
\longrightarrow & \forall x : T.\ f(x) \leqslant g(x) && \text{(Unfold)} \\
\longrightarrow & \forall x : T.\ f(x) + h(x) \leqslant g(x) + h(x) && \text{(Monotonicity of } \leqslant \text{ over } \mathbb{N}) \\
\longrightarrow & \forall x : T.\ (\lambda x.\ f(x) + h(x))(x) \leqslant (\lambda x.\ g(x) + h(x))(x) && (\beta\text{-abstraction}) \\
\longrightarrow & \forall x : T.\ (f \bullet h)(x) \leqslant (g \bullet h)(x) && \text{(Fold)} \\
\longrightarrow & (f \bullet h) \sqsubseteq (g \bullet h) && \text{(Fold)}
\end{aligned}
$$

Progressing To prove: $\forall f, g : T \to V.\ f \sqsubseteq (f \bullet g)$. Backwards proof: reduce to Lemma B.2.

$$
\begin{aligned}
& f \sqsubseteq (f \bullet g) \\
\longleftarrow & (f \bullet \varepsilon) \sqsubseteq (f \bullet g) && \text{(inner } \varepsilon \text{ is a unit)} \\
\longleftarrow & (\varepsilon \bullet f) \sqsubseteq (g \bullet f) && \text{(Lemma B.3, using inner } \bullet \text{ commutativity)} \\
\longleftarrow & \varepsilon \sqsubseteq g && \text{(Apply inner } \bullet\text{-}\sqsubseteq)
\end{aligned}
$$

□

Natural numbers

These proofs correspond to the results in § 6.1. As Lemmas 6.4 to 6.6 just rely on collecting trivial properties of natural numbers, we do not prove them here.

Lemma 6.7: naturals are ordered views semigroups

Proof. Per Definition 3.7, we must show that $\leqslant$ is a pre-ordering (reflexive and transitive), which is trivial. We must also prove the $\equiv$-$\sqsubseteq$ and $\sqsubseteq$-$\equiv$ laws, and that $+$ is increasing and inflationary with respect to $\leqslant$. over $\sqsubseteq$.

The $\equiv$-$\sqsubseteq$ law This is $\forall x, y : \mathbb{N}.\ x = y \implies x \leqslant y$, which we get by substitution and reflexivity.

The $\sqsubseteq$-$\equiv$ law This is $\forall x, y : \mathbb{N}.\ x \leqslant y \wedge y \leqslant x \implies x = y$, a well-known property of $\mathbb{N}$.

Increasing This is $\forall x, y, z : \mathbb{N}.\ x \leqslant y \implies (x + z) \leqslant (y + z)$, also well-known.

Inflation This is $\forall x, y : \mathbb{N}.\ x \leqslant (x + y)$; we can show it by transitivity through $(x + 0)$: $x \leqslant x + 0$ by 0 being the additive unit, and $x + 0 \leqslant x + y$ through compatibility and 0 being the least member of $\mathbb{N}$.

□

Lemma 6.8: naturals are subtractive views semigroups

Proof. By Definition 3.8, we must show that $\dot{-}$ is compatible with $\leqslant$, and that naturals have an adjoint property over $\leqslant$.

Compatibility To prove: $\forall x, y, z : \mathbb{N}.\, x \leqslant y \implies (x \dot{-} z) \leqslant (y \dot{-} z)$. First, unfold the RHS:

$$\begin{aligned} & (x \dot{-} z) \leqslant (y \dot{-} z) \\ \longrightarrow \quad & \mathsf{max}(0, x - z) \leqslant \mathsf{max}(0, y - z) \end{aligned}$$

When $x < z$, this reduces the RHS to $0 \leqslant k$ for some k, and, as 0 is the lowest natural number, we have a tautology. Otherwise, we have that the RHS is $x - z \leqslant y \dot{-} z$. When $y < z$, we can use the assumption that $x \leqslant y$ to show that, transitively, $x < z$. As a result, $x - z = 0$, and we have the tautology $0 \leqslant 0$.

For the remaining case ($x \geqslant z \wedge y \geqslant z$), the RHS becomes $x - z \leqslant y - z$, at point we appeal to the natural numbers and assumption $x \leqslant y$.

Adjoint To prove: $\forall x, y, z : \mathbb{N}.\, x \leqslant (y + z) \implies (x \dot{-} y) \leqslant z$.

Backwards proof, from RHS to LHS. First, unfold the RHS to $\mathsf{max}(0, x - y)$. Then, case split on $x < y$. If this is the case, then we have $0 \leqslant z$, which is always true as 0 is the least member of $\mathbb{N}$. Else, continue with backwards proof:

$$\begin{aligned} & \mathsf{max}(0, x - y) \leqslant z \\ \longleftarrow \quad & x - y \leqslant z && \text{(Apply case split)} \\ \longleftarrow \quad & x \leqslant y + z && \text{(Add } y \text{ to both sides, using case split to justify closure over } \mathbb{N}) \end{aligned}$$

i.e. the LHS.

□

View expressions

This section contains extra definitions and facts over view expressions.

Equivalence, and view expressions as setoids. Syntactic equivalence across view expressions is hard to calculate for two reasons. First, syntactically different view expressions can correspond to the same view through the laws of views semigroups. For example, $(@A)$, $(\bullet\ (@A)\ \mathit{1})$, and $(\bullet\ (\bullet\ (@A)\ \mathit{1})\ (\backslash\ (@A)\ (@A)))$ are all equivalent.

Second, different view expressions can correspond to the same view through the algebraic structure or reification of the underlying views monoid. For example, if our views monoid is idempotent ($\forall v, v \bullet v \equiv v$) then $(@A)$ becomes equivalent to $(\bullet\ (@A)\ (@A))$.

Instead of needing a complex, view-monoid-specific decision process for equivalence, we can define it in terms of equivalence on the underlying views.

Definition B.1. For all Atom, V, and r, the *view expression equivalence* $\equiv_{\mathsf{v}}$ relates view expressions v_1 and v_2 if, and only if, their interpretations are equivalent:

$$\equiv_{\mathsf{v}} : \mathsf{VExpr}(\mathsf{Atom}) \leftrightarrow \mathsf{VExpr}(\mathsf{Atom}) \qquad v_1 \equiv_{\mathsf{v}} v_2 \stackrel{\text{def}}{=} \mathrm{I}(\mathrm{r})(v_1) \equiv \mathrm{I}(\mathrm{r})(v_2)$$

This definition leads to a general result that all views expressions are setoids per Definition A.1, assuming that their underlying monoids are setoids.

Theorem B.4. For all Atom, V, and r, where V is a setoid, $\mathsf{VExpr}(\mathsf{Atom})$ is a setoid over $\equiv_{\mathsf{v}}$.

Proof. By proving reflexivity, symmetry, and transitivity:

Reflexivity To prove: $\forall v.\, v \equiv_{\mathsf{v}} v$. Unfold to $\mathrm{I}(\mathrm{r})(v) \equiv \mathrm{I}(\mathrm{r})(v)$; then, $\equiv$ reflexivity.

Symmetry To prove: $\forall v_1, v_2.\, v_1 \equiv_{\mathsf{v}} v_2 \implies v_2 \equiv_{\mathsf{v}} v_1$. Unfold to $\mathrm{I}(\mathrm{r})(v_1) \equiv \mathrm{I}(\mathrm{r})(v_2) \implies \mathrm{I}(\mathrm{r})(v_1) \equiv \mathrm{I}(\mathrm{r})(v_1)$; then, $\equiv$ symmetry.

Transitivity To prove: $\forall v_1, v_2, v_3.\, v_1 \equiv_{\mathsf{v}} v_2 \wedge v_2 \equiv_{\mathsf{v}} v_3 \implies v_1 \equiv_{\mathsf{v}} v_3$. Unfold:

$$\mathrm{I}(\mathrm{r})(v_1) \equiv \mathrm{I}(\mathrm{r})(v_2) \wedge \mathrm{I}(\mathrm{r})(v_2) \equiv \mathrm{I}(\mathrm{r})(v_3) \implies \mathrm{I}(\mathrm{r})(v_1) \equiv \mathrm{I}(\mathrm{r})(v_3)$$

Then, $\equiv$ transitivity.

□

Join, and views expressions as views semigroups. As with most of the view expression operators, the join operator maps directly to a production of the grammar, namely $\bullet$.

Definition B.2. The *view expression join* $*_{\mathsf{v}} : \mathsf{VExpr}(\mathsf{Atom}) \to \mathsf{VExpr}(\mathsf{Atom}) \to \mathsf{VExpr}(\mathsf{Atom})$, over an atom language, is: $v_1 *_{\mathsf{v}} v_2 \stackrel{\text{def}}{=} (\bullet\ v_1\ v_2)$.

Theorem B.5. For all Atom, V, and r, such that V is a views semigroup, $(\mathsf{VExpr}(\mathsf{Atom}), \bullet, \equiv)$ is a views semigroup.

Proof sketch. Unfold each property (commutativity, associativity, compatibility) until it becomes the equivalent property over views in V; then appeal to V's views semigroup.

We give a formal proof in Appendix B.2. □

Unit, and view expressions as views monoids. The unit expression also corresponds to a grammar production, namely 1.

Definition B.3. For all Atom, V, r, the *unit view expression* $1_{\mathsf{v}} : \mathsf{VExpr}(\mathsf{Atom})$ is *1*.

View expressions are views monoids. We now show that, if the underlying views model is a monoid, view expressions are also views monoids with 1_v as unit.

Theorem B.6. For all Atom, V, and r, if V is a views monoid, then $(\mathsf{VExpr}(\mathsf{Atom}), 1_v, *_v, \equiv_v)$ is also a views monoid.

Proof. By Definition 2.5), we must show that 1_v is a unit: $\forall v : \mathsf{VExpr}(\mathsf{Atom}).\ 1_v *_v v \equiv_v v$. To prove this, unfold:

$$\begin{aligned} 1_v *_v v &\equiv_v v \\ I(r)(1_v *_v v) &\equiv I(r)(v) \\ I(r)(\texttt{(* } 1_v \ v\texttt{)}) &\equiv I(r)(v) \\ I(r)(1_v) \bullet I(r)(v) &\equiv I(r)(v) \\ I(r)(1) \bullet I(r)(v) &\equiv I(r)(v) \\ 1 \bullet I(r)(v) &\equiv I(r)(v) \end{aligned}$$

Then, apply the unit property of the underlying views monoid. □

Order, and view expressions as ordered views semigroups. We can define inclusion on view expressions in the same way, substituting the underlying monoid's $\leqslant$ for $\equiv$, and arriving at Definition B.4.

Definition B.4. For all Atom, V, and r, the *view expression inclusion* $\sqsubseteq_v$ relates view expressions v_1 and v_2 if, and only if, the interpretation of v_1 is included in that of v_2 by the underlying inclusion relation:

$$\sqsubseteq_v : \mathsf{VExpr}(\mathsf{Atom}) \leftrightarrow \mathsf{VExpr}(\mathsf{Atom}) \qquad v_1 \sqsubseteq_v v_2 \stackrel{\text{def}}{=} I(r)(v_1) \sqsubseteq I(r)(v_2)$$

Theorem B.7. For all Atom, V, and r $(\mathsf{VExpr}(\mathsf{Atom}), *_v, \sqsubseteq, \equiv_v)$ is an ordered views semigroup if V is.

Proof sketch. Unfold each property, applying the definition of I where needed, until it becomes the equivalent property over views in V; then appeal to V's ordered views semigroup.

Appendix B.2 gives a formal proof. □

Subtraction, and view expressions as subtractive views semigroups. We also define the $\setminus$-operator for view expressions in the same way as we did the $\bullet$-operator.

Definition B.5. For all atom sets Atom, views monoids V, and atom projections r, we define the *view expression part* $\setminus_v : \mathsf{VExpr}(\mathsf{Atom}) \to \mathsf{VExpr}(\mathsf{Atom}) \to \mathsf{VExpr}(\mathsf{Atom})$ as follows:

$$v_1 \setminus_v v_2 \stackrel{\text{def}}{=} (\backslash\ v_1\ v_2)$$

With $\setminus_v$, view expressions form a subtractive views semigroup, provided they abstract over subtractive views semigroups themselves.

Theorem B.8. For all Atom, V, and r, $(\mathsf{VExpr}(\mathsf{Atom}), *_{\mathsf{v}}, \setminus_{\mathsf{v}}, \sqsubseteq_{\mathsf{v}}, \equiv_{\mathsf{v}})$ is an subtractive views semigroup.

Proof sketch. Unfold each property, applying the definition of I where needed, until it becomes the equivalent property over V; then appeal to V's subtractive views semigroup. □

View expression reification. If we treat view expressions as a thin layer on top of an existing views monoid, the most straightforward approach to reification is to compose the interpretation function with the reification we plan to use with the views monoid.

> **Definition B.6** (View expression reification)**.** Given an atom projection r and reification $\lfloor - \rfloor$, the *view expression reification* $\mathsf{VReify}(\mathrm{r})(\lfloor - \rfloor)$ is the function $\lfloor - \rfloor \circ \mathrm{I}(\mathrm{r})$.

This definition easily gives us the relationship between reification and equivalence we need from views reification functions.

Lemma B.9.

$$\forall \nu_1, \nu_2 : \mathsf{VExpr}(\mathsf{Atom}).$$
$$\nu_1 \equiv_{\mathsf{v}} \nu_2 \implies \mathsf{VReify}(\mathrm{r})(\nu_1) \subseteq \mathsf{VReify}(\mathrm{r})(\nu_2) \wedge \mathsf{VReify}(\mathrm{r})(\nu_2) \subseteq \mathsf{VReify}(\mathrm{r})(\nu_1)$$

Proof. Backwards proof from LHS to RHS:

$$\begin{aligned}
&\mathsf{VReify}(\mathrm{r})(\nu_1) \subseteq \mathsf{VReify}(\mathrm{r})(\nu_2) \wedge \mathsf{VReify}(\mathrm{r})(\nu_2) \subseteq \mathsf{VReify}(\mathrm{r})(\nu_1) \\
\longleftarrow\ &(\lfloor - \rfloor \circ \mathrm{I}(\mathrm{r}))(\nu_1) \subseteq (\lfloor - \rfloor \circ \mathrm{I}(\mathrm{r}))(\nu_2) \wedge (\lfloor - \rfloor \circ \mathrm{I}(\mathrm{r}))(\nu_2) \subseteq (\lfloor - \rfloor \circ \mathrm{I}(\mathrm{r}))(\nu_1) && \text{(Unfold VReify)} \\
\longleftarrow\ &\lfloor \mathrm{I}(\mathrm{r})(\nu_1) \rfloor \subseteq \lfloor \mathrm{I}(\mathrm{r})(\nu_2) \rfloor \wedge \lfloor \mathrm{I}(\mathrm{r})(\nu_2) \rfloor \subseteq \lfloor \mathrm{I}(\mathrm{r})(\nu_1) \rfloor && \text{(Apply composition)} \\
\longleftarrow\ &\mathrm{I}(\mathrm{r})(\nu_1) \equiv \mathrm{I}(\mathrm{r})(\nu_2) && \text{(Apply Definition 2.7 property)} \\
\longleftarrow\ &\nu_1 \equiv_{\mathsf{v}} \nu_2 && \text{(Fold definition of } \equiv_{\mathsf{v}})
\end{aligned}$$

□

Adjoint-rule compatiblity on view expressions. Unlike multisets, view expressions are not adjoint-compatible for all reifiers. Instead, they are adjoint-compatible when their underlying monoid (and *its* reifier) are adjoint-compatible.

Theorem B.10 (View expression adjoint-rule compatibility)**.** For all Atom, V, r, and $\lfloor - \rfloor$, if $(V, \bullet, \setminus, \sqsubseteq, \equiv)$ and $\lfloor - \rfloor$ are adjoint-rule compatible, then $(\mathsf{VExpr}(\mathsf{Atom}), *_{\mathsf{v}}, \setminus_{\mathsf{m}}, \sqsubseteq, \equiv_{\mathsf{v}})$ and $\mathsf{VReify}(\mathrm{r})(\lfloor - \rfloor)$ are adjoint-rule compatible.

Proof. By unfolding from the definition of adjoint-rule compatibility to the same property on the underlying semigroup, which we assume.

$$
\begin{aligned}
&\mathsf{VReify}(r)(\lfloor - \rfloor)(p *_v v) \subseteq \mathsf{VReify}(r)(\lfloor - \rfloor)(p *_v ((v *_v q) \setminus_v q)) \\
&= (\lfloor - \rfloor \circ \mathrm{I}(r))(p *_v v) \subseteq (\lfloor - \rfloor \circ \mathrm{I}(r))(p *_v ((v *_v q) \setminus_v q)) && \text{(Unfold VReify)} \\
&= \lfloor \mathrm{I}(r))(p *_v v) \rfloor \subseteq \lfloor \mathrm{I}(r))(p *_v (v *_v q) \setminus_v q) \rfloor && \text{(Unfold composition of VReify)} \\
&= \lfloor \mathrm{I}(r)(\texttt{(* p v)}) \rfloor \subseteq \lfloor \mathrm{I}(r)(\texttt{(* p ((v *}_v\texttt{ q) \}_v\texttt{ q))}) \rfloor && \text{(Unfold } *_v\text{)} \\
&= \lfloor \mathrm{I}(r)(\texttt{(* p v)}) \rfloor \subseteq \lfloor \mathrm{I}(r)(\texttt{(* p (\ (v *}_v\texttt{ q) q))}) \rfloor && \text{(Unfold } \setminus_v\text{)} \\
&= \lfloor \mathrm{I}(r)(\texttt{(* p v)}) \rfloor \subseteq \lfloor \mathrm{I}(r)(\texttt{(* p (\ (* v q) q))}) \rfloor && \text{(Unfold } *_v\text{)} \\
&= \lfloor \mathrm{I}(r)(p) \bullet \mathrm{I}(r)(v) \rfloor \subseteq \lfloor \mathrm{I}(r)(p) \bullet \mathrm{I}(r)(\texttt{(\ (* v q) q)}) \rfloor && \text{(Definition of I)} \\
&= \lfloor \mathrm{I}(r)(p) \bullet \mathrm{I}(r)(v) \rfloor \subseteq \lfloor \mathrm{I}(r)(p) \bullet (\mathrm{I}(r)(\texttt{(* v q)}) \setminus \mathrm{I}(r)(q)) \rfloor && \text{(Definition of I)} \\
&= \lfloor \mathrm{I}(r)(p) \bullet \mathrm{I}(r)(v) \rfloor \subseteq \lfloor \mathrm{I}(r)(p) \bullet ((\mathrm{I}(r)(v) \bullet \mathrm{I}(r)(q)) \setminus \mathrm{I}(r)(q)) \rfloor && \text{(Definition of I)}
\end{aligned}
$$

□

These proofs correspond to the results in § 6.2. Some of the proofs for views expressions are short enough to appear inline in the main text: we do not repeat them here.

Theorem B.5: multisets are views semigroups

Proof. By Definition 2.4), prove commutativity and associativity of $*_v$ over $\equiv_v$, and compatibility of $*_v$ with $\equiv_v$.

Commutativity To prove: $\forall v_1, v_2 : \mathsf{VExpr}(\mathsf{Atom}).\ (v_1 *_v v_2) \equiv_v (v_2 *_v v_1)$.

Unfold the body as follows:

$$
\begin{aligned}
& (v_1 *_v v_2) \equiv_v (v_2 *_v v_1) \\
\longrightarrow\quad & \mathrm{I}(r)(v_1 *_v v_2) \equiv \mathrm{I}(r)(v_2 *_v v_1) \\
\longrightarrow\quad & \mathrm{I}(r)(\texttt{(* } v_1\ v_2\texttt{)}) \equiv \mathrm{I}(r)(\texttt{(* } v_2\ v_1\texttt{)}) \\
\longrightarrow\quad & \mathrm{I}(r)(v_1) \bullet \mathrm{I}(r)(v_2) \equiv \mathrm{I}(r)(v_2) \bullet \mathrm{I}(r)(v_1)
\end{aligned}
$$

Then, appeal to commutativity of the underlying semigroup.

Associativity To prove: $\forall v_1, v_2, v_3 : \mathsf{VExpr}(\mathsf{Atom}).\ (v_1 *_v (v_2 *_v v_3)) \equiv_v ((v_1 *_v v_2) *_v v_3)$.

Unfold the body as follows:

$$
\begin{aligned}
& (v_1 *_v (v_2 *_v v_3)) \equiv_v ((v_1 *_v v_2) *_v v_3) \\
\longrightarrow\quad & \mathrm{I}(r)(v_1 *_v (v_2 *_v v_3)) \equiv \mathrm{I}(r)((v_1 *_v v_2) *_v v_3) \\
\longrightarrow\quad & \mathrm{I}(r)(\texttt{(* } v_1\ \texttt{(* } v_2 *_v v_3\texttt{))}) \equiv \mathrm{I}(r)(\texttt{(* (* } v_1 *_v v_2\texttt{) } v_3\texttt{)}) \\
\longrightarrow\quad & \mathrm{I}(r)(\texttt{(* } v_1\ \texttt{(* } v_2\ v_3\texttt{))}) \equiv \mathrm{I}(r)(\texttt{(* (* } v_1\ v_2\texttt{) } v_3\texttt{)}) \\
\longrightarrow\quad & \mathrm{I}(r)(v_1) \bullet \mathrm{I}(r)(\texttt{(* } v_2\ v_3\texttt{)}) \equiv \mathrm{I}(r)(\texttt{(* } v_1\ v_2\texttt{)}) \bullet v_3 \\
\longrightarrow\quad & \mathrm{I}(r)(v_1) \bullet (\mathrm{I}(r)(v_2) \bullet \mathrm{I}(r)(v_3)) \equiv (\mathrm{I}(r)(v_1) \bullet \mathrm{I}(r)(v_3)) \bullet v_3
\end{aligned}
$$

Then, appeal to associativity of the underlying semigroup.

Compatibility To prove: $\forall \nu_1, \nu_2, \nu_3 : \mathsf{VExpr}(\mathsf{Atom}).\, \nu_1 \equiv_{\mathsf{v}} \nu_2 \implies (\nu_1 *_{\mathsf{v}} \nu_3) \equiv_{\mathsf{v}} (\nu_2 *_{\mathsf{v}} \nu_3)$.

Backwards proof from RHS to LHS:

$$
\begin{array}{lll}
 & (\nu_1 *_{\mathsf{v}} \nu_3) \equiv_{\mathsf{v}} (\nu_2 *_{\mathsf{v}} \nu_3) & \\
\longleftarrow & \mathrm{I}(\mathrm{r})(\nu_1 *_{\mathsf{v}} \nu_3) \equiv \mathrm{I}(\mathrm{r})(\nu_2 *_{\mathsf{v}} \nu_3) & \text{(Unfold)} \\
\longleftarrow & \mathrm{I}(\mathrm{r})(\texttt{(* } \nu_1 \ \nu_3\texttt{)}) \equiv \mathrm{I}(\mathrm{r})(\texttt{(* } \nu_2 \ \nu_3\texttt{)}) & \text{(Unfold)} \\
\longleftarrow & \mathrm{I}(\mathrm{r})(\nu_1) \bullet \mathrm{I}(\mathrm{r})(\nu_3) \equiv \mathrm{I}(\mathrm{r})(\nu_2) \bullet \mathrm{I}(\mathrm{r})(\nu_3) & \text{(Definition of I)} \\
\longleftarrow & \mathrm{I}(\mathrm{r})(\nu_1) \equiv \mathrm{I}(\mathrm{r})(\nu_2) & \text{(Compatibility of underlying semigroup)} \\
\longleftarrow & \nu_1 \equiv_{\mathsf{v}} \nu_2 & \text{(Definition of I)}
\end{array}
$$

□

Theorem B.7: view expressions are ordered views semigroups

Proof. Per Definition 3.7, we must prove that $\sqsubseteq_{\mathsf{v}}$ is a pre-order (reflexive and transitive). We must also prove the $\equiv$–$\sqsubseteq$ and $\sqsubseteq$–$\equiv$ laws, and that $*_{\mathsf{v}}$ is increasing and inflationary over $\sqsubseteq_{\mathsf{v}}$.

Reflexivity Unfold to $\mathrm{I}(\mathrm{r})(\nu) \sqsubseteq \mathrm{I}(\mathrm{r})(\nu)$; then, by reflexivity of $\sqsubseteq$.

Transitivity Unfold:

$$\mathrm{I}(\mathrm{r})(\nu_1) \sqsubseteq \mathrm{I}(\mathrm{r})(\nu_2) \wedge \mathrm{I}(\mathrm{r})(\nu_2) \sqsubseteq \mathrm{I}(\mathrm{r})(\nu_3) \implies \mathrm{I}(\mathrm{r})(\nu_1) \sqsubseteq \mathrm{I}(\mathrm{r})(\nu_3)$$

Then, by transitivity of $\equiv$.

The $\equiv$–$\sqsubseteq$ law To prove:

$$\forall \nu_1, \nu_2 : \mathsf{VExpr}(\mathsf{Atom}).\, \nu_1 \equiv_{\mathsf{v}} \nu_2 \implies \nu_1 \sqsubseteq_{\mathsf{v}} \nu_2$$

Unfold:

$$\mathrm{I}(\mathrm{r})(\nu_1) \equiv \mathrm{I}(\mathrm{r})(\nu_2) \implies \mathrm{I}(\mathrm{r})(\nu_1) \sqsubseteq \mathrm{I}(\mathrm{r})(\nu_2)$$

Then, by $\equiv$–$\sqsubseteq$ over the underlying setoid.

The $\sqsubseteq$–$\equiv$ law To prove:

$$\forall \nu_1, \nu_2 : \mathsf{VExpr}(\mathsf{Atom}).\, \nu_1 \sqsubseteq_{\mathsf{v}} \nu_2 \wedge \nu_2 \sqsubseteq_{\mathsf{v}} \nu_1 \implies \nu_1 \equiv_{\mathsf{v}} \nu_2$$

Unfold:

$$\mathrm{I}(\mathrm{r})(\nu_1) \sqsubseteq \mathrm{I}(\mathrm{r})(\nu_2) \wedge \mathrm{I}(\mathrm{r})(\nu_2) \sqsubseteq \mathrm{I}(\mathrm{r})(\nu_1) \implies \mathrm{I}(\mathrm{r})(\nu_1) \equiv \mathrm{I}(\mathrm{r})(\nu_2)$$

Then, by $\sqsubseteq$–$\equiv$ over the underlying setoid.

Increasing To prove: $\forall \nu_1, \nu_2, \nu_3 : \mathsf{VExpr}(\mathsf{Atom}).\, \nu_1 \sqsubseteq_{\mathsf{v}} \nu_2 \implies (\nu_1 *_{\mathsf{v}} \nu_3) \sqsubseteq_{\mathsf{v}} (\nu_2 *_{\mathsf{v}} \nu_3)$.

Backwards proof from RHS to LHS:

$$
\begin{array}{lll}
 & (v_1 *_{\mathsf{v}} v_3) \sqsubseteq_{\mathsf{v}} (v_2 *_{\mathsf{v}} v_3) & \\
\longleftarrow & \mathrm{I}(r)(v_1 *_{\mathsf{v}} v_3) \sqsubseteq \mathrm{I}(r)(v_2 *_{\mathsf{v}} v_3) & \text{(Unfold)} \\
\longleftarrow & \mathrm{I}(r)((\bullet\ v_1\ v_3)) \sqsubseteq \mathrm{I}(r)((\bullet\ v_2\ v_3)) & \text{(Unfold)} \\
\longleftarrow & \mathrm{I}(r)(v_1) \bullet \mathrm{I}(r)(v_3) \sqsubseteq \mathrm{I}(r)(v_2) \bullet \mathrm{I}(r)(v_3) & \text{(Definition of I)} \\
\longleftarrow & \mathrm{I}(r)(v_1) \sqsubseteq \mathrm{I}(r)(v_2) & \text{(Compatibility of underlying semigroup)} \\
\longleftarrow & v_1 \sqsubseteq_{\mathsf{v}} v_2 & \text{(Definition of I)}
\end{array}
$$

Inflation To prove: $\forall v_1, v_2 : \mathsf{VExpr}(\mathsf{Atom}).\, v_1 \sqsubseteq_{\mathsf{v}} (v_1 *_{\mathsf{v}} v_2)$. Unfold:

$$
\begin{array}{ll}
 & v_1 \sqsubseteq_{\mathsf{v}} (v_1 *_{\mathsf{v}} v_2) \\
\longrightarrow & \mathrm{I}(r)(v_1) \sqsubseteq_{\mathsf{v}} \mathrm{I}(r)(v_1 *_{\mathsf{v}} v_2) \\
\longrightarrow & \mathrm{I}(r)(v_1) \sqsubseteq_{\mathsf{v}} \mathrm{I}(r)((\bullet\ v_1\ v_2)) \\
\longrightarrow & \mathrm{I}(r)(v_1) \sqsubseteq_{\mathsf{v}} \mathrm{I}(r)(v_1) \bullet \mathrm{I}(r)(v_2)
\end{array}
$$

Then apply inflation property of underlying ordered views semigroup.

□

Theorem B.8: view expressions are subtractive views semigroups

Proof. By Definition 3.8, we must show that $\setminus_{\mathsf{v}}$ is increasing with respect to $\sqsubseteq_{\mathsf{v}}$, and that $\mathsf{VExpr}(\mathsf{Atom})$ has an adjoint property over $\sqsubseteq_{\mathsf{v}}$.

Compatibility To prove: $\forall v_1, v_2, v_3 : \mathsf{VExpr}(\mathsf{Atom}).\, v_1 \sqsubseteq_{\mathsf{v}} v_2 \implies (v_1 \setminus_{\mathsf{v}} v_3) \sqsubseteq_{\mathsf{v}} (v_2 \setminus_{\mathsf{v}} v_3)$.

Backwards proof from RHS to LHS:

$$
\begin{array}{lll}
 & (v_1 \setminus_{\mathsf{v}} v_3) \sqsubseteq_{\mathsf{v}} (v_2 \setminus_{\mathsf{v}} v_3) & \\
\longleftarrow & \mathrm{I}(r)(v_1 \setminus_{\mathsf{v}} v_3) \sqsubseteq \mathrm{I}(r)(v_2 \setminus_{\mathsf{v}} v_3) & \text{(Unfold)} \\
\longleftarrow & \mathrm{I}(r)((\ \setminus v_1\ v_3)) \sqsubseteq \mathrm{I}(r)((\ \setminus v_2\ v_3)) & \text{(Unfold)} \\
\longleftarrow & \mathrm{I}(r)(v_1) \setminus \mathrm{I}(r)(v_3) \sqsubseteq \mathrm{I}(r)(v_2) \setminus \mathrm{I}(r)(v_3) & \text{(Definition of I)} \\
\longleftarrow & \mathrm{I}(r)(v_1) \sqsubseteq \mathrm{I}(r)(v_2) & \text{(Compatibility of underlying views semigroup)} \\
\longleftarrow & v_1 \sqsubseteq_{\mathsf{v}} v_2 & \text{(Definition of I)}
\end{array}
$$

Adjoint To prove: $\forall v_1, v_2, v_3 : \mathsf{VExpr}(\mathsf{Atom}).\, v_1 \sqsubseteq_{\mathsf{v}} (v_2 *_{\mathsf{v}} v_3) \implies (v_1 \setminus_{\mathsf{v}} v_2) \sqsubseteq_{\mathsf{v}} v_3$.

Backwards proof from RHS to LHS:

$$
\begin{aligned}
& (v_1 \setminus_v v_2) \sqsubseteq_v v_3 \\
\longleftarrow\ & I(r)(v_1 \setminus_v v_2) \sqsubseteq I(r)(v_3) && \text{(Unfold)} \\
\longleftarrow\ & I(r)(\texttt{(}\ \setminus v_1\ \ v_2\texttt{)}) \sqsubseteq I(r)(v_3) && \text{(Unfold)} \\
\longleftarrow\ & I(r)(v_1) \setminus I(r)(v_2) \sqsubseteq I(r)(v_3) && \text{(Definition of I)} \\
\longleftarrow\ & I(r)(v_1) \sqsubseteq I(r)(v_2) \bullet I(r)(v_3) && \text{(Adjoint property on underlying semigroup)} \\
\longleftarrow\ & I(r)(v_1) \sqsubseteq I(r)(\texttt{(*}\ v_2\ \ v_3\texttt{)}) && \text{(Definition of I)} \\
\longleftarrow\ & I(r)(v_1) \sqsubseteq I(r)(v_2 *_v v_3) && \text{(Definition of } *_v\text{)} \\
\longleftarrow\ & v_1 \sqsubseteq_v (v_2 *_v v_3) && \text{(Definition of I)}
\end{aligned}
$$

□

B.3 Local Views Framework

Lemma 5.14

Proof. By backwards rewrite:

$$
\begin{aligned}
& \forall l, l'.\, \mathsf{Sig}^{\uparrow}_{lo}(s_{lo}), (\widehat{\alpha}, l, l') \Vdash_{\mathsf{VFH}} \{p(l)\}\{q(l')\} \\
=\ & \forall l, l'.\, (\mathsf{Sem}^{\uparrow}_{lo}([\![-]\!]_{lo})(\widehat{\alpha}, l, l'))^*(\lfloor p(l) \rfloor) \subseteq \lfloor q(l') \rfloor && \text{(unfold Definition 2.16)} \\
=\ & \forall l, l'.\, \bigcup \left\{ (\mathsf{Sem}^{\uparrow}_{lo}([\![-]\!]_{lo})(\widehat{\alpha}, l, l'))^{\mathsf{id}}(\sigma) \;\middle|\; \sigma \in \lfloor p(l) \rfloor \right\} \subseteq \lfloor q(l') \rfloor \\
& && \text{(unfold Definition 2.12)} \\
=\ & \forall l, l', \sigma.\, \sigma \in \lfloor p(l) \rfloor \implies (\mathsf{Sem}^{\uparrow}_{lo}([\![-]\!]_{lo})(\widehat{\alpha}, l, l'))^{\mathsf{id}}(\sigma) \subseteq \lfloor q(l') \rfloor \\
& && \text{(expand out iterated union)} \\
\iff\ & \forall l, l', \sigma.\, \sigma \in \lfloor p(l) \rfloor \implies \mathsf{Sem}^{\uparrow}_{lo}([\![-]\!]_{lo})(\widehat{\alpha}, l, l'))(\sigma) \subseteq \lfloor q(l') \rfloor && \text{(label is not id)} \\
=\ & \forall l, l', \sigma.\, \sigma \in \lfloor p(l) \rfloor \implies \{\, \sigma' \mid ((l, \sigma), (l', \sigma')) \in [\![\widehat{\alpha}]\!]^{\mathsf{id}}_{lo} \,\} \subseteq \lfloor q(l') \rfloor && \text{(unfold)} \\
=\ & \forall l, l', \sigma, \sigma'.\, \sigma \in \lfloor p(l) \rfloor \implies \big(((l, \sigma), (l', \sigma')) \in [\![\widehat{\alpha}]\!]^{\mathsf{id}}_{lo} \implies \sigma' \in \lfloor q(l') \rfloor\big) \\
& && \text{(expand set-builder)} \\
=\ & \forall \sigma, \sigma' \in S.\, \forall l, l' \in L.\ \sigma \in \lfloor p(l) \rfloor \wedge ((l, \sigma), (l', \sigma')) \in [\![\alpha]\!]^{\mathsf{id}}_{lo} \implies \sigma' \in \lfloor p(l') \rfloor \\
& && \text{(rearrange quantifications and double implication)} \\
=\ & s_{lo}, \widehat{\alpha} \Vdash_{\mathsf{LVFH}} \{p\}\{q\} && \text{(Definition 5.5)}
\end{aligned}
$$

□

Theorem 5.2 (sketch)

Proof sketch. By co-induction. First, split the multi-thread semantic judgement into its two cases. In the first, all threads are `skip`, and we satisfy the semantic judgements by applying the corresponding result from each thread's single-thread judgement.

In the second case, we unfold Definition 5.13 to determine that there exists some c' such that $\mathbf{c}' = \mathbf{c}[t \mapsto c']$, and a single-thread transition exists between the two on thread t. We

then apply this to the single-thread semantic judgement to get the correct value for r and the action judgement. For the remaining case, we recur co-inductively, which gives us the obligation to show single-thread safety for each thread over the new triple $\{\mathbf{p}[t \mapsto r]\}\ \mathbf{c}'\ \{\mathbf{q}\}$. We do this by case analysis: where the thread is t, we use the safety result from the transition we just made; otherwise, we note that the view and program have not changed for that thread, and re-use the original result. □

Glossaries

Glossary

Symbols

$_{\mu}$Starling

Starling frontend supporting shared-state reasoning only; see § 4.4 49, 201

A

action judgement

Views judgement that states that a Hoare triple is safe modulo all possible contexts; see Definition 2.17 28, 29, 201, 202

atom set

set of primitive components used in view expressions 89, 201, 204

atomic action language

(*Views* parameter) the set of syntactic atomic actions for use inside atomic program steps; see Definition 2.8 26, 201, 204

atomic Hoare triple

Hoare triple over a single atomic action; see Definition 2.14 28, 201, 204

atomic label language

Atomic action language extended with id; see Definition 2.10 27, 201, 204

B

backend decomposition

Frontend element: map from atomic Hoare triples to backend conditions; see Definition 4.9 55, 201, 204

base downclosure

Iterated definition property: definition of empty pattern must imply all iterated definitions when the iterator is 0 121, 122, 201

C

constant function

A function of two parameters x and y that always returns x; see Definition A.2 177, 201, 205

F

Floyd/Hoare-style safety judgement

Traditional safety (or 'partial correctness') judgement on Hoare triples 19, 201, 204

free views instance

a views instance with an axiomatisation defined directly over the action judgement; see Definition 3.3 40, 201, 205

function erasure

Lifts a set of functional verification conditions to a set over its codomain; see Definition 5.22 78, 201, 204

I

inductive downclosure

Iterated definition property: iterated definitions at iterator $n + 1$ must imply the same definitions at iterator n 122, 201

L

label semantic function

lifting of semantic function to map id to the identity transformer; see Definition 2.11 27, 201, 202, 205

lifted semantic function

lifting of label semantic function to state sets; see Definition 2.12 27, 201, 205

list override

Operation that replaces one element of a list at a specified index; see Definition A.5 70, 177, 201, 205

local action judgement

Extension of the action judgement to local view functions; see Definition 5.6 65, 201, 205

local views–Hoare judgement

Extension of the views–Floyd/Hoare judgement to local view functions; see Definition 5.5 65, 201, 205

M

multiset

See Definition A.3 86, 177, 201, 205

P

proposition expression

syntactic representation of propositions over one states, representing a backend's view of view definitions; see Definition 4.4 53, 201, 204, 205

R

relation expression

syntactic representation of relations over two states, representing a backend's view of atomic actions; see Definition 4.5 53, 201, 204, 205

relational frame

Function to frame a proposition-as-relation expression over a set of variables; see Definition 6.18 100, 201, 205

S

semantic function

Views parameter: function from atomic actions to shared-state transformers. See definition 2.9 201, 202, 205

setoid

A set with an equivalence relation; see definition A.1 25, 87, 88, 177, 201, 204

structured propositions

Common language for $_{g}$Starling proposition expressions; see Definition 6.19 95, 101, 113, 201

T

truncated subtraction

Subtraction, closed over natural numbers, that saturates to zero; see definition A.4 177, 201, 205

V

verification condition

Hoare triple over backend predicate and relation expressions, representing solver input; see Definition 4.6 54, 116, 201, 202, 205

verification-condition Hoare judgement

Hoare-style judgement over verification conditions, which should be implied by solver predicates; see Definition 4.7 54, 78, 102, 164, 201, 204

view expressions

Common language for $_{\mathsf{g}}$Starling assertions; see Definition 6.2 89, 126, 201, 205

views decomposition

Frontend element: map from atomic Hoare triples to *Views* axioms; see Definition 4.9 55, 201, 205

views–Floyd/Hoare judgement

Atomic safety judgement over views; see Definition 2.16 28, 65, 201, 203, 205

Symbols

Notation	Description
α	(meta-syntactic) atomic action
$\widehat{\alpha}$	(meta-syntactic) atomic label
Atom	(meta-syntactic) atom set
$\mathsf{D}_{\mathsf{b}}^{lo}$	$_{\mathsf{lo}}$Starling backend decomposition; see Definition 5.27
$\mathsf{D}_{\mathsf{b}}^{\mu}$	$_{\mu}$Starling backend decomposition; see Definition 4.15
D_{b}	(meta-syntactic) backend decomposition
E_{Pr}	(meta-syntactic) set of proposition expressions
E_{Rl}	(meta-syntactic) set of relation expressions
A	(meta-syntactic) Atomic action language
$\Vdash_{\mathsf{VCFH}}$	verification-condition Hoare judgement
FErase	function erasure
$\mathsf{D}_{\mathsf{g}}^{lo}$	$_{\mathsf{lo}}$Starling decomposition from atomic Hoare triple–goal pairs to backend conditions; see Definition 5.26
$\mathsf{D}_{\mathsf{g}}^{\mu}$	$_{\mu}$Starling decomposition from atomic Hoare triple–goal pairs to backend conditions; see Definition 4.14
$\equiv$	setoid equivalence function
$\Vdash_{\mathsf{FH}}$	Floyd/Hoare-style safety judgement
id	Identity atomic action; see Definition 2.10
A^{id}	Atomic label language

Notation	Description
$[\![-]\!]^{\mathsf{id}}$	Label semantic function
$-, - \Vdash_{lo} \{-\}\{-\}$	local action judgement
$\mathsf{finst}(-)$	free views instance over the given signature
$- \sim (-;-)$	atomic composition; see Definition 5.30
rframe	relational frame
$[\![-]\!]$	semantic function
$[\![-]\!]^*$	lifted semantic function
$\mathsf{VConds}(\mathsf{E}_{\mathsf{Pr}}, \mathsf{E}_{\mathsf{Rl}})$	verification condition set over E_{Pr} and E_{Rl}
D_v	(meta-syntactic) views decomposition
$\mathsf{VExpr}(\mathsf{Atom})$	set of view expressions over Atom
$\langle\!\langle w \rangle\!\rangle$ c $\langle\!\langle \mathrm{g} \rangle\!\rangle$	verification condition over w, c, and g
c	(meta-syntactic) command relation expression
g	(meta-syntactic) goal proposition expression
w	(meta-syntactic) 'weakest-precondition' proposition expression
$\mathbf{l}[\mathrm{i} \mapsto \mathrm{x}]$	List override (replacing the ith element of $\mathbf{l}$ by x)
$\Vdash_{\mathsf{LVFH}}$	local views–Hoare judgement
bag T	multiset
$\dot{-}$	truncated subtraction
$\Vdash_{\mathsf{VFH}}$	views–Floyd/Hoare judgement
const	constant function

Bibliography

[1] M. Ben-Ari, *Principles of Concurrent Programming*. Prentice-Hall International, 1982.

[2] M. Windsor, M. Dodds, B. Simner, and M. J. Parkinson, "Starling: Lightweight concurrency verification with views," in *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*, 2017, pp. 544–569. [Online]. Available: https://doi.org/10.1007/978-3-319-63387-9_27

[3] H. Sutter, "The free lunch is over: a fundamental turn toward concurrency in software." *Dr. Dobb's Journal*, vol. 30, no. 3, Mar 2005. [Online]. Available: http://www.gotw.ca/publications/concurrency-ddj.htm

[4] A. W. Roscoe, *The Theory and Practice of Concurrency*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1997.

[5] H.-J. Boehm, "Position paper: Nondeterminism is unavoidable, but data races are pure evil," in *Proceedings of the 2012 ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability*, ser. RACES '12. New York, NY, USA: ACM, 2012, pp. 9–14. [Online]. Available: http://doi.acm.org/10.1145/2414729.2414732

[6] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming, Revised Reprint*. Morgan Kaufman, 2012.

[7] N. G. Leveson and C. S. Turner, "An investigation of the therac-25 accidents," *Computer*, vol. 26, no. 7, pp. 18–41, July 1993.

[8] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, Jul. 1990. [Online]. Available: http://doi.acm.org/10.1145/78969.78972

[9] T. Henzinger, A. Sezgin, and V. Vafeiadis, "Aspect-oriented linearizability proofs," in *CONCUR 2013 – Concurrency Theory*, ser. Lecture Notes in Computer Science, P. D'Argenio and H. Melgratti, Eds. Springer Berlin Heidelberg, 2013, vol. 8052, pp. 242–256. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-40184-8_18

[10] A. Gerrand, "Share memory by communicating," Jul. 2010. [Online]. Available: https://blog.golang.org/share-memory-by-communicating

[11] S. Marlow, R. Newton, and S. Peyton Jones, "A monad for deterministic parallelism," in *Proceedings of the 4th ACM Symposium on Haskell*, ser. Haskell '11. New York, NY, USA: ACM, 2011, pp. 71–82. [Online]. Available: http://doi.acm.org/10.1145/2034675.2034685

[12] M. J. Parkinson, "The next 700 separation logics - (invited paper)," in *Verified Software: Theories, Tools, Experiments, Third International Conference, VSTTE 2010, Edinburgh, UK, August 16-19, 2010. Proceedings*, 2010, pp. 169–182. [Online]. Available: https://doi.org/10.1007/978-3-642-15057-9_12

[13] L. Lamport, "Solved problems, unsolved problems and non-problems in concurrency," *SIGOPS Oper. Syst. Rev.*, vol. 19, no. 4, pp. 34–44, Oct. 1985. [Online]. Available: http://doi.acm.org/10.1145/858336.858339

[14] A. Laarman, J. van de Pol, and M. Weber, "Boosting multi-core reachability performance with shared hash tables," in *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*, ser. FMCAD '10. Austin, TX: FMCAD Inc, 2010, pp. 247–256. [Online]. Available: http://dl.acm.org/citation.cfm?id=1998496.1998541

[15] E. W. Dijkstra, "On the cruelty of really teaching computing science," Dec. 1988. [Online]. Available: http://www.cs.utexas.edu/users/EWD/ewd10xx/EWD1036.PDF

[16]

[17] R. P. Case and A. Padegs, "Architecture of the ibm system/370," *Commun. ACM*, vol. 21, no. 1, pp. 73–96, Jan. 1978. [Online]. Available: http://doi.acm.org/10.1145/359327.359337

[18] R. K. Treiber, *Systems programming: Coping with parallelism*. International Business Machines Incorporated, Thomas J. Watson Research Center, 1986.

[19] G. Peterson, "Myths about the mutual exclusion problem," *Information Processing Letters*, vol. 12, no. 3, pp. 115–116, 1981. [Online]. Available: http://www.sciencedirect.com/science/article/pii/002001908190106X

[20] E. M. Clarke and J. M. Wing, "Formal methods: State of the art and future directions," *ACM Comput. Surv.*, vol. 28, no. 4, pp. 626–643, Dec. 1996. [Online]. Available: http://doi.acm.org/10.1145/242223.242257

[21] C. Hoare, "Viewpoint: Retrospective: An axiomatic basis for computer programming," *Commun. ACM*, vol. 52, no. 10, pp. 30–32, Oct. 2009. [Online]. Available: http://doi.acm.org/10.1145/1562764.1562779

[22] C. B. Jones, I. J. Hayes, and R. J. Colvin, "Balancing expressiveness in formal approaches to concurrency," *Formal Aspects of Computing*, vol. 27, no. 3, pp. 475–497, May 2015. [Online]. Available: http://dx.doi.org/10.1007/s00165-014-0310-2

[23] Z. Manna and A. Pnueli, "Axiomatic approach to total correctness of programs," *Acta Informatica*, vol. 3, no. 3, pp. 243–263, Sep 1974. [Online]. Available: https://doi.org/10.1007/BF00288637

[24] V. Vafeiadis, "Modular fine-grained concurrency verification," University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-726, Jul 2008. [Online]. Available: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-726.pdf

[25] W.-P. De Roever, *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Cambridge University Press, 2001, vol. 54.

[26] C. Flanagan, S. N. Freund, S. Qadeer, and S. A. Seshia, "Modular verification of multithreaded programs," *Theoretical Computer Science*, vol. 338, no. 1, pp. 153 – 183, 2005. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0304397504007935

[27] J. Hoenicke, R. Majumdar, and A. Podelski, "Thread modularity at many levels: A pearl in compositional verification," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL 2017. New York, NY, USA: ACM, 2017, pp. 473–485. [Online]. Available: http://doi.acm.org/10.1145/3009837.3009893

[28] J. Reynolds, "Separation logic: a logic for shared mutable data structures," in *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, 2002, pp. 55–74.

[29] K. Svendsen and L. Birkedal, "Impredicative concurrent abstract predicates," in *Programming Languages and Systems*, ser. Lecture Notes in Computer Science, Z. Shao, Ed. Springer Berlin Heidelberg, 2014, vol. 8410, pp. 149–168. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-54833-8_9

[30] R. W. Floyd, "Assigning meanings to programs," in *Proceedings of Symposia in Applied Mathematics*, vol. 19, 1967, pp. 19–32.

[31] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, Oct. 1969. [Online]. Available: http://doi.acm.org/10.1145/363235.363259

[32] T. Hoare and S. van Staden, "In praise of algebra," *Formal Aspects of Computing*, vol. 24, no. 4, pp. 423–431, Jul 2012. [Online]. Available: https://doi.org/10.1007/s00165-012-0249-0

[33] C. Hoare, "Towards a theory of parallel programming," in *The Origin of Concurrent Programming*, P. Hansen, Ed. Springer New York, 2002, pp. 231–244. [Online]. Available: http://dx.doi.org/10.1007/978-1-4757-3472-0_6

[34] S. Owicki and D. Gries, "An axiomatic proof technique for parallel programs i," *Acta Informatica*, vol. 6, no. 4, pp. 319–340, Dec 1976. [Online]. Available: http://dx.doi.org/10.1007/BF00268134

[35] E. W. Dijkstra, "A constructive approach to the problem of program correctness," *BIT Numerical Mathematics*, vol. 8, no. 3, pp. 174–186, Sep 1968. [Online]. Available: http://dx.doi.org/10.1007/BF01933419

[36] O. Lahav and V. Vafeiadis, "Owicki-gries reasoning for weak memory models," in *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part II*, ser. Lecture Notes in Computer Science, M. M. Halldórsson, K. Iwama, N. Kobayashi, and B. Speckmann, Eds., vol. 9135. Springer, 2015, pp. 311–323. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-47666-6_25

[37] C. B. Jones, "Developing methods for computer programs including a notion of interference," Ph.D. dissertation, University of Oxford, UK, 1981. [Online]. Available: http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.259064

[38] ——, "Specification and design of (parallel) programs," in *Information Processing, 1983. Proceedings. 9th World Computer Congress on*. Elsevier Science Publishers B.V. (North-Holland), 1983, pp. 321–322.

[39] I. J. Hayes, C. B. Jones, and R. J. Colvin, "Reasoning about concurrent programs: Refining rely-guarantee thinking," School of Computing Science, Newcastle University, Tech. Rep. CS-TR-1395, September 2013.

[40] C. Morgan, *Programming from specifications, 2nd Edition*, ser. Prentice Hall International series in computer science. Prentice Hall, 1994.

[41] C. Calcagno, P. W. O'Hearn, and H. Yang, "Local action and abstract separation logic," in *22nd IEEE Symposium on Logic in Computer Science (LICS 2007), 10-12 July 2007, Wroclaw, Poland, Proceedings*. IEEE Computer Society, 2007, pp. 366–378. [Online]. Available: http://dx.doi.org/10.1109/LICS.2007.30

[42] T. Dinsdale-Young, L. Birkedal, P. Gardner, M. Parkinson, and H. Yang, "Views: Compositional reasoning for concurrent programs," in *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '13. New York, NY, USA: ACM, 2013, pp. 287–300. [Online]. Available: http://doi.acm.org/10.1145/2429069.2429104

[43] R. Jung, R. Krebbers, J.-H. Jourdan, A. Bizjak, L. Birkedal, and D. Dreyer, "Iris from the ground up: A modular foundation for higher-order concurrent separation logic," *Journal of Functional Programming*, vol. 28, p. e20, 2018.

[44] I. Wehrman, C. A. R. Hoare, and P. W. O'Hearn, "Graphical models of separation logic," *Inf. Process. Lett.*, vol. 109, no. 17, pp. 1001–1004, 2009. [Online]. Available: https://doi.org/10.1016/j.ipl.2009.06.003

[45] J. Boyland, *Checking Interference with Fractional Permissions*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 55–72. [Online]. Available: http://dx.doi.org/10.1007/3-540-44898-5_4

[46] T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis, "Concurrent abstract predicates," in *ECOOP 2010 – Object-Oriented Programming: 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings*, T. D'Hondt, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 504–528. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-14107-2_24

[47] T. Dinsdale-Young, P. da Rocha Pinto, K. J. Andersen, and L. Birkedal, "Caper - automatic verification for fine-grained concurrency," in *Programming Languages and Systems: 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22–29, 2017, Proceedings*, H. Yang, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 420–447. [Online]. Available: https://doi.org/10.1007/978-3-662-54434-1_16

[48] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, C. R. Ramakrishnan and J. Rehof, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340. [Online]. Available: https://doi.org/10.1007/978-3-540-78800-3_24

[49] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, "Boogie: A modular reusable verifier for object-oriented programs," in *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 364–387. [Online]. Available: https://doi.org/10.1007/11804192_17

[50] N. Bjørner, A. Gurfinkel, K. McMillan, and A. Rybalchenko, "Horn clause solvers for program verification," in *Fields of Logic and Computation II: Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*, L. D. Beklemishev, A. Blass, N. Dershowitz, B. Finkbeiner, and W. Schulte, Eds. Cham: Springer International Publishing, 2015, pp. 24–51. [Online]. Available: https://doi.org/10.1007/978-3-319-23534-9_2

[51] A. Gupta, C. Popeea, and A. Rybalchenko, *Threader: A Constraint-Based Verifier for Multi-threaded Programs*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 412–417. [Online]. Available: https://doi.org/10.1007/978-3-642-22110-1_32

[52] S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko, "Synthesizing software verifiers from proof rules," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12. New York, NY, USA: ACM, 2012, pp. 405–416. [Online]. Available: http://doi.acm.org/10.1145/2254064.2254112

[53] I. Sergey, A. Nanevski, and A. Banerjee, "Mechanized verification of fine-grained concurrent programs," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '15. New York, NY, USA: ACM, 2015, pp. 77–87. [Online]. Available: http://doi.acm.org/10.1145/2737924.2737964

[54] L. Lamport, "The temporal logic of actions," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 3, pp. 872–923, May 1994. [Online]. Available: http://doi.acm.org/10.1145/177492.177726

[55] R. Piskac, T. Wies, and D. Zufferey, "Grasshopper - complete heap verification with mixed specifications," in *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, 2014, pp. 124–139. [Online]. Available: https://doi.org/10.1007/978-3-642-54862-8_9

[56] A. Khyzha, A. Gotsman, and M. Parkinson, "A generic logic for proving linearizability," in *FM 2016: Formal Methods*, J. Fitzgerald, C. Heitmeyer, S. Gnesi, and A. Philippou, Eds. Cham: Springer International Publishing, 2016, pp. 426–443.

[57] A. W. Appel, *Program Logics - for Certified Compilers*. Cambridge University Press, 2014. [Online]. Available: http://www.cambridge.org/de/academic/subjects/computer-science/programming-languages-and-applied-logic/program-logics-certified-compilers?format=HB

[58] G. Hutton, "A tutorial on the universality and expressiveness of fold," *J. Funct. Program.*, vol. 9, no. 4, pp. 355–372, 1999. [Online]. Available: http://journals.cambridge.org/action/displayAbstract?aid=44275

[59] Q. Cao, S. Wang, A. Hobor, and A. W. Appel, "Proof pearl: Magic wand as frame," Feb 2018, unpublished. [Online]. Available: http://www.cs.princeton.edu/~appel/papers/wand-frame.pdf

[60] B. Huffman, "Formal verification of monad transformers," *CoRR*, vol. abs/1207.3208, 2012. [Online]. Available: http://arxiv.org/abs/1207.3208

[61] C. A. R. Hoare and H. Jifeng, *Unifying theories of programming*. Prentice Hall Englewood Cliffs, 1998, vol. 14.

[62] C. Barrett, P. Fontaine, and C. Tinelli, "The SMT-LIB Standard: Version 2.6," Department of Computer Science, The University of Iowa, Tech. Rep., 2017, available at http://www.SMT-LIB.org.

[63] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Trans. Comput. Syst.*, vol. 9, no. 1, pp. 21–65, 1991. [Online]. Available: http://doi.acm.org/10.1145/103727.103729

[64] G. P. Huet, "The zipper," *J. Funct. Program.*, vol. 7, no. 5, pp. 549–554, 1997. [Online]. Available: http://journals.cambridge.org/action/displayAbstract?aid=44121

[65] "The go programming language specification," May 2018. [Online]. Available: https://golang.org/ref/spec

[66] (2018, May) Intel 64 and ia-32 architectures software developer's manual. Intel Corporation. [Online]. Available: https://software.intel.com/en-us/articles/intel-sdm

[67] E. E. Dijkstra, "An assertional proof of an algorithm by g. l. peterson," Feb. 1981. [Online]. Available: https://www.cs.utexas.edu/users/EWD/ewd07xx/EWD779.PDF

[68] S. Klabnik and C. Nichols, *The Rust Programming Language*. No Starch Press, Jun. 2018. [Online]. Available: https://doc.rust-lang.org/book/second-edition/index.html

[69] D. Dreyer, "Deductive verification of concurrent programs: logical foundations," May 2016, presentation slides. [Online]. Available: https://people.mpi-sws.org/~dreyer/talks/talk-dagstuhl16.pdf

[70] T. S. Craig, "Building fifo and priority-queuing spin locks from atomic swap," University of Washington, Tech. Rep. 93-02-02, Feb 1993.

[71] P. S. Magnusson, A. Landin, and E. Hagersten, "Queue locks on cache coherent multiprocessors," in *Proceedings of the 8th International Symposium on Parallel Processing, Cancún, Mexico, April 1994*, 1994, pp. 165–171. [Online]. Available: https://doi.org/10.1109/IPPS.1994.288305

[72] A. Chlipala, *Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, 2013. [Online]. Available: http://mitpress.mit.edu/books/certified-programming-dependent-types

[73] K. R. M. Leino, "Dafny: An automatic program verifier for functional correctness," in *Logic for Programming, Artificial Intelligence, and Reasoning*, E. M. Clarke and A. Voronkov, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 348–370.

[74] Dafny: A language and program verifier for functional correctness. Microsoft Research. [Online]. Available: https://www.microsoft.com/en-us/research/project/dafny-a-language-and-program-verifier-for-functional-correctness/

[75] B. Jacobs and F. Piessens, "The verifast program verifier," Department of Computer Science, Katholieke Universiteit Leuven, Belgium, Tech. Rep., Aug. 2008. [Online]. Available: https://people.cs.kuleuven.be/~bart.jacobs/verifast/verifast.pdf

[76] B. Jacobs, J. Smans, and F. Piessens. (2017, Nov.) The verifast program verifier: A tutorial. [Online]. Available: https://people.cs.kuleuven.be/~bart.jacobs/verifast/tutorial.pdf

[77] A. Chlipala, "Mostly-automated verification of low-level programs in computational separation logic," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, 2011, pp. 234–245. [Online]. Available: http://doi.acm.org/10.1145/1993498.1993526

[78] (2017, Sep) Verified software toolchain. Princeton University. [Online]. Available: http://vst.cs.princeton.edu

[79] V. Vafeiadis and C. Narayan, "Relaxed separation logic: a program logic for C11 concurrency," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, 2013, pp. 867–884. [Online]. Available: https://doi.org/10.1145/2509136.2509532

[80] A. Turon, V. Vafeiadis, and D. Dreyer, "Gps: Navigating weak memory with ghosts, protocols, and separation," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '14. New York, NY, USA: ACM, 2014, pp. 691–707. [Online]. Available: http://doi.acm.org/10.1145/2660193.2660243

[81] J. Kaiser, H. Dang, D. Dreyer, O. Lahav, and V. Vafeiadis, "Strong logic for weak memory: Reasoning about release-acquire consistency in iris," in *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*, 2017, pp. 17:1–17:29. [Online]. Available: https://doi.org/10.4230/LIPIcs.ECOOP.2017.17

[82] M. Dodds, X. Feng, M. J. Parkinson, and V. Vafeiadis, "Deny-guarantee reasoning," in *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, 2009, pp. 363–377. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-00590-9_26

[83] (2019, Feb) Fdr4 - the csp refinement checker. University of Oxford.

[84] M. Dodds, A. Haas, and C. M. Kirsch, "A scalable, correct time-stamped stack," *SIGPLAN Not.*, vol. 50, no. 1, pp. 233–246, Jan. 2015. [Online]. Available: http://doi.acm.org/10.1145/2775051.2676963

[85] B. W. Kernighan and D. Ritchie, *The C Programming Language, Second Edition*. Prentice-Hall, 1988.