Starling: Lightweight Concurrency Verification With Views

Matt Windsor[1], Mike Dodds[1], Ben Simner[1], and Matthew J. Parkinson[2]

[1] University of York, UK.
{mbw500,mike.dodds,bs829}@york.ac.uk
[2] Microsoft Research. mattpark@microsoft.com

Abstract. Modern program logics have made it feasible to verify the most complex concurrent algorithms. However, many such logics are complex, and most lack automated tool support. We propose *Starling*, a new lightweight logic and automated tool for concurrency verification. Starling takes a proof outline written in an abstracted Hoare-logic style, and converts it into proof terms that can be discharged by a sequential solver. Starling's approach is generic in its structure, making it easy to target different solvers. In this paper we verify shared-variable algorithms using the Z3 SMT solver, and heap-based algorithms using the GRASShopper solver. We have applied our approach to a range of concurrent algorithms, including Rust's atomic reference counter, the Linux ticketed lock, the CLH queue-lock, and a fine-grained list algorithm.

1 Introduction

Shared-memory concurrent algorithms are critical components of many systems, for example as locks, reference counters, work-queues, and garbage collectors [12]. These algorithms must achieve high performance, while also enforcing properties such as mutual exclusion and safe memory reclamation. In pursuit of performance, modern algorithms have become increasingly complex. As a result, by-hand correctness arguments are unreliable, and formal verification remains very challenging.

Concurrent algorithms often depend on intangible concepts such as thread-local *ownership* of resources, and *protocols* between threads. For example, a thread that acquires a lock takes ownership of the guarded resource, and the mutual exclusion protocol forbids other threads from accessing the lock at the same time. Beginning with Concurrent Separation Logic (CSL) [18], program logics have integrated these concepts directly in reasoning, which has enabled the verification of many challenging algorithms (see §7, Related Work).

However, these logics derived from CSL are very complex, with auxiliary proof constructs such as fractional permissions, shared regions, and labelled transition systems. Complexity makes these logics difficult to learn and difficult to reason with, and non-standard proof constructs make tooling hard to develop, and therefore rare. As a result, there are substantial barriers to applying these logics in practice.

We present *Starling*, a new program logic and verification tool for concurrent algorithms. Our approach is inspired by CSL and its relatives, but we dispense with heavyweight auxiliary proof concepts. Starling's proofs are lightweight, easy to read, and easy to automate – but powerful enough to verify challenging concurrent algorithms.

Starling's approach is based on *views* – units of linear, invariant information that can be held by a single thread. Proofs in Starling are written in a lightweight proof-outline style, with views annotating program points and *constraints* defining their meaning in the underlying domain. Notions such as ownership and protocol can be expressed through interactions between views. For example, we can have a view expressing that the thread holds a lock, then express mutual exclusion by forbidding two threads from holding this view at the same time.

Starling's reasoning is built on the pre-existing *Views framework* [6]: this was designed as an off-the-shelf metatheory for encoding other logics, but we instead instantiate it directly as a simple view-based logic. The Views framework works by reducing a concurrent proof to multiple applications of a single core proof rule. We use this to reduce a Starling proof to a collection of verification conditions that can be discharged using a sequential solver. Building on the Views framework means that Starling requires minimal extra metatheory and can easily be automated.

Our approach is agnostic to the underlying data domain: we require only an appropriate sequential solver. In this paper, we instantiate our approach with two domains. First, for algorithms that use shared variables and linear arithmetic, we generate SMT queries, which are discharged using Z3 [5]. For algorithms that use dynamic linked data-structures, we generate queries written in separation logic, which we discharge using GRASShopper [20]. In both cases, our approach lets us map uniformly from concurrent reasoning into sequential verification conditions.

We have tested Starling on a collection of real-world concurrent algorithms. Many of these are synchronisation algorithms, one of the most important class of concurrent algorithm. Our running example is Rust's Atomic Reference-Count algorithm, which prevents reuse of an object after it has been freed. We also verify several different lock algorithms including the CLH queue-lock algorithm, Peterson's algorithm, and a fine-grained list algorithm. As is often the case in concurrency, these algorithms are small in size but exhibit killer subtleties that make verification very challenging. Other approaches would require considerably more proof annotations, or customised auxiliary proof constructs. We show that these algorithms can be verified using a lightweight, automated approach.

Our tool is open source (MIT license) and available on GitHub:

`https://github.com/septract/starling-tool`

2 Motivating Example: ARC

The Atomic Reference-Count (ARC) algorithm is used to ensure that a shared object is not disposed before all threads are finished with it. In Rust, the ARC forms an important part of the concurrency model [23]. Our version of the ARC has three operations:

`clone`: Clone the ARC reference and increment the counter.
`access`: Fetch or modify the object stored in the ARC.
`drop`: Destroy an ARC reference by decrementing the counter. If the count is 0, dispose the shared object.

2.1 Specification

To specify the ARC using our approach, we first declare the *view atom* $\mathsf{arc}()$. A view atom is a unit of linear, invariant information that can be held by a thread. The atom $\mathsf{arc}()$ states the thread holds a single reference to the ARC object. We do not specify the meaning of $\mathsf{arc}()$ in the program state yet (in this way view atoms resemble the *abstract predicates* of Dinsdale-Young et al [8]).

View atoms can be conjoined into unboundedly large *views* using the composition operator, $*$. This operator is linear, not standard conjunction: for example the view $\mathsf{arc}() * \mathsf{arc}() * \mathsf{arc}()$ asserts that the thread holds three separate references to the ARC object. A thread could also hold zero references to the ARC, represented by the special unit view emp. The $*$ operator is generalised from separating conjunction in separation logic, but views need not have disjoint heap representations.

Using $\mathsf{arc}()$ and emp, we give the ARC operations Hoare-style specifications:

$$\begin{array}{ccc} \{\mathsf{arc}()\} & \texttt{clone()} & \{\mathsf{arc}() * \mathsf{arc}()\} \\ \{\mathsf{arc}()\} & \texttt{access()} & \{\mathsf{arc}()\} \\ \{\mathsf{arc}()\} & \texttt{drop()} & \{\mathsf{emp}\} \end{array}$$

The `clone` method creates a new reference, represented by a duplicate $\mathsf{arc}()$ atom in its postcondition. The `access` method requires an ARC reference to ensure the object has not been disposed: the $\mathsf{arc}()$ atom in its precondition represents this. The `drop` method takes an ARC reference, represented by an $\mathsf{arc}()$ atom, and destroys it leaving emp.

In our tool, specifications are implicitly *framed* with arbitrary views. The frame represents other views held locally or by other threads. For example, the thread might hold three ARC references, and then call `drop()`:

$$\{\mathsf{arc}() * \mathsf{arc}() * \mathsf{arc}()\} \quad \texttt{drop()} \quad \{\mathsf{arc}() * \mathsf{arc}()\}$$

As can be seen, the frame $\mathsf{arc}()*\mathsf{arc}()$ is unaffected by calling `drop()`. Likewise, if some other thread held $\mathsf{arc}() * \mathsf{arc}()$ it would be unaffected by the call.

Framing means that every view must continue to hold irrespective of the behaviour of other threads. However, $\mathsf{arc}()$ atoms are not independent in their underlying representation, nor between each other. In their representation, all the $\mathsf{arc}()$ views refer to the same shared variables. Also, the reference count must not be smaller than the *total* number of $\mathsf{arc}()$ atoms across all threads – otherwise a thread could access the object after it has been disposed. Reasoning about this combination of thread-local views and inter-thread interaction is the core problem that our approach solves.

```
// View atom declarations
view iter arc();
view countCopy(int c);

// Create a new reference to the ARC
method clone() {
  {| arc() |}
    <| count++; |>
  {| arc() * arc() |}
}

// Remove an ARC reference and dispose if possible
method drop() {
  {| arc() |}
    <| c = count--; |>
  {| countCopy(c) |}
    if (c == 1) {
      {| countCopy(1) |}
        <| free = true; |>
      {| emp |}
    }
  {| emp |}
}

// Access the ARC contents - Model with a test of free
method access() {
  {| arc() |}
    <| f = free; |>
  {| if (f) { false } else { arc() } |}
    if (f) {
      {| false |}
        <| error; |>  // Models a bad dereference.
      {| false |}
    }
  {| arc() |}
}

// Constraints on countCopy()
constraint countCopy(c)                 ->  c == 1 => (!free && count == 0);
constraint countCopy(m) * countCopy(n)  ->  (m != 1) || (n != 1);

// Iterated constraint on arc()
constraint iter[n] arc()  ->  n > 0 => (!free && n <= count);
```

Fig. 1. Shared-variable version of ARC, and proof.

2.2 Proof

Figure 1 shows an ARC implementation, and a proof that it satisfies our specification. (Here, and elsewhere, we elide some details such as variable declarations.)

In this implementation we model a single ARC instance by shared variables. The integer variable `count` holds the reference count, while disposal is modelled by the boolean variable `free`. This simplification to variables means we can discharge the proof using an SMT solver. Below, we verify a heap-allocated ARC using the GRASShopper separation-logic solver.

Our programming language is a standard while-language, with atomic commands written with angle-brackets, `<| |>`. The proof itself consists of Hoare-style assertions, written in views, that are interleaved into the program. These assertions are written using assertion brackets `{| |}`. As well as plain views, views can hold conditional on local variables: for example, in Figure 1 we write `{| if (f) { false } else { arc() } |}`. The complete syntax for Starling's input language is given in Appendix A.

In addition to the $\mathsf{arc}()$ atom discussed above, the proof uses the additional atom $\mathsf{countCopy}(c)$, which represents the fact that c was previously observed as the value of `count`. (It does not mean that `count` is currently c, as `count` can change through the action of other threads).

The meaning of the views in the underlying program state is given by *constraints*. There are unboundedly many possible composite views, but we need only give meanings for a minimal set of *defining* views – meanings for others are derived from these. Section 3 explains how this derivation works.

In Figure 1, the meaning of a single $\mathsf{countCopy}(c)$ atom is given by the following constraint:

```
constraint countCopy(c)  ->  c == 1 => (!free && count == 0);
```

Once a thread observes `count` as 1 in a fetch-and-decrement, the ARC cannot be disposed by any other thread, and the value of `count` will always be zero. This depends on `count` accurately recording the number of references to the ARC: once `count` is 1, the only thread with access is the current one.

Constraints can also specify interactions between views. Interactions can be be between views on the same or multiple different threads – we make no distinction between the two. In Figure 1, two $\mathsf{countCopy}(c)$ atoms have the following meaning:

```
constraint countCopy(m) * countCopy(n)  ->  (m != 1) || (n != 1);
```

If two threads take copies of `count`, only one of them can equal 1: again, this depends on the counter accurately recording the number of references.

The final important properties represented in the proof are, first, that the ARC is not disposed until all references are removed; and, second, that `count` accurately records the number of references. Each $\mathsf{arc}()$ atom represents a reference, so we need the following:

$$\overbrace{\mathsf{arc}() * \mathsf{arc}() * \cdots * \mathsf{arc}()}^{n > 0 \ \textit{atoms}} \implies \neg \textit{disposed} \wedge n \leq \textit{ref-count}.$$

In the proof, this is expressed directly by the following constraint on views:

```
constraint iter[n] arc()  ->  n > 0 => (!free && n <= count);
```

The `iter[n]` keyword indicates that we have `n` instances of the arc() atom on the same thread or across different threads.

2.3 Heap-allocated ARC

The implementation in Figure 1 modelled a single ARC by shared variables – as a result, we can discharge this proof using an SMT back-end. In Figure 2, we give a more realistic implementation where ARCs are heap-allocated structs. To discharge this proof, we use GRASShopper, a solver for separation logic [20].

The most important implementation change is a new method `init` which allocates a new ARC. This method has the following specification:

$$\{\mathsf{emp}\} \;\; \texttt{init()} \;\; \{\mathsf{arc}(\mathit{ret})\}$$

A further difference is that heap commands are written in GRASShopper's input language. We embed these using the special brackets `%{ }`, and allow variables to be referenced using the inner brackets `[| |]`. For example, in `clone`, we write the following for an atomic increment:

```
    <| %{[|x|].count := [|x|].count + 1}; |>
```

By combining heap commands we can build complex atomic operations – for example an atomic fetch-and-decrement operation, as used in `drop`:

```
   <| c = %{ [|x|].count };  %{ [|x|].count := [|x|].count - 1 }; |>
```

Despite the fact that this implementation targets a much richer domain than shared variables, we can apply the same proof strategy as Figure 1. The same views are needed, though they are now parameterised by the address of the ARC. Likewise, the same constraints are needed, modified to use GRASShopper's constraint language. As with commands, we embed GRASShopper assertions using the special brackets `%{ }`. For example, this is the constraint on a single $\mathsf{countCopy}(x, c)$ atom:

```
   constraint countCopy(x, c) ->
       c == 1 => %{ [|x|] in ArcFoot && [|x|].count == 0};
```

Here, `[|x|] in ArcFoot` requires that `x` is in the set of allocated ARCs – this corresponds to the requirement that `free` is false in Figure 1. Likewise, `[|x|].count == 0` corresponds to the constraint on the value of `count`.

With both the variable-based and heap-based versions of the ARC, our approach gives a simple proof that captures the algorithm's linear nature. Our approach lets us convert these lightweight proofs into verification conditions that can be discharged by either SMT or GRASShopper as appropriate. We next explain how this translation works.

```
struct ArcNode {
  var count: Int;
  var val: Int;
}

view iter arc(ArcNode x);
view countCopy(ArcNode x, Int c);

method init() {
  {| emp |}
    <| ret = %{new ArcNode};
       %{ [|ret|].count := 1 }; |>
  {| arc(ret) |}
}

method clone(ArcNode x) {
  {| arc(x) |}
    <| %{ [|x|].count := [|x|].count + 1 }; |> // Atomic increment
  {| arc(x) * arc(x) |}
}

method drop(ArcNode x) {
  {| arc(x) |}
    <| c = %{ [|x|].count };  // Atomic fetch-and-decrement
       %{ [|x|].count := [|x|].count - 1 }; |>
  {| countCopy(x, c) |}
    if (c == 1) {
      {| countCopy(x, 1) |}
         <| %{ free([|x|]) }; |>
      {| emp |}
    }
  {| emp |}
}

method access(ArcNode x) {
  {| arc(x) |}
    <| pval = %{ [|x|].val }; |>
  {| arc(x) |}
}

constraint countCopy(x, c) ->
    c == 1 => %{ [|x|] in ArcFoot && [|x|].count == 0 };
constraint countCopy(x, m) * countCopy(y, n) ->
    x == y => ((m != 1) || (n != 1));

constraint iter[n] arc(x) ->
    n > 0 => %{ [|x|] in ArcFoot && [|n|] <= [|x|].count };
```

Fig. 2. Heap-allocated version of ARC, and proof.

3 Theory

Starling's theory works by recasting the pre-existing Views framework [6] into a form suitable for automation. As the Views framework has been proved sound in Coq, this gives us a simple way of justifying the soundness of our translation into a set of verification conditions.

3.1 Owicki-Gries

For comparison, we first consider the Owicki-Gries method [19], one of the simplest approaches to Hoare-style verification of a concurrent program. Owicki-Gries presents us with a single core rule for validating a proof outline.[3] Let Axioms be the set of atomic Hoare triples of the proof; $\mathsf{Formula}$ the set of all formulas used in the outline; and $\models_{\mathrm{Hoare}}$ the entailment rule for Hoare logic. Then, the Owicki-Gries proof rule is written as:

$$\forall \{P\}\, c\, \{Q\} \in \mathsf{Axioms}.\ \forall F \in \mathsf{Formula}.\ \models_{\mathrm{Hoare}} \{P \land F\}\, c\, \{Q \land F\}$$

This rule expresses two key correctness properties for a concurrent system. First, each command behaves correctly in a sequential setting – the post-state Q is established from the pre-state P. Second, no command interferes with any properties needed by other threads – the frame F is preserved by c.

To achieve completeness, Owicki-Gries needs *auxiliary variables*: additional variables that capture key aspects of the local state of each thread. To encode Starling into Owicki-Gries, we would need to use auxiliary variables to encode the more rich interactions our constraint system permits. However, these variables can hide the details of the verification and make proof discovery harder. We need a different approach.

3.2 Views

We eliminate the need for auxiliary variables, while keeping much of the shape and simplicity of Owicki-Gries, by building on the Views framework [6]. Views was originally an off-the-shelf metatheory for proving the soundness of concurrent reasoning systems; we recast it as an Owicki-Gries-style proof rule. In this paper, we introduce just enough of the Views framework to support Starling's theory – this fits with the framework's purpose as reusable metatheory.

The Views framework is designed to allow a broad range of reasoning systems to be encoded into a small set of parameters. If these parameters satisfy a few key properties, the encoded reasoning system is sound.

The parameters that must be instantiated include the sets Views, from which all assertions in the logic are derived; Cmds, containing atomic commands; and Axioms, containing the atomic Hoare triples over views and commands. The reasoning system must also define a view composition operator $*$ and unit view

[3] We simplify Owicki-Gries to a setting where all threads execute the same code.

emp, which together must form a monoid with Views; a reification function $\lfloor \_ \rfloor$ mapping Views to their representation in the underlying state; and a semantic function $[\![\_]\!]$ mapping atomic commands to state transformers.

Taken together, these parameters must satisfy the key property of *axiom soundness*:

$$\forall \{P\}\, c\, \{Q\} \in \mathsf{Axioms}.\ \forall V \in \mathsf{Views}.\ [\![c]\!] \lfloor P * V \rfloor \subseteq \lfloor Q * V \rfloor \tag{1}$$

This rule requires that every atomic Hoare triple generated by the reasoning system upholds sequential correctness, and inter-thread non-interference, just as we saw in Owicki-Gries. As the Views approach makes no distinction between contexts that on the same thread or other threads, it captures both Concurrent Separation Logic's Frame and Parallel rules:

$$\frac{\{P\}\ C\ \{Q\}}{\{P * F\}\ C\ \{Q * F\}}\ \textsc{Frame} \qquad \frac{\{P_1\}\ C_1\ \{Q_1\} \qquad \{P_2\}\ C_2\ \{Q_2\}}{\{P_1 * P_2\}\ C_1 \parallel C_2\ \{Q_1 * Q_2\}}\ \textsc{Parallel}$$

In Starling, we recast Rule (1) to generate verification conditions from proofs. In comparison to Owicki-Gries, the Views proof rule allows us to avoid auxiliary variables in most cases. In Owicki-Gries, assertions and contexts are joined by conjunction, but in the Views rule they are joined by view composition, $*$, and their reification is defined separately. This means that we can define interactions between views that go beyond their individual reifications – for example to enforce mutual exclusion between views. This gives our proof system its power.

3.3 Instantiating the Views rule

We first instantiate the Views framework parameters in a way that is suitable for Starling's reasoning. For Starling, *view atoms* consist of a name and a sequence of value arguments, and views are multisets of view atoms. More formally, we define Views as:

$$\begin{aligned} \mathsf{ViewAtoms} &\triangleq \mathsf{String} \times \mathit{seq}\ \mathsf{Value} \\ \mathsf{Views} &\triangleq \mathit{multiset}\ \mathsf{ViewAtoms} \end{aligned}$$

(Below we sometimes call these *plain views* to distinguish them from constructs such as view patterns.)

Starling Views form a monoid with the multiset union $\cup_{\mathsf{m}}$ as the view composition $*$, and the empty multiset $\emptyset$ as the unit view emp.

We first change Rule (1) by making the state accessed by a command explicit. We model the state as a pair (l, s) of thread-local and shared components. The command semantics $[\![c]\!]$ is then a relation over these states. We write $\lfloor P \rfloor(s)$ to say that state s is in the representation of P, and (for now) ignore the local state. The resulting rule is:

$$\begin{aligned} &\forall \{P\}\, c\, \{Q\} \in \mathsf{Axioms}. \\ &\forall ((l, s), (l', s')) \in [\![c]\!].\ \forall V \in \mathsf{Views}.\ \lfloor P * V \rfloor(s) \Rightarrow \lfloor Q * V \rfloor(s') \end{aligned} \tag{2}$$

For example, in Figure 1, of the atomic triples in Axioms is:

$$\{\mathsf{arc}()\} \quad \texttt{<| count++; |>} \quad \{\mathsf{arc}() * \mathsf{arc}()\}$$

Rule (2) yields a proof term with the following shape for each combination of this triple and frame V:

$$\forall((l, s), (l', s')) \in [\![\texttt{count++}]\!].\, \forall V \in \mathsf{Views}.\ \lfloor \mathsf{arc}() * V \rfloor(s) \Rightarrow \lfloor \mathsf{arc}() * \mathsf{arc}() * V \rfloor(s')$$

3.4 Integrating Local State

Rule (2) is not sufficient for the ARC proof in Figure 1. First, the view atom `countCopy(c)` refers to a local variable `c`, not a value. Second, the view `arc()` is defined using the iterator variable `n`. Finally, we need the ability to choose whether atoms appear in a view based on local conditions to encode assertions such as `{| if (f) { false } else { arc() } |}`.

To incorporate these local-state properties into the rule, we introduce syntactic *view expressions*, with the following syntax:

$$P \ ::= \ \mathsf{emp} \mid (B \to a[n](\overline{e})) * P$$

View expressions are used to encode Starling's assertion syntax. Each view expression P is a $*$-composition of *atom expressions*. These have a name a, a list $\overline{e}$ of integer or boolean argument expressions, an integer iterator expression n, and a boolean guard expression B. The argument, iterator, and guard expressions are all interpreted in the local state.

To map a view expression to a view, we must interpret its local-state expressions. Given a local state l and expression X, we write $l(X)$ for the value of X in l. Using this, we define a function $[\![-]\!]_l$ which maps from view expressions into views:[4]

$$[\![\mathsf{emp}]\!]_l \triangleq \emptyset$$
$$[\![B \to a[n](\overline{e}) * P]\!]_l \triangleq [\![P]\!]_l \cup_{\mathsf{m}} \begin{cases} \{a(l(\overline{e})) \mapsto l(n)\} & \text{if } l(B) \\ \emptyset & \text{otherwise} \end{cases}$$

Here, the empty view expression maps to an empty multiset, i.e. the unit plain view. Other view expressions map to the appropriate view atoms, dictated by the values of the local-state expressions. The argument expressions dictate the values of the view atom's arguments. The guard expression controls whether any view atoms are created, and the iterator expression dictates the number of instances of the view atom.

To integrate this into our core proof rule, we amend Axioms so that pre- and post-conditions are view expressions, not plain views. This means that they must be interpreted by the semantic function $[\![-]\!]_l$. Our modified rule is as follows:

$$\begin{array}{l} \forall \{P\}\, c\, \{Q\} \in \mathsf{Axioms}. \\ \forall((l, s), (l', s')) \in [\![c]\!].\, \forall V \in \mathsf{Views}.\ \lfloor [\![P]\!]_l * V \rfloor(s) \Rightarrow \lfloor [\![Q]\!]_{l'} * V \rfloor(s') \end{array} \qquad (3)$$

[4] Note that we have a composition operator $*$ and unit emp in both view expressions and plain views. This definition links the two levels: to avoid confusion here, for plain views we use their semantic definitions $\cup_{\mathsf{m}}$ and $\emptyset$.

3.5 Context Reduction

The quantification $\forall V$ over context views means that Rule (3) cannot be used directly for automated verification. As two smaller views can be composed into a larger one, there are arbitrarily many possible values of V, and by default we must consider them all.

Other logics allow a degree of context reduction here. For example, in Owicki-Gries, if two threads separately assert F_1 and F_2, and each is preserved, we need not consider the context $F_1 \wedge F_2$. This means we can validate our proof outline for an unbounded number of threads by considering a finite set of entailments.

We cannot use this simple context reduction, because in Views any context may contribute information not represented in its sub-views. This generality is desirable – it is what gives our proof system its power. We can preserve it while gaining context reduction by defining reification in a particular way.

Defining function. The first restriction on reification is we only consider functions where the reification of a composite view implies the conjunction of its sub-view reifications. In other words, view composition cannot lose information, which lets us avoid considering sub-views of composite views. More formally, we require that for all views, $\lfloor P * Q \rfloor \Rightarrow \lfloor P \rfloor \wedge \lfloor Q \rfloor$.

The second restriction is that we bound the set of views that can contribute information to the reification. Intuitively, this means that we only need to consider these *defining* sub-views in our proof rule. To enforce this, we require that the reification function is derived from a syntactic *defining function.*

In a Starling proof, the defining function is given precisely by the constraints. For example, in Figure 1 we have:

```
constraint  countCopy(m) * countCopy(n)  ->  (m != 1) || (n != 1);
```

On the left we have a *view pattern* `countCopy(m) * countCopy(n)`, while on the right we have a formula giving the meaning for this pattern.

View patterns allow a definition to match many different views with similar shapes. A view pattern r has the syntax:

$$r \ ::= \ \mathsf{emp} \mid a[n](\overline{x}) * r$$

A pattern is either emp, or a $*$-composition of *pattern atoms.* Each atom has a name a, variable arguments $\overline{x}$ which bind to the arguments of a view atom, and an iterator variable n which records the number of view atoms matched.

A *definition* is then a tuple $(\overline{y}, r, p)$ where, r is a view pattern, p is a formula of the underlying theory, and $\overline{y}$ is a set of free variables used in the definition. In the example constraint above, $\overline{y}$ is the set of variables $\{m, n\}$, the pattern r is $\mathsf{countCopy}[1](m) * \mathsf{countCopy}[1](n)$, and the formula p is $(m \neq 1) \vee (n \neq 1)$.

A defining function D is then a finite set of definitions (derived from the constraints in the proof). Using such a D, we can then induce a reification function where only definitions contribute information. The reification of a view-expression

$$
\begin{array}{ll}
\multicolumn{2}{l}{\forall V \in \mathsf{Views}.\ \lfloor [\![P]\!]_l \cup_{\mathsf{m}} V \rfloor(s) \Rightarrow \lfloor [\![Q]\!]_{l'} \cup_{\mathsf{m}} V \rfloor(s')} \\
 & \textit{[Definition of reification]} \\
\Longleftarrow & \forall V \in \mathsf{Views}.\ \lfloor [\![P]\!]_l \cup_{\mathsf{m}} V \rfloor(s) \Rightarrow \forall(\overline{y}, r, p) \in D.\ \hat{\forall}\overline{y}.\ (r \subseteq_{\mathsf{m}} ([\![Q]\!]_{l'} \cup_{\mathsf{m}} V) \Rightarrow p(s')) \\
 & \textit{[Lift out quantifiers]} \\
\Longleftarrow & \forall V \in \mathsf{Views}.\ \forall(\overline{y}, r, p) \in D.\ \hat{\forall}\overline{y}.\ \lfloor [\![P]\!]_l \cup_{\mathsf{m}} V \rfloor(s) \Rightarrow (r \subseteq_{\mathsf{m}} ([\![Q]\!]_{l'} \cup_{\mathsf{m}} V) \Rightarrow p(s')) \\
 & \textit{[View adjoint lemma]} \\
\Longleftarrow & \forall V \in \mathsf{Views}.\ \forall(\overline{y}, r, p) \in D.\ \hat{\forall}\overline{y}.\ \lfloor [\![P]\!]_l \cup_{\mathsf{m}} V \rfloor(s) \Rightarrow ((r \setminus_{\mathsf{m}} [\![Q]\!]_{l'}) \subseteq_{\mathsf{m}} V \Rightarrow p(s')) \\
 & \textit{[Move subset condition]} \\
\Longleftarrow & \forall V \in \mathsf{Views}.\ \forall(\overline{y}, r, p) \in D.\ \hat{\forall}\overline{y}.\ (r \setminus_{\mathsf{m}} [\![Q]\!]_{l'}) \subseteq_{\mathsf{m}} V \Rightarrow (\lfloor [\![P]\!]_l \cup_{\mathsf{m}} V \rfloor(s) \Rightarrow p(s')) \\
 & \textit{[Multiset subset is preserved under union]} \\
\Longleftarrow & \forall V \in \mathsf{Views}.\ \forall(\overline{y}, r, p) \in D.\ \hat{\forall}\overline{y}. \\
 & (([\![P]\!]_l \cup_{\mathsf{m}} (r \setminus_{\mathsf{m}} [\![Q]\!]_{l'})) \subseteq_{\mathsf{m}} ([\![P]\!]_l \cup_{\mathsf{m}} V)) \Rightarrow (\lfloor [\![P]\!]_l \cup_{\mathsf{m}} V \rfloor(s) \Rightarrow p(s')) \\
 & \textit{[Reification monotone]} \\
\Longleftarrow & \forall V \in \mathsf{Views}.\ \forall(\overline{y}, r, p) \in D.\ \hat{\forall}\overline{y}. \\
 & (([\![P]\!]_l \cup_{\mathsf{m}} (r \setminus_{\mathsf{m}} [\![Q]\!]_{l'})) \subseteq_{\mathsf{m}} ([\![P]\!]_l \cup_{\mathsf{m}} V)) \Rightarrow (\lfloor [\![P]\!]_l \cup_{\mathsf{m}} (r \setminus_{\mathsf{m}} [\![Q]\!]_{l'}) \rfloor(s) \Rightarrow p(s')) \\
 & \textit{[Remove V]} \\
\Longleftarrow & \forall(\overline{y}, r, p) \in D.\ \hat{\forall}\overline{y}.\ \lfloor [\![P]\!]_l \cup_{\mathsf{m}} (r \setminus_{\mathsf{m}} [\![Q]\!]_{l'}) \rfloor(s) \Rightarrow p(s')
\end{array}
$$

Fig. 3. Derivation of Rule (4), with outer quantifiers elided.

V, for a shared state s, is the conjunction of all the definitions that match some sub-view of V.

$$\lfloor V \rfloor(s) \ \triangleq \bigwedge_{(\overline{y}, r, p) \in D} \hat{\forall}\overline{y}.\ r \subseteq_{\mathsf{m}} V \implies p(s)$$

We write $r \subseteq_{\mathsf{m}} V$ (using multiset subset) to indicate that r is a sub-view of V, meaning there is a pattern match.

A pattern may be matched under any instantiations of its free variables $\overline{y}$. We express this using the special quantification $\hat{\forall}\overline{y}$. Given a formula X that includes r and p, $\hat{\forall}\overline{y}. X$ is shorthand for quantifying over all possible assignments to $\overline{y}$, and substituting in r and p. This has the effect of converting r into a plain view. Many theories, such as SMT, can natively handle the $\hat{\forall}\overline{y}$ construction without further expansion.

Rule context reduction. Using this definition, we can modify Rule (3) to reduce the contexts we consider to just those in the defining function.

First we introduce two lemmas. The first lemma (*reification monotone*) states that the reifications of larger views are more restrictive than those of smaller views. This justifies us considering only defining views in the premise of the proof rule, because any larger context will be more restrictive.

Lemma 1 (Reification monotone). $V_1 \subseteq_{\mathsf{m}} V_2 \implies (\forall s.\, \lfloor V_2 \rfloor(s) \Rightarrow \lfloor V_1 \rfloor(s))$

The second lemma (*view adjoint*) defines the relationship between multiset union $\cup_{\mathsf{m}}$, multiset subset $\subseteq_{\mathsf{m}}$, and multiset minus $\setminus_{\mathsf{m}}$. We use $\setminus_{\mathsf{m}}$ in our new rule to construct a ‘weakest context’, analogous to a weakest precondition.

Lemma 2 (View adjoint). $(V_1 \setminus_{\mathsf{m}} V_2) \subseteq_{\mathsf{m}} V_3 \implies V_1 \subseteq_{\mathsf{m}} (V_2 \cup_{\mathsf{m}} V_3)$

Now we take Rule (3) and (eliding the two outer quantifiers) rewrite it as shown in Figure 3. This at last gives us Starling's core proof rule:

$$\boxed{\begin{aligned} &\forall \{P\}\, c\, \{Q\} \in \mathsf{Axioms}.\, \forall((l, s), (l', s')) \in \llbracket c \rrbracket. \\ &\qquad \forall(\overline{y}, r, p) \in D.\ \hat{\forall}\overline{y}.\ \lfloor \llbracket P \rrbracket_l \cup_{\mathsf{m}} (r \setminus_{\mathsf{m}} \llbracket Q \rrbracket_{l'}) \rfloor(s) \Rightarrow p(s') \end{aligned} \qquad (4)}$$

This is the rule that we use to generate verification conditions from Starling input proofs such as Figure 1. The atomic steps of the program form the set Axioms; the built-in semantics of commands specify $\llbracket c \rrbracket$; and the constraints specify the defining function D and the reification $\lfloor - \rfloor$. The significant advantage of this rule is that, rather than quantify over an infinite set of context views, it quantifies only over finite sets, and therefore generates a finite set of proof terms.

Consider the $\mathsf{arc}()$ proof term we examined in §3.3. If rather than using Rule (2), we apply our new rule, we get the following outcome:

$$\begin{aligned} &\forall((l, s), (l', s')) \in \llbracket \texttt{count++} \rrbracket. \\ &\forall(\overline{y}, r, p) \in D.\ \hat{\forall}\overline{y}.\ \lfloor \llbracket \mathsf{arc}() \rrbracket_l \cup_{\mathsf{m}} (r \setminus_{\mathsf{m}} \llbracket \mathsf{arc}() * \mathsf{arc}() \rrbracket_{l'}) \rfloor(s) \Rightarrow p(s') \end{aligned}$$

3.6 Finite Pattern Matching

Rule (4) gives us a finite set of proof terms. However, we must also translate each term into finitely many verification conditions. The key issue is ensuring that the number of pattern matches in each reification is finite.

Most cases of pattern matching are trivially finite, but iterated views require careful treatment. An iterated view expression $B \to a[n](\overline{y})$ can produce n many subviews. As a result, if a view pattern r and view V are both iterated, there may be unboundedly many valid distinct matches (for $i = 1, 2, \ldots$).

To solve this, a definition $(\overline{y}, r, p)$ where p is dependent on an iterator n must satisfy the following *downclosure* properties:

$$\lfloor \mathsf{emp} \rfloor(s) \implies p[0/n](s) \qquad \text{(base downclosure)}$$

$$\forall x \in \mathbb{Z}^+.\, p[x/n](s) \implies p[x-1/n](s) \qquad \text{(inductive downclosure)}$$

These properties let us just consider the largest iterator value when constructing pattern matches. Our tool checks downclosure as an extra proof obligation.

A further subtlety is that iterated definitions can match against combinations of atoms when they can be made equal through parameter equality. For example, $\mathsf{A}[n](x)$ matches $(B_1 \to \mathsf{A}[i](y)) * (B_2 \to \mathsf{A}[j](z))$ to form $((B_1 \wedge B_2 \wedge y = z) \to \mathsf{A}[i + j](y))$. We can solve this by expanding out the equalities as if they are separate view atoms before matching – this does not change the view's meaning.

4 SMT Back-End

We now have a proof outline for the ARC (§2) and a proof rule to convert it into verification conditions (§3). We now show how to verify these conditions using an SMT solver – in our case, Z3 [5]. To do this, we must convert the defining function, multiset minus, and command semantics into forms supported by Z3.

Definition quantification. We begin by eliminating the defining function. Consider the following term we generated from our running example at the end of §3.5:

$$\begin{array}{l}\forall((l,s),(l',s')) \in [\![\texttt{count++}]\!].\\ \forall(\overline{y},r,p) \in D.\ \hat{\forall}\overline{y}.\ \lfloor [\![\mathsf{arc}()]\!]_l \cup_{\mathsf{m}} (r \setminus_{\mathsf{m}} [\![\mathsf{arc}() * \mathsf{arc}()]\!]_{l'}) \rfloor(s) \Rightarrow p(s')\end{array}$$

As the defining function D is bounded, we can expand the quantification into a finite set of terms. For example, for the pattern $\mathsf{arc}[n]()$, we get the following term:

$$\begin{array}{l}\forall((l,s),(l',s')) \in [\![\texttt{count++}]\!].\ \forall n.\\ \lfloor [\![\mathsf{arc}()]\!]_l \cup_{\mathsf{m}} ([\![\mathsf{arc}[n]()]\!]_{l'} \setminus_{\mathsf{m}} [\![\mathsf{arc}() * \mathsf{arc}()]\!]_{l'}) \rfloor(s)\\ \qquad\qquad \Rightarrow (n > 0 \Rightarrow \neg \mathit{free} \wedge n \leq \mathit{count})(s')\end{array}$$

We get this by substituting the view pattern into the left of the implication in place of r, and the corresponding formula into the right in place of p. We also eliminate the $\hat{\forall}\overline{y}$ by quantifying over the single variable n that is bound in $\overline{y}$. For simplicity later, we treat r as a view expression over l'.

Multiset minus. We next eliminate multiset minus. We can easily reduce our proof term so that all instances of $\setminus_{\mathsf{m}}$ have the following shape:

$$[\![B_1 \to a[n_1](\overline{y_1})) * P]\!]_{l'} \setminus_{\mathsf{m}} [\![B_2 \to a[n_2](\overline{y_2})]\!]_{l'}$$

We eliminate this shape by case-splitting on the relationship between B_1 and B_2, n_1 and n_2, and $\overline{y_1}$ and $\overline{y_2}$. The main subtlety is that some, but not all instances in the iterator $a[n_1]$ may be subtracted, i.e. we may be left with the iterator $a[n_1 - n_2]$. If we are left with anything on the right of the $\setminus_{\mathsf{m}}$, we then apply the simplification step to the remainder formula P.

In our example, subtracting $[\![\mathsf{arc}() * \mathsf{arc}()]\!]_{l'}$ from $[\![\mathsf{arc}[n]()]\!]_{l'}$ leaves $n-2$ copies of $\mathsf{arc}()$. If $n \leq 2$, nothing is left: we express this as a guarded view. The multiset minus rewrite yields the following term:

$$\begin{array}{l}\forall((l,s),(l',s')) \in [\![\texttt{count++}]\!].\ \forall n.\\ \lfloor [\![\mathsf{arc}()]\!]_l \cup_{\mathsf{m}} [\![(n > 2 \to \mathsf{arc}[n-2]())]\!]_{l'} \rfloor \Rightarrow (n > 0 \Rightarrow \neg \mathit{free} \wedge n \leq \mathit{count})(s')\end{array}$$

Commands as predicates. To eliminate the command, we recast it as a boolean predicate over pre- and post-states. To do so, we instantiate two copies of each variable: one set for (l,s), and another (primed) set for (l',s'). We conjoin this command predicate into the proof term, replacing the outer quantification with implicit ones over the variable sets. Expanding out the reification and the local-state interpretations, and ensuring we handle the subtleties in §3.6, we get:

$$\begin{array}{l}\left(\begin{array}{l}\mathit{count}' = \mathit{count} + 1 \wedge \mathit{free}' = \mathit{free} \wedge c' = c\\ \wedge\ (1 > 0 \Rightarrow \neg \mathit{free} \wedge 1 \leq \mathit{count})\\ \wedge\ (n > 2 \Rightarrow (n - 1 > 0 \Rightarrow \neg \mathit{free} \wedge n - 1 \leq \mathit{count}))\\ \wedge\ (n > 2 \Rightarrow (n - 2 > 0 \Rightarrow \neg \mathit{free} \wedge n - 2 \leq \mathit{count}))\end{array}\right)\\ \qquad\qquad\qquad\qquad \Longrightarrow\ (n > 0 \Rightarrow \neg \mathit{free}' \wedge n \leq \mathit{count}')\end{array}$$

SMT term. Finally, we negate the outer implication for each condition, so Z3 tries to find counter-example instantiations for the condition's variables. We can also simplify the term. For example, we remove the $n - 2$ case, as it is implied by the $n - 1$ case. The resulting term, in the SMT-LIB language accepted by Z3, is:

```
(and (= count' (+ count 1)) (= free' free) (= c' c) (not free) (<= 1 count)
     (=> (> n 2) (<= (- n 1) count))
     (not (=> (> n 0) (and (not free') (<= n count')))))
```

5 GRASShopper Back-End

For heap-based programs like the ARC in Figure 2, we target the GRASShopper solver [20] rather than Z3. GRASShopper is a separation-logic solver, but its underlying model is based on sets of heap locations and reachability properties over sets. For example, the following GRASShopper predicate asserts that the set of locations `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)}
    }
```

Here, `acc(Footprint)` is a spatial assertion claiming ownership of the locations in `Footprint`. The `Btwn(next,x,z,y)` predicate asserts that `z` is reachable between `x` and `y` by following the `next` field – in other words, `z` is in the list starting at `x` and ending at `y`. The set comprehension `{z: Node:: Btwn(next,x,z,y)}` therefore contains the set of locations in the list.

Most of the pipeline for producing GRASShopper proofs is similar to the SMT case. However, the presence of a heap model causes some differences. Suppose we try to model the allocated ARC equivalent of our previous working example,

$$\{\mathsf{arc}(x)\} \quad \texttt{<| count++; |>} \quad \{\mathsf{arc}(x) * \mathsf{arc}(x)\}$$

Given a context of $\mathsf{arc}(x) * \mathsf{arc}(x)$ (that is, the same x as in the local state of the thread), our translation would give the following in pseudo-SMT format:

```
(and %{ [|x|].count := [|x|].count + 1; }
     %{ [|x|] in ArcFoot && 1 <= [|x|].count }
     (=> (> n 2) (and %{ [|x|] in ArcFoot } (<= (- n 1) %{ [|x|].count })))
     (not (=> (> n 0) (and %{ [|x|] in ArcFoot } (<= n %{ [|x|].count })))))
```

As we cannot discharge this term using SMT, we convert it into a GRASShopper procedure. Input and output variables are represented by arguments to the 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 the `requires` and `ensures` clause existentially quantify over a *footprint set* representing the whole heap – in the ARC, this is the `ArcFoot` set. This allows predicates to require access to the footprint, represented by `acc(ArcFoot)`, and to conjoin constraints on this shared footprint arising from the views.

In general, it would not be sound to introduce an arbitrary existential to the consequent side of the term. The problem is that existential might be witnessed differently across different terms (see the derivation in §3). However, our encoding into GRASShopper is sound, because GRASShopper will always witness the footprint the same way, as the set of all available heap locations.

With this translation, the above pseudo-SMT query 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; }
```

In some cases we need to model the mutation of variables. To do this, we declare fresh GRASShopper variables in the procedure body, and connect them to the input and output variables by assertion.

5.1 Example: CLH Queue Lock

GRASShopper's support for dynamic data-structures allows us to target much more complex algorithms than the ARC. In this section we verify the queue-based CLH lock [16], which also demonstrates a subtle ownership-transfer pattern between threads. For space reasons, we give the main proof in Appendix B, and here only explain the key details.

The code and inline views are given in Figure 4. In the CLH lock, each participating thread owns a single node. To contend for the lock, a thread adds its own node to the queue, and waits on its predecessor. Releasing the lock means setting the node's `lock` flag to false. Once the predecessor is released, the thread can take hold of the lock.

This protocol is reflected in the views in Figure 4. A node starts life `dormant`, i.e. not on the queue. It is then made `active` when its `lock` flag is set, and then is `queued`. Once the algorithm establishes that the node is at the end of the queue, it becomes `locked`. Finally, once the lock is released the node leaves the queue, and it becomes `dormant` again.

The key property of the CLH lock (and any lock) is mutual exclusion: each node is held exclusively, and the lock as a whole can only be held by one thread. In our approach, we can specify this using constraints, for example:

```
constraint queued(a, ap) * queued(b, bp) -> a != b;
constraint locked(a, ap) * locked(b, bp) -> false;
```

```
1  method lock() {
2   {| dormant(mynode) |}
3    <| %{ [|mynode|].lock := true }; |>
4   {| active(mynode) |}
5    <| mypred = tail; tail = mynode;
6       %{[|tail|].pred := [|mypred|]}; |>
7   {| queued(mynode, mypred) |}
8    do {
9      {| queued(mynode, mypred) |}
10      <| test = %{ [|mypred|].lock }; |>
11     {| if (test) {queued(mynode, mypred)}
12        else {locked(mynode, mypred)} |}
13   } while (test);
14  {| locked(mynode, mypred) |}
15 }
```

```
1 method unlock() {
2  {| locked(mynode, mypred) |}
3   <| %{ [|mynode|].lock := false };
4      %{ [|mynode|].pred := null };
5      head = mynode; |>
6  {| dormant(mypred) |}
7   mynode = mypred;
8  {| dormant(mynode) |}
9 }
```

Fig. 4. CLH queue-based lock algorithm. Note that the `head` pointer and `pred` field are ghost code necessary to verify the algorithm.

The queue data-structure is similarly defined by constraints. For example the locked() atom is defined using GRASShopper assertions similar to the `list_segment` predicate above.

```
constraint locked(node, pred) -> %{
    [|node|] in Foot && [|pred|] in Foot
    && Btwn(pred, [|tail|], [|node|], [|head|])
    && [|node|].pred == [|pred|] && [|pred|] == [|head|] };
```

The most subtle reasoning step happens in lines 2–6 of `unlock` in Figure 4, when the thread releases the lock. As some other thread may be waiting on its current node, it cannot be reused immediately. Instead the thread takes ownership of its dormant *predecessor*. Thus threads always have a single exclusively-held node, but the exact node held varies over time.

This ownership transfer is reflected in the proof in Figure 4 and the mutual exclusion constraints above. The terms passed to GRASShopper precisely encode the required properties, even though GRASShopper itself cannot reason about ownership transfer. Other reasoning approaches would capture this through regions or shared protocols: we encode it through views.

6 Examples and Performance Results

We have tested Starling on a range of examples: the ARC algorithm discussed in §2; a standard compare-and-swap spinlock; a ticket-based FIFO lock, as used in Linux [2]; a reader-writer lock which combines the classic Courtois et al algorithm [3] with tickets; Peterson's algorithm; the CLH queue-lock discussed in §5 [16]; and a lock-coupling list algorithm previously verified by Vafeiadis [26]

Algorithm	No. lines Starling input	No. lines auxiliary input	No. lines proof input	No. lines gen GH	No. generated terms	No. SMT-elim terms	Time, total excl GH (s)	Time, on tool (s)	Time, SMT (s)	Time, GH (s)	Mem use, Starling (MiB)	Mem use, GH (MiB)
SMT/Z3:												
ARC (static)	52	-	19	-	40	40	1.62	1.55	0.08	-	118	-
Ticket lock (static)	47	-	16	-	18	18	1.49	1.44	0.05	-	94	-
Spinlock (static)	35	-	10	-	12	12	1.51	1.47	0.04	-	87	-
Reader/writer lock	109	-	45	-	160	160	1.85	1.67	0.19	-	192	-
Peterson's algo.	94	-	27	-	72	72	2.35	2.05	0.30	-	136	-
GRASShopper:												
ARC (alloc)	59	13	32	482	20	5	1.55	1.54	0.02	1.56	92	10.2
Ticket lock (alloc)	59	80	104	1054	66	30	1.48	1.46	0.02	3.64	87	10.8
Spin lock (alloc)	54	18	38	689	56	31	1.57	1.56	0.02	2.45	88	10.6
CLH queue-lock	124	10	58	1407	50	21	1.47	1.45	0.02	3.87	84	11.3
Lock-coupling list	79	118	154	5019	240	116	1.96	1.94	0.02	35.31	96	30.2

Fig. 5. Benchmarks for example algorithms.

(note we verify memory safety, not linearizability). For several of these we have verified both a static version encoded in shared variables (using SMT) and a version allocated on the heap (using GRASShopper).

These algorithm are small in size, but all are challenging to verify, and each demonstrates an aspect of Starling's reasoning. Verifying the ARC example would typically require a primitive notion of "permissions" in separation logic – Starling can directly handle it without resorting to new metatheory. The CLH lock has an implied protocol between threads that performs ownership transfer of the node from one thread to the next, again handled directly by the theory. The other synchronisation algorithms similarly involve subtle protocols between threads that, in other reasoning systems, would need auxiliary proof constructs. The lock-coupling example shows that we can reason about complex fine-grained data-structures where the protocol is entwined with the list nodes.

Figure 5 gives performance statistics for our examples. From left to right we give statistics for: the total lines of input code and proof (including auxiliary GRASShopper code); the approximate number of which are proof annotations; the lines of generated GRASShopper output; the total number of proof terms generated; the number of those successfully discharged using SMT/Z3 (the remainder are sent to GRASShopper); the total proof time (excluding GRASShopper); of that time, the total spent on the tool itself, and on SMT/Z3; the total memory in the .NET runtime working set at the end of the proof, in mebibytes; and the average maximum resident set size over 3 runs of GRASShopper on the output from Starling, in mebibytes (these loosely approximate the total memory used).

Times reported are the average of 3 runs. Benchmarks were run on a 2016 series MacBook Pro, with 8GB RAM and a 2.9GHz dual-core Intel Core i5.

7 Related Work

Our approach builds on Views [6], and thus is part of the family of logics descended from Concurrent Separation Logic [18]. These logics all use separating conjunction to reason about distinct threads, and many of these logics have introduced auxiliary constructs to assist with reasoning. For example, Svendsen et al's iCAP [24] combines reasoning about interference (derived from Rely-Guarantee [14]), abstraction through abstract predicates, a rich system of protocols based on capabilities, and higher-order propositions. Other significant logics include CaReSL [25], TaDA [22], FCSL [17], and others – each comes with a different collection of auxiliary constructs.

As discussed in §3, our approach also has similarities to Owicki-Gries reasoning [19]. In Owicki-Gries, many kinds of interaction between threads need to be encoded through auxiliary variables. Views allow us to capture these interactions directly in a more intuitive style.

Starling inherits much of the generality of the Views framework – see [6] for encodings of multiple previous logics. We can encode many of the auxiliary proof constructs used in other logics. For example, Boyland-style fractional permissions [1] can be encoded by a view with a permission-value argument, which can then be split and joined by entailment. iCAP-style protocols can be encoded by making each protocol state into a view, and using constraints to enforce mutual exclusion between these state-views.

A few CSL-style logics have automated tool support. FCSL [17] and Verifast [13] both support automated proof-checking, albeit with a considerable annotation burden as all steps must be given explicitly. SmallfootRG [26] supports proof-checking for the RGsep logic, but requires annotations of invariants and rely-conditions – in our system these are defined implicitly by the constraints.

Caper [7] is the tool most similar to ours. It supports reasoning about functional specifications that our tool cannot presently handle – for example that an element is correctly inserted into a bag. However, Caper's logic is built on auxiliary guard algebras, shared regions, and actions. It is therefore significantly more complex than our approach both in reasoning and in metatheory. Caper uses Z3, as do we, but its heap reasoning is custom-built, and we are uncertain whether it could verify an example of the complexity of the CLH lock or lock-coupling list. We handle these examples using the GRASShopper heap solver [20], and our approach is designed to be generic in the choice of back-end solver.

We have not undertaken a precise comparison, but we believe for our heap-based examples, all competing tools would require significantly more annotations.

For example, the CLH lock is our most challenging algorithm: in Verifast, its code and proof require 343 lines, while Starling requires 134 lines.[5]

Several other tools share similarities with our approach. VCC [4] is a verifier based on Z3 which has been used to verify large-scale concurrent C programs. In VCC, concepts such as permission and ownership are encoded through auxiliary state. Our approach encodes these properties through view interactions.

QED [9] is a refinement-based approach to verification: concurrent programs are related to their atomic specifications by a series of sound refinement steps. We are hopeful that our approach could be combined with this style of reasoning as well as CSL-style program logic.

Our SMT/Z3 back-end has similarities to Threader [11], and unlike our tool, Threader can infer invariants using a Horn-clause solver. However, it only targets shared-variable algorithms – we can handle heap-based algorithms. Invariant inference in our approach is a topic of future work.

There is a lot of work on model-checking concurrent systems – e.g [21,27]. In model-checking terms we require significant annotation, but our context reduction means that our proofs apply to an unbounded number of threads, context switches and unrolling of loops.

8 Conclusions

We have presented a new logic-based approach to verifying concurrent programs. Our approach is lightweight, automated, and based on a sound bedrock of existing theory. Because we build on the generic Views framework, we believe our approach could be reused by other concurrent logics as a way to target sequential solvers.

One next step will be invariant inference for Starling. Our proof terms are already in quasi-Horn clause form, and preliminary experiments suggest we can infer view definitions using an off-the-shelf solver such as HSF [10]. We also plan to extend Starling with modular reasoning, meaning that proofs of libraries and clients can be performed separately, as in iCAP [24]. Finally, we plan to extend Starling to prove algorithm linearizability rather than pre-post specifications, as in Vafeiadis [26] and Liang et al [15].

[5] `https://github.com/verifast/verifast/blob/master/examples/clhlock/clhlock.c`, accessed May 2017.

References

1. John Boyland. Checking interference with fractional permissions. In *Proceedings of the 10th International Conference on Static Analysis*, SAS'03, pages 55–72, Berlin, Heidelberg, 2003. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=1760267.1760273.
2. Jonathan Corbet. Ticket spinlocks. LWN.net, https://lwn.net/Articles/267968/, 2008.
3. P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with "readers" and "writers". *Commun. ACM*, 14(10):667–668, October 1971. URL: http://doi.acm.org/10.1145/362759.362813, doi:10.1145/362759.362813.
4. M. Dahlweid, M. Moskal, T. Santen, S. Tobies, and W. Schulte. Vcc: Contract-based modular verification of concurrent c. In *31st International Conference on Software Engineering*, pages 429–430, May 2009. doi:10.1109/ICSE-COMPANION.2009.5071046.
5. Leonardo De Moura and Nikolaj Bjørner. *Z3: An Efficient SMT Solver*, pages 337–340. TACAS'08/ETAPS'08. Springer-Verlag, Berlin, Heidelberg, 2008. URL: http://dl.acm.org/citation.cfm?id=1792734.1792766.
6. Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew Parkinson, and Hongseok Yang. Views: Compositional reasoning for concurrent programs. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 287–300, New York, NY, USA, 2013. ACM. URL: http://doi.acm.org/10.1145/2429069.2429104, doi:10.1145/2429069.2429104.
7. Thomas Dinsdale-Young, Pedro da Rocha Pinto, Kristoffer Just Andersen, and Lars Birkedal. Caper. In Hongseok Yang, editor, *Programming Languages and Systems: 26th European Symposium on Programming, ESOP 2017*, pages 420–447, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg. URL: http://dx.doi.org/10.1007/978-3-662-54434-1_16, doi:10.1007/978-3-662-54434-1_16.
8. Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In *Proceedings of the 24th European Conference on Object-oriented Programming*, ECOOP'10, pages 504–528, Berlin, Heidelberg, 2010. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=1883978.1884012.
9. Tayfun Elmas. Qed: A proof system based on reduction and abstraction for the static verification of concurrent software. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 507–508, New York, NY, USA, 2010. ACM. URL: http://doi.acm.org/10.1145/1810295.1810454, doi:10.1145/1810295.1810454.
10. Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. Synthesizing software verifiers from proof rules. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 405–416, New York, NY, USA, 2012. ACM. URL: http://doi.acm.org/10.1145/2254064.2254112, doi:10.1145/2254064.2254112.
11. Ashutosh Gupta, Corneliu Popeea, and Andrey Rybalchenko. *Threader: A Constraint-Based Verifier for Multi-threaded Programs*, pages 412–417. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. URL: http://dx.doi.org/10.1007/978-3-642-22110-1_32, doi:10.1007/978-3-642-22110-1_32.
12. Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming.* Elsevier, 2008.

13. Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. Verifast: A powerful, sound, predictable, fast verifier for c and java. In *Proceedings of the Third International Conference on NASA Formal Methods*, NFM'11, pages 41–55, Berlin, Heidelberg, 2011. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=1986308.1986314.
14. C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, October 1983. URL: http://doi.acm.org/10.1145/69575.69577, doi:10.1145/69575.69577.
15. Hongjin Liang and Xinyu Feng. Modular verification of linearizability with non-fixed linearization points. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 459–470, New York, NY, USA, 2013. ACM. URL: http://doi.acm.org/10.1145/2491956.2462189, doi:10.1145/2491956.2462189.
16. Peter S. Magnusson, Anders Landin, and Erik Hagersten. Queue locks on cache coherent multiprocessors. In *Proceedings of the 8th International Symposium on Parallel Processing*, pages 165–171, Washington, DC, USA, 1994. IEEE Computer Society. URL: http://dl.acm.org/citation.cfm?id=645604.662740.
17. Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. Communicating state transition systems for fine-grained concurrent resources. In *Proceedings of the 23rd European Symposium on Programming Languages and Systems - Volume 8410*, pages 290–310, New York, NY, USA, 2014. Springer-Verlag New York, Inc. URL: http://dx.doi.org/10.1007/978-3-642-54833-8_16, doi:10.1007/978-3-642-54833-8_16.
18. Peter W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, April 2007. URL: http://dx.doi.org/10.1016/j.tcs.2006.12.035, doi:10.1016/j.tcs.2006.12.035.
19. Susan Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM*, 19(5):279–285, May 1976. URL: http://doi.acm.org/10.1145/360051.360224, doi:10.1145/360051.360224.
20. Ruzica Piskac, Thomas Wies, and Damien Zufferey. GRASShopper: Complete heap verification with mixed specifications. In *Tools and Algorithms for the Construction and Analysis of Systems: 20th International Conference, TACAS 2014*, pages 124–139, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. URL: http://dx.doi.org/10.1007/978-3-642-54862-8_9, doi:10.1007/978-3-642-54862-8_9.
21. Shaz Qadeer and Jakob Rehof. Context-bounded model checking of concurrent software. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'05, pages 93–107, Berlin, Heidelberg, 2005. Springer-Verlag. URL: http://dx.doi.org/10.1007/978-3-540-31980-1_7, doi:10.1007/978-3-540-31980-1_7.
22. Pedro Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. Tada: A logic for time and data abstraction. In *Proceedings of the 28th European Conference on ECOOP 2014 — Object-Oriented Programming - Volume 8586*, pages 207–231, New York, NY, USA, 2014. Springer-Verlag New York, Inc. URL: http://dx.doi.org/10.1007/978-3-662-44202-9_9, doi:10.1007/978-3-662-44202-9_9.
23. Rust `std::sync` module. https://doc.rust-lang.org/std/sync/struct.Arc.html.
24. Kasper Svendsen and Lars Birkedal. Impredicative concurrent abstract predicates. In *Proceedings of the 23rd European Symposium on Programming Languages and Systems - Volume 8410*, pages 149–168, New York, NY, USA, 2014. Springer-Verlag New York, Inc. URL: http://dx.doi.org/10.1007/978-3-642-54833-8_9, doi:10.1007/978-3-642-54833-8_9.

25. Aaron Turon, Derek Dreyer, and Lars Birkedal. Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 377–390, New York, NY, USA, 2013. ACM. URL: http://doi.acm.org/10.1145/2500365.2500600, doi:10.1145/2500365.2500600.
26. Viktor Vafeiadis. *Modular Fine-Grained Concurrency Verification.* PhD thesis, University of Cambridge, July 2007.
27. Eran Yahav and Mooly Sagiv. Verifying safety properties of concurrent heap-manipulating programs. *ACM Trans. Program. Lang. Syst.*, 32(5):18:1–18:50, May 2008. URL: http://doi.acm.org/10.1145/1745312.1745315, doi:10.1145/1745312.1745315.

A Starling Assertion and Command Languages

We define the syntax of the Starling assertion and command languages using the grammars below. We assume the existence of grammars for `<lvalue>` (assignable locations), `<expr>` (expressions), and `<identifier>` (valid identifiers).

A.1 Assertions

```
<assertion>      ::= <assertion-item>
                  |  <assertion-item> "*" <assertion>

<assertion-item> ::= "emp"
                  |  "false"
                  |  <identifier> "(" <arglist> ")"
                  |  "local" "{" <expr> "}"
                  |  "if" "(" <expr> ")" "{" <assertion> "}" <assertion-else>
                  |  "(" <assertion> ")"
<assertion-else> ::= ""  |  "else" "{" <assertion> "}"

<arglist>        ::= ""  |  <arglist-1>
<arglist-1>      ::= <expr>  |  <expr> "," <arglist-1>
```

A.2 Commands

Atomic commands, ie. those within `<| angle braces |>`, are described by `<atomic-cmds>`, and may refer to thread-local and shared state variables in their expressions. Local commands are described by `<local-cmds>`, and may only refer to thread-local variables.

```
<atomic-cmds>   ::= ""  |  <atomic-cmd> <atomic-cmds>

<atomic-cmd>    ::= <primitive-cmd> ";"
                 |  "assert" "(" <expr> ")" ";"
                 |  "if" "(" <expr> ")" "{" <atomic-cmds> "}" <atomic-else>
                 |  "CAS" "(" <lvalue> "," <lvalue> ","<expr> ")" ";"
<atomic-else>   ::= ""  |  "else" "{" <atomic-cmds> "}"

<local-cmds>    ::= ""  |  <primitive-cmd> <local-cmds>

<primitive-cmd> ::= <lvalue> "=" <expr>
                 |  <lvalue> "=" <lvalue> <postfix>
                 |  "havoc" <lvalue>
                 |  <lvalue> <postfix>
                 |  "assume" "(" <expr> ")"
                 |  ""

<postfix>       ::= "++" | "--"
```

B The CLH Lock Proof

```
typedef int Node;

// Shared pointers to nodes
shared Node tail;
shared Node head;  // (Ghost code)

// Thread-local pointers to nodes
thread Node mynode, mypred;
thread bool test;  // Used when trying to take the lock.

// Views
view dormant(Node node);
view active(Node node);
view queued(Node node, Node pred);
view locked(Node node, Node pred);

// Goal constraint
constraint locked(a, ap) * locked(b, bp) -> false;

// Other constraints
constraint emp -> %{
    [|head|] in Foot
    && [|tail|] in Foot
    && Reach(pred, [|tail|], [|head|])
    && ![|head|].lock
    && (forall x : Node ::
          (x in Foot && x.pred != null) ==> x.lock)
    && (forall x : Node ::
          (x in Foot && Reach(pred, [|tail|], x) && !x.lock)
           ==> x == [|head|])
};

constraint dormant(node) -> %{
    [|node|] in Foot && [|node|] != [|head|] && [|node|].pred == null
    && [|node|].lock == false
};
constraint active(node) -> %{
    [|node|] in Foot && [|node|] != [|head|] && [|node|].pred == null
    && [|node|].lock == true
};

constraint queued(node, pred) -> %{
    [|node|] in Foot
    && [|pred|] in Foot
    && [|node|].pred == [|pred|]
    && [|node|].lock
    && Btwn(pred, [|tail|], [|node|], [|head|])
};
constraint locked(node, pred) -> %{
    [|node|] in Foot
    && [|pred|] in Foot
    && [|node|].pred == [|pred|]
    && Btwn(pred, [|tail|], [|node|], [|head|])
    && [|pred|] == [|head|]
};

constraint dormant(a) * dormant(b) -> a != b;
constraint active(a) * active(b) -> a != b;
constraint queued(a, ap) * queued(b, bp) -> a != b;
constraint queued(a, ap) * locked(b, bp) -> a != b;

// Proof outline
method lock() {
  {| dormant(mynode) |}
    <| %{ [|mynode|].lock := true }; |>
  {| active(mynode) |}
    <| mypred = tail; tail = mynode;
       %{[|tail|].pred := [|mypred|]}; /* Ghost code */ |>
  {| queued(mynode, mypred) |}
    do {
      {| queued(mynode, mypred) |}
        <| test = %{ [|mypred|].lock }; |>
      {| if (test) { queued(mynode, mypred) }
              else { locked(mynode, mypred) } |}
    } while (test);
  {| locked(mynode, mypred) |}
}

method unlock() {
  {| locked(mynode, mypred) |}
    <| %{ [|mynode|].lock := false };
       %{ [|mynode|].pred := null }; head = mynode; /* Ghost code */ |>
  {| dormant(mypred) |}
    mynode = mypred;
  {| dormant(mynode) |}
}
```

The CLH lock proof depends on the following auxiliary definition written in GRASShopper's assertion language:

```
struct Node {
    var lock: Bool;
    var pred: Node;  // Ghost field
}
```