Modular specification and verification for higher-order languages with state

PhD Dissertation

Kasper Svendsen
IT University of Copenhagen

October 23, 2012

Abstract

The overall topic of this thesis is modular reasoning for higher-order languages with state. The thesis consists of four mostly independent chapters that each deal with a different aspect of reasoning about higher-order languages with state. The unifying theme throughout all four chapters is higher-order separation logic.

The first chapter presents a higher-order separation logic for a higher-order subset of C♯. One of the interesting issues that arises when reasoning about higher-order code in an imperative language, is the combination of mutable variables and variable capture. In C♯, anonymous methods can have externally observable state effects on both the heap and the stack, through captured variables. However, despite state effects on the stack it is still not possible to alias stack variables in this subset of C♯. We exploit this to define a logic that allows us to treat captured variables as resources, without breaking Hoare's assignment rule.

The second chapter is concerned with the problem of reasoning about sharing in a higher-order concurrent language. When specifying shared mutable data structures, the appropriate specification depends on how clients intend to share instances of the data structures. To ensure modular reasoning, we need a generic specification that clients can refine with their desired sharing discipline. In the second chapter we present a new higher-order separation logic and a new style of specification that clients can refine with a sharing discipline of their choice.

The third chapter of the thesis is a case study of the C♯ joins library. What makes this library interesting as a case study is that it combines a lot of advanced features (higher-order code with effects, concurrency, recursion through the store, shared mutable state, and fine-grained synchronization) to implement a high-level interface for defining synchronization primitives in C♯. Due to the declarative nature of this interface, synchronization primitives implemented using the joins library admit fairly simple informal correctness arguments. We present an abstract specification of the joins library that extends these informal correctness arguments to fully formal partial correctness proofs. Using the logic developed in the second chapter, we verify a lock-based implementation of the joins library against this abstract joins specification.

The last chapter is concerned with the problem of extending a pure dependent type theory with effects. The motivation is to take advantage of an existing implementation of a pure dependent type theory to obtain an implementation of a formal proof system based on higher-order separation logic. In particular, we extend the Calculus of Inductive Constructions with monadically encapsulated stateful and potentially non-terminating computations. The monadic computation types are indexed with pre- and postconditions and act as partial correctness specifications. The type theory supports local reasoning about state through a notion of disjointness strongly inspired by separation logic.

Acknowledgements

I would like to thank my supervisor Lars Birkedal for his guidance, for taking an active interest in my research, and for helping make the PLS group such a wonderful place to work. Thanks to Aleksandar Nanevski for hosting me for 5 months at IMDEA and for sharing his proof-theoretic perspective on Hoare Type Theory. Thanks to Matthew Parkinson for hosting me for 3 months at Microsoft Research Cambridge and for teaching me verification of concurrent programs.

Thanks to all the current and former members of the PLS group at ITU for creating a very welcoming and friendly environment. Thanks to my family for their love and support.

Contents

Introduction

Almost all software contains errors that cause incorrect results or unexpected behavior with varying consequences. It has been a long-standing vision of computer science to eliminate such errors by proving the absence of errors through mathematical techniques. However, despite over 40 years of research the cost-benefit ratio of formal verification is still too high for widespread adoption. One barrier to cost-effective verification is the lack of techniques for modular specification and verification for realistic programming languages.

Programming languages typically provide various ways of decomposing code into libraries that interact through well-defined interfaces. Provided these interfaces do not expose internal implementation details, this decomposition allows libraries to be developed, tested, and maintained independently. Modular reasoning techniques should support a similar decomposition, allowing libraries to be verified independently, through formal interface specifications that abstract the intended behavior of each library.

Higher-order separation logic is one attempt at achieving modular reasoning about partial correctness properties for higher-order languages with state. In the following section we give a brief introduction to higher-order separation logic, followed by a discussion of open problems.

Background

The languages that we consider in this thesis all feature global state and allow libraries to interact through shared mutable data structures. Shared mutable data structures are difficult to reason about due to the possibility of *aliases* – i.e., syntactically distinct references pointing to the same location in memory.

Local reasoning about state. Separation logic [21] is an extension of Hoare logic [12], developed to support local reasoning about shared mutable data structures. It achieves this by internalizing a concept of a *resource* and a concept of *disjointness* in the logic. Disjointness is internalized through a new connective, $*$,

called separating conjunction. $\mathsf{P} * \mathsf{Q}$ asserts ownership of a resource that can be split into two disjoint parts, such that one satisfies P and the other satisfies Q.

Intuitively, a resource represents some information about the current state and some *rights* to change this state [8]. Standard separation logic takes heap cells as its primitive resources. In particular, $\mathsf{x.f} \mapsto \mathsf{v}$ is a resource representing the fact that the current value of heap cell $\mathsf{x.f}$ is v and the exclusive right to change the value of this heap cell. Since this resource asserts exclusive right to change heap cell $\mathsf{x.f}$, the information it represents is invariant under any rights asserted by any disjoint resources.

To update the state, specifications must assert explicit ownership of resources with rights justifying the given update. Hence, specifications preserve any assertions about disjoint resources, as expressed by the frame rule:

$$\frac{\{\mathsf{P}\}\mathsf{c}\{\mathsf{Q}\}}{\{\mathsf{P} * \mathsf{R}\}\mathsf{c}\{\mathsf{Q} * \mathsf{R}\}}$$

Informally, due to separation, R is invariant under any changes to the state permitted by rights owned by P, and any changes to the state made by c are justified by rights owned by P. Hence, the assertion R is invariant under any changes to the state made by c.

This allows separation logic to support local reasoning about shared mutable data structures. In particular, specifications need only mention the resources used to justify potential state effects, as any disjoint resources will automatically be preserved.

Modular reasoning about libraries. Operations on data structures are typically specified by defining a representation predicate that relates a model of the data structure with a concrete heap representation. The concrete state effects of an operation can then be expressed in terms of the model. For instance, we might specify a push method for a stack as follows:

$$\forall \alpha : \mathsf{seq\ Val}.\ \{\mathsf{stack}(\mathsf{x}, \alpha)\}\mathsf{x.Push(y)}\{\mathsf{stack}(\mathsf{x}, \mathsf{y} :: \alpha)\}$$

Here $\mathsf{stack}(\mathsf{x}, \alpha)$ is a representation predicate, relating the model – a list of elements in the case of a stack – with a concrete heap representation. If the stack is implemented in terms of a singly-linked list, the stack representation predicate might be defined as follows:

$$\mathsf{stack}(\mathsf{x}, \alpha) = \exists \mathsf{y} : \mathsf{Val}.\ \mathsf{x.head} \mapsto \mathsf{y} * \mathsf{lst}(\mathsf{y}, \alpha) \tag{1}$$

in terms of the following list representation predicate:

$$\begin{aligned} \mathsf{lst}(\mathsf{x}, \varepsilon) &= \mathsf{x} =_{\mathsf{Val}} \mathsf{null} \\ \mathsf{lst}(\mathsf{x}, \mathsf{y} :: \alpha) &= \exists \mathsf{z} : \mathsf{Val}.\ \mathsf{x.next} \mapsto \mathsf{z} * \mathsf{x.value} \mapsto \mathsf{y} * \mathsf{lst}(\mathsf{z}, \alpha) \end{aligned}$$

Higher-order separation logic [3] extends separation logic with higher-order quantification. This allows library specifications to hide their internal data representation from clients, by existentially quantifying over representation predicates used to specify the library. For instance, we can specify a simple stack library as follows:

$$\begin{array}{l}\exists \mathsf{stack} : \mathsf{Val} \times \mathsf{seq}\ \mathsf{Val} \to \mathsf{Prop}.\\ \quad \{\mathrm{emp}\}\mathbf{new}\ \mathsf{Stack}()\{\mathsf{ret}.\ \mathsf{stack}(\mathsf{ret}, \varepsilon)\} \wedge\\ \quad \forall \alpha : \mathsf{seq}\ \mathsf{Val}.\ \{\mathsf{stack}(\mathsf{x}, \alpha)\}\mathsf{x}.\mathsf{Push}(\mathsf{y})\{\mathsf{stack}(\mathsf{x}, \mathsf{y} :: \alpha)\} \wedge\\ \quad \forall \alpha : \mathsf{seq}\ \mathsf{Val}.\ \forall \mathsf{y} : \mathsf{Val}.\ \{\mathsf{stack}(\mathsf{x}, \mathsf{y} :: \alpha)\}\mathsf{x}.\mathsf{Pop}()\{\mathsf{ret}.\ \mathsf{stack}(\mathsf{x}, \alpha) * \mathsf{ret} = \mathsf{y}\}\end{array}$$

Clients can thus be verified against an abstract library specification and linked with any library implementation satisfying the abstract library specification. Conversely, when reasoning about higher-order library methods, we can use universal quantification over representation predicates to reason abstractly about function arguments. To illustrate, consider extending the stack library with a `ForEach` method that applies the given function argument to each element of the stack.

$$\begin{array}{l}\forall \mathsf{I} : \mathsf{seq}\ \mathsf{Val} \times \mathsf{seq}\ \mathsf{Val} \to \mathsf{Prop}.\ \forall \alpha : \mathsf{seq}\ \mathsf{Val}.\\ \quad \{\mathsf{stack}(\mathsf{x}, \alpha) * \mathsf{I}(\alpha, \varepsilon) * \forall \beta, \eta : \mathsf{seq}\ \mathsf{Val}.\ \mathsf{b} \mapsto (\mathsf{y}).\{\mathsf{I}(\mathsf{y} :: \beta, \eta)\}\{\mathsf{I}(\beta, \mathsf{y} :: \eta)\}\}\\ \qquad \mathsf{x}.\mathsf{ForEach}(\mathsf{b})\\ \quad \{\mathsf{stack}(\mathsf{x}, \alpha) * \mathsf{I}(\varepsilon, \mathsf{rev}(\alpha))\}\end{array} \tag{2}$$

Here I is a representation predicate chosen by the caller of `ForEach`, to abstractly describe the effects of the given function argument. Intuitively, $\mathsf{I}(\alpha, \beta)$ describes the state after having called the given function argument on each element of β, with the elements in α still to go. We thus assert, using a nested Hoare triple [23], that the given function argument (b) should satisfy the specification:

$$\forall \beta, \eta : \mathsf{seq}\ \mathsf{Val}.\ \mathsf{b} \mapsto (\mathsf{y}).\{\mathsf{I}(\mathsf{y} :: \beta, \eta)\}\{\mathsf{I}(\beta, \mathsf{y} :: \eta)\}$$

That is, when called with argument y in a state where y is the next stack element up for processing, the function argument should process y according to the chosen representation predicate (I).

Higher-order separation logic thus supports local reasoning about disjoint resources and formal interface specifications that abstract internal data representations across library boundaries. This is a step towards modular reasoning. In the next section we discuss some open problems.

Open problems

Separation logic is a rapidly advancing field and some of the problems discussed in this section have already been addressed by others within the last 4 years, some we address in this thesis, and some are still open.

Logical resources. As we sketched in the previous section, separation logic achieves local reasoning about state through *resources* and the concept of *rights*. Standard separation logic provides primitive resources based on individual heap cells with a very coarse-grained notion of rights (such as the exclusive right to update a given heap cell). This induces a very coarse-grained notion of disjointness. Representation predicates defined in terms of primitive resources simply inherit this coarse-grained notion of disjointness. For instance, the stack representation predicate defined in (1) inherits its notion of disjointness from the primitive heap cell resources used to specify the data representation. Since these primitive heap cell resources cannot be shared, the stack as a whole cannot be shared:

$$\mathsf{stack}(\mathsf{x}, \alpha) * \mathsf{stack}(\mathsf{x}, \beta) \Rightarrow \bot$$

To support local reasoning at the level of abstract data structures, we need ways of lifting the notion of rights from the underlying heap cells, to the operations exposed by the abstract data structure. By lifting the notion of rights, we can construct *logical resources* that support sharing disciplines expressed at the level of the abstract data structure instead of the underlying data representation.

Specifications. In the introduction we defined modular reasoning as allowing libraries to be verified independently through formal interface specifications that abstractly describe the intended behavior. This definition implicitly assumes that for the libraries we are interested in verifying we can define a single abstract specification that covers all intended uses of the library. This is not an obviously valid assumption, especially in the presence of higher-order features or sharing.

For instance, it is not at all clear that the specification of `ForEach` proposed above, (2), is sufficiently general to capture the state effects of all sensible function arguments for `ForEach`.

Furthermore, as sketched above, to support local reasoning about abstract data structures, we need logical resources with rights corresponding to the operations of the data structure. Since the appropriate notion of rights depends on the intended use of the data structure, *clients* should be able to construct these logical resources. To support local and modular reasoning about abstract data structures we thus need a single formal interface specification that hides internal data representations through abstract representation predicates *and* allows *clients* to construct logical resources from these abstract representation predicates.

Real world. Finally, to verify larger and more realistic examples we need separation logics for more realistic programming languages and computer support for carrying out proofs in these logics.

Contents

This dissertation consists of four articles and three technical reports organized into four chapters. The unifying goal throughout all four chapters is modular reasoning about partial correctness properties using higher-order separation logic, and each chapter deals with one of the open problems sketched above. With the exception of Chapters 2 and 3, each chapter can be read independently.

Each chapter starts with an article containing a self-contained introduction. This section provides a more informal and high-level overview of each chapter. Since the chapters are mostly independent, related and future work is addressed in each chapter. We include some references in this introduction; however, we refer the reader to the articles for more detailed references.

Chapter 1: State in C♯

Most of the early work on separation logic focused on low-level imperative languages without higher-order features. More recently, we have seen separation logics for higher-order functional languages with state [14, 16]. The first chapter of the thesis concerns the problem of reasoning about state in a higher-order imperative language. In particular, we present a new higher-order separation logic for a subset of C♯ that includes delegates (i.e., type-safe function pointers), anonymous methods, and generics.

Variable capture. The main difficulty introduced in C♯ is the combination of mutable variables and variable capture. In particular, in C♯ anonymous methods capture the location rather than the value of captured variables and the lifetime of captured variables is extended to the lifetime of the capturing delegate. Delegates can thus modify local variables from enclosing scopes. To illustrate the issue, consider the following snippet of C♯ code:

```
public int Sum(Stack⟨int⟩ s) {
  int sum = 0;
  s.ForEach((n) => { sum = sum + n; });
  return sum;
}
```

This method computes the sum of the elements of the given stack of integers. It achieves this by applying the anonymous delegate, `(n) => { sum = sum + n; }`, to each element of the stack, thus adding each element, `n`, to the running total, `sum`. As the above example illustrates, delegates can thus have externally observable state effects on both the heap *and the stack*.

This poses a problem for the treatment of variables in separation logic. In particular, in separation logic, pre- and postconditions are usually assertions about the heap, expressed in terms of the stack. This suffices for describing state effects on the heap, but not the stack. Furthermore, since variables cannot be aliased, it is sound to reason about variable assignments using Hoare's assignment rule (i.e., substitution). Variable capture does not introduce aliasing of variables, so we should still be able to reason about assignments using Hoare's assignment rule. In the first chapter, we present a higher-order separation logic with assertions for reasoning about the stack, *without breaking Hoare's assignment rule*. Verification of code without variable capture is thus unaffected by this extension of the logic.

Variables-as-resources. We present a higher-order separation logic extended with assertions for reasoning about the location of a stack variable and the contents and ownership of a given stack location. In particular, $\mathsf{x} \overset{s}{\mapsto} \mathsf{v}$ is a new primitive resource representing the fact that the stack location of stack variable x contains v and the exclusive right to modify this location. Furthermore, $\&\mathsf{x}$ denotes the stack location of the stack variable x. With these ingredients we can specify the anonymous delegate from the Sum method as follows:

$$\forall \mathsf{v}.\ (\mathsf{n}).\{\&\mathsf{sum} \overset{s}{\mapsto} \mathsf{v}\}\{\&\mathsf{sum} \overset{s}{\mapsto} \mathsf{v} + \mathsf{n}\}$$

Intuitively, this specification expresses that the delegate, if it terminates, adds its argument (n) to sum. Note that if we had simply referred directly to sum in the postcondition, then this would have referred to sum's current value. Instead, we use the stack points-to predicate to *assert* the value of sum at the *entry and exit* of a call to the anonymous delegate.

The logic thus features two ways of reasoning about stack variables – through the context and through variables-as-resources [5, 18]. While a variable is still in scope, we can switch freely between these two modes of reasoning. For instance, reading from the bottom up, the following rule switches from reasoning about a stack variable x through the context (ϕ, x) to stack assertions $(l \overset{s}{\mapsto} \mathsf{x})$:

$$\frac{\phi; \psi \vdash \{\exists \mathsf{x} : \mathsf{Val}.\ (l \overset{s}{\mapsto} \mathsf{x} * \mathsf{P})\} C \{\exists \mathsf{x} : \mathsf{Val}.\ (l \overset{s}{\mapsto} \mathsf{x} * \mathsf{Q})\} \qquad \mathsf{x} \notin \phi}{\phi, \mathsf{x}; \psi \vdash \{\&\mathsf{x} = l \wedge \mathsf{P}\} C \{\mathsf{Q}\}}$$

Pre- and postconditions are thus assertions on pairs of heaps and stacks, expressed in terms of the stack (i.e., the context). To ensure that these two modes of reasoning about the stack do not introduce aliasing (which would break Hoare's assignment rule), the stack context implicitly asserts ownership of the locations of its stack variables. This ensures that for each variable we only reason about its current value using one of these modes at any given point in a proof. The above

proof rule thus requires that $\mathsf{x} \notin \phi$. This means we can use stack assertions to *specify* the state effects of delegates on the stack and switch to context reasoning when *verifying* delegate bodies against these specifications.

Abstraction. We usually use existentially quantified representation predicates to abstract over internal state on the heap. Since our representation predicates now describe both the stack and the heap, we can also abstract over internal state on the stack. To illustrate, consider the following C♯ snippet:

```
public Action⟨int⟩ Counter() {
  int count = 0;
  return (() => { count++; return count; });
}
```

This method returns a delegate that returns the number of times it has been invoked. We can give this method a concrete specification that reveals the local state:

$$\{\mathsf{emp}\}$$
$$\quad \mathsf{Counter()}$$
$$\{\mathsf{ret}.\ \exists l : \mathsf{Loc}.\ l \overset{s}{\mapsto} 0 * \forall \mathsf{n}.\ \mathsf{ret} \mapsto \{l \overset{s}{\mapsto} \mathsf{n}\}\{\mathsf{ret}.\ \mathsf{ret} = \mathsf{n}+1 * l \overset{s}{\mapsto} \mathsf{n}+1\}\}$$

or use existential quantification to abstract over the stack points-to predicate. As this example illustrates, we can also reason about captured variables after they have gone out of scope, by existentially quantifying over the stack location of the out-of-scope captured variable.

The first chapter consists of an article and a technical appendix. The article gives an overview of the logic and its semantics. The full logic and semantics is defined in the accompanying technical appendix. The technical appendix is not a self-contained work and assumes the reader is familiar with the preceding article. The article has been published at ECOOP 2010.

Chapter 2: State, sharing and concurrency in C♯

One of the open problems we mentioned in the introduction was the need for specifications of abstract data structures that allow clients to choose a sharing discipline that matches their intended use. In the second chapter we take a step towards solving this problem, in the context of a higher-order concurrent subset of C♯.

Abstract data structure resources. To illustrate the problem, consider a simple counter with methods to increment and read the current count. Assume this counter satisfies the following specification, which enforces a single exclusive owner and allows that owner to track the *exact value* of the counter:

$$\begin{array}{ll} \{\mathsf{counter}_e(\mathsf{x},\mathsf{n})\} & \mathsf{x.Read}() \quad \{\mathsf{ret}.\ \mathsf{counter}_e(\mathsf{x},\mathsf{n}) \wedge \mathsf{ret} = \mathsf{n}\} \\ \{\mathsf{counter}_e(\mathsf{x},\mathsf{n})\} & \mathsf{x.Increment}() \quad \{\mathsf{counter}_e(\mathsf{x},\mathsf{n}+1)\} \end{array}$$
$$\mathsf{counter}_e(\mathsf{x},\mathsf{n}) * \mathsf{counter}_e(\mathsf{x},\mathsf{m}) \Rightarrow \bot$$

The $\mathsf{counter}_e(\mathsf{x},\mathsf{n})$ resource represents the fact that the current count is exactly n and the *exclusive right* to increment the current count. Assume this counter also satisfies the following specification, which allows unrestricted sharing of the counter by only providing each owner with a *lower-bound* on the current count [20]:

$$\begin{array}{ll} \{\mathsf{counter}_s(\mathsf{x},\mathsf{n})\} & \mathsf{x.Read}() \quad \{\mathsf{ret}.\ \mathsf{counter}_s(\mathsf{x},\mathsf{ret}) \wedge \mathsf{n} \leq \mathsf{ret}\} \\ \{\mathsf{counter}_s(\mathsf{x},\mathsf{n})\} & \mathsf{x.Increment}() \quad \{\mathsf{counter}_s(\mathsf{x},\mathsf{n}+1)\} \end{array}$$
$$\mathsf{counter}_s(\mathsf{x},\mathsf{n}) \Leftrightarrow \mathsf{counter}_s(\mathsf{x},\mathsf{n}) * \mathsf{counter}_s(\mathsf{x},\mathsf{n})$$

The $\mathsf{counter}_s(\mathsf{x},\mathsf{n})$ resource represents the fact that the current count is at least n and the *non-exclusive right* to increment the current count. This illustrates two different ways of constructing a counter resource that results in two different sharing disciplines. The appropriate sharing discipline depends on the intended use of the counter. It should thus be up to the *client* to pick a sharing discipline that matches the client's intended use. To support modular reasoning, we thus want a single abstract counter specification that allows us to *define* $\mathsf{counter}_e$ and $\mathsf{counter}_s$ and *derive* both of these specifications.

In a first-order sequential setting without reentrancy, the first counter specification suffices. In the absence of reentrancy and interleavings, clients can construct new counter resources in terms of the abstract $\mathsf{counter}_e$ predicate. In particular, clients can define $\mathsf{counter}_s$ from $\mathsf{counter}_e$ and derive the $\mathsf{counter}_s$ specification. Unfortunately, this is not sound in a higher-order concurrent setting with reentrancy. For instance, there exists non thread-safe counter implementations (without internal synchronization) that satisfy the first counter specification, but not the second.

In the second chapter of the thesis we propose a new higher-order separation logic and a new style of specification for thread-safe implementations of shared mutable data structures. This new style of specification allow clients to reason about the atomic points *inside* library methods where the abstract state changes. This gives clients the ability to construct logical resources with a sharing discipline that matches the intended use of the thread-safe data structure in question.

Higher-order resources. Resources allow us to *share* ownership of mutable data structures through the notion of *rights*. However, we also use shared mutable data structures to control access to other mutable data structures. The canonical example is a lock. In separation logic, we control access to mutable data structures through ownership transfer. To support reasoning about sharing of mutable data structures through a shared mutable data structure, our new logic features *higher-order resources*. In particular, we extend the concept of rights to allow rights to specify ownership transfer protocols of resources.

To illustrate, consider a library for scheduling independent tasks for parallel execution. One might implement such a library using a bag to share all the tasks scheduled for execution between a set of worker threads. For this particular use-case the appropriate bag specification might be as follows:

$$\begin{array}{rcl} \{\mathsf{bag}_s(\mathsf{x},\mathsf{P}) * \mathsf{P}(\mathsf{y})\} & \mathsf{x.Push(y)} & \{\mathsf{bag}_s(\mathsf{x},\mathsf{P})\} \\ \{\mathsf{bag}_s(\mathsf{x},\mathsf{P})\} & \mathsf{x.Pop()} & \{\mathsf{ret}.\ \mathsf{bag}_s(\mathsf{x},\mathsf{P}) * (\mathsf{ret} = \mathsf{null} \vee \mathsf{P}(\mathsf{ret}))\} \end{array}$$

$$\mathsf{bag}_s(\mathsf{x},\mathsf{P}) \Leftrightarrow \mathsf{bag}_s(\mathsf{x},\mathsf{P}) * \mathsf{bag}_s(\mathsf{x},\mathsf{P})$$

This specification allows unrestricted sharing of the bag, but does not allow owners to track the contents of the bag. Instead, it allows clients of the bag to associate additional resources with each element of the bag, through the P predicate. Upon pushing an element y, the client is thus required to transfer $\mathsf{P}(\mathsf{y})$ to the bag. Conversely, upon popping an element, if the bag returns a non-null element ret, then $\mathsf{P}(\mathsf{ret})$ is transferred from the bag to the client.

The bag_s resource thus represents: the fact that there exists a bag with some multiset of elements X that owns the resources $\circledast_{\mathsf{x}\in\mathsf{X}}\mathsf{P}(\mathsf{x})$, the right to add elements to this bag by transferring the resources associated with the given element from client to bag, and the right to remove elements from this bag by transferring the resources associated with the given elements from bag to client. This makes bag_s a higher-order resource.

Once we start reasoning about sharing of a data structure through a shared mutable data structure, we immediately encounter the problem of sharing a shared mutable data structure through itself. This is tricky to reason about. A previous logic with higher-order resources [10] failed to address this issue and was consequently unsound. To ensure soundness, we impose certain restrictions on higher-order resources in our logic, including a stratification of the construction of resources. In particular, we introduce an ordering on resources and prevent resources from specifying ownership transfer protocols on resources greater than or equal to itself in the ordering. [1]

[1] Since the submission of this thesis, we have presented a new model and accompanying logic without these restriction [24].

The logic. The higher-order separation logic that we propose in Chapter 2 is a general-purpose logic for reasoning about C♯ code that combines concurrency, state, sharing and reentrancy. It features a higher-order extension of concurrent abstract predicates [9] to reason about shared state in a concurrent setting. Concurrent abstract predicates partitions the state into regions with protocols governing how the state in each region is allowed to evolve. To allow clients to construct logical resources from instances of abstract data structures, we extend concurrent abstract predicates with a way of synchronizing two regions so they evolve in lock-step. To support higher-order resources we extend concurrent abstract predicates with state-independent higher-order protocols. The logic features guarded recursion [4] to reason about reentrancy. We prove soundness of the logic by constructing a model. The presentation of the model is strongly inspired by the Views framework [8].

When reasoning about stability and atomic updates, most previous work on concurrent abstract predicates have resorted to working directly in the model. To avoid this, we define an extensive proof system for reasoning about concurrent abstract predicates in the logic.

Chapter 2 consists of an article and a technical report. The article contains an overview of the logic and its semantics, but is mainly concerned with our specification pattern for thread-safe shared mutable data structures. The full logic and semantics is given in the accompanying technical report. The article has been published at ESOP 2013.

Chapter 3: The joins library

The third chapter of the thesis is a case study of the C♯ joins library [2, 22]. This is a C♯ library for defining synchronization primitives, based on the joins calculus [11]. What makes it interesting as a verification and specification challenge is that it combines a lot of advanced features, including higher-order code, state, concurrency, sharing, and reentrancy. We specify and verify the library using the higher-order separation logic from Chapter 2. In addition to being an interesting specification and verification challenge the case study is thus also intended to demonstrate that the logic from Chapter 2 scales to realistic examples.

Joins. To define a synchronization primitive using the joins library one declares a set of *channels* and *chords*. Clients interact by sending messages on channels, which are received by chords. There are two types of channels, *synchronous* and *asynchronous* channels. Sending a message on a synchronous channel blocks the sender until the message has been received. Chords consist of a channel pattern

and a continuation given as a delegate. A chord may *fire* when the pattern matches a set of messages by atomically *consuming* the matched messages before executing the continuation. Once the continuation terminates, the consumed messages are *received* and any blocked senders of the received messages are allowed to continue.

To illustrate how the library works and the challenges involved in reasoning about clients, consider the following joins implementation of a reader/writer lock.

```
class RWLock {
  public SyncChannel acqR, acqW, relR, relW;
  private AsyncChannel unused, shared, writer;
  private int readers = 0;

  public RWLock() {
    Join join = new Join();
    // ... initialize channels ...

    join.When(acqR).And(unused).Do(() => { readers++; shared.Call(); });
    join.When(acqR).And(shared).Do(() => { readers++; shared.Call(); });
    join.When(acqW).And(unused).Do(() => { writer.Call(); });
    join.When(relW).And(writer).Do(() => { unused.Call(); });
    join.When(relR).And(shared).Do(() => {
      if (--readers == 0) unused.Call() else shared.Call(); });

    unused.Call();
  }
}
```

This example declares four public synchronous channels for clients of the lock to acquire and release the lock (acqR, ...), and three private asynchronous channels (unused, ...). The statement join.When(ch1).And(ch2).Do(b) defines a new chord with continuation b and a pattern that matches a pending message from each channel ch1 and ch2. The statement ch.Call() sends a message on the channel ch.

To see how this implements a reader/writer lock, note that each chord consumes exactly one asynchronous message (i.e., a message on an asynchronous channel) and sends exactly one asynchronous message. Furthermore, the unused channel is initialized with exactly one pending message. The asynchronous channels thus satisfy the invariant that there is at most one pending asynchronous message at any given point in time. If this pending asynchronous message is on the unused channel, there are currently no readers or writers; if it is on the shared channel, there is at least one reader; and if it is on the writer channel, there is exactly one writer. The asynchronous channels thus encode the state of the lock, and the readers field encode the actual number of readers in the shared state. One can thus

read the first chord as asserting that the lock may grant a read request when there are no readers or writers, by incrementing the number of readers and transitioning to a state with at least one reader.

Challenges. Note that several of the continuations in the above example have state effects (i.e., they modify the readers field). Note further that despite the fact that multiple continuations may execute in parallel, the continuations in the above example *do not* synchronize access to the readers field. Intuitively, this is sound because of the invariant on the number of pending asynchronous messages, which ensures that between consuming and sending an asynchronous message only the continuation of the chord that consumed the message is running. Note further that all of the continuations make *reentrant* calls back into the joins library by sending messages on channels.

To reason about joins clients like the reader/writer lock we thus need a specification that supports continuations with state effects that make reentrant calls back into the joins library. The second chord even matches and sends a message on the same channel (shared). Since one can encode recursion by matching and sending on the same channel, the joins specification must also support continuations that encode recursion through the store through the joins library. This illustrates some of the challenges of specifying the joins library.

Joins specification. In the third chapter of the thesis we propose an abstract specification of the joins library that supports stateful reentrant continuations. The basic idea behind the abstract joins specification is to allow clients of the joins library to pick an ownership transfer protocol for each channel. An ownership transfer protocol consists of a channel precondition and a channel postcondition. The channel precondition describes the resources the sender is required to transfer to the recipient upon sending a message on the channel. The channel postcondition describes the resources the recipient is required to transfer back to the sender upon receiving a message on the channel. The client picks an ownership transfer protocol for each channel up front. As the client adds chords, the client then has to prove that each chord satisfies the chosen protocol. To support reentrant calls, the client is allowed to assume channels obey the chosen protocol when proving that continuations satisfy the chosen protocol.

To demonstrate that this abstract joins specification is sufficiently strong to reason about interesting clients, we verify a series of classic synchronization primitives implemented using the joins library. To demonstrate that the joins specification is not too strong (i.e., that it is actually implementable), we verify a naive lock-based implementation of the joins library against our abstract joins specification.

The two main challenges in verifying the joins implementation is reentrancy

and sharing. Since our abstract joins specification explicitly supports reentrant continuations, some of the abstract joins representation predicates have to refer to themselves recursively. We thus define these abstract join representation predicates by guarded recursion. Moreover, since the joins library is itself implemented using shared mutable state, we need to prove that these low-level shared mutable data structures enforce the high-level ownership transfer protocol chosen by the client. We use the higher-order extension of concurrent abstract predicates from Chapter 2 to prove this.

Chapter 3 consists of a paper and a technical report. The paper presents the abstract joins specification and illustrates how to use it by verifying a series of classic synchronization primitives. The technical report includes a full proof outline of the lock-based joins implementation against the abstract joins specification. The paper has been published at ECOOP 2013. The paper is self-contained and can be read independently of the rest of the thesis. The technical report assumes the reader is familiar with the logic introduced in Chapter 2.

Chapter 4: State and non-termination in CIC

The fourth chapter of the thesis addresses the problem of extending the Calculus of Inductive Constructions [1] with stateful, potentially non-terminating computations.

Dependent type theories such as the Calculus of Inductive Constructions (CIC) provide powerful languages for integrated programming, specification, and verification. However, to maintain soundness, they typically require all computations to be pure and terminating, severely limiting their use as general purpose programming languages. The goal of the Hoare Type Theory project is to extend a pure dependent type theory with effects, to obtain a general-purpose programming language with integrated support for specification and verification [16, 17].

Hoare Type Theory extends pure dependent type theories with effects by encapsulating effectful computations with monadic computation types. To extend the principle of specifications-as-types to effectful computations these computation types – also called Hoare types – are further indexed with pre- and postconditions. Hoare types, written $\{p\}x : \tau\{q\}$, act as partial correctness specifications. Like separation logic, Hoare Type Theory internalizes a notion of disjointness to support local reasoning about state. The Hoare Type Theory approach has previously been used to extend both the Calculus of Constructions [6] and the Extended Calculus of Constructions [15] with state and potentially non-terminating computations [16, 19].

Neither of these prior versions of Hoare Type Theory embed into CIC. In the fourth chapter of the thesis we present a new version of Hoare Type Theory that

extends CIC with stateful, potentially non-terminating computations. The reason why CIC is so important and our main reason for targeting CIC is Coq – the implementation of CIC. By presenting our type theory as an extension of CIC, we *get an implementation for free*, as an axiomatic extension of Coq.

While our treatment of state is strongly inspired by previous versions of Hoare Type Theory, our treatment of non-termination is new. For the purposes of this introduction, we will thus focus on our treatment of non-termination.

Non-termination in dependent type theories. Allowing unrestricted recursion breaks the principle of propositions-as-types, as one can construct a non-terminating term of *false* (i.e., the type of proofs of the proposition *false*). Following Constable and Smith, we add partiality by introducing a type $O(\tau)$ of potentially non-terminating computations of type τ, along with the following fixed point principle for typing recursively defined computations:

$$\mathsf{fix}_\tau : (O(\tau) \to O(\tau)) \to O(\tau)$$

Unfortunately, in sufficiently expressive dependent type theories, there exist types for which the above fixed point principle is unsound [7]. For instance, in type theories with subset-types, the fixed point principle allows reasoning by a form of fixed point induction, which is only sound for admissible predicates. Previous type theories based on the idea of partial types that admit fixed points have approached the admissibility issue in roughly two different ways:

- The notion of admissibility is axiomatized in the type theory and explicit admissibility conditions are required in order to use fix. This approach has, e.g., been investigated by Crary in the context of Nuprl [7]. The resulting type theory is expressive, but admissibility conditions lead to significant proof obligations, particularly when using Σ types.

- The underlying dependent type theory is restricted in such a way that one can only form types that are trivially admissible. This was the approach used in prior work on Hoare Type Theory to extend the Calculus of Constructions with effects [19]. That version of Hoare Type Theory excluded strong Σ types (and, more generally, dependent inductive types) from the *Set* universe, to ensure that all types were trivially admissible. Since the *Set* universe of CIC *does* feature dependent inductive types, this approach does not scale to CIC.

In the fourth chapter we explore a third approach, which ensures that all types are admissible, not by limiting the underlying standard dependent type theory, but by limiting only the partial types. The limitation on partial types consists of equating all effectful computations at a given type: if M and N are both of

type $O(\tau)$, then they are propositionally equal. Thus, with this approach, the only way to reason about effectful computations is through their type, rather than via equality or predicates. With sufficiently expressive types, the type of an effectful computation can serve as a partial correctness specification of the computation. This approach allows us to restrict attention to a subset of admissible types that is closed under the standard dependent type formers, including dependent inductive types.

Impredicative Hoare Type Theory. Using this new approach to partiality we present a new impredicative variant of Hoare Type Theory – dubbed iHTT – that embeds into CIC. To demonstrate the soundness of iHTT we construct a model of iHTT. The model is based on a standard realizability model of partial equivalence relations (PERs) and assemblies over a combinatory algebra [13]. These give rise to a model of the Calculus of Constructions, that models the *Set* universe using PERs. Restricting to *complete* PERs (i.e., PERs closed under limits of ω-chains) over a suitable universal domain yields a model of recursion in a simply-typed setting and a dependently-typed setting without dependent inductive types. Our contribution is in identifying a set of *complete monotone* PERs that models a full dependent type universe (including dependent inductive types) *and* recursion.

Our model shows the soundness of Hoare Type Theory with higher-order store and justifies the first implementation of Hoare Type Theory with higher-order store as an axiomatic extension of Coq. The implementation is available at:

`http://www.itu.dk/people/kasv/ihtt.tgz`

Chapter 4 consists of a paper. The paper is an extended version of a paper published at TLCA 2011. The paper presents our new approach to non-termination, a new version of impredicative Hoare Type Theory, and a model of iHTT.

Summary of contributions

In this thesis we present two new higher-order separation logics for C♯. The first logic is concerned with the problem of reasoning about state effects on the stack, without introducing explicit reasoning about aliasing on the stack. We achieve this with a new non-standard variables-as-resources extension of higher-order separation logic.

The second logic is concerned with the problem of reasoning about shared mutable data structures in a higher-order concurrent setting. The second logic is a higher-order variant of concurrent abstract predicates, extended with support for synchronizing multiple regions and higher-order protocols. We present a new way of specifying thread-safe shared mutable data structures in this logic that allow clients to refine specifications with a sharing discipline that matches the intended use. We also present an extensive proof system for reasoning about concurrent abstract predicates.

We demonstrate that this second logic scales to realistic examples by verifying the C♯ joins library. This library combines state, sharing, concurrency, and reentrancy to realize a high-level interface for defining synchronization primitives. We give an abstract specification of the joins library and illustrates its use by verifying a series of classic synchronization primitives implemented using the joins library. To demonstrate that the specification is realizable, we verify a naive lock-based implementation against this abstract joins specification.

Finally, we present a new approach for extending pure dependent type theories with monadically encapsulated potentially non-terminating computations. By collapsing the propositional equality on these potentially non-terminating computations, we can restrict attention to a subset of admissible types that includes a full dependent type universe and dependent inductive types. We apply this approach to extend CIC with stateful, potentially non-terminating computations. This justifies the first implementation of Hoare Type Theory with higher-order store.

List of Publications

During my PhD I've coauthored five articles and three technical reports. This thesis includes the four articles and three technical reports for which I am the primary author. Below is a list of the articles and technical reports.

The following articles are joint work with Lars Birkedal and Matthew Parkinson:

- Verifying Generics and Delegates.
- Modular Reasoning about Separation for Concurrent Data Structures.
- Joins: A Case Study in Modular Specification of a Concurrent Reentrant Higher-order Library.

and so are the following technical reports:

- Verifying Generics and Delegates: Technical Appendix.
- Higher-order Concurrent Abstract Predicates.
- Verification of the Joins Library in Higher-order Separation Logic.

The first article was published at ECOOP 2010, the second at ESOP 2013 and the third at ECOOP 2013.

The following article is joint work with Lars Birkedal and Aleksandar Nanevski:

- Partiality, State and Dependent Types.

The fourth article was published at TLCA 2011. The article included in the thesis is an extended version of the published article.

The fifth article is joint work with Neelakantan Krishnaswami, Jonathan Aldrich, Lars Birkedal and Alexandre Buisse:

- Design Patterns in Separation Logic.

The fifth article was published at TLDI 2009. My contribution to this work was primarily related to the Coq formalization; hence, this work is not part of the thesis.

Bibliography

[1] *Coq Reference Manual, Version 8.3*.

[2] Nick Benton, Luca Cardelli, and Cédric Fournet. Modern Concurrency Abstractions for C♯. *ACM Transactions of Programming Languages and Systems*, 26(5), 2004.

[3] Bodil Biering, Lars Birkedal, and Noah Torp-Smith. BI-Hyperdoctrines, Higher-order Separation Logic, and Abstraction. *ACM Transactions on Programming Languages and Systems*, 2007.

[4] Lars Birkedal, Rasmus Møgelberg, Jan Schwinghammer, and Kristian Støvring. First steps in synthetic guarded domain theory: step-indexing in the topos of trees. In *Proceedings of LICS*, 2011.

[5] Richard Bornat, Cristiano Calcagno, and Hongseok Yang. Variables as Resources in Separation Logic. In *Proceedings of MFPS*, pages 125–146, 2005.

[6] Thierry Coquand and Gerard Huet. The calculus of constructions. *Information and Computation*, 76(2-3):95–120, 1988.

[7] Karl Crary. Admissibility of Fixpoint Induction over Partial Types. In *Automated Deduction - CADE-15*, 1998.

[8] Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew Parkinson, and Hongseok Yang. Views: Compositional Reasoning for Concurrent Programs. In *Proceedings of POPL*, 2013.

[9] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. Concurrent Abstract Predicates. In *Proceedings of ECOOP*, 2010.

[10] Mike Dodds, Suresh Jagannathan, and Matthew J. Parkinson. Modular reasoning for deterministic parallelism. In *Proceedings of POPL 2011*, POPL '11, pages 259–270, 2011.

[11] Cédric Fournet and Georges Gonthier. The Join Calculus: A Language for Distributed Mobile Programming. In *Proceedings of APPSEM*, pages 268–332, 2000.

[12] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[13] Bart Jacobs. *Categorical Logic and Type Theory*. Elsevier Science, 1999.

[14] Neelakantan R. Krishnaswami. *Verifying Higher-Order Imperative Programs with Higher-Order Separation Logic*. PhD thesis, Carnegie Mellon University, 2012.

[15] Zhaohui Luo. *Computation and reasoning: a type theory for computer science*. Oxford University Press, 1994.

[16] Aleksandar Nanevski, Amal Ahmed, Greg Morrisett, and Lars Birkedal. Abstract Predicates and Mutable ADTs in Hoare Type Theory. In *ESOP*, pages 189–204, 2007.

[17] Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Hoare Type Theory, Polymorphism and Separation. *Journal of Functional Programming*, 18(5–6):865–911, 2008.

[18] Matthew J. Parkinson, Richard Bornat, and Cristiano Calcagno. Variables as resource in Hoare Logic. In *Proceedings of LICS*, pages 137–146. IEEE, 2006.

[19] Rasmus L. Petersen, Lars Birkedal, Aleksandar Nanevski, and Greg Morrisett. A Realizability Model of Impredicative Hoare Type Theory. In *Proceedings of ESOP 2008*, 2008.

[20] Alexandre Pilkiewicz and Franccois Pottier. The essence of monotonic state. In *Proceedings of TLDI*, pages 73–86, 2011.

[21] John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*, pages 55–74, 2002.

[22] Claudio V. Russo. The Joins Concurrency Library. In *Proceedings of PADL*, pages 260–274, 2007.

[23] Jan Schwinghammer, Lars Birkedal, Bernhard Reus, and Hongseok Yang. Nested Hoare Triples and Frame Rules for Higher-order Store. In *Proceedings of CSL*, April 2009.

[24] Kasper Svendsen and Lars Birkedal. Impredicative Concurrent Abstract Predicates. Under submission, 2013.

Chapter 1

State in C♯

Verifying Generics and Delegates

Kasper Svendsen[1], Lars Birkedal[1], and Matthew Parkinson[2]

[1] IT University of Copenhagen, {kasv,birkedal}@itu.dk
[2] University of Cambridge, Matthew.Parkinson@cl.cam.ac.uk

Abstract. Recently, object-oriented languages, such as C♯, have been extended with language features prevalent in most functional languages: parametric polymorphism and higher-order functions. In the OO world these are called generics and delegates, respectively. These features allow for greater code reuse and reduce the possibilities for runtime errors. However, the combination of these features pushes the language beyond current object-oriented verification techniques.
In this paper, we address this by extending a higher-order separation logic with new assertions for reasoning about delegates and variables. We faithfully capture the semantics of C♯ delegates including their capture of the l-value of a variable, and that "stack" variables can live beyond their "scope". We demonstrate that our logic is sound and illustrate its use by specifying and verifying a series of interesting and challenging examples.

1 Introduction

There has been a recent trend for object-oriented languages, like C♯, to adopt features such as generics (parametric polymorphism) and delegates (first class functions). These features help the programmer improve code reuseability and reliability by providing greater abstraction at the level of types. However, C♯ type safety falls short of proving that programs do the right thing, it simply prevents certain classes of errors.

Program verification enables proofs that programs do the right thing, and there has been a lot of work on verifying object-oriented languages, for example [6, 8, 2, 1]. However, the majority of this work falls short of reasoning about features such as generics or delegates.

Higher-order separation logic (HOSL) [3, 9] and Hoare Type Theory (HTT) [11, 18] have both been developed to reason about higher-order functional languages. HOSL and HTT uses quantification over propositions to allow reasoning about parametric polymorphism, and HTT uses nested Hoare triples to allow reasoning about first class functions.

We borrow ideas from HOSL to extend a separation logic for object-oriented programs [16, 14, 15] to reason about generics and delegates. Unfortunately, the combination is not straightforward. Neither, HTT nor HOSL deal with the combination of mutable variables and first class functions that we have in C♯. In particular, anonymous C♯ delegates have a surprising behaviour for capturing

variables. First, they capture the l-value, i.e., the location of the variable, rather than just its value, and secondly, captured variables can live beyond their static scope, if they have been captured, and can thus not always be allocated on the stack. These two aspects complicate the logic.

Just as C♯ hides these implementation details from the programmer, so should the program logic and in particular one should be able to reason about captured and escaped variables as if they were on the stack. Our key aims for reasoning about delegates and captured/escaped variables are thus that (1) Hoare's assignment rule should remain valid; and (2) reasoning should be standard for programs that do not use these complicated features.

Simply throwing all variables onto the heap would violate both (1) and (2). Likewise, treating variables as resource [5, 17] would also complicate proofs of programs that do not take advantage of delegates with captured variables. Instead, we will use an operational semantics where local variables do reside on the stack and extend the assertion logic with new assertions for reasoning about the location and value of stack variables.

For escaped variables we need to be able to assert the existence of an escaped variable on the stack and reason about its value, and for a delegate with captured variables we need to be able to reason about the captured variables' current values as well as their value at the call- and exit-site of the delegate. To accomplish this, we introduce a new assertion, `lookup L as x in P`. Here `L` is a term denoting a location on the stack and `lookup` simply binds the contents of this stack location to `x` in the assertion `P`. Of course, we need to ensure that this does not introduce any aliasing in reasoning about the stack, as this would invalidate Hoare's assignment rule.

Our resulting system has the following properties:

1. Programs that use generics (but do not use delegates) can be verified using higher-order separation logic reasoning.
2. Programs that use named delegates can be verified using nested triples [19].
3. Programs that use anonymous delegates with escaping local variables can be verified using `lookup` to refer to escaped variables.

In summary, the key contributions of the paper are:

- Application of higher-order separation logic, in particular quantification over predicates in the assertion logic, to allow reasoning about programs that use generics.
- The first logic to deal with C♯ 2.0 style delegates, involving anonymous methods and variable capture.
- Illustration of the utility of our logic with a series of examples.
- Formal semantics and soundness of our logic.

We stress that the soundness of the logic is non-trivial; indeed, we develop a new model of separation logic in order to reason about delegates that can capture mutable variables.

In this paper, we do not address inheritance. However, we believe this is an orthogonal issue that can be dealt with in the same way as Parkinson and Bierman [16, 15].

The rest of the paper is structured as follows. We begin, in §2, by showing how a higher-order separation logic can be used to reason about object-oriented programs using generics. Then, in §3, we carefully explain the intricacies of C♯ delegates and how they can capture variables, and extend the logic to reason about delegates. In §4, we present a short case study using the logic. We then, §5, rigorously define the formal semantics of the logic and demonstrate its soundness. Finally, we close by discussing related work, §6 and conclusions and future work, §7.

See [20] for an extended technical appendix to the present paper with more details and proofs.

2 Generics

Generics allow for greater code reuse, by allowing the programmer to abstract over types. In this section we illustrate how to reason about generic methods using higher-order features of the logic.

Consider a simple example of a Node class, that stores items of a generic type:

```
public class Node⟨X⟩ {
  Node⟨X⟩ next;
  X item;
}
```

Here we see the Node class takes a type parameter X that is the type of the elements it stores.

Typically, in reasoning about Java or C♯ programs the representation of classes in separation logic is done through predicates. Generics extends the notion of a class to enable it to take other classes as arguments, hence we must similarly extend the logical representation to predicates that can take predicates as arguments. Thus we leave first-order separation logic and move to higher-order separation logic.

Our assertion logic is an intuitionistic higher-order logic over a simply typed term language, derived from [3]. We use ω to range over types, which are generated by the following grammar:

$$\omega ::= \omega \times \omega \mid \omega \to \omega \mid \texttt{Val} \mid \texttt{Class} \mid \texttt{Prop} \mid \texttt{Int}.$$

The set of types is closed under products and function spaces. `Val` is the type of mathematical values; it includes all $C^\sharp$ values and strings, and is closed under formation of pairs, such that mathematical sequences and other mathematical objects can be conveniently represented.[3] The type `Class` represents the set of

[3] We use a single universe `Val` for the universe of mathematical values to avoid also having to quantify over types in the logic and because such a single universe is also used in the jStar tool [7].

all syntactic generic class identifiers as generated by the grammar presented in Section 5. Assertions are terms of type `Prop`.

Terms are typed with the typing judgment

$$\phi; \psi \vdash \mathsf{M} : \omega,$$

where ϕ is a program variable context, and ψ a logical variable context. Throughout the paper we assume the two contexts to be disjoint and that variables are not repeated in either context. (In the extended version [20] of this paper, there is also a context Δ of type variables; but since we do not need that for the examples in this paper we have omitted it, to simplify the notation.)

The usual specification style of separation logic for describing data structures is to define a predicate at the meta-level which describes how a mathematical structure is represented on the heap. Using higher-order logic allows us to define these representation predicates inside the logic [3], and to define them abstractly in terms of representation predicates for abstracted types.

For the generic Node class defined above we can, for instance, define a generic representation predicate of the following type:

$$\vdash \mathit{list} : \mathtt{Val} \times \mathtt{Val} \times (\mathtt{Val} \times \mathtt{Val} \to \mathtt{Prop}) \to \mathtt{Prop}.$$

The predicate is defined as follows:

$$\begin{aligned} \mathit{list}(n, [], \mathsf{P}) &\stackrel{\text{def}}{=} n =_{\mathtt{Val}} \mathtt{null} \\ \mathit{list}(n, v :: xs, \mathsf{P}) &\stackrel{\text{def}}{=} \exists n', \mathsf{x} : \mathtt{Val}.\ n.\mathit{next} \mapsto n' * n.\mathit{item} \mapsto \mathsf{x} \\ &\qquad * \mathsf{P}(\mathsf{x}, v) * \mathit{list}(n', xs, \mathsf{P}). \end{aligned}$$

To simplify notation, we use standard notation for mathematical sequences ($[]$ for the empty sequence, and $v :: xs$ for the sequence with head element v and tail xs) that are officially represented as elements in the type `Val`.

Thus $\mathit{list}(n, xs, \mathsf{P})$ expresses that n is a representation of a list of objects, where each object represents the corresponding value in the sequence xs, as described by representation predicate P. A reference n represents the sequence xs if n is null and xs is the empty sequence, or if n is a reference to an object o such that o's item field is a representation of the head of xs and o's next field is a representation of the tail of the sequence.

We can now give the constructor the following specification:

```
public Node(X item, Node next) { ... }
∀v : Val, ys : Val, P : Val × Val →Prop,
  { P(item, v) ∗ list(next, ys, P) }
  { r. list(r, v::ys, P) }
```

where the r in the post-condition is a binder for the return value. So the constructor takes a reference to an X object, which represents the value v and a reference to a Node⟨X⟩ object, representing the list of values ys and produces a Node⟨X⟩ object which represents the list of values $v :: ys$.

Here is a simple example of how we can specify a generic append operation:

```
class List {
  public static Node⟨X⟩ append⟨X⟩(Node⟨X⟩ front,
                                  Node⟨X⟩ tail) {
    if(front == null)
      return tail;
    else {
      Node⟨X⟩ tmp = append⟨X⟩(front.next, tail);
      front.next = tmp;
      return front;
    }
  } ∀P : Val × Val →Prop, xs, ys : Val.
      { list(front, xs, P) ∗ list(tail, ys, P) }
      { r. list(r, xs@ys, P) }
}
```

The proof is straightforward, and we provide a detailed proof outline in the extended version [20].

Next we consider a client of this class that appends two lists of numbers.

```
{ list(xs,[1,2,3],Int) ∗ list(ys,[4,5,6],Int) }
   zs = List.append⟨Integer⟩(xs,ys);
{ list(zs,[1,2,3,4,5,6],Int) }
```

Here $Int : \texttt{Val} \times \texttt{Val} \to \texttt{Prop}$ is a predicate such that $Int(i, v)$ holds if i points to an Integer object representing the number v.

Thus we see that it is very simple to specify generic programs by being correspondingly generic in the logic, via quantification over predicates. This is as one would hope, given the earlier work on HOSL and HTT mentioned in the introduction. We now turn to the more challenging issue of delegates.

3 Delegates

In this section we recall the semantics of C♯ delegates (Subsection 3.1); show how to reason about methods using delegates via the delegate call rule (Subsection 3.2); and then show how to specify named delegates (Subsection 3.3) and anonymous delegates (Subsection 3.4).

3.1 Understanding C♯ delegates

A C♯ delegate type describes the parameter types and return type of a method, and is thus very similar to an ML function type. An instance of a C♯ delegate type refers to a method with a compatible signature, and is thus very similar to an ML function (in this paper we ignore that a delegate can refer to multiple such methods, all of which get invoked when the delegate is invoked).

We will use the following delegate types in our examples in this paper:

```
public delegate void Action();
public delegate void Action⟨X⟩ (X x);
public delegate Y Func⟨Y⟩ ();
public delegate Y Func⟨X,Y⟩ (X x);
```

The first two are used for delegates that do not return a result, and the last two for delegates that return a result of type Y. The first and third do not take an argument and the second and fourth take an argument of type X. Overloading will resolve which Action or Func type is meant.

Methods can then take delegates as arguments; for instance, here is how one writes an apply method that takes a delegate with no formal parameters as argument and calls it:

```
public void apply(Action f) { f(); }
```

This might look very "functional" and simple, but, of course, the argument delegate may have lots of effects, using local state on both the stack and the heap.

As an example of the kind of programs we are interested in verifying, consider the following imperative fold method:

```
public static void fold⟨X⟩(Node⟨X⟩ lst, Action⟨Node⟨X⟩⟩ f) {
  if(lst != null) { Node⟨X⟩ tmp = lst.next; f(lst); fold(tmp, f); }
}
```

This method takes a list and a delegate as arguments and applies the delegate to each element in the list, from left to right. An important feature of this implementation is that it remembers the value of the next pointer before calling the delegate f. This allows the delegate to update the next pointer of the current node while preserving that fold applies the delegate to all the nodes of the list. No accumulator value is passed explicitly as an argument to the delegate, since the delegate can maintain an accumulator value itself using local state, as in the following example:

```
class Reverse⟨X⟩ {
  Node⟨X⟩ head;
  public void flip(Node⟨X⟩ x) { x.next = head; head = x; }
  public Node⟨X⟩ reverse(Node⟨X⟩ lst)
    { head = null; fold⟨X⟩(lst, flip); return head; }
}
```

This method uses the fold method to reverse a list in-place. It uses the head field to point to the head of the part of the list reversed so far and folds the flip method, which adds a node to the front of head, over the given list, thus ending up with a reversed list. In Section 3.3, we show how to reason about delegates referring to named methods.

Anonymous methods introduce new complications in the form of captured and potentially escaping variables, because they are declared inline and are allowed to refer to local variables from enclosing scopes. To illustrate, consider the following example:

```
public Func⟨int⟩ counter() {
  int x=0;
  return delegate () { return ++x; };
}
```

This method returns a delegate that has captured the l-value (the location) of the local variable x. Calling the returned delegate will return the number of times the delegate has been called. The strange aspect to this code is that x is scoped to the body of the method counter and thus escapes its scope when the delegate is returned.

One can thus also use captured variables to associate local state with a delegate; for instance, one can implement reverse using a captured variable in place of the head field as follows:

```
public static Node⟨X⟩ reverse⟨X⟩(Node⟨X⟩ lst) {
  Node⟨X⟩ head = null;
  fold⟨X⟩(lst, delegate (Node⟨X⟩ x) { x.next = head; head = x; });
  return head;
}
```

C♯ compilers compile such programs by rewriting them into equivalent C♯ programs without the inline delegate or the captured variable, by introducing a new class with a field corresponding to the captured variable and a method corresponding to the inline delegate. So, while head appears to be a local variable on the stack to the programmer, in fact it ends up on the heap, after the rewriting done by the C♯ compiler.

One could verify the rewritten program, without local state on the stack, with a higher-order separation logic with nested Hoare triples. However, this would mean explicitly reasoning about aliasing of captured local variables in the logic, even though there is no aliasing of captured local variables in the original program. Alternatively, one could devise a storage model for the program logic with no stack at all, but only a heap and an environment, so that all mutation would happen in the heap (as for ML-like languages). For imperative languages, such as Java and C♯, such an approach would, however, lead to proof rules that are more complicated to use than the standard Hoare / Separation logic rules. Instead, we define a storage model where all local variables do reside on the stack and a program logic that treats captured variables as normal stack variables. The operational semantics thus never pops values from the stack, to allow for references to escaped variables.

3.2 Reasoning about Methods that use Delegates

To reason about delegates we extend our higher-order assertion logic with nested Hoare triples [19], written

$$\mathsf{M} \mapsto \langle(\bar{\mathsf{u}}).\{\mathsf{P}\}\_\{\mathsf{d}.\mathsf{Q}\}\rangle,$$

for asserting that M denotes a reference to a method satisfying the given specification. The specification includes a context, $\bar{u}$, that specifies the delegate's formal parameters. Variable d binds the return value in Q. When d does not occur in Q we omit d. The typing rule for the new assertion is given below.

$$\frac{\phi;\psi \vdash \mathsf{M} : \mathtt{Val} \qquad \phi;\psi,\bar{u} \vdash \mathsf{P} : \mathtt{Prop} \qquad \phi;\psi,\bar{u},\mathsf{d} : \mathtt{Val} \vdash \mathsf{Q} : \mathtt{Prop}}{\phi;\psi \vdash \mathsf{M} \mapsto \langle(\bar{u}).\{\mathsf{P}\}\_\{\mathsf{d}.\mathsf{Q}\}\rangle : \mathtt{Prop}} \quad (1)$$

As the typing rule shows, we allow P and Q to refer to program variables and logical variables from the context. In the case of program variables, references to $x \in \phi$ in P and Q refer to x's current value on the stack.

Using this nested triple assertion, we can specify apply as follows:

public void apply(Action f) { f(); }
∀P,Q : Prop.{ P ∗ f ↦ ⟨{P}_{Q}⟩ } { Q ∗ f ↦ ⟨{P}_{Q}⟩ }

The specification universally quantifies over the delegate's pre and post-condition, to allow apply to be called with any delegate. For modularity, it is essential that this specification is strong enough for verifying calls with delegates with local state on the heap and/or stack — in the following sections we show that this is indeed the case by demonstrating how to verify calls to apply with named delegates (with local state on the heap) and anonymous delegates (with local state on both the heap and the stack).

The pre and post-condition both contain the delegate assertion f ↦⟨{P}_{Q}⟩. This specifies what the delegate parameter will do. The pre-condition additionally contains the delegate's pre-condition P, which enables the body to call the delegate. The post-condition of apply contains the post-condition of the delegate. We can see the verification outline as

{ P ∗ f ↦ ⟨{P}_{Q}⟩ }
  f()
{ Q ∗ f ↦ ⟨{P}_{Q}⟩ }

To call a delegate we require two things: an assertion about the delegate being called, here f ↦⟨{P}_{Q}⟩, and the pre-condition specified in that assertion, here P. The post-condition of the call is simply the pre-condition with the delegate's pre-condition replaced by the delegate's post-condition, Q.

The apply method did not deal with parameters. We can adapt the apply method for delegates with a single argument as follows

public void apply⟨X⟩(Action⟨X⟩ f, X x) { f(x); }
∀P,Q : Val →Prop.{ P(x) ∗ f ↦ ⟨(v).{P(v)}_{Q(v)}⟩ }
{ Q(x) ∗ f ↦ ⟨(v).{P(v)}_{Q(v)}⟩ }

The logical variables P and Q take a parameter for the argument given to the delegate. In the pre-condition of apply, we have P(x), which is the pre-condition of the delegate instantiated with the argument with which it will be called. We give an outline of the verification of the body below.

$\{\ \mathsf{P(x)} * \mathsf{f} \mapsto \langle(\mathsf{v}).\{\mathsf{P(v)}\}\_\{\mathsf{Q(v)}\}\rangle\ \}$
$\quad\{\ \mathsf{P(v)[x/v]} * \mathsf{f} \mapsto \langle(\mathsf{v}).\{\mathsf{P(v)}\}\_\{\mathsf{Q(v)}\}\rangle\ \}$
$\quad\mathsf{f(x)}$
$\quad\{\ \mathsf{Q(v)[x/v]} * \mathsf{f} \mapsto \langle(\mathsf{v}).\{\mathsf{P(v)}\}\_\{\mathsf{Q(v)}\}\rangle\ \}$
$\{\ \mathsf{Q(x)} * \mathsf{f} \mapsto \langle(\mathsf{v}).\{\mathsf{P(v)}\}\_\{\mathsf{Q(v)}\}\rangle\ \}$

Here we see that we must provide the pre-condition of the delegate with the argument substituted into the parameter, $\mathsf{P(v)[x/v]}$, and similarly for the post-condition.

Now we have presented informally how we can call delegates; next, we present the formal details. We begin by presenting the overall form of our proof system and then the general proof rule for calling a delegate. Hoare triples in our specification logic take the form:

$$\Gamma; \phi; \psi \vdash \{\mathsf{P}\}\mathsf{s}\{\mathsf{Q}\} \triangleleft \mathsf{M}$$

where Γ is a method context, assigning specifications to methods, and ϕ and ψ are as for assertions;[4] P and Q are assertions, both well-typed in ϕ, ψ; and M is a finite set of variables (an over-approximation of the set of variables in ϕ that s might modify, explained in Section 3.4.)

The proof rule for calling a delegate is shown below.

$$\frac{\mathsf{r} \notin \mathrm{FV}(\mathsf{R}) \qquad \mathsf{R} = \mathsf{y} \mapsto \langle(\bar{\mathsf{u}}).\{\mathsf{P}\}\_\{\mathsf{d.Q}\}\rangle}{\Gamma; \mathsf{r}, \mathsf{y}, \bar{\mathsf{x}}; \psi \vdash \{\mathsf{R} * \mathsf{P}[\bar{\mathsf{x}}/\bar{\mathsf{u}}]\}\mathsf{r} = \mathsf{y}(\bar{\mathsf{x}})\{\mathsf{R} * \mathsf{Q}[\bar{\mathsf{x}}/\bar{\mathsf{u}}, \mathsf{r}/\mathsf{d}]\} \triangleleft \{\mathsf{r}\}}$$

The rule expresses that if y points to a delegate, then we can call the delegate if its precondition P holds, with actual arguments $\bar{\mathsf{x}}$ substituted in for the formal arguments $\bar{\mathsf{u}}$. To simplify the presentation of the proof system, we do not allow delegates to modify their formal arguments (this is formalized in the delegate definition rule below). Furthermore, the rule only applies to delegates that have not captured r, y or $\bar{\mathsf{x}}$ (we will see why in Section 3.4). Hence, neither the formal nor the actual parameters will be modified by the call, so we can also substitute the actual arguments for the formal arguments in the post-condition along with r for the return-value binder d.

3.3 Reasoning about Named Delegates

A delegate referring to a named method can refer to a static method, an open instance method or a closed instance method; in this section we will consider delegates referring to closed instance methods (i.e., a reference to a method and a target object) as we can associate local state with such delegates through the target object. Furthermore, we will only consider delegates referring to exactly one method.

The difficulty in reasoning about delegates is that they can maintain local state, however, since C$^\sharp$ lacks global variables, methods and named delegates can

[4] Again, in the extended version [20] there is also a context, Δ, of type variables.

only maintain local state across calls on the heap. We can reason about methods with local state on the heap by referring to the fields with the local state in the pre- and post-condition of the method. For instance, an increment method implemented with a count field would get the following specification:

```
public class Counter {
  public int count;
  int increment() { return this.count++;}
  ∀n. { this.count ↦ n } { this.count ↦ n + 1 }
}
```

Note, we could abstract the details of the field name using standard techniques for abstraction in separation logic [16]. For simplicity, we leave it concrete here.

We can specify an increment delegate as follows: (the assertion x : Counter holds iff x has dynamic type Counter)

```
x = new Counter();
{ x.count ↦ 0 ∗ x : Counter }
  { x : Counter }
f = x.increment;
  { ∀n. f ↦ ⟨{(this.count ↦ n)}_{(this.count ↦ n + 1)}⟩[x/this]}
{ x.count ↦ 0 ∗ ∀n. f ↦ ⟨{x.count ↦ n}_{x.count ↦ n + 1}⟩ }
```

To create a named delegate, we take the original method specification and replace **this** with the target object x. Since delegate specifications can refer to logical variables from the context, universal quantification in the assertion logic can be used to introduce the logical variables used in the method's specification, here n.

We can duplicate delegate assertions, which enables the logical variables in a delegate specification to be instantiated without losing the general specification. For example, the following holds in the logic:

```
∀n. f ↦ ⟨{x.count ↦ n}_{x.count ↦ n+1}⟩
  ⊢ ∀n. f ↦ ⟨{x.count ↦ n}_{x.count ↦ n+1}⟩
      ∗ ∀n. f ↦ ⟨{x.count ↦ n}_{x.count ↦ n+1}⟩
```

This is due to the storage model defined in Section 5: the heap is split into a field heap and a closure heap and since the closure heap is never modified and only ever extended, it does not have to be separated by separating conjunction.

The proof rule for creating a delegate referring to a closed instance method is given below:

$$\frac{\Gamma(\mathsf{C},\mathsf{m}) = \langle(\bar{\mathsf{u}};\psi).\{\mathsf{P}\}\_\{\mathsf{d.Q}\}\rangle}{\Gamma;\mathsf{x},\mathsf{y};- \vdash \{\mathsf{y}:\mathsf{C}\}\mathsf{x} = \mathsf{y.m}\{\forall\psi.\ \mathsf{x} \mapsto \langle(\bar{\mathsf{u}}).\{\mathsf{P}[\mathsf{y}/\mathtt{this}]\}\_\{\mathsf{d.Q}[\mathsf{y}/\mathtt{this}]\}\rangle\} \triangleleft \{\mathsf{x}\}}$$

where $\Gamma(\mathsf{C},\mathsf{m}) = \langle(\bar{\mathsf{u}};\psi).\{\mathsf{P}\}\_\{\mathsf{d.Q}\}\rangle$ looks up the specification of the method to which the delegate is being created, and ψ are the logical variables for the specification.

To illustrate that the specification given to the previously verified apply method is strong enough for verifying calls with delegates with local state on

the heap, consider calling apply with an increment counter. In a state where the current count is m one can verify a call to apply with delegate f from above by instantiating P and Q with the terms x.count ↦m and x.count ↦m+1.

```
{ x.count ↦ m ∗ ∀n. f ↦ ⟨{x.count ↦ n}_{x.count ↦ n + 1}⟩ }
  { x.count ↦ m ∗ f ↦ ⟨{x.count ↦ m}_{x.count ↦ m+1}⟩ }
    { (P ∗ f ↦ ⟨{P}_{Q}⟩)[x.count ↦ m/P, x.count ↦ m+1/Q] }
  apply(f);
    { (Q ∗ f ↦ ⟨{P}_{Q}⟩)[x.count ↦ m/P, x.count ↦ m+1/Q] }
  { x.count ↦ m+1 ∗ f ↦ ⟨{x.count ↦ m}_{x.count ↦ m+1}⟩ }
{ x.count ↦ m+1 }
```

3.4 Reasoning about Anonymous Delegates

In this section we extend our treatment of delegates to cover anonymous methods with captured variables that can escape their scope. First, we consider reasoning about delegates with captured variables while the captured variables are still in scope, followed by delegates with captured variables that have gone out of scope.

Reasoning about captured variables still in scope Anonymous delegates can have local state, not only on the heap as we saw with named delegates above, but also on the stack. For example, the following code snippet

```
int x = 0;
Func⟨int⟩ f = delegate () { x++; };
apply(f);
assert(x==1);
```

binds a delegate to f, which captures the local stack variable x. The call to apply causes a call to the delegate, which increments the x variable from the enclosing scope. Hence, the assertion will succeed.

Now let us consider the specification of the delegate. Intuitively, the delegate specification should express that for any stack and heap where x has the value n, the delegate is safe to execute and if it terminates, x will have the value $n+1$ on the terminal stack. That is, in the pre- and postcondition of the delegate, we want to refer to the value of x on the stack upon entry to, and exit from, the delegate, respectively.

To express this we extend the logic with two new terms, written

$$\mathsf{L} \overset{s}{\mapsto} \mathsf{N} \quad \text{and} \quad \&\mathsf{x}$$

and a new type Loc of stack locations. The & operator can only be applied to stack variables, and &x denotes the location of x on the stack. The $\mathsf{L} \overset{s}{\mapsto} \mathsf{N}$ term is a points-to predicate for the stack: it asserts that the location denoted by L is allocated on the stack and that this location contains the value denoted by N. The typing rules for $\mathsf{L} \overset{s}{\mapsto} \mathsf{N}$ and & are given below.

$$\frac{\phi;\psi \vdash \mathsf{L} : \mathtt{Loc} \qquad \phi;\psi \vdash \mathsf{N} : \mathtt{Val}}{\phi;\psi \vdash \mathsf{L} \overset{s}{\mapsto} \mathsf{N} : \mathtt{Prop}} \qquad \frac{\mathsf{x} \in \phi}{\phi;\psi \vdash \&\mathsf{x} : \mathtt{Loc}} \tag{2}$$

With these extensions, the specification can be expressed as follows:

$\mathsf{Func}\langle\textbf{int}\rangle\ \mathsf{f} = \textbf{delegate}\ ()\ \{\ \mathsf{x}{+}{+};\ \};$
$\{\ \forall \mathsf{n}.\ \mathsf{f} \mapsto \langle \{\&\mathsf{x} \overset{s}{\mapsto} \mathsf{n}\}\_\{\&\mathsf{x} \overset{s}{\mapsto} \mathsf{n}{+}1\}\rangle\ \}$

Note that if we simply referred directly to x in the the specification then this would refer to x's current value. Whereas, when used in a pre/post-condition of a delegate assertion, the stack points-to predicate refers to the stack upon entry/exit of a call to the delegate. In effect, the stack points-to predicate thus allows us to delay the evaluation of a variable, and the & term allows us to specify the variable to evaluate.

In the above specification, we only need to refer to the value of the captured variable once in the pre- and post-condition and can thus use the stack points-to predicate directly. However, in general, it is more convenient with a term $\texttt{lookup}\ \mathsf{L}\ \texttt{as}\ \mathsf{x}\ \texttt{in}\ \mathsf{P}$ for asserting that the location denoted by L is allocated and that P holds with the contents of this stack location bound to x. We can define such a term in terms of the stack points-to predicate as follows:[5]

$$\texttt{lookup}\ \mathsf{L}\ \texttt{as}\ \mathsf{x}\ \texttt{in}\ \mathsf{P} \overset{\text{def}}{=} \exists \mathsf{x} : \texttt{Val}.\ (\mathsf{L} \overset{s}{\mapsto} \mathsf{x} * \mathsf{P})$$

The creation of an anonymous delegate is given by the following rule:

$$\frac{\bar{\mathsf{u}} \cap \mathsf{M} = \emptyset \qquad \bar{\mathsf{y}}, \bar{\mathsf{u}} \cap \mathrm{FVA}(\mathsf{P}, \mathsf{Q}) = \emptyset \qquad \bar{\mathsf{y}} \subseteq \mathrm{FV}(\mathsf{B}) \qquad \Gamma; \bar{\mathsf{y}}, \bar{\mathsf{u}}; \psi \vdash \{\mathsf{P}\}\mathsf{B}\{\mathsf{d}.\mathsf{Q}\} \triangleleft \mathsf{M}}{\begin{array}{l}\Gamma; \bar{\mathsf{y}}, \mathsf{x}; \psi \vdash \{\texttt{emp}\} \\ \qquad \mathsf{x} = \texttt{delegate}(\bar{\mathsf{G}}\bar{\mathsf{u}})\ \{\mathsf{B}\} \\ \qquad \{\mathsf{x} \mapsto \langle(\bar{\mathsf{u}}).\{\texttt{lookup}\ \&\bar{\mathsf{y}}\ \texttt{as}\ \bar{\mathsf{y}}\ \texttt{in}\ \mathsf{P}\}\_\{\mathsf{d}.\texttt{lookup}\ \&\bar{\mathsf{y}}\ \texttt{as}\ \bar{\mathsf{y}}\ \texttt{in}\ \mathsf{Q}\}\rangle\} \triangleleft \{\mathsf{x}\}\end{array}}$$

where $\mathrm{FVA}(\mathsf{P})$ denotes the set of variables x, such that $\&\mathsf{x}$ occurs in P and B is a method body,

$$\mathsf{B} ::= \mathsf{G}\,\bar{\mathsf{x}}; \mathsf{s}; \texttt{return}\ \mathsf{x};$$

Note that, implicitly $\bar{\mathsf{y}}$ and $\bar{\mathsf{u}}$ are disjoint.

To ensure that substituting the actual arguments for the formal arguments in the delegate call rule correctly captures $\mathrm{C}^\sharp$'s calling convention, we prevent the delegate from modifying or capturing its arguments, hence the premises $\bar{\mathsf{u}} \notin \mathsf{M}$ and $\bar{\mathsf{u}} \notin \mathrm{FVA}(\mathsf{P}, \mathsf{Q})$. The captured variables used in the body from the context, $\bar{\mathsf{y}}$, are bound in the specification using a `lookup` term to allow their evaluation to be postponed until the point of call, and return. The premise $\bar{\mathsf{y}} \notin \mathrm{FVA}(\mathsf{P}, \mathsf{Q})$, which prevents nested delegates from capturing variables from any but the innermost enclosing scope, is just to simplify the rule slightly. The technical report [20] contains a generalized rule without this restriction.

It might seem like $\mathsf{L} \overset{s}{\mapsto} \mathsf{N}$ and & opens up the possibility of aliasing stack variables in the logic, which would invalidate Hoare's assignment rule. This is not the case, as we will explain in Section 5.1 (basically, variables in the program

[5] Whenever we use this abbreviation we will implicitly assume that $x \notin \mathrm{FV}(\mathsf{L})$.

variable context are implicitly separated from stack variables mentioned in the pre- and post-condition of a Hoare triple, outside of a delegate assertion.)

When we come to calling the delegate, we need to be able to evaluate lookup in the current state. We use a structural rule to enable this evaluation:

$$\frac{\Gamma; \phi; \psi \vdash \{\texttt{lookup}\ l\ \texttt{as}\ \mathsf{x}\ \texttt{in}\ \mathsf{P}\}\mathsf{s}\{\texttt{lookup}\ l\ \texttt{as}\ \mathsf{x}\ \texttt{in}\ \mathsf{Q}\} \lhd \mathsf{M}}{\Gamma; \phi, \mathsf{x}; \psi \vdash \{\&\mathsf{x} = l \wedge \mathsf{P}\}\mathsf{s}\{\mathsf{Q}\} \lhd \mathsf{M} \cup \{\mathsf{x}\}} \tag{3}$$

The rule is hiding two different evaluations of x, one at the beginning and one at the end. These may evaluate to different values, so we must ensure that we have not framed any facts about x that could be invalidated, hence this rule includes x in the modified set. Note that the assertions P and Q cannot refer to the address of x, since `lookup` L `as` x `in` P binds x as a *logical* variable in P. Furthermore, by our implicit assumption about the program variable context, this rule only applies if $\mathsf{x} \notin \phi$. The $\mathsf{x} \notin \phi$ restriction is vital to the soundness of the rule, as will be explained in Section 5.2.

Using (3) we can verify the call of the anonymous increment delegate given above:

```
{ x = 0 ∗ f ↦ ⟨{lookup &x as y in y=0}_{lookup &x as y in y=1}⟩ }
{ &x = l ∗ x = 0 ∗ f ↦ ⟨{lookup l as y in y=0}_{lookup l as y in y=1}⟩ }
   { lookup l as x in (x = 0 ∗ f ↦ ⟨{lookup l as y in y=0}_{lookup l as y in y=1}⟩) }
   { (lookup l as x in x = 0) ∗ f ↦ ⟨{lookup l as y in y=0}_{lookup l as y in y=1}⟩ }
      apply(f);
   { (lookup l as x in x = 1) ∗ f ↦ ⟨{lookup l as y in y=0}_{lookup l as y in y=1}⟩ }
   { lookup l as x in (x = 1 ∗ f ↦ ⟨{lookup l as y in y=0}_{lookup l as y in y=1}⟩) }
{ x = 1 ∗ f ↦ ⟨{lookup l as y in y=0}_{lookup l as y in y=1}⟩ }
```

Here we instantiate P and Q from the apply specification as lookup l **as** y **in** y=0, and lookup l **as** y **in** y=1, respectively. This proof outline uses the following key property of the lookup binder to rearrange the assertions:

$$(\texttt{lookup}\ \mathsf{L}\ \texttt{as}\ \mathsf{x}\ \texttt{in}\ \mathsf{P}) * \mathsf{Q} \quad \Leftrightarrow \quad \texttt{lookup}\ \mathsf{L}\ \texttt{as}\ \mathsf{x}\ \texttt{in}\ (\mathsf{P} * \mathsf{Q}) \qquad (\mathsf{x} \notin \mathrm{FV}(\mathsf{Q}))$$

Reasoning about captured variables that have escaped their scope Consider the following counter method from the introduction:

```
public Func⟨int⟩ counter() {
  int x = 0;
  return delegate () { return ++x; };
}
```

The interesting aspect is how to verify the body of counter.

```
{emp}
  int x = 0;
  {x=0}
    Func⟨int⟩ f = delegate () { return ++x; };
  {x=0 ∗ ∀n. S(f,&x,n)) }
```

```
    return f;
  {d. x=0 ∗ ∀n. S(d,&x,n)) }
{???}
```

where

$$\mathsf{S} = \lambda(\mathsf{f},\mathsf{l},\mathsf{n}).\ \mathsf{f} \mapsto \{\ \mathsf{lookup\ l\ as\ x\ in\ x} = \mathsf{n}\ \}\ \{\ \mathsf{d.\ lookup\ l\ as\ x\ in\ x} = \mathsf{d} = \mathsf{n} + 1\ \}$$

The final step of the proof, marked ???, deals with leaving the scope of the variable x. Typically, this would be dealt with by existential quantification. However, since the variable x is still accessible through the delegate and its value may change, existential quantification does not suffice. Instead, we assert that there exists a stack location:

$$\{\mathsf{d.}\ \exists \mathsf{l.\ lookup\ l\ as\ x\ in\ x}{=}0 \ast \forall \mathsf{n.\ S(d,l,n))}\ \}$$

The existential is used to quantify over the location of the variable, and lookup allows us to bind its value. Importantly, lookup asserts that there is a stack location disjoint from those already in scope. Indeed we have the following implication in the logic:

$$(\texttt{lookup}\ \mathsf{L}\ \texttt{as}\ x\ \texttt{in}\ \mathsf{P}) \ast (\texttt{lookup}\ \mathsf{L}'\ \texttt{as}\ x\ \texttt{in}\ \mathsf{Q}) \quad \Rightarrow \quad \mathsf{L} \neq \mathsf{L}' \tag{4}$$

Thus, if we call counter twice, we will reason correctly about the different stack locations for their internal state.

The general case is given by the variable declaration rule:

$$\frac{\Gamma; \phi, \mathsf{z}; \psi \vdash \{\mathsf{P} \wedge \mathsf{z} = \texttt{null}\}\mathsf{B}\{\mathsf{d.Q}\} \triangleleft \mathsf{M}}{\Gamma; \phi; \psi \vdash \{\mathsf{P}\}\mathsf{Gz}; \mathsf{B}\{\mathsf{d}.\exists l : \texttt{Loc}.\ \texttt{lookup}\ l\ \texttt{as}\ \mathsf{z}\ \texttt{in}\ \mathsf{Q}[l/\&\mathsf{z}]\} \triangleleft \mathsf{M} \setminus \mathsf{z}}$$

This degenerates into the standard rule if $\mathsf{z} \notin FV(\mathsf{Q})$.

The frame rule With the introduction of captured variables, it is no longer possible to determine syntactically from a statement which stack variables it might modify. Hence, we have extended our Hoare triples with a finite set M of variables:

$$\Gamma; \phi; \psi \vdash \{\mathsf{P}\}\mathsf{s}\{\mathsf{Q}\} \triangleleft \mathsf{M}$$

to give an explicit over-approximation of which variables in ϕ, s modifies. We restrict M to variables in scope, since we have no way of referring to variables not in scope, nor any useful syntactic approximation of which assertions a given assertion makes about out-of-scope variables.

To ensure that one cannot frame on assertions about potentially modified out-of-scope variables, we let separating conjunction separate out of scope variables (as expressed by (4)). Our language satisfies the standard heap frame property and heap safety monotonicity (used for showing the standard frame rule sound), and it also satisfies a corresponding stack frame property and stack safety monotonicity. We can thus prove the soundness of the following frame rule:

$$\frac{\Gamma; \phi; \psi \vdash \mathsf{R} : \texttt{Prop} \qquad \Gamma; \phi; \psi \vdash \{\mathsf{P}\}\mathsf{s}\{\mathsf{Q}\} \triangleleft \mathsf{M} \qquad \mathsf{M} \cap \mathrm{FVV}(\mathsf{R}) = \emptyset}{\Gamma; \phi; \psi \vdash \{\mathsf{P} \ast \mathsf{R}\}\mathsf{s}\{\mathsf{Q} \ast \mathsf{R}\} \triangleleft \mathsf{M}}$$

since P has to assert the existence of any out-of-scope variables potentially modified by s. Here FVV(R) denotes the set of free value variables, which is defined as follows for variables and &:

$$\mathrm{FVV}(x) = \{x\} \qquad\qquad \mathrm{FVV}(\&x) = \emptyset$$

and like FV(R) for every other case. Hence, one can frame on assertions that refer to the address of a variable; intuitively, since the address cannot be modified, only the contents stored at that address.

4 Case study

In this section we return to the fold/reverse example from Section 3. We show how to specify fold, such that one can verify calls with delegates with local state and use it to verify the second reverse method, which maintains its local state on the stack.

We can give fold the following specification:

```
public void fold⟨X⟩(Node⟨X⟩ lst, Action⟨Node⟨X⟩⟩ f) {...}
  ∀xs,ys : Val. ∀P : Val × Val →Prop.
  ∀Q : Val × Val →Prop.
    { list(lst,xs,P) ∗ Q(lst,ys) ∗ ∀v,ys : Val, ∀n', x : Val,
        f ↦ ⟨(n).{n.next ↦ n' ∗ n.item ↦ x ∗ P(x, v) ∗ Q(n, ys)}_{Q(n', v::ys)}⟩ }
    { Q(null, rev(xs) @ ys) }
```

The code and specification is easiest to understand as an imperative form of a fold-left function, where the accumulator is not passed around explicitly, but rather maintained as local state by the delegate. Then Q(n,ys) is an accumulator predicate, intended to describe the current state, after having folded over ys and where the next node to be visited is n. In the pre-condition of the folding-delegate we explicitly mention n.next and n.item, to allow the delegate to modify these fields. The rev in the post-condition is a function for reversing a list in the logic.

Define Q' as follows:

$$\mathsf{Q'} = \lambda \mathsf{l} : \mathsf{Loc}.\ \lambda \mathsf{n} : \mathsf{Val}.\ \lambda \mathsf{ys} : \mathsf{Val}.\ \mathsf{lookup\ l\ as\ head\ in\ list(head,\ ys,\ P)}$$

Then we can verify the reverse method as follows by instantiating fold's accumulator predicate Q with Q'(l), where l is a logical variable introduced to refer to the location of the captured head variable:

```
public static Node⟨X⟩ reverse⟨X⟩(Node⟨X⟩ lst) {
{ list(lst, xs, P) }
  Node⟨X⟩ head = null;
{ list(lst, xs, P) ∗ head = null }
  Action⟨Node⟨X⟩⟩ f = delegate (Node⟨X⟩ n) {
  { n.next ↦ n' ∗ n.item ↦ x ∗ P(x, v) ∗ list(head, ys, P) }
    n.next = head; head = n;
  { list(head, v::ys, P) }
```

```
  };
{ l = &head ∧ list(lst, xs, P) ∗ head = null ∗ ∀v,ys,x,n' : Val,
      f ↦ ⟨(n).{lookup l as h in n.next ↦ n' ∗ n.item ↦ x ∗ P(x, v) ∗ list(h, ys, P)}
               {lookup l as h in list(h, v::ys, P)}⟩ }
  { lookup l as head in list(lst, xs, P) ∗ head = null ∗ ∀v,ys,x,n' : Val,
      f ↦ ⟨(n).{n.next ↦ n' ∗ n.item ↦ x ∗ P(x, v) ∗ Q'(l, n, ys)}
               {Q'(l, n', v::ys)}⟩ }
  { list(lst, xs, P) ∗ Q'(l, lst, []) ∗ ∀v,ys,x,n' : Val,
      f ↦ ⟨(n).{n.next ↦ n' ∗ n.item ↦ x ∗ P(x, v) ∗ Q'(l, n, ys)}
               {Q'(l, n', v::ys)}⟩ }
  fold⟨X⟩(lst, f);
  { Q'(l, null, rev(xs)@[]) }
  { lookup l as head in list(head, rev(xs)@[], P) }
{ list(head, rev(xs), P) }
  return head;
{ r. list(r, rev(xs), P) }
}
```

In this example, we did not use the first parameter of Q, which is a reference to the next node in the list. This field is primarily useful for delegates which do not modify the node's next field, to express that there is a list segment ending with next-pointer n. Consider, for instance a map method which takes a delegate whose action on a list element is described by a function f in the logic. Then we could take the accumulator predicate to be:

$$\mathsf{Q(n,\ ys)} = \mathsf{list{-}segment(lst,\ n,\ map(f,\ ys),\ P)}$$

where the second argument to list−segment is the value of the next field of the last node in the list, if the list is non-empty.

Filter As a second example we consider a generic filter method for the List class, that takes as argument a delegate which is called on each element of the list to determine whether or not it should be included in the filtered list:

```
class List {
  public static Node⟨X⟩ filter⟨X⟩(Node⟨X⟩ lst, Func⟨X,bool⟩ f) {
    if(lst == null) return null;
    else {
      Node⟨X⟩ tmp = filter(lst.next, f);
      if(f(lst.item)) { lst.next = tmp; return lst; } else return tmp;
    }
  } ∀xs : Val, p : Val →Val, P : Val →Prop,
      { list(lst, xs, P) ∗ ∀v : Val. f ↦ ⟨(x).{P(x,v)}_{r. P(x,v) ∗ r = p(v) }⟩ }
      { r. list(r, filter(xs, p), P) }
}
```

The filter(xs, p) in the post-condition is a mathematical function, which filters the sequence xs using the predicate p. The precondition asserts that f should be

a reference to a delegate, which, when called with a reference x representing the value v, should return $p(v)$.

We can verify a simple client of the filter class that filters the even numbers in a list of integers.

```
{ m : Math ∗ list(x,[1,2,3,4],Int) }
 d = m.isEven;
{ list(x,[1,2,3,4],Int) ∗ ∀v : Val. d ↦ ⟨(i).{Int(i,v)}_{r. Int(i,v) ∗ r=even(v)}⟩ }
 List⟨Integer⟩.filter(x,d);
{ list(x,filter([1,2,3,4],even),Int) }
{ list(x,[2,4],Int) }
```

where we assume the Math class has the following isEven method:

```
public boolean isEven(Integer i) { return i.intValue()%2==0; }
∀v : Val. { Int(i,v) } { r. Int(i, v) ∗ r = even(v) }
```

5 Semantics and Soundness of Proof Rules

In this section we formalize the syntax and operational semantics of the fragment of the C♯ programming language that we consider and the semantics of our logic. For reasons of space, many details and proofs have been omitted, please see [20] for details.

The language that we consider is a subset of C♯ with the most basic object-oriented and imperative features of C♯ and with a restricted syntax to simplify the presentation of the proof system. The syntax of the language is given in Figure 1. In the syntax we use the following metavariables: f ranges over field names, m over method names, C over class names, x, y and z over program variables, and T over type variables. We denote the set of field names by $\mathbb{F}$, the set of method names by $\mathbb{M}$, the set of class names by $\mathbb{C}$, the set of generic class names by $\mathbb{T}$, and the set of variables by $\mathbb{A}_p$. We use an overbar for sequences.

For the operational semantics we assume countable disjoint infinite sets $\mathbb{O}$, $\mathbb{L}_s$, and $\mathbb{L}_h$ of object identifiers, stack locations and heap locations, respectively. We take values to be object identifiers, heap locations, and null. An environment is a finite function from variables to stack locations and a stack is a finite function from stack locations to values:

$$\mathbb{V} \stackrel{\text{def}}{=} \mathbb{O} \uplus \mathbb{L}_h \uplus \{\text{null}\} \qquad \mathbb{E}_p \stackrel{\text{def}}{=} \mathbb{A}_p \stackrel{\text{fin}}{\to} \mathbb{L}_s \qquad \mathbb{S} \stackrel{\text{def}}{=} \mathbb{L}_s \stackrel{\text{fin}}{\to} \mathbb{V}$$

A heap is a tuple (h_v, h_t, h_c) of finite functions, where h_v is a field heap, mapping object identifiers and field names to values; h_t is a type heap, mapping object identifiers to class names; and h_c is a closure heap, mapping heap locations to delegates. A delegate is either an object identifier and a method name or an environment and an anonymous method body.

$$\mathbb{H} \stackrel{\text{def}}{=} (\mathbb{O} \times \mathbb{F} \stackrel{\text{fin}}{\to} \mathbb{V}) \times (\mathbb{O} \stackrel{\text{fin}}{\to} \mathbb{T}) \times (\mathbb{L}_h \stackrel{\text{fin}}{\to} \mathbb{D})$$
$$\mathbb{D} \stackrel{\text{def}}{=} (\mathbb{O} \times \mathbb{M}) \uplus (\mathbb{E}_p \times \mathbb{A}_p^* \times \mathbb{A}_p^* \times \mathbb{P} \times \mathbb{A}_p)$$

G	$::= \mathsf{C}\langle\overline{\mathsf{G}}\rangle \mid \mathsf{T}$	Generic class
L	$::= \texttt{class}\ \mathsf{C}\langle\overline{\mathsf{T}}\rangle : \mathsf{G}\ \{\bar{\mathsf{G}}\,\bar{\mathsf{f}}; \bar{\mathsf{M}}\}$	Class definition
M	$::= \mathsf{G}\,\mathsf{m}(\bar{\mathsf{G}}\,\bar{\mathsf{z}})\ \{\mathsf{B}\}$	Method definition
B	$::= \bar{\mathsf{G}}\,\bar{\mathsf{x}}; \mathsf{s}; \texttt{return}\ \mathsf{x};$	Method body
s	$::=$	Statement
	$\mathsf{x}\ \texttt{=}\ \mathsf{y}$	assignment
	$\mathsf{x}\ \texttt{=}\ \texttt{null}$	initialization
	$\mathsf{x}\ \texttt{=}\ \mathsf{y}\texttt{.}\mathsf{f}$	field access
	$\mathsf{x}\texttt{.}\mathsf{f}\ \texttt{=}\ \mathsf{y}$	field update
	$\mathsf{x}\ \texttt{=}\ \mathsf{y}\texttt{.}\mathsf{m}(\bar{\mathsf{z}})$	method invocation
	$\mathsf{x}\ \texttt{=}\ (\mathsf{G})\mathsf{y}$	cast
	$\texttt{if}\ (\mathsf{x}\ \texttt{==}\ \mathsf{y})\ \{\mathsf{s}_1\}\ \texttt{else}\ \{\mathsf{s}_2\}$	conditional
	$\mathsf{x}\ \texttt{=}\ \texttt{new}\ \mathsf{C}\langle\bar{\mathsf{G}}\rangle()$	object creation
	$\mathsf{x}\ \texttt{=}\ \texttt{delegate}\ (\bar{\mathsf{G}}\,\bar{\mathsf{z}})\ \{\mathsf{B}\}$	anonymous delegate
	$\mathsf{x}\ \texttt{=}\ \mathsf{y}\texttt{.}\mathsf{m}$	named delegate
	$\mathsf{x}\ \texttt{=}\ \mathsf{y}(\bar{\mathsf{z}})$	delegate application
	$\mathsf{s}_1; \mathsf{s}_2$	sequential composition

Fig. 1. Syntax of a simplified $\mathrm{C}^\sharp$

The operational semantics is defined as a big-step semantics with step-indices corresponding to a small-step semantics. It takes configurations, (P, E, S, H, s), consisting of a program P, mapping class and method names to method bodies, an environment E, a stack S, a heap H, and a statement s to err or a terminal stack and heap. The operational semantics of the fragment of the language without generics and delegates is standard and omitted. Instead, we just give the following two rules for constructing and invoking an anonymous delegate. When constructing an anonymous delegate we store the relevant part of the current stack environment along with the delegate; we restore it again when invoking the delegate:

$$\frac{\begin{array}{cc} l \notin \mathrm{Dom}(H_c) & S' = S[E(\mathsf{x}) \mapsto l] \\ E_c = E|_{\mathrm{FV}(\mathsf{s},\mathsf{r})\setminus(\bar{\mathsf{x}}\cup\bar{\mathsf{z}})} & H' = H_c[l \mapsto (E_c, \bar{\mathsf{x}}, \bar{\mathsf{z}}, \mathsf{s}, \mathsf{r})] \end{array}}{(P, E, S, H, \mathsf{x}\ \texttt{=}\ \texttt{delegate}\ (\bar{\mathsf{G}}\bar{\mathsf{x}})\ \{\bar{\mathsf{G}}\bar{\mathsf{z}};\ \mathsf{s}; \texttt{return}\ \mathsf{r}\}) \Downarrow_1 (S', H')}$$

$$\frac{\begin{array}{c} \bar{l}_x, \bar{l}_z \notin \mathrm{Dom}(S) \qquad H_c(S(E(\mathsf{y}))) = (E_c, \bar{\mathsf{x}}, \bar{\mathsf{z}}, \mathsf{s}, \mathsf{r}) \\ (P, E_c[\bar{\mathsf{x}} \mapsto \bar{l}_x, \bar{\mathsf{z}} \mapsto \bar{l}_z], S[\bar{l}_x \mapsto S(E(\bar{\mathsf{u}})), \bar{l}_z \mapsto \mathrm{null}], H, \mathsf{s}) \Downarrow_n (S', H') \end{array}}{(P, E, S, H, \mathsf{x} = \mathsf{y}(\bar{\mathsf{u}})) \Downarrow_{n+1} (S'[E(\mathsf{x}) \mapsto S'(E_c[\bar{\mathsf{x}} \mapsto \bar{l}_x, \bar{\mathsf{z}} \mapsto \bar{l}_z](\mathsf{r}))], H')}$$

We use the notation H_c to refer to the closure heap of H and $H_c[x \mapsto y]$ for the heap H where the closure heap has been extended to map x to y. For each rule we implicitly assume all applications with finite functions to be defined.

A configuration is safe for n steps if it cannot fault in n steps or less:

Definition 1. $(P, E, S, H, \mathtt{s}) : \mathtt{safe}_n \quad \textit{iff} \quad \forall m \leq n.\ (P, E, S, H, \mathtt{s}) \not\Downarrow_m \text{err}$

In addition to the usual safety monotonicity and frame property the language satisfies the following stack monotonicity and stack frame properties:

Lemma 1. *If* $(P, E, S_1, H, \mathtt{s}) : \mathtt{safe}_n$ *and* $S_1 \# S_2$ *then* $(P, E, S_1 \cup S_2, H, \mathtt{s}) : \mathtt{safe}_n$.

Lemma 2. *If* $(P, E, S_1, H, \mathtt{s}) : \mathtt{safe}_n$, $S_1 \# S_2$, *and* $(P, E, S_1 \cup S_2, H, \mathtt{s}) \Downarrow_n (S', H')$ *then there exists an* S_1' *such that* $S' = S_1' \cup S_2$, $S_1' \# S_2$, *and* $(P, E, S_1, H, \mathtt{s}) \Downarrow_n (S_1', H')$.

5.1 Assertion logic

The proof system consists of two layers: an assertion logic for reasoning about program states and a specification logic for reasoning about the effects of programs. In this subsection we formalize the syntax and semantics of the assertion logic. The assertion logic is an intuitionistic higher-order logic over a typed term language. The terms of the language are generated by the following grammar:

$$\begin{array}{rl} \mathsf{P}, \mathsf{Q}, \mathsf{L}, \mathsf{M}, \mathsf{N} ::= & x \mid \lambda x : \omega.\mathsf{M} \mid \mathsf{M}\ \mathsf{N} \mid (\mathsf{M}, \mathsf{N}) \mid \mathtt{fst}\ \mathsf{M} \mid \mathtt{snd}\ \mathsf{M} \\ \mid & \bot \mid \top \mid \mathsf{P} \vee \mathsf{Q} \mid \mathsf{P} \wedge \mathsf{Q} \mid \mathsf{P} \Rightarrow \mathsf{Q} \mid \forall x : \omega.\mathsf{M} \mid \exists x : \omega.\mathsf{M} \mid \mathsf{M} =_\omega \mathsf{N} \\ \mid & \mathsf{P} * \mathsf{Q} \mid \mathsf{P} \mathrel{-\!\!*} \mathsf{Q} \mid \mathtt{emp} \mid \mathsf{M}.\mathsf{f} \mapsto \mathsf{N} \mid \mathsf{M} : \mathsf{N} \mid \mathtt{null} \\ \mid & \mathsf{M} \mapsto \langle (\bar{\mathsf{u}}).\{\mathsf{P}\}\_\{\mathsf{d}.\mathsf{Q}\}\rangle \mid \mathsf{L} \overset{s}{\mapsto} \mathsf{N} \mid \&\mathsf{x} \end{array}$$

where ω ranges over types generated by the following grammar:

$$\omega ::= \omega \to \omega \mid \omega \times \omega \mid \mathtt{Prop} \mid \mathtt{Loc} \mid \mathtt{Val} \mid \mathtt{Int} \mid \mathtt{Class}$$

Assertions are terms of type `Prop`. We will follow the convention of using P and Q for assertions and predicates, L for terms of type `Loc`, and M and N for general terms. The terms are typed with the typing judgment, $\phi; \psi \vdash \mathsf{M} : \omega$, where ϕ is a program variable context, and ψ a logical variable context. The typing rules include all the usual rules of higher-order separation logic [3], extended with the typing rule for delegate assertions, $\overset{s}{\mapsto}$, and the address-of operator (rules (1) and (2)).

To ensure the soundness of Hoare's assignment rule, we need to ensure that $\overset{s}{\mapsto}$ does not introduce any aliasing of stack variables into the logic. Intuitively, we achieve this by only allowing $\overset{s}{\mapsto}$ to assert the existence and value of out-of-scope stack locations. More formally, the meaning of an assertion, $\phi; \psi \vdash \mathsf{P} : \mathtt{Prop}$, is given in terms of the meaning of the program variables in ϕ, i.e., the part of the stack in scope. To reason about out-of-scope stack locations, we interpret assertions as sets of stacks and heaps. Intuitively, this second stack is the "rest of the stack" in the *specification logic*; see the definition of the semantics of triples in Section 5.2. Aliasing is thus avoided, by restricting $\overset{s}{\mapsto}$ to this second stack (see the semantics of $\overset{s}{\mapsto}$ below), which is disjoint from the part of the stack in scope.

We follow [15] in indexing the interpretation of the *specification logic* with step-indices, to allow verification of mutually recursive methods. Furthermore, since we now have specifications in the assertion logic, we also step-index the interpretation of assertions. This idea comes from current work by the second author jointly with Thamsborg and Støvring.

Definition 2. *The types are interpreted as follows:*

$$\llbracket \omega \to \omega' \rrbracket = \llbracket \omega \rrbracket \to \llbracket \omega' \rrbracket \qquad \llbracket \omega \times \omega' \rrbracket = \llbracket \omega \rrbracket \times \llbracket \omega' \rrbracket \qquad \llbracket \mathtt{Class} \rrbracket = \mathbb{T}$$
$$\llbracket \mathtt{Val} \rrbracket = \mathrm{Val} \qquad \llbracket \mathtt{Loc} \rrbracket = \mathbb{L}_s \qquad \llbracket \mathtt{Int} \rrbracket = \mathbf{Z}$$

$$\llbracket \mathtt{Prop} \rrbracket = \{U \in \mathcal{P}^{\uparrow}(\mathsf{N} \times \mathbb{S} \times \mathbb{H}) \mid \forall \pi \in \mathrm{Perm}(\mathbb{A}_p).\ \forall a \in U.\ \pi(a) \in U\}$$

where Val *is the least set satisfying,*

$$\mathrm{Val} \cong \mathbb{V} \uplus \mathrm{Strings} \uplus \mathrm{Val} \times \mathrm{Val},$$

and where the ordering on $\mathsf{N} \times \mathbb{S} \times \mathbb{H}$ *is defined as follows*

$$(n, S, H) \leq (m, S', H') \quad \textit{iff} \quad m \leq n \wedge S \sqsubseteq S' \wedge H \sqsubseteq H'$$

with $\sqsubseteq$ *given by the point-wise extension of the following order on finite functions:*

$$f \leq g \quad \textit{iff} \quad \mathrm{Dom}(f) \subseteq \mathrm{Dom}(g) \wedge \forall \mathsf{x} \in \mathrm{Dom}(f).\ f(x) = g(x).$$

and the permutation action is given by atom-permutation on $\mathbb{A}_p$ *and* $\mathbb{P}$ *and the trivial action on* $\mathsf{N}, \mathbb{O}, \mathbb{F}, \mathbb{C}, \mathbb{L}_s, \mathbb{L}_h$*, and* $\mathbb{V}$*.*

Assertions are thus interpreted as step-indexed subsets of stacks and heaps, downwards-closed in the step-index and upwards closed in extensions of the stack and heap and equivariant under permutations of the closures and environments on the closure heap. Permutations are used to ensure that program and logical variables in the specification logic context are α-convertible (see Rule (α) in [20]).

Lemma 3. *Let* $\mathcal{L} = (\llbracket \mathtt{Prop} \rrbracket, \subseteq)$*, then* $\mathcal{L}$ *is a complete BI-algebra [3], with BI structure* $(I, *, -\!\!*)$ *given by:*

$$\begin{aligned} I &= \emptyset \\ U * V &= \{(n, C \cup C', (h_v \cup h'_v, h_t, h_c)) \mid C \# C' \wedge h_v \# h'_v \ \wedge \\ &\qquad (n, C, (h_v, h_t, h_c)) \in U \wedge (n, C', (h'_v, h_t, h_c)) \in V\} \\ U -\!\!* V &= \bigcup \{W \in \llbracket \mathtt{Prop} \rrbracket \mid W * U \subseteq V\} \end{aligned}$$

for $U, V \in \llbracket \mathtt{Prop} \rrbracket$*.*

Definition 3. *A term-in-context, $\phi;\psi \vdash \mathsf{M} : \omega$, is interpreted as a set-theoretic function:*

$$[\![\phi;\psi \vdash \mathsf{M} : \omega]\!] : [\![\phi]\!] \times [\![\psi]\!] \to [\![\omega]\!]$$

where

$$[\![\phi]\!] = \{(E,S) \in \mathbb{E}_p \times \mathbb{S} \mid E \text{ injective} \wedge \phi = \mathrm{Dom}(E) \wedge \mathrm{Rng}(E) = \mathrm{Dom}(S)\}$$
$$[\![\psi]\!] = \Pi x \in \mathrm{Dom}(\psi).\, [\![\psi(x)]\!].$$

The standard part of the assertion logic is interpreted using a BI-hyperdoctrine over the category **Set**, *induced by the complete BI algebra $\mathcal{L}$ (as in Example 6 in [3]). The interpretation is written out in full in [20].*

The new assertion forms are interpreted as follows:

$$[\![\phi;\psi \vdash \&\mathsf{x} : \mathtt{Loc}]\!]((E,S),\vartheta) = E(\mathsf{x})$$
$$[\![\phi;\psi \vdash \mathsf{L} \overset{s}{\mapsto} \mathsf{N} : \mathtt{Prop}]\!]((E,S);\vartheta) =$$
$$\{(n,C,H) \in \mathsf{N} \times \mathbb{S} \times \mathbb{H} \mid l \in \mathrm{Dom}(C) \wedge C(l) = [\![\phi;\psi \vdash \mathsf{N} : \mathtt{Val}]\!]((E,S);\vartheta)\}$$

where $l = [\![\phi;\psi \vdash \mathsf{L} : \mathtt{Loc}]\!]((E,S),\vartheta)$. For delegate assertions, we just show the interpretation of anonymous delegate assertions:

$$[\![\phi;\psi \vdash \mathsf{R} \mapsto \langle(\bar{\mathsf{u}}).\{\mathsf{P}\}\_\{\mathsf{d}.\mathsf{Q}\}\rangle : \mathtt{Prop}]\!]((E,S),\vartheta) =$$
$$\{(n,\_,(h_v,h_t,h_c)) \in \mathsf{N} \times \mathbb{S} \times \mathbb{H} \mid \exists E_c, \bar{\mathsf{x}}, \bar{\mathsf{z}}, \mathsf{s}, \mathsf{r}.$$
$$(h_c([\![\phi;\psi \vdash \mathsf{R} : \mathtt{Val}]\!]((E,S),\vartheta) = (E_c, \bar{\mathsf{x}}, \bar{\mathsf{z}}, \mathsf{s}, \mathsf{r}) \wedge$$
$$\forall m \le n.\ \forall k \le m.\ \forall C \in \mathbb{S}.\ \forall H \in \mathbb{H}.\ \forall \bar{l}_x, \bar{l}_z\ \in \mathbb{L}_s \setminus (\mathrm{Dom}(C) \cup \mathrm{Rng}(E_c)).\ \forall \bar{v}_x \in \mathbb{V}.$$
$$(m-1,C,H) \in [\![\phi;\psi,\bar{\mathsf{u}} \vdash \mathsf{P} : \mathtt{Prop}]\!]((E,S),\vartheta[\bar{\mathsf{u}} \mapsto \bar{v}_x]) \ \Rightarrow$$
$$(E', C[\bar{l}_x \mapsto \bar{v}_x, \bar{l}_z \mapsto \mathtt{null}], H, \mathsf{s}) : \mathtt{safe}_k \wedge$$
$$(E', C[\bar{l}_x \mapsto \bar{v}_x, \bar{l}_z \mapsto \mathtt{null}], H, \mathsf{s}) \Downarrow_k (C', H') \ \Rightarrow$$
$$(m-k, C' \setminus \bar{l}_x; H') \in [\![\phi;\psi,\bar{\mathsf{u}},\mathsf{d} \vdash \mathsf{Q} : \mathtt{Prop}]\!]((E,S),\vartheta[\bar{\mathsf{u}} \mapsto \bar{v}_x, \mathsf{d} \mapsto C'(E'(\mathsf{r}))])$$

where $E' = E_c[\bar{\mathsf{x}} \mapsto \bar{l}_x, \bar{\mathsf{z}} \mapsto \bar{l}_z]$ and $f \setminus U$ is shorthand for $f|_{\mathrm{Dom}(f)\setminus U}$.

Note that in the interpretation of the delegate assertion, we use the current stack to give meaning to the program variables in ϕ in the pre- and postcondition, but we do not use this stack when running the body. This allows us to refer to the value of captured variables upon entry to and exit from a delegate call using $\overset{s}{\mapsto}$, even for variables currently in scope.

Theorem 1. *The standard part of the higher-order assertion logic is sound.*

Proof. As in [3].

Theorem 2. *The following rules are sound.*

$$\frac{\phi;\psi \vdash \mathsf{M} : \mathtt{Val} \qquad \phi;\psi,\bar{\mathsf{u}} \mid \mathsf{P}' \vdash \mathsf{P} \qquad \phi;\psi,\bar{\mathsf{u}},\mathsf{d} \mid \mathsf{Q} \vdash \mathsf{Q}'}{\phi;\psi \mid \mathsf{M} \mapsto \langle(\bar{\mathsf{u}}).\{\mathsf{P}\}\_\{\mathsf{d}.\mathsf{Q}\}\rangle \vdash \mathsf{M} \mapsto \langle(\bar{\mathsf{u}}).\{\mathsf{P}'\}\_\{\mathsf{d}.\mathsf{Q}'\}\rangle}$$

$$\frac{\phi;\psi \vdash \mathsf{L}, \mathsf{L}' : \mathit{Loc} \qquad \phi;\psi,\mathsf{x} \vdash \mathsf{P}, \mathsf{Q} : \mathit{Prop}}{\phi;\psi \mid \mathit{lookup}\ \mathsf{L}\ \mathit{as}\ \mathsf{x}\ \mathit{in}\ \mathsf{P} * \mathit{lookup}\ \mathsf{L}'\ \mathit{as}\ \mathsf{x}\ \mathit{in}\ \mathsf{Q} \vdash \mathsf{L} \neq \mathsf{L}'}$$

$$\frac{\phi;\psi \vdash \mathsf{L} : \mathit{Loc} \qquad \phi;\psi,\mathsf{x} \vdash \mathsf{P} : \mathit{Prop} \qquad \phi;\psi \vdash \mathsf{Q} : \mathit{Prop}}{\phi;\psi \mid (\mathit{lookup}\ \mathsf{L}\ \mathit{as}\ \mathsf{x}\ \mathit{in}\ \mathsf{P}) * \mathsf{Q} \dashv\vdash \mathit{lookup}\ \mathsf{L}\ \mathit{as}\ \mathsf{x}\ \mathit{in}\ (\mathsf{P} * \mathsf{Q})}$$

The soundness of the second rule follows from the BI-structure on $\mathcal{L}$, and the soundness of the third rule follows from the fact that the meaning of a term is independent of the meaning of logical variables that do not appear free in the term.

5.2 Specification Logic

The specification logic has Hoare triples as its only propositions (it can straightforwardly be extended to a full first-order logic over Hoare triples as atomic propositions). A triple-in-context is interpreted as a downwards-closed set of step-indices:

$$[\![\phi;\psi \vdash \{\mathsf{P}\}\mathsf{s}\{\mathsf{Q}\} \triangleleft \mathsf{M}]\!] : [\![\phi]\!] \times [\![\psi]\!] \to \mathcal{P}^{\downarrow}(\mathsf{N})$$

as follows:

$$\begin{aligned}
&[\![\phi;\psi \vdash \{\mathsf{P}\}\mathsf{s}\{\mathsf{Q}\} \triangleleft \mathsf{M} : \mathtt{Spec}]\!]((E,S);\vartheta) = \{n \in \mathsf{N} \mid \\
&\quad \forall m \leq n.\ \forall k \leq m.\ \forall C \in \mathbb{S}.\ \forall H \in \mathbb{H}. \\
&\qquad (m-1, C, H) \in [\![\phi;\psi \vdash \mathsf{P} : \mathtt{Prop}]\!]((E,S);\vartheta) \wedge C \# S \ \Rightarrow \\
&\qquad\quad (E; C \cup S; H; \mathsf{s}) : \mathtt{safe}_k \wedge \\
&\qquad\quad (E; C \cup S; H; \mathsf{s}) \Downarrow_k (S'; H') \Rightarrow \\
&\qquad\qquad (m-k; S' \setminus E(\phi); H') \in [\![\phi;\psi \vdash \mathsf{Q} : \mathtt{Prop}]\!]((E, S'|_{E(\phi)});\vartheta) \wedge \\
&\qquad\qquad \forall \mathsf{x} \in \phi \setminus \mathsf{M}.\ S(E(\mathsf{x})) = S'(E(\mathsf{x}))\}
\end{aligned}$$

and entailment is interpreted as:

$$\begin{aligned}
[\![\Gamma;\phi;\psi \vdash \{\mathsf{P}\}\mathsf{s}\{\mathsf{Q}\} \triangleleft \mathsf{M}]\!] = \forall n \in \mathsf{N}.\ \forall (E,S) \in [\![\phi]\!].\ \forall \vartheta \in [\![\psi]\!]. \\
n \in [\![\Gamma : \mathtt{MCtx}]\!] \Rightarrow n+1 \in [\![\phi;\psi \vdash \{\mathsf{P}\}\mathsf{s}\{\mathsf{Q}\} \triangleleft \mathsf{M}]\!]((E,S),\vartheta)
\end{aligned}$$

where $n \in [\![\Gamma : \mathtt{MCtx}]\!]$ expresses that all the method specifications in Γ hold for at least n steps.

In the interpretation of Hoare triples above, S corresponds to the part of the stack in scope and C to the "rest of the stack". Since we restrict attention to disjoint S and C, the following specification holds trivially for any ϕ, ψ, s, Q, N, and M,

$$\phi, \mathsf{x}; \psi \vdash \{\&\mathsf{x} \overset{s}{\mapsto} \mathsf{N}\}\mathsf{s}\{\mathsf{Q}\} \triangleleft \mathsf{M}$$

as x's stack location must be in the domain of S and any stack C in the interpretation of the precondition.

Theorem 3. *The specification logic is sound.*

6 Related and Future Work

Our work has built on Parkinson and Bierman's separation logic for object-oriented programs [16, 14, 15]. Their work did not deal with generics or delegates. In the functional programming world, higher-order separation logic [3, 9] and Hoare type theory [11, 18] have both dealt with parametric polymorphism and first-class functions. We have used ideas from those approaches to extend the separation logic for object-oriented programs.

Our logic uses an assertion that contains a delegate specification. Something similar was used in Hoare Type Theory [11] (using types as specifications), for reasoning about ML-like functions. Recently, a foundational study of the interaction between nested triples and higher-order frame rules have been performed by Schwinghammer *et al.* [19], who restricted attention to an idealized language with immutable variables and storable code (rather than functions / delegates). Here our focus has instead been on treating delegates as they appear in a real language like C♯; future work will show whether we can combine our present logic with more advanced higher-order frame rules.

There have been other approaches to reasoning about delegates in object-oriented languages. Müller and Ruskiewicz [10] have a specification for each delegate type, and then every instance of that type must satisfy the specification. This makes it difficult to specify a generic filter method, as you would require a different type for each semantic filter operation. If the delegate passed to the filter is pure, that is it doesn't modify the heap, then it can be used directly in the specification. However, impure methods cannot be used in specifications.

To address this, Nordio *et al.* [12] add to the assertions the ability to abstractly assert that a delegates' pre- and post-conditions, ($d.pre(x)$ and $d.post(x)$) hold for the argument supplied (x). This means they can express the filter specification, by saying every element of the list satisfies the delegate's post-condition, even if the delegate is not pure. However, more complex examples where the implementation must impose a structure on the delegates' specification, such as the fold method, cannot be handled by [12] as it stands. In the fold example, we require one step's post-condition to tie up with the next steps pre-condition, that is, we accumulate a result. With filter, each delegate call is independent of the others. It is unclear how Nordio *et al.*'s work could be extended to the fold example. Nordio *et al.*'s work has been implemented on top of Spec♯. It remains future work to implement our solution.

Both of these works focus on C♯1.0 style delegates, so they do not need to address the issues of anonymous methods, or the subtleties of variable escape in C♯. This is one of the key contributions of this paper.

Yoshida *et al.* [21] have studied such a language with higher-order functions and local state. Their hiding quantifier has some similarity with our `lookup` assertion. They make a distinction between l-values and r-values of a local variable, unlike Hoare logic, so to regain Hoare's assignment axiom, they extend the definition of substitution. It would be interesting to see if our approach to escaping local state could be used in their language.

Variable side-conditions have been a problem for concurrent separation logic [13]. Bornat *et al.*'s [5, 17] variables as resources is another approach to treating variables by separating them using a separating conjunction. We do not believe our new $\overset{s}{\mapsto}$ assertion can be used to reason about shared variable concurrency, because we cannot split knowledge of a variable using permissions [4]. However, for the sequential setting our approach does not complicate reasoning about standard programs. If we had adapted variables as resources [5, 17] to our setting, proofs not involving escaping variables would have to be altered. Now we can simply treat the majority of variables in the same way as Hoare logic.

In future work we plan to combine the present development with the earlier work of Parkinson and Bierman [16, 15] on inheritance. Since the semantics of our logic is based on the semantics used earlier in [16, 15] we are confident that this will be possible. We also plan to extend jStar [7] to allow for semi-automatic verification of programs with generics.

7 Conclusion

We have shown how to apply higher-order separation logic, in particular quantification over predicates, to reason about generics. Earlier work on separation logic for OO languages [15] used restricted forms of quantification over predicates for reasoning about abstraction; here we show how pleasingly straightforward quantification over predicates applies to reasoning about generics. This is as should be expected from the earlier work on HOSL and HTT mentioned above.

Moreover, we have developed the first logic for reasoning about C♯ 2.0 style delegates, involving anonymous methods and capture of variables. To reason about escaping variables, we introduced the assertion

lookup l **as** x **in** P

and the term &x denoting the address of local variable x, with associated proof rules. Soundness was proved by a new model of separation logic in which the truth value of an assertion, relative to a stack corresponding to the topmost stack frame, is a subset of pairs of heaps and stacks (these stacks containing values for escaped variables). We have demonstrated the applicability of the logic via several small, but non trivial, examples.

8 Acknowledgements

We would like to thank our anonymous reviewers for their comments and in particular, for suggesting that `lookup` could be defined in terms of $\overset{s}{\mapsto}$.

References

1. W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY tool. *Software and System Modeling*, 4:32–54, 2005.

2. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec$^\sharp$ programming system: An overview. In *CASSIS*, pages 49–69, 2005.
3. B. Biering, L. Birkedal, and N. Torp-Smith. Bi hyperdoctrines and higher-order separation logic. In *In Proceedings of European Symposium on Programming 2005*, volume 3444 of *Lecture Notes in Computer Science*, pages 233–247, 2005.
4. R. Bornat, C. Calcagno, P. W. O'Hearn, and M. J. Parkinson. Permission accounting in separation logic. In *Proceedings of POPL*, pages 259–270, 2005.
5. R. Bornat, C. Calcagno, and H. Yang. Variables as resources in separation logic. In *Proceedings of MFPS*, pages 125–146, 2005.
6. L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. In *FMICS*, pages 73–89, 2003.
7. D. Distefano and M. J. Parkinson. jstar: towards practical verification for java. In *OOPSLA*, pages 213–226, 2008.
8. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *PLDI*, pages 234–245, 2002.
9. N. R. Krishnaswami, J. Aldrich, L. Birkedal, K. Svendsen, and A. Buisse. Design patterns in separation logic. In *In Proceedings of TLDI'09*, pages 105–116, 2009.
10. P. Müller and J. N. Ruskiewicz. A modular verification methodology for C# delegates. In U. Glässer and J.-R. Abrial, editors, *Rigorous Methods for Software Construction and Analysis*, 2007. To appear.
11. A. Nanevski, A. Ahmed, G. Morrisett, and L. Birkedal. Abstract predicates and mutable adts in hoare type theory. In *In Proceedings of ESOP'08*, pages 189–204, 2007.
12. M. Nordio, C. Calcagno, B. Meyer, and P. Müller. Reasoning about function objects. Technical Report 615, ETH Zurich, 2009.
13. P. W. O'Hearn. Resources, concurrency and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.
14. M. Parkinson. *Local Reasoning for Java.* PhD thesis, University of Cambridge, November 2005.
15. M. Parkinson and G. Bierman. Separation logic, abstraction and inheritance. In *ACM Symposium on Principles of Programming Languages (POPL'08)*. ACM Press, January 2008.
16. M. J. Parkinson and G. M. Bierman. Separation logic and abstraction. In *Proceedings of POPL*, pages 247–258, 2005.
17. M. J. Parkinson, R. Bornat, and C. Calcagno. Variables as resource in Hoare logic. In *Proceedings of LICS*, pages 137–146. IEEE, 2006.
18. R. L. Petersen, L. Birkedal, A. Nanevski, and G. Morrisett. A realizability model for impredicative hoare type theory. In *In Proceedings of ESOP'08*, pages 337–352, 2008.
19. J. Schwinghammer, L. Birkedal, B. Reus, and H. Yang. Nested Hoare triples and frame rules for higher-order store. In *Proceedings of CSL 2009*, Apr. 2009.
20. K. Svendsen, L. Birkedal, and M. Parkinson. Verifying generics and delegates (technical appendix). Technical report, 2009. Available at `http://www.itu.dk/people/kasv/generics-delegates-tr.pdf`.
21. N. Yoshida, K. Honda, and M. Berger. Logical reasoning for higher-order functions with local state. In *FoSSaCS*, pages 361–377, 2007.

Verifying Generics and Delegates
(Technical Appendix)

Kasper Svendsen
IT University of Copenhagen
kasv@itu.dk

Lars Birkedal
IT Univeristy of Copenhagen
birkedal@itu.dk

Matthew Parkinson
University of Cambridge
Matthew.Parkinson@cl.cam.ac.uk

Contents

1 Programming Language

1.1 Syntax

We use the notation $\bar{\mathsf{x}}$ for finite sequences.

G	::=	$\mathsf{C}\langle\overline{\mathsf{G}}\rangle \mid \mathsf{T}$	Generic class
L	::=	$\texttt{class}\ \mathsf{C}\langle\overline{\mathsf{T}}\rangle : \mathsf{G}\ \{\bar{\mathsf{G}}\bar{\mathsf{f}}; \bar{\mathsf{M}}\}$	Class definition
M	::=	$\mathsf{G}\ \mathsf{m}\ (\bar{\mathsf{G}}\bar{\mathsf{u}})\ \{\mathsf{B}\}$	Method definition
B	::=	$\bar{\mathsf{G}}\bar{\mathsf{z}}; \mathsf{s}; \texttt{return}\ \mathsf{r};$	Method body
s	::=		Statement
		$\mathsf{x}\ \texttt{=}\ \mathsf{y}$	assignment
		$\mathsf{x}\ \texttt{=}\ \texttt{null}$	initialization
		$\mathsf{x}\ \texttt{=}\ \mathsf{y}\texttt{.}\mathsf{f}$	field access
		$\mathsf{x}\texttt{.}\mathsf{f}\ \texttt{=}\ \mathsf{y}$	field update
		$\mathsf{x}\ \texttt{=}\ \mathsf{y}\texttt{.}\mathsf{m}(\bar{\mathsf{z}})$	method invocation
		$\mathsf{x}\ \texttt{=}\ (\mathsf{G})\mathsf{y}$	cast
		$\texttt{if}\ (\mathsf{x}\ \texttt{==}\ \mathsf{y})\ \{\mathsf{s}_1\}\ \texttt{else}\ \{\mathsf{s}_2\}$	conditional
		$\mathsf{x}\ \texttt{=}\ \texttt{new}\ \mathsf{C}\langle\bar{\mathsf{G}}\rangle()$	object creation
		$\mathsf{x}\ \texttt{=}\ \texttt{delegate}\ (\bar{\mathsf{G}}\bar{\mathsf{z}})\ \{\mathsf{B}\}$	inline delegate
		$\mathsf{x}\ \texttt{=}\ \texttt{delegate}\ \mathsf{y}\texttt{.}\mathsf{m}$	named delegate
		$\mathsf{x}\ \texttt{=}\ \mathsf{y}(\bar{\mathsf{z}})$	delegate application
		$\mathsf{s}_1; \mathsf{s}_2$	sequential composition

1.2 Operational Semantics

For the operational semantics we assume disjoint countably infinite sets of stack locations, $\mathbb{L}_s$, heap locations, $\mathbb{L}_h$, variables, $\mathbb{A}_p$, type variables $\mathbb{A}_t$, class identifiers, $\mathbb{C}$, method identifiers, $\mathbb{M}$, field identifiers, $\mathbb{F}$, and object identifiers, $\mathbb{O}$.

l	$\in$	$\mathbb{L}_s$	locations
o	$\in$	$\mathbb{O}$	object identifiers
v	$\in$	$\mathbb{V} = \mathbb{L}_h \uplus \mathbb{O} \uplus \{\texttt{null}\}$	values
$\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{r}, \mathsf{u}, \mathsf{x}, \mathsf{y}, \mathsf{z}$	$\in$	$\mathbb{A}_p$	variables
C, D	$\in$	$\mathbb{C}$	class identifiers
f	$\in$	$\mathbb{F}$	field identifiers
m	$\in$	$\mathbb{M}$	method identifiers
T	$\in$	$\mathbb{A}_t$	type variables
		$\mathbb{B} \stackrel{\text{def}}{=} L(\mathsf{s})$	statements
		$\mathbb{T} \stackrel{\text{def}}{=} \{w \in L(\mathsf{G}) \mid \text{FTV}(w) = \emptyset\}$	generic classes
δ	$\in$	$\mathbb{E}_t \stackrel{\text{def}}{=} \mathbb{A}_t \stackrel{\text{fin}}{\to} \mathbb{T}$	type environment
E	$\in$	$\mathbb{E}_p \stackrel{\text{def}}{=} \mathbb{A}_p \stackrel{\text{fin}}{\to} \mathbb{L}_s$	environment
C, S	$\in$	$\mathbb{S} \stackrel{\text{def}}{=} \mathbb{L}_s \stackrel{\text{fin}}{\to} \mathbb{V}$	stack
H	$\in$	$\mathbb{H} \stackrel{\text{def}}{=} (\mathbb{O} \times \mathbb{F} \stackrel{\text{fin}}{\to} \mathbb{V}) \times (\mathbb{O} \stackrel{\text{fin}}{\to} \mathbb{T} \times \mathbb{E}_t) \times (\mathbb{L}_h \stackrel{\text{fin}}{\to} \mathbb{D})$	heap
		$\mathbb{D} \stackrel{\text{def}}{=} (\mathbb{E}_t \times \mathbb{E}_p \times \mathbb{A}_p^* \times \mathbb{A}_p^* \times \mathbb{B} \times \mathbb{A}_p) \uplus (\mathbb{O} \times \mathbb{M})$	delegate
P	$\in$	$\mathbb{P} \stackrel{\text{def}}{=} (\mathbb{C} \stackrel{\text{fin}}{\to} \mathbb{A}_t^* \times \mathbb{F}^*) \times (\mathbb{C} \times \mathbb{M} \stackrel{\text{fin}}{\to} \mathbb{A}_p^* \times \mathbb{A}_p^* \times \mathbb{B} \times \mathbb{A}_p)$	program

where FTV is the set of free type variables. We use the notation A^* for the set of finite lists of A elements. We take the permutation action on to be atom-permutation on $\mathbb{A}_p$, $\mathbb{A}_t$, and $\mathbb{B}$, and

the trivial action on $\mathbb{L}_s$, $\mathbb{O}$, $\mathbb{V}$, $\mathbb{C}$, $\mathbb{F}$, $\mathbb{M}$.

The operational semantics is given as a big-step semantics, with step-indices corresponding to a small-step semantics.

$$\frac{S' = S[E(\mathsf{x}) \mapsto S(E(\mathsf{y}))]}{(P, \delta, E, S, H, \mathsf{x} = \mathsf{y}) \Downarrow_1 (S', H)} \qquad \frac{S' = S[E(\mathsf{x}) \mapsto \mathrm{null}]}{(P, \delta, E, S, H, \mathsf{x} = \mathtt{null}) \Downarrow_1 S', H)}$$

$$\frac{S' = S[E(\mathsf{x}) \mapsto H(S(E(\mathsf{y})), \mathsf{f})]}{(P, \delta, E, S, H, \mathsf{x} = \mathsf{y}.\mathsf{f}) \Downarrow_1 (S', H)} \qquad \frac{H' = H[(S(E(\mathsf{x})), \mathsf{f}) \mapsto S(E(\mathsf{y}))]}{(P, \delta, E, S, H, \mathsf{x}.\mathsf{f} = \mathsf{y}) \Downarrow_1 (S, H')}$$

$$\frac{\begin{array}{c}\bar{l}_x, \bar{l}_z, l_t \notin \mathrm{Dom}(S) \\ H_t(E(S(\mathsf{y}))) = (\mathsf{C}, \delta_m) \qquad P(\mathsf{C}, \mathsf{m}) = (\bar{\mathsf{x}}, \bar{\mathsf{z}}, \mathsf{s}, \mathsf{r}) \qquad E' = [\mathtt{this} \mapsto l_t, \bar{\mathsf{x}} \mapsto \bar{l}_x, \bar{\mathsf{z}} \mapsto \bar{l}_z] \\ (P, \delta_m, E', S[l_t \mapsto S(E(\mathsf{y})), \bar{l}_x \mapsto S(E(\bar{\mathsf{u}})), \bar{l}_z \mapsto \mathtt{null}], H, \mathsf{s}) \Downarrow_n (S', H')\end{array}}{(P, \delta, E, S, H, \mathsf{x} = \mathsf{y}.\mathsf{m}(\bar{\mathsf{u}})) \Downarrow_{n+1} (S'[S(E(\mathsf{x})) \mapsto S'(E'(\mathsf{r}))], H')}$$

$$\frac{H_3(S(E(\mathsf{y}))) \leq [\![\mathsf{G}]\!](\delta) \qquad S' = S[E(\mathsf{x}) \mapsto S(E(\mathsf{y}))]}{(P, \delta, E, S, H, \mathsf{x} = (\mathsf{G})\mathsf{y}) \Downarrow_1 (S', H)}$$

$$\frac{S(E(\mathsf{y})) = \mathrm{null} \qquad S' = S[E(\mathsf{x}) \mapsto \mathrm{null}]}{(P, \delta, E, S, H, \mathsf{x} = (\mathsf{G})\mathsf{y}) \Downarrow_1 (S', H)}$$

$$\frac{S(E(\mathsf{x})) = S(E(\mathsf{y})) \qquad (P, \delta, E, S, \mathsf{s}_1) \Downarrow_n (S', H')}{(P, \delta, E, S, H, \mathtt{if}\ (\mathsf{x} == \mathsf{y})\ \mathtt{then}\ \mathsf{s}_1\ \mathtt{else}\ \mathsf{s}_2) \Downarrow_{n+1} (S', H')}$$

$$\frac{\begin{array}{c}o \notin \mathrm{Dom}(H_t) \qquad H' = H[o \mapsto \mathsf{C}\langle [\![\bar{\mathsf{G}}]\!](\delta)\rangle, (o, \bar{\mathsf{f}}) \mapsto \mathtt{null}] \\ P(\mathsf{C}) = (\bar{\mathsf{T}}, \bar{\mathsf{f}}) \qquad \left|\bar{\mathsf{T}}\right| = \left|\bar{\mathsf{G}}\right| \qquad \mathrm{FTV}(\bar{\mathsf{G}}) = \emptyset \qquad S' = S[E(\mathsf{x}) \mapsto o]\end{array}}{(P, \delta, E, S, H, \mathsf{x} = \textbf{\textit{new}}\ \mathsf{C}\langle\bar{\mathsf{G}}\rangle()) \Downarrow_1 (S', H')}$$

$$\frac{l \notin \mathrm{Dom}(H_c) \qquad S' = S[E(\mathsf{x}) \mapsto l] \qquad E_c = E|_{\mathrm{FV}(\mathsf{s},\mathsf{r}) \setminus (\bar{\mathsf{x}} \cup \bar{\mathsf{z}})} \qquad H' = H[l \mapsto (\delta, E_c, \bar{\mathsf{x}}, \bar{\mathsf{z}}, \mathsf{s}, \mathsf{r})]}{(P, \delta, E, S, H, \mathsf{x}\ \mathtt{=}\ \mathtt{delegate}\ (\bar{\mathsf{G}}\bar{\mathsf{x}})\ \{\bar{\mathsf{G}}\bar{\mathsf{z}};\ \mathsf{s}; \mathtt{return}\ \mathsf{r}\}) \Downarrow_1 (S', H')}$$

$$\frac{l \notin \mathrm{Dom}(H_c) \qquad H' = H_c[l \mapsto (S(E(\mathsf{y})), \mathsf{m})] \qquad S' = S[E(\mathsf{x}) \mapsto l]}{(P, \delta, E, S, H, \mathsf{x}\ \mathtt{=}\ \mathtt{delegate}\ \mathsf{y}.\mathsf{m}) \Downarrow_1 (S', H')}$$

$$\frac{\begin{array}{c}\bar{l}_x, \bar{l}_z, l_t \notin \mathrm{Dom}(S) \qquad H_c(S(E(\mathsf{y}))) = (o, \mathsf{m}) \\ H_t(o) = (\mathsf{C}, \delta_c) \qquad P(\mathsf{C}, \mathsf{m}) = (\bar{\mathsf{x}}, \bar{\mathsf{z}}, \mathsf{s}, \mathsf{r}) \qquad E' = [\mathtt{this} \mapsto l_t, \bar{\mathsf{x}} \mapsto \bar{l}_x, \bar{\mathsf{z}} \mapsto \bar{l}_z] \\ (P, \delta_c, E', S[l_t \mapsto S(E(\mathsf{y})), \bar{l}_x \mapsto S(E(\bar{\mathsf{u}})), \bar{l}_z \mapsto \mathtt{null}], H, \mathsf{s}) \Downarrow_n (S', H')\end{array}}{(P, \delta, E, S, H, \mathsf{x} = \mathsf{y}(\bar{\mathsf{u}})) \Downarrow_{n+1} (S'[S(E(\mathsf{x})) \mapsto S'(E'(\mathsf{r}))], H')}$$

$$\frac{\bar{l}_x, \bar{l}_z \notin \mathrm{Dom}(S) \qquad H_c(S(E(\mathsf{y}))) = (\delta_c, E_c, \bar{\mathsf{x}}, \bar{\mathsf{z}}, \mathsf{s}, \mathsf{r}) \quad (P, \delta_c, E_c[\bar{\mathsf{x}} \mapsto \bar{l}_x, \bar{\mathsf{z}} \mapsto \bar{l}_z], S[\bar{l}_x \mapsto S(E(\bar{\mathsf{u}})), \bar{l}_z \mapsto \mathrm{null}], H, \mathsf{s}) \Downarrow_n (S', H')}{(P, \delta, E, S, H, \mathsf{x} = \mathsf{y}(\bar{\mathsf{u}})) \Downarrow_{n+1} (S'[E(\mathsf{x}) \mapsto S'(E'(\mathsf{r}))], H')}$$

$$\frac{(P, \delta, E, S, H, \mathsf{s}_1) \Downarrow_n (S', H') \qquad (P, \delta, E, S', H', \mathsf{s}_2) \Downarrow_m T}{(P, \delta, E, S, H, \mathsf{s}_1; \mathsf{s}_2) \Downarrow_{n+m} T}$$

$$\frac{(P, \delta, E, S, H, \mathsf{s}_1) \Downarrow_n \mathrm{err}}{(P, \delta, E, S, H, \mathsf{s}_1; \mathsf{s}_2) \Downarrow_n \mathrm{err}}$$

We use the notation $P(\mathsf{C})$ for $\pi_1(P)(\mathsf{C})$ and $P(\mathsf{C}, \mathsf{m})$ for $\pi_2(P)(\mathsf{C}, \mathsf{m})$. Furthermore, we use $[\bar{x} \mapsto \bar{\mathsf{y}}]$ as shorthand for $[x_1 \mapsto y_1, ..., x_n \mapsto y_n]$, with the implicit assumption that the two sequences have the same length. We omit most of the rules for exceptional termination.

1.3 Metatheory

Lemma 1.

$$(P, \delta, E, S, H, \mathsf{s}) \Downarrow_n (S', H') \Rightarrow (P, \delta, \pi(E), S, \pi(H), \pi(\mathsf{s})) \Downarrow_n (S', \pi(H'))$$

and

$$(P, \delta, E, S, H, \mathsf{s}) : \mathit{safe}_n \Rightarrow (P, \delta, \pi(E), S, \pi(H), \pi(\mathsf{s})) : \mathit{safe}_n$$

Lemma 2. *If* $\mathsf{x} \notin FV(\mathsf{s})$ *then*

$$(P, \delta, E, S, H, \mathsf{s}) \Downarrow_n (S', H') \Rightarrow (P, \delta, E \setminus \mathsf{x}, S, H, \mathsf{s}) \Downarrow_n (S', H')$$

Lemma 3 (Safety monotonicity)**.** *If* $S_1 \# S_2$, $H_1 \# H_2$, *and* $(P, \delta, E, S_1, H_1, \mathsf{s}) : \mathit{safe}_n$ *then*

$$(P, \delta, E, S_1 * S_2, H_1 * H_2, \mathsf{s}) : \mathit{safe}_n$$

Lemma 4 (Heap frame property)**.** *If*

$$(P, \delta, E, S, H_1 * H_2, \mathsf{s}) \Downarrow_n (S', H')$$

and $(P, \delta, E, S, H_1, \mathsf{s}) : \mathit{safe}_n$ *then there exists a* H_1' *such that* $H' = H_1' * H_2$ *and*

$$(P, \delta, E, S, H_1, \mathsf{s}) \Downarrow_n (S', H_1')$$

Lemma 5 (Stack frame property)**.** *If*

$$(P, \delta, E, S_1 * S_2, H, \mathsf{s}) \Downarrow_n (S', H')$$

and $(P, \delta, E, S_1, H, \mathsf{s}) : \mathit{safe}_n$ *then there exists an* S_1' *such that* $S' = S_1' * S_2$ *and*

$$(P, \delta, E, S_1, H, \mathsf{s}) \Downarrow_n (S_1', H')$$

2 Assertion Logic

2.1 Syntax

$$
\begin{array}{rcll}
\omega, \omega' & ::= & \omega \to \omega' \mid \omega \times \omega' \mid \texttt{Prop} \mid \texttt{Class} \mid \texttt{Val} \mid \texttt{Int} \mid \texttt{Loc} & \text{Types} \\
\mathsf{M}, \mathsf{N}, \mathsf{L}, \mathsf{P}, \mathsf{Q}, \mathsf{R} & ::= & \mathsf{P} \vee \mathsf{Q} \mid \mathsf{P} \wedge \mathsf{Q} \mid \mathsf{P} \Rightarrow \mathsf{Q} \mid \top \mid \bot \mid \forall \mathsf{x} : \omega . \mathsf{P} \mid \exists \mathsf{x} : \omega . \mathsf{P} & \text{Propositions} \\
 & \mid & \mathsf{P} * \mathsf{Q} \mid \mathsf{P} \mathrel{-\!\!*} \mathsf{Q} \mid \texttt{emp} \mid \mathsf{M}.\mathsf{f} \mapsto \mathsf{N} \mid \mathsf{M} =_{\omega} \mathsf{N} & \\
 & \mid & \mathsf{L} \overset{s}{\mapsto} \mathsf{N} \mid \mathsf{M} \mapsto \langle (\bar{\mathsf{x}}) . \{\mathsf{P}\}\_\{\mathsf{d}.\mathsf{Q}\} \rangle \mid \mathsf{M} : \mathsf{N} & \\
 & \mid & \lambda \mathsf{x} : \omega . \mathsf{M} \mid \mathsf{M}\ \mathsf{N} \mid \mathsf{x} \mid \&\mathsf{x} \mid \texttt{null} & \text{Other terms}
\end{array}
$$

The judgments of the assertion logic are of the forms:

$$\Delta; \phi; \psi \vdash \mathsf{M} : \omega, \quad \Delta; \phi; \psi \vdash \mathsf{M} = \mathsf{N} : \omega, \quad \Delta; \phi; \psi \mid \mathsf{P}_1, ..., \mathsf{P}_n \vdash \mathsf{Q}$$

where Δ is the type variable context, ϕ is the program variable context and ψ is the logic variable context, which are defined as follows:

$$
\begin{array}{rcll}
\Delta & ::= & \Delta, \mathsf{T} \mid \epsilon & \text{type variable context} \\
\phi & ::= & \phi, \mathsf{x} : \texttt{Val} \mid \epsilon & \text{program variable context} \\
\psi & ::= & \psi, \mathsf{a} : \omega \mid \epsilon & \text{logic variable context}
\end{array}
$$

Variables cannot be repeated in the program or logic variable context and the same variable cannot appear in both the program and logic variable context. Since program variables are always of type `Val` we will never write the type.

Definition 1 (Value substitution)**.**

$$
\begin{aligned}
(\mathsf{R} \mapsto \langle (\bar{\mathsf{u}}) . \{\mathsf{P}\}\_\{\mathsf{d}.\mathsf{Q}\} \rangle)[\mathsf{M}/\mathsf{x}] &= \mathsf{R}[\mathsf{M}/\mathsf{x}] \mapsto \langle (\bar{\mathsf{u}}) . \{\mathsf{P}[\mathsf{M}/\mathsf{x}]\}\_\{\mathsf{d}.\mathsf{Q}[\mathsf{M}/\mathsf{x}]\} \rangle \\
(\mathsf{L} \overset{s}{\mapsto} \mathsf{N})[\mathsf{M}/\mathsf{x}] &= \mathsf{L}[\mathsf{M}/\mathsf{x}] \overset{s}{\mapsto} \mathsf{N}[\mathsf{M}/\mathsf{x}] \\
\&\mathsf{y}[\mathsf{M}/\mathsf{x}] &= \&\mathsf{x} \\
\mathsf{y}[\mathsf{M}/\mathsf{x}] &= \begin{cases} \mathsf{M} & \textit{if } \mathsf{y} = \mathsf{x} \\ \mathsf{y} & \textit{otherwise} \end{cases}
\end{aligned}
$$

assuming $\bar{\mathsf{u}}$, d, *and* y *are fresh for* x.

Definition 2 (Location substitution)**.**

$$
\begin{aligned}
(\mathsf{R} \mapsto \langle (\bar{\mathsf{u}}) . \{\mathsf{P}\}\_\{\mathsf{d}.\mathsf{Q}\} \rangle)[\mathsf{M}/\&\mathsf{x}] &= \mathsf{R}[\mathsf{M}/\&\mathsf{x}] \mapsto \langle (\bar{\mathsf{u}}) . \{\mathsf{P}[\mathsf{M}/\&\mathsf{x}]\}\_\{\mathsf{d}.\mathsf{Q}[\mathsf{M}/\&\mathsf{x}]\} \rangle \\
(\mathsf{L} \overset{s}{\mapsto} \mathsf{N})[\mathsf{M}/\&\mathsf{x}] &= \mathsf{L}[\mathsf{M}/\&\mathsf{x}] \overset{s}{\mapsto} \mathsf{N}[\mathsf{M}/\&\mathsf{x}] \\
\&\mathsf{y}[\mathsf{M}/\&\mathsf{x}] &= \begin{cases} \mathsf{M} & \textit{if } \mathsf{x} = \mathsf{y} \\ \&\mathsf{y} & \textit{otherwise} \end{cases} \\
\mathsf{y}[\mathsf{M}/\&\mathsf{x}] &= \mathsf{y}
\end{aligned}
$$

assuming $\bar{\mathsf{u}}$, d, *and* y *are fresh for* x.

Definition 3 (Free variables)**.**

$$
\begin{aligned}
\mathrm{FV}(\mathsf{M} \mapsto \langle (\bar{\mathsf{u}}) . \{\mathsf{P}\}\_\{\mathsf{d}.\mathsf{Q}\} \rangle) &= \mathrm{FV}(\mathsf{M}) \cup (\mathrm{FV}(\mathsf{P}) \cup \mathrm{FV}(\mathsf{Q})) \setminus (\bar{\mathsf{u}} \cup \{\mathsf{d}\}) \\
\mathrm{FV}(\mathsf{L} \overset{s}{\mapsto} \mathsf{N}) &= \mathrm{FV}(\mathsf{L}) \cup \mathrm{FV}(\mathsf{N}) \\
\mathrm{FV}(\&\mathsf{x}) &= \{\mathsf{x}\} \\
\mathrm{FV}(\mathsf{x}) &= \{\mathsf{x}\}
\end{aligned}
$$

Definition 4 (Free value variables)**.**

$$\begin{aligned}\mathrm{FVV}(\mathsf{M} \mapsto \langle(\bar{\mathsf{u}}).\{\mathsf{P}\}\_\{\mathsf{d}.\mathsf{Q}\}\rangle) &= \mathrm{FVV}(\mathsf{M}) \cup (\mathrm{FVV}(\mathsf{P}) \cup \mathrm{FVV}(\mathsf{Q})) \setminus (\bar{\mathsf{u}} \cup \{\mathsf{d}\}) \\ \mathrm{FVV}(\mathsf{L} \overset{s}{\mapsto} \mathsf{N}) &= \mathrm{FV}(\mathsf{L}) \cup \mathrm{FVV}(\mathsf{N}) \\ \mathrm{FVV}(\&\mathsf{x}) &= \emptyset \\ \mathrm{FVV}(\mathsf{x}) &= \{\mathsf{x}\}\end{aligned}$$

Definition 5 (Free location variables)**.**

$$\begin{aligned}\mathrm{FVA}(\mathsf{M} \mapsto \langle(\bar{\mathsf{u}}).\{\mathsf{P}\}\_\{\mathsf{d}.\mathsf{Q}\}\rangle) &= \mathrm{FVA}(\mathsf{M}) \cup (\mathrm{FVA}(\mathsf{P}) \cup \mathrm{FVA}(\mathsf{Q})) \setminus (\bar{\mathsf{u}} \cup \{\mathsf{d}\}) \\ \mathrm{FVA}(\mathsf{L} \overset{s}{\mapsto} \mathsf{N}) &= \mathrm{FVA}(\mathsf{L}) \cup \mathrm{FVA}(\mathsf{N}) \\ \mathrm{FVA}(\&\mathsf{x}) &= \{\mathsf{x}\} \\ \mathrm{FVA}(\mathsf{x}) &= \emptyset\end{aligned}$$

Definition 6 (Lookup)**.**

$$\mathit{lookup}\ \mathsf{L}\ \mathit{as}\ \mathsf{x}\ \mathit{in}\ \mathsf{P} \overset{def}{=} \exists \mathsf{x} : \mathit{Val}.\ (\mathsf{L} \mapsto \mathsf{x} * \mathsf{P})$$

2.2 Typing rules

Well-formed terms $\boxed{\Delta; \phi; \psi \vdash \mathsf{R} : \omega}$

$$\frac{\Delta; \phi; \psi \vdash \mathsf{M} : \mathtt{Val} \qquad \Delta; \phi; \psi, \bar{\mathsf{u}} : \mathtt{Val} \vdash \mathsf{P} : \mathtt{Prop} \qquad \Delta; \phi; \psi, \bar{\mathsf{u}} : \mathtt{Val}, \mathsf{d} : \mathtt{Val} \vdash \mathsf{Q} : \mathtt{Prop}}{\Delta; \phi; \psi \vdash \mathsf{M} \mapsto \langle(\bar{\mathsf{u}}).\{\mathsf{P}\}\_\{\mathsf{d}.\mathsf{Q}\}\rangle : \mathtt{Prop}}$$

$$\frac{\Delta; \phi; \psi, \mathsf{x} : \omega \vdash \mathsf{P} : \mathtt{Prop} \qquad Q \in \{\exists, \forall\}}{\Delta; \phi; \psi \vdash Q\mathsf{x} : \omega.\ \mathsf{P} : \mathtt{Prop}} \qquad \frac{\Delta; \phi; \psi \vdash \mathsf{L} : \mathtt{Loc} \qquad \Delta; \phi; \psi \vdash \mathsf{N} : \mathtt{Val}}{\Delta; \phi; \psi \vdash \mathsf{L} \overset{s}{\mapsto} \mathsf{N} : \mathtt{Prop}}$$

$$\frac{}{\Delta; \phi; \psi \vdash \top : \mathtt{Prop}} \qquad \frac{}{\Delta; \phi; \psi \vdash \bot : \mathtt{Prop}} \qquad \frac{}{\Delta; \phi; \psi \vdash \mathtt{emp} : \mathtt{Prop}}$$

$$\frac{op \in \{\wedge, \vee, *, -\!\!*, \Rightarrow\} \qquad \Delta; \phi; \psi \vdash \mathsf{P} : \mathtt{Prop} \qquad \Delta; \phi; \psi \vdash \mathsf{Q} : \mathtt{Prop}}{\Delta; \phi; \psi \vdash \mathsf{P}\ op\ \mathsf{Q} : \mathtt{Prop}}$$

$$\frac{\Delta; \phi; \psi \vdash \mathsf{M} : \mathtt{Val} \qquad \Delta; \phi; \psi \vdash \mathsf{N} : \mathtt{Class}}{\Delta; \phi; \psi \vdash \mathsf{M} : \mathsf{N} : \mathtt{Prop}} \qquad \frac{\mathsf{x} \in \phi}{\Delta; \phi; \psi \vdash \&\mathsf{x} : \mathtt{Loc}}$$

$$\frac{}{\Delta, \mathsf{T}; \phi; \psi \vdash \mathsf{T} : \mathtt{Class}} \qquad \frac{\Delta; \phi; \psi \vdash \bar{\mathsf{G}} : \mathtt{Class}}{\Delta; \phi; \psi \vdash \mathsf{C}\langle\bar{\mathsf{G}}\rangle : \mathtt{Class}} \qquad \frac{}{\Delta; \phi; \psi \vdash \mathtt{null} : \mathtt{Val}}$$

$$\frac{}{\Delta; \phi, \mathsf{x}; \psi \vdash \mathsf{x} : \mathtt{Val}} \qquad \frac{}{\Delta; \phi; \psi, \mathsf{a} : \omega \vdash \mathsf{a} : \omega}$$

$$\frac{\Delta; \phi; \psi, \mathsf{x} : \omega \vdash \mathsf{M} : \omega'}{\Delta; \phi; \psi \vdash \lambda \mathsf{x} : \omega.\ \mathsf{M} : \omega \to \omega'} \qquad \frac{\Delta; \phi; \psi \vdash \mathsf{M} : \omega \to \omega' \qquad \Delta; \phi; \psi \vdash \mathsf{M} : \omega}{\Delta; \phi; \psi \vdash \mathsf{M}\ \mathsf{N} : \omega'}$$

$$\frac{\Delta; \phi; \psi \vdash \mathsf{M} : \mathtt{Val} \qquad \Delta; \phi; \psi \vdash \mathsf{N} : \mathtt{Val}}{\Delta; \phi; \psi \vdash \mathsf{M}.\mathsf{f} \mapsto \mathsf{N} : \mathtt{Prop}} \qquad \frac{\Delta; \phi; \psi \vdash \mathsf{M} : \omega \qquad \Delta; \phi; \psi \vdash \mathsf{N} : \omega}{\Delta; \phi; \psi \vdash \mathsf{M} =_\omega \mathsf{N} : \mathtt{Prop}}$$

2.3 Proof rules

Standard HO intuitionistic separation logic, extended with the following rules:

$$\frac{\Delta; \phi; \psi \vdash \mathsf{M} : \mathtt{Val} \qquad \Delta; \phi; \psi, \bar{\mathsf{u}} \mid \mathsf{P}' \vdash \mathsf{P} \qquad \Delta; \phi; \psi, \bar{\mathsf{u}}, \mathsf{d} \mid \mathsf{Q} \vdash \mathsf{Q}'}{\Delta; \phi; \psi \mid \mathsf{M} \mapsto \langle (\bar{\mathsf{u}}).\{\mathsf{P}\}\_\{\mathsf{d}.\mathsf{Q}\}\rangle \vdash \mathsf{M} \mapsto \langle (\bar{\mathsf{u}}).\{\mathsf{P}'\}\_\{\mathsf{d}.\mathsf{Q}'\}\rangle}$$

$$\frac{\Delta; \phi; \psi \vdash \mathsf{L}, \mathsf{L}' : \mathtt{Loc} \qquad \Delta; \phi; \psi, \mathsf{x} \vdash \mathsf{P}, \mathsf{Q} : \mathtt{Prop}}{\Delta; \phi; \psi \mid \mathtt{lookup}\ \mathsf{L}\ \mathtt{as}\ \mathsf{x}\ \mathtt{in}\ \mathsf{P} * \mathtt{lookup}\ \mathsf{L}'\ \mathtt{as}\ \mathsf{x}\ \mathtt{in}\ \mathsf{Q} \vdash \mathsf{L} \neq \mathsf{L}'}$$

$$\frac{\Delta; \phi; \psi \vdash \mathsf{L} : \mathtt{Var} \qquad \Delta; \phi; \psi, \mathsf{x} \vdash \mathsf{P} : \mathtt{Prop} \qquad \Delta; \phi; \psi \vdash \mathsf{Q} : \mathtt{Prop}}{\Delta; \phi; \psi \mid (\mathtt{lookup}\ \mathsf{L}\ \mathtt{as}\ \mathsf{x}\ \mathtt{in}\ \mathsf{P}) * \mathsf{Q} \dashv\vdash \mathtt{lookup}\ \mathsf{L}\ \mathtt{as}\ \mathsf{x}\ \mathtt{in}\ (\mathsf{P} * \mathsf{Q})}$$

2.4 Semantics

Types

$$\boxed{[\![\omega]\!] \in \mathrm{Set}}$$

$$\begin{aligned}
[\![\omega \to \omega']\!] &= [\![\omega]\!] \to [\![\omega']\!] \\
[\![\omega \times \omega']\!] &= [\![\omega]\!] \times [\![\omega']\!] \\
[\![\mathtt{Prop}]\!] &= \{U \in \mathcal{P}^{\uparrow}(\mathsf{N} \times \mathbb{S} \times \mathbb{H}) \mid \forall \pi \in \mathrm{Perm}(\mathbb{A}_p).\ \forall a \in U.\ \pi(a) \in U\} \\
[\![\mathtt{Val}]\!] &= \mathrm{Val} \\
[\![\mathtt{Loc}]\!] &= \mathbb{L}_s \\
[\![\mathtt{Class}]\!] &= \mathbb{T} \\
[\![\mathtt{Int}]\!] &= \mathbf{Z}
\end{aligned}$$

where Val is the least set satisfying:

$$\mathrm{Val} \cong \mathbb{V} \uplus \mathrm{Strings} \uplus \mathrm{Val} \times \mathrm{Val}$$

The order on $\mathsf{N} \times \mathbb{S} \times \mathbb{H}$ is given as follows:

$$\begin{aligned}
&(n, S, (h_v, h_t, h_c)) \leq (m, S', (h'_v, h'_t, h'_c)) \text{ iff} \\
&\qquad m \leq n \wedge S \leq S' \wedge h_v \leq h'_v \wedge h_t \leq h'_t \wedge h_c \leq h'_c
\end{aligned}$$

where all the finite functions are ordered as follows:

$$f \leq g \quad \text{iff} \quad \mathrm{Dom}(f) \subseteq \mathrm{Dom}(g) \wedge \forall x \in \mathrm{Dom}(f).\ f(x) = g(x)$$

Contexts $\boxed{[\![\phi]\!], [\![\psi]\!], [\![\Delta]\!] \in \mathrm{Set}}$

$$
\begin{aligned}
S \in [\![\phi]\!] &= \{(E,S) \in \mathbb{E}_p \times \mathbb{S} \mid E \text{ injective} \wedge \mathrm{Dom}(E) = \phi \wedge \mathrm{Rng}(E) = \mathrm{Dom}(S)\} \\
\vartheta \in [\![\psi]\!] &= \Pi(\mathsf{x} : \omega) \in \psi.\ [\![\omega]\!] \\
\delta \in [\![\Delta]\!] &= \{\delta \in \mathbb{E}_t \mid \mathrm{Dom}(\delta) = \Delta\}
\end{aligned}
$$

Propositions and Terms $\boxed{[\![\Delta; \phi; \psi \vdash \mathsf{M} : \omega]\!] : [\![\Delta]\!] \times [\![\phi]\!] \times [\![\psi]\!] \to [\![\omega]\!]}$

$$
\begin{aligned}
[\![\Delta;\phi;\psi \vdash \top : \mathtt{Prop}]\!](\delta;(E,S);\vartheta) &= \mathsf{N} \times \mathbb{S} \times \mathbb{H} \\
[\![\Delta;\phi;\psi \vdash \bot : \mathtt{Prop}]\!](\delta;(E,S);\vartheta) &= \emptyset \\
[\![\Delta;\phi;\psi \vdash \mathtt{emp} : \mathtt{Prop}]\!](\delta;(E,S);\vartheta) &= \mathsf{N} \times \mathbb{S} \times \mathbb{H} \\
[\![\Delta;\phi;\psi \vdash \mathsf{P} \wedge \mathsf{Q} : \mathtt{Prop}]\!](\delta;(E,S);\vartheta) &= \{B \in \mathsf{N} \times \mathbb{S} \times \mathbb{H} \mid \\
&\qquad B \in [\![\Delta;\phi;\psi \vdash \mathsf{P} : \mathtt{Prop}]\!](\delta;(E,S);\vartheta) \wedge \\
&\qquad B \in [\![\Delta;\phi;\psi \vdash \mathsf{Q} : \mathtt{Prop}]\!](\delta;(E,S);\vartheta)\} \\
[\![\Delta;\phi;\psi \vdash \mathsf{P} \vee \mathsf{Q} : \mathtt{Prop}]\!](\delta;(E,S);\vartheta) &= \{B \in \mathsf{N} \times \mathbb{S} \times \mathbb{H} \mid \\
&\qquad B \in [\![\Delta;\phi;\psi \vdash \mathsf{P} : \mathtt{Prop}]\!](\delta;(E,S);\vartheta) \vee \\
&\qquad B \in [\![\Delta;\phi;\psi \vdash \mathsf{Q} : \mathtt{Prop}]\!](\delta;(E,S);\vartheta)\} \\
[\![\Delta;\phi;\psi \vdash \mathsf{P} * \mathsf{Q} : \mathtt{Prop}]\!](\delta;(E,S);\vartheta) &= \{(n,C,(h_v,h_t,h_c)) \in \mathsf{N} \times \mathbb{S} \times \mathbb{H} \mid \exists C_1, C_2, h_1, h_2. \\
&\qquad C_1 \# C_2 \wedge h_1 \# h_2 \wedge C = C_1 \cup C_2 \wedge h_v = h_1 \cup h_2 \wedge \\
&\qquad (n, C_1, (h_1, h_t, h_c)) \in [\![\Delta;\phi;\psi \vdash \mathsf{P} : \mathtt{Prop}]\!](\delta;(E,S);\vartheta) \wedge \\
&\qquad (n, C_2, (h_2, h_t, h_c)) \in [\![\Delta;\phi;\psi \vdash \mathsf{Q} : \mathtt{Prop}]\!](\delta;(E,S);\vartheta)\} \\
[\![\Delta;\phi;\psi \vdash \mathsf{P} \Rightarrow \mathsf{Q} : \mathtt{Prop}]\!](\delta;(E,S);\vartheta) &= \{B \in \mathsf{N} \times \mathbb{S} \times \mathbb{H} \mid \forall B' \geq B. \\
&\qquad B' \in [\![\Delta;\phi;\psi \vdash \mathsf{P} : \mathtt{Prop}]\!](\delta;(E,S);\vartheta) \Rightarrow \\
&\qquad B' \in [\![\Delta;\phi;\psi \vdash \mathsf{Q} : \mathtt{Prop}]\!](\delta;(E,S);\vartheta)\} \\
[\![\Delta;\phi;\psi \vdash \forall \mathsf{a} : \omega.\mathsf{P} : \mathtt{Prop}]\!](\delta;(E,S);\vartheta) &= \bigcap_{v \in [\![\omega]\!]} [\![\Delta;\phi;\psi,\mathsf{a} : \omega \vdash \mathsf{P} : \mathtt{Prop}]\!](\delta;(E,S);\vartheta,\mathsf{a} \mapsto v) \\
[\![\Delta;\phi;\psi \vdash \exists \mathsf{a} : \omega.\mathsf{P} : \mathtt{Prop}]\!](\delta;(E,S);\vartheta) &= \bigcup_{v \in [\![\omega]\!]} [\![\Delta;\phi;\psi,\mathsf{a} : \omega \vdash \mathsf{P} : \mathtt{Prop}]\!](\delta;(E,S);\vartheta,\mathsf{a} \mapsto v) \\
[\![\Delta;\phi,\psi \vdash \mathsf{M} =_\omega \mathsf{N} : \mathtt{Prop}]\!](\delta;(E,S);\vartheta) &= \{B \in \mathsf{N} \times \mathbb{S} \times \mathbb{H} \mid \exists m, n \in [\![\omega]\!]. \\
&\qquad m = [\![\Delta;\phi;\psi \vdash \mathsf{M} : \omega]\!](\delta;(E,S);\vartheta) \wedge \\
&\qquad n = [\![\Delta;\phi;\psi \vdash \mathsf{N} : \omega]\!](\delta;(E,S);\vartheta) \ \wedge m = n\}
\end{aligned}
$$

$$
\begin{aligned}
&[\![\Delta;\phi;\psi,\mathsf{a} : \omega \vdash \mathsf{a} : \omega]\!](\delta;(E,S);\vartheta) = \vartheta(\mathsf{a}) \\
&[\![\Delta;\phi;\psi \vdash \mathsf{M}\ \mathsf{N} : \omega]\!](\delta;(E,S),\vartheta) = \\
&\quad ([\![\Delta;\phi;\psi \vdash \mathsf{M} : \omega' \to \omega]\!](\delta;(E,S);\vartheta))([\![\Delta;\phi;\psi \vdash \mathsf{N} : \omega']\!](\delta;(E,S);\vartheta)) \\
&[\![\Delta;\phi;\psi \vdash \lambda \mathsf{a} : \omega.\mathsf{M} : \omega \to \omega']\!](\delta;(E,S);\vartheta) = \\
&\quad \lambda v : [\![\omega]\!].\ [\![\Delta;\phi;\psi,\mathsf{a} : \omega \vdash \mathsf{M} : \omega']\!](\delta;(E,S);\vartheta[\mathsf{a} \mapsto v])
\end{aligned}
$$

$$
\begin{array}{rcl}
[\![\Delta;\phi;\psi \vdash \mathsf{C}\langle\bar{\mathsf{G}}\rangle : \mathtt{Class}]\!](\delta;(E,S);\vartheta) & = & \mathsf{C}\langle[\![\Delta;\phi;\psi \vdash \bar{\mathsf{G}} : \mathtt{Class}]\!](\delta;(E,S);\vartheta)\rangle \\
[\![\Delta;\phi;\psi \vdash \mathsf{e} : \mathsf{C} : \mathtt{Prop}]\!](\delta;(E,S);\vartheta) & = & \{(n,C,(h_v,h_t,h_c)) \in \mathsf{N}\times\mathbb{S}\times\mathbb{H} \mid \exists o \in \mathbb{O}, \mathsf{C} \in \mathbb{C}, \delta' \in \mathbb{E}_t. \\
 & & \quad in_{\mathbb{O}}(o) = [\![\Delta;\phi;\psi \vdash \mathsf{e} : \mathtt{Val}]\!](\delta;(E,S);\vartheta) \ \wedge \\
 & & \quad h_t(o) = [\![\Delta;\phi;\psi \vdash \mathsf{C} : \mathtt{Class}]\!](\delta;(E,S);\vartheta)\} \\
[\![\Delta;\phi;\psi \vdash \mathsf{M.f} \mapsto \mathsf{N} : \mathtt{Prop}]\!](\delta;(E,S);\vartheta) & = & \{(n,C,(h_v,h_t,h_c)) \in \mathsf{N}\times\mathbb{S}\times\mathbb{H} \mid \exists o \in \mathbb{O}. \\
 & & \quad in_{Oid}(o) = [\![\Delta;\phi;\psi \vdash \mathsf{M} : \mathtt{Val}]\!](\delta;(E,S);\vartheta) \ \wedge \\
 & & \quad h_v(o,\mathsf{f}) = [\![\Delta;\phi;\psi \vdash \mathsf{N} : \mathtt{Val}]\!](\delta;(E,S);\vartheta)\} \\
[\![\Delta,\mathsf{T};\phi;\psi \vdash \mathsf{T} : \mathtt{Class}]\!](\delta;(E,S);\vartheta) & = & \delta(\mathsf{T}) \\
[\![\Delta;\phi,\mathsf{x};\psi \vdash \mathsf{x} : \mathtt{Val}]\!](\delta;(E,S);\vartheta) & = & S(E(\mathsf{x})) \\
[\![\Delta;\phi;\psi \vdash \mathtt{null} : \mathtt{Val}]\!](\delta;(E,S);\vartheta) & = & \mathrm{null} \\
[\![\Delta;\phi,\mathsf{x};\psi \vdash \&\mathsf{x} : \mathtt{Loc}]\!](\delta;(E,S);\vartheta) & = & E(\mathsf{x}) \\
[\![\Delta;\phi;\psi \vdash \mathsf{L} \overset{s}{\mapsto} \mathsf{N} : \mathtt{Prop}]\!](\delta;(E,S);\vartheta) & = & \{(n,C,H) \in \mathsf{N}\times\mathbb{S}\times\mathbb{H} \mid l \in \mathrm{Dom}(C) \wedge \\
 & & \qquad C(l) = [\![\Delta;\phi;\psi \vdash \mathsf{N} : \mathtt{Val}]\!](\delta;(E,S);\vartheta)\}
\end{array}
$$

where $l = [\![\Delta;\phi;\psi \vdash \mathsf{L} : \mathtt{Loc}]\!](\delta;(E,S);\vartheta)$.

$$
\begin{array}{l}
[\![\Delta;\phi;\psi \vdash \mathsf{M} \mapsto \langle(\bar{\mathsf{u}}).\{\mathsf{P}\}\_\{\mathsf{d.Q}\}\rangle : \mathtt{Prop}]\!](\delta,(E,S),\vartheta) = \\
\quad \{(n,\_,(h_v,h_t,h_c)) \in \mathsf{N}\times\mathbb{S}\times\mathbb{H} \mid \exists o,\delta_c,E_c,\bar{\mathsf{x}},\bar{\mathsf{z}},\mathsf{s},\mathsf{r}. \\
\qquad h_c([\![\Delta;\phi;\psi \vdash \mathsf{M} : \mathtt{Val}]\!](\delta,(E,S),\vartheta) = (\delta_c,E_c,\bar{\mathsf{x}},\bar{\mathsf{z}},\mathsf{s},\mathsf{r}) \wedge \\
\qquad \forall m \le n.\ \forall k \le m.\ \forall C \in \mathbb{S}.\ \forall H \in \mathbb{H}.\ \forall \bar{l}_x,\bar{l}_z\ \in \mathrm{Loc} \setminus (\mathrm{Dom}(C) \cup \mathrm{Rng}(E_c)).\ \forall \bar{v}_x \in \mathrm{Val}. \\
\qquad (m-1,C,H) \in [\![\Delta;\phi;\psi,\bar{\mathsf{u}} \vdash \mathsf{P} : \mathtt{Prop}]\!](\delta,(E,S),\vartheta[\bar{\mathsf{u}} \mapsto \bar{v}_x]) \ \Rightarrow \\
\qquad\quad (\delta_c;E'_c,C[\bar{l}_x \mapsto \bar{v}_x,\bar{l}_z \mapsto \mathtt{null}],H,\mathsf{s}) : \mathtt{safe}_k \wedge \\
\qquad\quad (\delta_c;E'_c,C[\bar{l}_x \mapsto \bar{v}_x,\bar{l}_z \mapsto \mathtt{null}],H,\mathsf{s}) \Downarrow_k (C',H') \ \Rightarrow \\
\qquad\qquad (m-k,C' \setminus \bar{l}_x;H') \in [\![\Delta;\phi;\psi,\bar{\mathsf{u}},\mathsf{d} \vdash \mathsf{Q} : \mathtt{Prop}]\!](\delta;(E,S),\vartheta[\bar{\mathsf{u}} \mapsto \bar{v}_x,\mathsf{d} \mapsto C'(E'_c(\mathsf{r}))]) \\
\quad \vee \\
\qquad h_c([\![\Delta;\phi;\psi \vdash \mathsf{M} : \mathtt{Val}]\!](\delta,(E,S),\vartheta) = (o,\mathsf{m}) \wedge h_t(o) = (\mathsf{C},\delta_c) \wedge P(\mathsf{C},\mathsf{m}) = (\bar{\mathsf{x}},\bar{\mathsf{z}},\mathsf{s},\mathsf{r}) \wedge \\
\qquad \forall m \le n.\ \forall k \le m.\ \forall C \in \mathsf{S}.\ \forall H \in \mathbb{H}.\ \forall \bar{l}_x,\bar{l}_z,l_t \in \mathbb{L}_s \setminus \mathrm{Dom}(C).\ \forall \bar{v}_x \in \mathbb{V}. \\
\qquad (m-1,C,H) \in [\![\Delta;\phi;\psi,\bar{\mathsf{u}} \vdash \mathsf{P} : \mathtt{Prop}]\!](\delta,(E,S),\vartheta[\bar{\mathsf{u}} \mapsto \bar{v}_x]) \ \Rightarrow \\
\qquad\quad (\delta_c,E',C[\bar{l}_u \mapsto \bar{v}_x,\bar{l}_z \mapsto \mathtt{null},l_t \mapsto o],H,\mathsf{s}) : \mathtt{safe}_k \wedge \\
\qquad\quad (\delta_c,E',C[\bar{l}_u \mapsto \bar{v}_x,\bar{l}_z \mapsto \mathtt{null},l_t \mapsto o],H,\mathsf{s}) \Downarrow_k (C',H') \ \Rightarrow \\
\qquad\qquad (m-k,C' \setminus \bar{l}_x;H') \in [\![\Delta,\phi;\psi,\bar{\mathsf{u}},\mathsf{d} \vdash \mathsf{Q} : \mathtt{Prop}]\!](\delta,(E,S),\vartheta[\bar{\mathsf{u}} \mapsto \bar{v}_x,\mathsf{d} \mapsto C'(E'(\mathsf{r}))])
\end{array}
$$

where $E'_c = E_c[\bar{\mathsf{x}} \mapsto \bar{l}_x,\bar{\mathsf{z}} \mapsto \bar{l}_z]$ and $E' = [\mathtt{this} \mapsto l_t,\bar{\mathsf{x}} \mapsto \bar{l}_x,\bar{\mathsf{z}} \mapsto \bar{l}_z]$.

Entailment $\boxed{[\![\Delta;\phi;\psi \mid \mathsf{P}_1,...,\mathsf{P}_n \vdash \mathsf{Q}]\!] : 2}$

$$
[\![\Delta;\phi;\psi \mid \mathsf{P}_1,...,\mathsf{P}_n \vdash \mathsf{Q}]\!] = \forall \delta \in [\![\Delta]\!].\ \forall (E,S) \in [\![\phi]\!].\ \forall \vartheta \in [\![\psi]\!].
$$
$$
\left(\bigcap_{1\le i\le n} [\![\Delta;\phi;\psi \vdash \mathsf{P}_i : \mathtt{Prop}]\!](\delta;(E,S);\vartheta)\right) \subseteq [\![\Delta;\phi;\psi \vdash \mathsf{Q} : \mathtt{Prop}]\!](\delta;(E,S);\vartheta)
$$

2.5 Metatheory

Lemma 6. *Let*

$$\mathbb{P} = \{U \in \mathcal{P}^{\uparrow}(\mathsf{N} \times \mathbb{S} \times \mathbb{H}) \mid \forall \pi \in \mathrm{Perm}(\mathbb{A}_p).\ \forall a \in U.\ \pi(a)\}$$

Then $(\mathbb{P}, \subseteq)$ *is a complete BI-algebra, with BI structure* $(I, *, -\!\!*)$ *given by:*

$$\begin{aligned} I &= \emptyset \\ U * V &= \{(n, C \cup C', (h_v \cup h'_v, h_t, h_c)) \mid C \# C' \wedge h_v \# h'_v \wedge \\ &\qquad (n, C, (h_v, h_t, h_c)) \in U \wedge (n, C', (h'_v, h_t, h_c)) \in V\} \\ U -\!\!* V &= \bigcup \{W \in [\![\mathtt{Prop}]\!] \mid W * U \subseteq V\} \end{aligned}$$

for $U, V \in [\![\mathtt{Prop}]\!]$.

Lemma 7 (Alpha renaming)**.** *If* $\Delta; \phi; \psi \vdash \mathsf{P} : \omega$ *then,*

$$\begin{aligned} &\forall \delta \in [\![\Delta]\!].\ \forall (E, S) \in [\![\phi]\!].\ \forall \vartheta \in [\![\psi]\!]. \\ &\quad [\![\Delta; \phi; \psi \vdash \mathsf{P} : \omega]\!](\delta; (E, S); \vartheta) = [\![\Delta; \pi(\phi); \pi(\psi) \vdash \pi(\mathsf{P}) : \omega]\!](\delta; (\pi(E), S); \pi(\vartheta)) \end{aligned}$$

Lemma 8 (Weakening and strengthening)**.** *If* $\Delta; \phi; \psi \vdash \mathsf{P} : \omega$*,* $\mathsf{x} \notin \phi \cup \psi$ *then,*

$$\begin{aligned} &\forall \delta \in [\![\Delta]\!].\ \forall (E, S) \in [\![\phi]\!].\ \forall \vartheta \in [\![\psi]\!].\ \forall l \in \mathbb{L}_s \setminus \mathrm{Rng}(S).\ \forall v_1 \in \mathrm{Val}.\ \forall v_2 \in [\![\omega']\!]. \\ &\quad [\![\Delta; \phi; \psi \vdash \mathsf{P} : \omega]\!](\delta; (E, S); \vartheta) = [\![\Delta; \phi, \mathsf{x}; \psi \vdash \mathsf{P} : \omega]\!](\delta; (E[\mathsf{x} \mapsto l], S[l \mapsto v_1]); \vartheta) \\ &\qquad\qquad = [\![\Delta; \phi; \psi, \mathsf{x} : \omega' \vdash \mathsf{P} : \omega]\!](\delta; (E, S); \vartheta[\mathsf{x} \mapsto v_2]) \end{aligned}$$

Lemma 9. *If* $\Delta; \phi; \psi \vdash \mathsf{P} : \omega$ *then,*

$$\begin{aligned} &\forall \delta \in [\![\Delta]\!].\ \forall E, S_1, S_2.\ \forall \vartheta \in [\![\psi]\!].\ ((E, S_1), (E, S_2) \in [\![\phi]\!] \wedge \forall \mathsf{x} \in \mathrm{FVV}(\mathsf{P}).\ S_1(E(\mathsf{x})) = S_2(E(\mathsf{x}))) \ \Rightarrow \\ &\quad [\![\Delta; \phi; \psi \vdash \mathsf{P} : \omega]\!](\delta; (E, S_1); \vartheta) = [\![\Delta; \phi; \psi \vdash \mathsf{P} : \omega]\!](\delta; (E, S_2); \vartheta) \end{aligned}$$

Lemma 10 (Substitution (Logical variable))**.** *If* $\Delta; \phi; \psi, \mathsf{a} : \omega \vdash \mathsf{P} : \omega'$ *and* $\Delta; \phi; \psi \vdash \mathsf{M} : \omega$ *then,*

$$\begin{aligned} &\forall \delta \in [\![\Delta]\!].\ \forall (E, S) \in [\![\phi]\!].\ \forall \vartheta \in [\![\psi]\!]. \\ &\quad [\![\Delta; \phi; \psi, \mathsf{a} : \omega \vdash \mathsf{P} : \omega']\!](\delta; (E, S); \vartheta, \mathsf{a} \mapsto [\![\Delta; \phi; \psi \vdash \mathsf{M} : \omega]\!](\delta; (E, S); \vartheta)) \\ &\qquad = [\![\Delta; \phi; \psi \vdash \mathsf{P}[\mathsf{M}/\mathsf{a}] : \omega']\!](\delta; (E, S); \vartheta) \end{aligned}$$

Lemma 11 (Substitution (Program variable))**.** *If* $\Delta; \phi, \mathsf{x}; \psi \vdash \mathsf{P} : \omega$ *and* $\Delta; \phi, \mathsf{x}; \psi \vdash \mathsf{M} : \mathtt{Val}$ *then,*

$$\begin{aligned} &\forall \delta \in [\![\Delta]\!].\ \forall (E, S) \in [\![\phi, \mathsf{x}]\!].\ \forall \vartheta \in [\![\psi]\!]. \\ &\quad [\![\Delta; \phi, \mathsf{x}; \psi \vdash \mathsf{P} : \omega]\!](\delta; (E, S[E(\mathsf{x}) \mapsto [\![\Delta; \phi, \mathsf{x}; \psi \vdash \mathsf{M} : \mathtt{Val}]\!](\delta; (E, S); \vartheta)]); \vartheta) \\ &\qquad = [\![\Delta; \phi, \mathsf{x}; \psi \vdash \mathsf{P}[\mathsf{M}/\mathsf{x}] : \omega]\!](\delta; (E, S); \vartheta) \end{aligned}$$

Lemma 12 (Splitting)**.** *If* $\Delta; \phi, \mathsf{x}; \psi \vdash \mathsf{P} : \omega$ *then,*

$$\begin{aligned} &\forall \delta \in [\![\Delta]\!].\ \forall (E, S) \in [\![\phi]\!].\ \forall \vartheta \in [\![\psi]\!].\ \forall l \in \mathbb{L}_s \setminus \mathrm{Dom}(S).\ \forall v \in \mathbb{V}. \\ &\quad [\![\Delta; \phi, \mathsf{x}; \psi \vdash \mathsf{P} : \omega]\!](\delta; (E[x \mapsto l], S[l \mapsto v]); \vartheta) \\ &\qquad = [\![\Delta; \phi; \psi, \mathsf{x} : \mathtt{Val}, \mathsf{y} : \mathtt{Loc} \vdash \mathsf{P}[\mathsf{y}/\&\mathsf{x}] : \omega]\!](\delta; (E, S); \vartheta[\mathsf{x} \mapsto v, \mathsf{y} \mapsto l]) \end{aligned}$$

Corollary 1 (Splitting)**.** *If* $\Delta; \phi; \psi, \mathsf{x} : \mathtt{Val} \vdash \mathsf{P} : \omega$ *then,*

$$\forall \delta \in [\![\Delta]\!].\ \forall (E,S) \in [\![\phi]\!].\ \forall \vartheta \in [\![\psi]\!].\ \forall l \in \mathbb{L}_s \setminus \mathrm{Dom}(S).\ \forall v \in \mathrm{Val}.$$
$$[\![\Delta; \phi, \mathsf{x}; \psi \vdash \mathsf{P} : \omega]\!](\delta; (E[\mathsf{x} \mapsto l], S[l \mapsto v]); \vartheta) = [\![\Delta; \phi; \psi, \mathsf{x} : \mathtt{Val} \vdash \mathsf{P} : \omega]\!](\delta; (E,S); \vartheta[\mathsf{x} \mapsto v])$$

Lemma 13. *If* $\Delta; \phi; \psi \vdash \mathsf{L} : \mathtt{Loc}$ *and* $\Delta; \phi; \psi, \mathsf{x} : \mathtt{Val} \vdash \mathsf{P} : \mathtt{Prop}$ *then,*

$$\begin{array}{l}\forall \delta \in [\![\Delta]\!].\ \forall (E,S) \in [\![\phi]\!].\ \forall \vartheta \in [\![\psi]\!].\\ \quad [\![\Delta; \phi; \psi \vdash \mathtt{lookup}\ \mathsf{L}\ \mathtt{as}\ \mathsf{x}\ \mathtt{in}\ \mathsf{P} : \mathtt{Prop}]\!](\delta; (E,S); \vartheta) =\\ \qquad \{(n, C, H) \in \mathsf{N} \times \mathbb{S} \times \mathbb{H} \mid l \in \mathrm{Dom}(C) \wedge\\ \qquad\quad (n, C \setminus l, H) \in [\![\Delta; \phi; \psi, \mathsf{x} : \mathtt{Val} \vdash \mathsf{P} : \mathtt{Prop}]\!](\delta; (E,S); \vartheta[\mathsf{x} \mapsto C(l)])\}\end{array}$$

where $l = [\![\Delta; \phi; \psi \vdash \mathsf{L} : \mathtt{Loc}]\!](\delta; (E,S); \vartheta)$.

3 Specification Logic

3.1 Syntax

$$\begin{array}{lcll} \mathsf{S}, \mathsf{T} & ::= & \{\mathsf{P}\}\mathsf{s}\{\mathsf{Q}\} \triangleleft \mathsf{M} \mid \{\mathsf{P}\}\mathsf{s}\{\mathsf{d}.\mathsf{Q}\} \triangleleft \mathsf{M} & \text{Specifications} \\ \mathsf{M} & \in & \mathcal{P}_{fin}(\mathbb{A}_p) & \end{array}$$

$$\begin{array}{lcll} MS & ::= & \mathsf{C}\langle\Delta\rangle.\mathsf{m} : \langle(\phi;\psi).\{\mathsf{P}\}\_\{\mathsf{d}.\mathsf{Q}\}\rangle & \text{method specification} \\ \Gamma & ::= & \Gamma, MS \mid \epsilon & \text{program context} \end{array}$$

We use the notation $\Gamma(\mathsf{C})$ to lookup C's type variables and $\Gamma(\mathsf{C}, \mathsf{m})$ to lookup m's specification.

3.2 Typing rules

Well-formed Specifications $\boxed{\Delta; \phi; \psi \vdash \mathsf{S} : \mathtt{Spec}}$

$$\frac{\Delta; \phi; \psi \vdash \mathsf{P} : \mathtt{Prop} \qquad \Delta; \phi; \psi \vdash \mathsf{Q} : \mathtt{Prop} \qquad \mathsf{M} \subseteq \phi}{\Delta; \phi; \psi \vdash \{\mathsf{P}\}\mathsf{s}\{\mathsf{Q}\} \triangleleft \mathsf{M} : \mathtt{Spec}}$$

$$\frac{\Delta; \phi; \psi \vdash \mathsf{P} : \mathtt{Prop} \qquad \Delta; \phi; \psi, \mathsf{d} : \mathtt{Val} \vdash \mathsf{Q} : \mathtt{Prop} \qquad \mathsf{M} \subseteq \phi}{\Delta; \phi; \psi \vdash \{\mathsf{P}\}\mathsf{B}\{\mathsf{d}.\mathsf{Q}\} \triangleleft \mathsf{M} : \mathtt{Spec}}$$

Well-formed Contexts $\boxed{\Gamma : \mathtt{Context}}$

$$\frac{\Delta; -; \psi, \phi, \mathtt{this} \vdash \mathsf{P} : \mathtt{Prop} \qquad \Delta; -; \psi, \phi, \mathtt{this}, \mathsf{d} \vdash \mathsf{Q} : \mathtt{Prop}}{\mathsf{C}\langle\Delta\rangle.\mathsf{m} : \langle(\phi;\psi).\{\mathsf{P}\}\_\{\mathsf{d}.\mathsf{Q}\}\rangle : \mathtt{Context\text{-}Spec}}$$

Note that we do not allow P and Q to refer to the location of `this` or ϕ.

$$\frac{}{\epsilon : \mathtt{Context}} \qquad \frac{\Gamma : \mathtt{Context} \qquad MS : \mathtt{Context\text{-}Spec}}{\Gamma, MS : \mathtt{Context}}$$

3.3 Proof rules

Statements $\boxed{\Gamma; \phi; \psi \vdash \{\mathsf{P}\}\mathtt{s}\{\mathsf{Q}\} \triangleleft \mathsf{M}}$

$$\frac{\Delta; \phi, \mathsf{x}, \mathsf{y}; \psi \vdash \mathsf{P} : \mathtt{Prop}}{\Gamma; \Delta; \phi, \mathsf{x}, \mathsf{y}; \psi \vdash \{[\mathsf{y}/\mathsf{x}]\mathsf{P}\}\mathsf{x}\ \mathtt{=}\ \mathsf{y}\{\mathsf{P}\} \triangleleft \{\mathsf{x}\}}$$

$$\frac{\Delta; \phi, \mathsf{x}; \psi \vdash \mathsf{P} : \mathtt{Prop}}{\Gamma; \Delta; \phi, \mathsf{x}; \psi \vdash \{[\mathtt{null}/\mathsf{x}]\mathsf{P}\}\mathsf{x} = \mathtt{null}\{\mathsf{P}\} \triangleleft \{\mathsf{x}\}}$$

$$\frac{}{\Gamma; \Delta; \phi, \mathsf{x}, \mathsf{y}; \vdash \{\mathsf{x}.\mathsf{f} \mapsto \_\}\mathsf{x}.\mathsf{f} = \mathsf{y}\{\mathsf{x}.\mathsf{f} \mapsto \mathsf{y}\} \triangleleft \emptyset}$$

$$\frac{}{\Gamma; \Delta; \phi, \mathsf{x}, \mathsf{y}; \mathsf{a} \vdash \{\mathsf{y}.\mathsf{f} \mapsto \mathsf{a}\}\mathsf{x} = \mathsf{y}.\mathsf{f}\{\mathsf{y}.\mathsf{f} \mapsto \mathsf{a} \wedge \mathsf{x} = \mathsf{a}\} \triangleleft \{\mathsf{x}\}}$$

$$\frac{fields(\mathsf{C}) = f_1, ..., f_n}{\Gamma; \Delta; \mathsf{x}; - \vdash \{\mathtt{emp}\}\mathsf{x} = \mathtt{new}\ \mathsf{C}\langle\Delta\rangle()\{\mathsf{x} : \mathsf{C}\langle\Delta\rangle \wedge \mathsf{x}.f_1 \mapsto \mathtt{null} * \cdots * \mathsf{x}.f_n \mapsto \mathtt{null}\} \triangleleft \{\mathsf{x}\}}$$

$$\frac{\Gamma(\mathsf{C}) = \Delta \qquad \Gamma(\mathsf{C}, \mathsf{m}) = \langle(\bar{\mathsf{x}}; \psi).\{\mathsf{P}\}\_\{\mathsf{d}.\mathsf{Q}\}\rangle}{\Gamma; \Delta; \mathsf{r}, \mathsf{y}, \bar{\mathsf{u}}; \psi \vdash \{[\bar{\mathsf{u}}/\bar{\mathsf{x}}, \mathsf{y}/\mathtt{this}]\mathsf{P} \wedge \mathsf{y} : \mathsf{C}\langle\Delta\rangle\}\mathsf{r} = \mathsf{y}.\mathsf{m}(\bar{\mathsf{u}})\{[\bar{\mathsf{u}}/\bar{\mathsf{x}}, \mathsf{y}/\mathtt{this}, \mathsf{r}/\mathsf{d}]\mathsf{Q}\} \triangleleft \{\mathsf{r}\}}$$

$$\frac{\Gamma(\mathsf{C}) = \Delta \qquad \Gamma(\mathsf{C}, \mathsf{m}) = \langle(\bar{\mathsf{u}}; \psi).\{\mathsf{P}\}\_\{\mathsf{d}.\mathsf{Q}\}\rangle}{\Gamma; \Delta; \mathsf{x}, \mathsf{y}; - \vdash \{\mathsf{y} : \mathsf{C}\langle\Delta\rangle\}\mathsf{x} = \mathsf{y}.\mathsf{m}\{\forall\psi.\ \mathsf{x} \mapsto \langle(\bar{\mathsf{u}}).\{\mathsf{P}[\mathsf{y}/\mathtt{this}]\}\_\{\mathsf{d}.\mathsf{Q}[\mathsf{y}/\mathtt{this}]\}\rangle\} \triangleleft \{\mathsf{x}\}}$$

$$\frac{\bar{\mathsf{u}} \notin \mathsf{M} \qquad \bar{\mathsf{u}} \notin \mathrm{FVA}(\mathsf{P}, \mathsf{Q}) \qquad \bar{\mathsf{y}} \subseteq \mathrm{FV}(\mathsf{B}) \qquad \Gamma; \Delta; \bar{\mathsf{y}}, \bar{\mathsf{u}}; \psi \vdash \{\mathsf{P}\}\mathsf{B}\{\mathsf{d}.\mathsf{Q}\} \triangleleft \mathsf{M}}{\begin{array}{l}\Gamma; \Delta; \bar{\mathsf{y}}, \mathsf{x}; \psi, \bar{l} \vdash \{\bar{l} = \&\bar{\mathsf{y}}\} \\ \qquad \mathsf{x} = \mathtt{delegate}(\bar{\mathsf{G}}\bar{\mathsf{u}})\ \{\mathsf{B}\} \\ \qquad \{\mathsf{x} \mapsto \langle(\bar{\mathsf{u}}).\{\mathtt{lookup}\ \bar{l}\ \mathtt{as}\ \bar{\mathsf{z}}\ \mathtt{in}\ \mathsf{P}[\bar{l}/\&\bar{\mathsf{y}}][\bar{\mathsf{z}}/\bar{\mathsf{y}}]\}\_\{\mathsf{d}.\mathtt{lookup}\ \bar{l}\ \mathtt{as}\ \bar{\mathsf{z}}\ \mathtt{in}\ \mathsf{Q}[\bar{l}/\&\bar{\mathsf{y}}][\bar{\mathsf{z}}/\bar{\mathsf{y}}]\}\rangle\} \triangleleft \{\mathsf{x}\}\end{array}}\ (\text{anondel})$$

$$\frac{\mathsf{R} = \mathsf{y} \mapsto \langle(\bar{\mathsf{u}}).\{\mathsf{P}\}\_\{\mathsf{d}.\mathsf{Q}\}\rangle \qquad \mathsf{x} \notin \mathrm{FV}(\mathsf{R}) \qquad \mathsf{y} \in \phi}{\Gamma; \Delta; \phi, \bar{\mathsf{x}}, \mathsf{x}; \psi \vdash \{\mathsf{R} * \mathsf{P}[\bar{\mathsf{x}}/\bar{\mathsf{u}}]\}\mathsf{x} = \mathsf{y}(\bar{\mathsf{x}})\{\mathsf{R} * \mathsf{Q}[\bar{\mathsf{x}}/\bar{\mathsf{u}}, \mathsf{r}/\mathsf{d}]\} \triangleleft \{\mathsf{x}\}}\ (\text{delcall})$$

$$\frac{\Gamma; \Delta; \phi, \mathsf{x}, \mathsf{y}; \psi \vdash \{\mathsf{P} \wedge \mathsf{x} = \mathsf{y}\}\mathsf{s}_1\{\mathsf{Q}\} \triangleleft \mathsf{M}_1 \qquad \Gamma; \Delta; \phi, \mathsf{x}, \mathsf{y}; \psi \vdash \{\mathsf{P} \wedge \neg(\mathsf{x} = \mathsf{y})\}\mathsf{s}_2\{\mathsf{Q}\} \triangleleft \mathsf{M}_2}{\Gamma; \Delta; \phi, \mathsf{x}, \mathsf{y}; \psi \vdash \{\mathsf{P}\}\mathtt{if}\ (\mathsf{x}\ \mathtt{==}\ \mathsf{y})\ \{\mathsf{s}_1\}\ \mathtt{else}\ \{\mathsf{s}_2\}\{\mathsf{Q}\} \triangleleft \mathsf{M}_1 \cup \mathsf{M}_2}$$

Structural Rules $\boxed{\Gamma; \Delta; \phi; \psi \vdash \{\mathsf{P}\}\mathsf{s}\{\mathsf{Q}\} \triangleleft \mathsf{M}}$

$$\frac{\Gamma; \Delta; \phi; \psi, \mathsf{a} : \omega \vdash \{\mathsf{P}\}\mathsf{s}\{\mathsf{Q}\} \triangleleft \mathsf{M}_2 \qquad \mathsf{a} \notin FV(\mathsf{P})}{\Gamma; \Delta; \phi; \psi \vdash \{\mathsf{P}\}\mathsf{s}\{\forall \mathsf{a} : \omega.\mathsf{Q}\} \triangleleft \mathsf{M}_2}$$

$$\frac{\Gamma; \Delta; \phi; \psi \vdash \{\mathsf{P}\}\mathsf{s}_1\{\mathsf{Q}\} \triangleleft \mathsf{M}_1 \qquad \Gamma; \Delta; \phi; \psi \vdash \{\mathsf{Q}\}\mathsf{s}_2\{\mathsf{R}\} \triangleleft \mathsf{M}_2}{\Gamma; \Delta; \phi; \psi \vdash \{\mathsf{P}\}\mathsf{s}_1; \mathsf{s}_2\{\mathsf{R}\} \triangleleft \mathsf{M}_1 \cup \mathsf{M}_2} \quad \text{(seq)}$$

$$\frac{\Delta; \phi; \psi \mid \mathsf{P} \vdash \mathsf{P}' \qquad \Gamma; \Delta; \phi; \psi \vdash \{\mathsf{P}'\}\mathsf{s}\{\mathsf{Q}'\} \triangleleft \mathsf{M} \qquad \Delta; \phi; \psi \mid \mathsf{Q}' \vdash \mathsf{Q}}{\Gamma; \Delta; \phi; \psi \vdash \{\mathsf{P}\}\mathsf{s}\{\mathsf{Q}\} \triangleleft \mathsf{M}}$$

$$\frac{\Delta; \phi; \psi \vdash \mathsf{R} : \mathtt{Prop} \qquad \Gamma; \Delta; \phi; \psi \vdash \{\mathsf{P}\}\mathsf{s}\{\mathsf{Q}\} \triangleleft \mathsf{M} \qquad \mathrm{FVV}(\mathsf{R}) \cap \mathsf{M} = \emptyset}{\Gamma; \Delta; \phi; \psi \vdash \{\mathsf{P} * \mathsf{R}\}\mathsf{s}\{\mathsf{Q} * \mathsf{R}\} \triangleleft \mathsf{M}} \quad \text{(frame)}$$

$$\frac{\Gamma; \Delta; \phi; \psi \vdash \{\mathsf{P}\}\mathsf{s}\{\mathsf{Q}\} \triangleleft \mathsf{M}}{\Gamma; \Delta; \phi, \mathsf{x}; \psi \vdash \{\mathsf{P}\}\mathsf{s}\{\mathsf{Q}\} \triangleleft \mathsf{M}}$$

$$\frac{\Gamma; \Delta; \phi; \psi \vdash \{\mathsf{P}\}\mathsf{s}\{\mathsf{Q}\} \triangleleft \mathsf{M}}{\Gamma; \Delta; \pi(\phi); \pi(\psi) \vdash \{\pi(\mathsf{P})\}\pi(\mathsf{s})\{\pi(\mathsf{Q})\} \triangleleft \pi(\mathsf{M})} \quad (\alpha)$$

$$\frac{\Gamma; \Delta; \phi; \psi \vdash \{\mathsf{P}_1\}\mathsf{s}\{\mathsf{Q}_1\} \triangleleft \mathsf{M}_1 \qquad \Gamma; \Delta; \phi; \psi \vdash \{\mathsf{P}_2\}\mathsf{s}\{\mathsf{Q}_2\} \triangleleft \mathsf{M}_2 \qquad op \in \{\wedge, \vee\}}{\Gamma; \Delta; \phi; \psi \vdash \{\mathsf{P}_1 \ op \ \mathsf{P}_2\}\mathsf{s}\{\mathsf{Q}_1 \ op \ \mathsf{Q}_2\} \triangleleft \mathsf{M}_1 \cup \mathsf{M}_2}$$

$$\frac{\Gamma; \Delta; \phi; \psi, \mathsf{x} : \omega \vdash \{\mathsf{P}\}\mathsf{s}\{\mathsf{Q}\} \triangleleft \mathsf{M} \qquad \Delta; \phi; \psi \vdash \mathsf{R} : \omega \qquad \mathrm{FV}(\mathsf{R}) \cap \mathsf{M} = \emptyset}{\Gamma; \Delta; \phi; \psi \vdash \{\mathsf{P}[\mathsf{R}/\mathsf{x}]\}\mathsf{s}\{\mathsf{Q}[\mathsf{R}/\mathsf{x}]\} \triangleleft \mathsf{M}}$$

$$\frac{\Gamma; \Delta; \phi; \psi \vdash \{\mathtt{lookup}\ l\ \mathtt{as}\ \mathsf{x}\ \mathtt{in}\ \mathsf{P}\}\mathsf{s}\{\mathtt{lookup}\ l\ \mathtt{as}\ \mathsf{x}\ \mathtt{in}\ \mathsf{Q}\} \triangleleft \mathsf{M}}{\Gamma; \Delta; \phi, \mathsf{x}; \psi \vdash \{\&\mathsf{x} = l \wedge \mathsf{P}\}\mathsf{s}\{\mathsf{Q}\} \triangleleft \mathsf{M} \cup \{\mathsf{x}\}} \quad \text{(lookup)}$$

Method definitions $\boxed{\Gamma \vdash \mathsf{M} : \Gamma'}$

$$\frac{}{\Gamma; \Delta, \phi, \mathsf{x}; \psi \vdash \{\mathsf{P}[\mathsf{x}/\mathsf{d}]\}\mathtt{return}\ \mathsf{x}; \{\mathsf{d}.\mathsf{P}\} \triangleleft \emptyset}$$

$$\frac{\Gamma; \Delta; \phi; \psi \vdash \{\mathsf{P}\}\mathsf{s}\{\mathsf{Q}\} \triangleleft \mathsf{M}_1 \qquad \Gamma; \Delta; \phi; \psi \vdash \{\mathsf{Q}\}\mathsf{s}; \mathtt{return}\ \mathsf{x}\{\mathsf{d}.\mathsf{R}\} \triangleleft \mathsf{M}_2}{\Gamma; \Delta; \phi; \psi \vdash \{\mathsf{P}\}\mathsf{s}; \mathsf{s}; \mathtt{return}\ \mathsf{x}\{\mathsf{d}.\mathsf{R}\} \triangleleft \mathsf{M}_1 \cup \mathsf{M}_2}$$

$$\frac{\Delta; \phi; \psi \mid \mathsf{P} \vdash \mathsf{P}' \qquad \Gamma; \Delta; \phi; \psi \vdash \{\mathsf{P}'\}\mathsf{B}\{\mathsf{d}.\mathsf{Q}'\} \triangleleft \mathsf{M} \qquad \Delta; \phi; \psi \mid \mathsf{Q}' \vdash \mathsf{Q}}{\Gamma; \Delta; \phi; \psi \vdash \{\mathsf{P}\}\mathsf{B}\{\mathsf{d}.\mathsf{Q}\} \triangleleft \mathsf{M}}$$

$$\frac{\Gamma; \Delta; \phi, \bar{\mathsf{z}}; \psi \vdash \{\mathsf{P} \wedge \bar{\mathsf{z}} = \mathtt{null}\}\mathsf{s}; \mathtt{return}\ \mathsf{x}\{\mathsf{d}.\mathsf{Q}\} \triangleleft \mathsf{M}}{\Gamma; \Delta; \phi; \psi \vdash \{\mathsf{P}\}\bar{\mathsf{G}}\bar{\mathsf{z}}; \mathsf{s}; \mathtt{return}\ \mathsf{x}\{\mathsf{d}.\exists \bar{l} : \mathtt{Var}.\ \mathtt{lookup}\ \bar{l}\ \mathtt{as}\ \bar{\mathsf{z}}\ \mathtt{in}\ \mathsf{Q}[\bar{l}/\&\bar{\mathsf{z}}]\} \triangleleft \mathsf{M} \setminus \bar{\mathsf{z}}} \quad \text{(localvar)}$$

$$\frac{\begin{array}{c}\Gamma(\mathsf{C}) = \Delta \qquad \mathtt{this}, \bar{\mathsf{u}} \notin \mathsf{M} \cup \mathrm{FVA}(\mathsf{P}, \mathsf{Q}) \\ MS = \mathsf{C}\langle\Delta\rangle.\mathsf{m} : \langle(\bar{\mathsf{u}}; \psi).\{\mathsf{P}\}\_\{\mathsf{d}.\mathsf{Q}\}\rangle \\ \Gamma, MS; \Delta; \bar{\mathsf{u}}, \mathtt{this}; \psi \vdash \{\mathsf{P}\}\bar{\mathsf{G}}\bar{\mathsf{z}}; \mathsf{s}; \mathtt{return}\ \mathsf{x}\{\mathsf{d}.\mathsf{Q}\} \triangleleft \mathsf{M}\end{array}}{\Gamma, MS \vdash \mathsf{G}\ \mathsf{m}(\bar{\mathsf{G}}\bar{\mathsf{u}})\ \{\bar{\mathsf{G}}\bar{\mathsf{z}};\ \mathtt{s}\ \mathtt{return}\ \mathsf{x};\} : MS}$$

Class definitions $\boxed{\Gamma \vdash \mathsf{L} : \Gamma'}$

$$\frac{\Gamma \vdash \mathsf{K} : \Gamma_{\mathsf{K}} \qquad \bar{\mathsf{M}} = \mathsf{M}_1 \cdots \mathsf{M}_n \qquad \forall i \in \{1, \cdots, n\}.\ \Gamma \vdash \mathsf{M}_i : \Gamma_i}{\Gamma \vdash \mathtt{class}\ \mathsf{C}\langle\bar{\mathsf{T}}\rangle : \mathsf{D}\ \{\mathtt{public}\ \bar{\mathsf{C}}\bar{\mathsf{f}}; \bar{\mathsf{A}}\mathsf{K}\bar{\mathsf{M}}\} : \Gamma_{\mathsf{K}}, \Gamma_1, \cdots, \Gamma_n}$$

Programs $\boxed{\psi \vdash \{\mathsf{P}\}\bar{\mathsf{L}}; \bar{\mathsf{C}}\ \bar{\mathsf{x}}; \mathsf{s}\{\mathsf{Q}\} \triangleleft \mathsf{M}}$

$$\frac{\bar{\mathsf{L}} = \mathsf{L}_1 \cdots \mathsf{L}_n \qquad \Gamma = \Gamma_1, \cdots, \Gamma_n \qquad \Gamma \vdash \mathsf{L}_i : \Gamma_i, \quad \forall i \in \{1, \cdots, n\} \qquad \Gamma; -; \bar{\mathsf{x}}; \psi \vdash \{\mathsf{P}\}\mathsf{s}\{\mathsf{Q}\} \triangleleft \mathsf{M}}{\psi \vdash \{\mathsf{P}\}\bar{\mathsf{L}}; \bar{\mathsf{C}}\ \bar{\mathsf{x}}; \mathsf{s}\{\mathsf{Q}\} \triangleleft \mathsf{M}}$$

3.4 Semantics

Specifications $\boxed{[\![\Delta;\phi;\psi \vdash S : \texttt{Spec}]\!] : [\![\Delta]\!] \times [\![\phi]\!] \times [\![\psi]\!] \to \{U \in \mathcal{P}^{\downarrow}(\mathsf{N}) \mid 0 \in U\}}$

$$
\begin{array}{l}
[\![\Delta;\phi;\psi \vdash \{\mathsf{P}\}\mathsf{s}\{\mathsf{Q}\} \triangleleft \mathsf{M} : \texttt{Spec}]\!](\delta;(E,S);\vartheta) = \{n \in \mathsf{N} \mid \\
\quad \forall m \le n.\ \forall k \le m.\ \forall C \in \mathbb{S}.\ \forall H \in \mathbb{H}. \\
\quad\quad (m-1,C,H) \in [\![\Delta;\phi;\psi \vdash \mathsf{P} : \texttt{Prop}]\!](\delta;(E,S);\vartheta) \wedge C\#S \ \Rightarrow \\
\quad\quad\quad (\delta;E;C \uplus S;H;\mathsf{s})) : \texttt{safe}_k \wedge \\
\quad\quad\quad (\delta;E;C \uplus S;H;\mathsf{s}) \Downarrow_k (S';H') \Rightarrow \\
\quad\quad\quad\quad (m-k;S' \setminus E(\phi);H') \in [\![\Delta;\phi;\psi \vdash \mathsf{Q} : \texttt{Prop}]\!](\delta;(E,S'|_{E(\phi)});\vartheta) \wedge \\
\quad\quad\quad\quad \forall \mathsf{x} \in \phi \setminus \mathsf{M}.\ S(E(\mathsf{x})) = S'(E(\mathsf{x}))\}
\end{array}
$$

$$
\begin{array}{l}
[\![\Delta;\phi;\psi \vdash \{\mathsf{P}\}\bar{\mathsf{G}}\bar{\mathsf{z}};\ \mathsf{s};\texttt{return}\ \mathsf{x}\{\mathsf{d}.\mathsf{Q}\} \triangleleft \mathsf{M} : \texttt{Spec}]\!](\delta;(E,S);\vartheta) = \{n \in \mathsf{N} \mid \\
\quad \forall m \le n.\ \forall k \le m.\ \forall C \in \mathbb{S}.\ \forall H \in \mathbb{H}.\ \forall \bar{l}_z \in \mathrm{Loc} \setminus (\mathrm{Dom}(C) \cup \mathrm{Dom}(S)). \\
\quad\quad (m-1,C,H) \in [\![\Delta;\phi;\psi \vdash \mathsf{P} : \texttt{Prop}]\!](\delta,(E,S);\vartheta) \wedge C\#S \ \Rightarrow \\
\quad\quad\quad (\delta;E';C \uplus S[\bar{l}_z \mapsto \texttt{null}];H;\mathsf{s}) : \texttt{safe}_k \wedge \\
\quad\quad\quad (\delta;E';C \uplus S[\bar{l}_z \mapsto \texttt{null}];H;\mathsf{s}) \Downarrow_k (S';H') \Rightarrow \\
\quad\quad\quad\quad (m-k;S' \setminus E(\phi);H') \in [\![\Delta;\phi;\psi \vdash \mathsf{Q} : \texttt{Prop}]\!](\delta;(E,S'|_{E(\phi)});\vartheta[\mathsf{d} \mapsto S'(E'(\mathsf{x}))]) \wedge \\
\quad\quad\quad\quad \forall \mathsf{x} \in \phi \setminus \mathsf{M}.\ S(E(\mathsf{x})) = S'(E(\mathsf{x}))\}
\end{array}
$$

where $E' = E[\bar{\mathsf{z}} \mapsto \bar{l}_z]$.

Context Specification $\boxed{[\![MS : \texttt{Context-Spec}]\!] : \{U \in \mathcal{P}^{\downarrow}(\mathsf{N}) \mid 0 \in U\}}$

$$
\begin{array}{l}
[\![\mathsf{C}\langle\Delta\rangle.\mathsf{m} : \langle(\bar{\mathsf{u}};\psi).\{\mathsf{P}\}\_\{\mathsf{d}.\mathsf{Q}\}\rangle : \texttt{Context-Spec}]\!] = \{n \in \mathsf{N} \mid \\
\quad \forall m \le n.\ \forall k \le m.\ \forall \bar{\mathsf{x}},\bar{\mathsf{z}},\mathsf{s},\mathsf{r}.\ \forall C \in \mathbb{S}.\ \forall \vartheta \in [\![\psi]\!].\ \forall \delta \in [\![\Delta]\!].\ \forall l_t,\bar{l}_x,\bar{l}_z \notin \mathrm{Dom}(C).\ \forall v_t,\bar{v}_x \in \mathbb{V}. \\
\quad P(\mathsf{C}) = \Delta \wedge P(\mathsf{C},\mathsf{m}) = (\bar{\mathsf{x}},\bar{\mathsf{z}},\mathsf{s},\mathsf{r}) \wedge \\
\quad (m-1,C,H) \in [\![\Delta;-;\psi,\texttt{this},\bar{\mathsf{u}} \vdash \mathsf{P} : \texttt{Prop}]\!](\delta;([],[]);\vartheta[\texttt{this} \mapsto v_t,\bar{\mathsf{u}} \mapsto \bar{v}_x]) \ \Rightarrow \\
\quad\quad (\delta;E,C[l_t \mapsto v_t,\bar{l}_x \mapsto \bar{v}_x,\bar{l}_z \mapsto \texttt{null}],H,\mathsf{s}) : \texttt{safe}_k \wedge \\
\quad\quad (\delta;E,C[l_t \mapsto v_t,\bar{l}_x \mapsto \bar{v}_x,\bar{l}_z \mapsto \texttt{null}],H,\mathsf{s})) \Downarrow_k (S',H') \ \Rightarrow \\
\quad\quad\quad (m-k,S' \setminus l_t \cup \bar{l}_z,H') \in [\![\Delta;-;\psi,\texttt{this},\bar{\mathsf{u}},\mathsf{d} \vdash \mathsf{Q} : \texttt{Prop}]\!](\delta,([],[]);\vartheta')\}
\end{array}
$$

where $E = [\texttt{this} \mapsto l_t, \bar{\mathsf{x}} \mapsto \bar{l}_x, \bar{\mathsf{z}} \mapsto \bar{l}_z]$ and $\vartheta' = \vartheta[\texttt{this} \mapsto v_t, \bar{\mathsf{u}} \mapsto \bar{v}_x, \mathsf{d} \mapsto S'(E(\mathsf{r}))]$.

Context $\boxed{[\![\Gamma : \texttt{Context}]\!] : \{U \in \mathcal{P}^{\downarrow}(\mathsf{N}) \mid 0 \in U\}}$

$$[\![\Gamma : \texttt{Context}]\!] = \bigcap_{MS \in \Gamma} [\![MS : \texttt{Context-Spec}]\!]$$

Entailment $\boxed{[\![\Gamma;\Delta;\phi;\psi \vdash \mathsf{S} : \texttt{Spec}]\!] : 2}$

$$
\begin{array}{l}
[\![\Gamma;\Delta;\phi;\psi \vdash \mathsf{S} : \texttt{Spec}]\!] = \forall n \in \mathsf{N}.\ \forall (E,S) \in [\![\phi]\!].\ \forall \vartheta \in [\![\psi]\!].\ \forall \delta \in [\![\Delta]\!]. \\
\quad n \in [\![\Gamma : \texttt{Context}]\!] \Rightarrow n+1 \in [\![\Gamma;\Delta;\phi;\psi \vdash \mathsf{S} : \texttt{Spec}]\!](\delta;(E,S);\vartheta)
\end{array}
$$

Others

$$[\![\Gamma \vdash \texttt{public C m}(\bar{\mathsf{C}}\bar{\mathsf{u}})\ \mathsf{B} : MS]\!] = \forall n \in \mathbb{N}.\ n \in [\![\Gamma : \texttt{Context}]\!] \Rightarrow n+1 \in [\![\mathsf{C.m} : MS : \texttt{Context-Spec}]\!]$$

$$[\![\Gamma \vdash \texttt{class C}\langle\bar{\mathsf{T}}\rangle : \mathsf{G}\ \{\texttt{public}\ \bar{\mathsf{C}}\bar{\mathsf{f}}; \bar{\mathsf{A}}\mathsf{K}\bar{\mathsf{M}}\} : MS_K, \bar{MS}]\!] = \\ \mathrm{Dom}(\bar{MS}) = \mathrm{Dom}(\bar{\mathsf{M}}) \wedge [\![\Gamma; \bar{\mathsf{T}} \vdash K : \Gamma_K]\!] \wedge \forall i \in \mathrm{Dom}(\bar{\mathsf{M}}).\ [\![\Gamma; \bar{\mathsf{T}} \vdash \bar{\mathsf{M}}_i : \bar{MS}_i]\!]$$

$$[\![\psi \vdash \{\mathsf{P}\}\bar{\mathsf{L}}; \bar{\mathsf{C}}\bar{\mathsf{x}}; \mathsf{s}\{\mathsf{Q}\} \triangleleft \mathsf{M}]\!] = [\![\Gamma; -; \bar{\mathsf{x}}; \psi \vdash \{\mathsf{P}\}\mathsf{s}\{\mathsf{Q}\} \triangleleft \mathsf{M}]\!] \wedge \forall i \in \mathrm{Dom}(\bar{\mathsf{L}}).\ [\![\Gamma \vdash \bar{\mathsf{L}}_i : \Gamma_i]\!]$$

where $\Gamma_i = spec(\bar{\mathsf{L}}_i)$ and $\Gamma = \Gamma_1, ..., \Gamma_n$.

3.5 Metatheory

Lemma 14. *If* $\Gamma; \Delta; \phi; \psi \vdash \{\mathsf{P}\}\mathsf{s}\{\mathsf{Q}\} \triangleleft \mathsf{M}$ *then* $FV(\mathsf{s}) \subseteq \phi$.

Lemma 15. *If* $\forall i \in \mathrm{Dom}(\bar{\mathsf{L}}).\ [\![\Gamma \vdash \bar{\mathsf{L}}_i : \Gamma_i]\!]$ *then* $[\![\Gamma : \textit{Context}]\!] = \mathbb{N}$.

Lemma 16. *Rule* (α) *is sound.*

$$\forall \pi \in \mathrm{Perm}(\mathbb{A}_p).\ [\![\Gamma; \Delta; \phi; \psi \vdash \{\mathsf{P}\}\mathsf{s}\{\mathsf{Q}\} \triangleleft \mathsf{M}]\!] = [\![\Gamma; \Delta; \pi(\phi); \pi(\psi) \vdash \{\pi(\mathsf{P})\}\pi(\mathsf{s})\{\pi(\mathsf{Q})\} \triangleleft \pi(\mathsf{M})]\!]$$

Proof. Let $n, m, k \in \mathbb{N}$ such that $k \leq m \leq n+1$. Assume $n \in [\![\Gamma : \texttt{Context}]\!]$,

$$(m-1, C, H) \in [\![\Delta; \pi(\phi); \pi(\psi) \vdash \pi(\mathsf{P}) : \texttt{Prop}]\!](\delta; (E, S); \vartheta)$$

and $C \# S$. By alpha-renaming and equivariance it follows that,

$$(m-1, C, \pi^{-1}(H)) \in [\![\Delta; \phi; \psi \vdash \mathsf{P} : \texttt{Prop}]\!](\delta; (\pi^{-1}(E), S); \pi^{-1}(\vartheta))$$

Hence,

$$(\pi^{-1}(E), C \uplus S, \pi^{-1}(H), \mathsf{s}) : \texttt{safe}_m$$

and thus by alpha-renaming,

$$(E, C \uplus S, H, \pi(\mathsf{s})) : \texttt{safe}_m$$

Correctness: If

$$(E, C \uplus S, H, \pi(\mathsf{s})) \Downarrow_k (S', H')$$

then by alpha-renaming,

$$(\pi^{-1}(E), C \uplus S, \pi^{-1}(H), \mathsf{s}) \Downarrow_k (S', \pi^{-1}(H'))$$

and thus,

$$(m-k, S' \setminus \pi^{-1}(E)(\phi), \pi^{-1}(H')) \in [\![\Delta; \phi; \psi \vdash \mathsf{Q} : \texttt{Prop}]\!](\delta; (\pi^{-1}(E), S'|_{\pi^{-1}(E)(\phi)}); \pi^{-1}(\vartheta))$$

and by alpha-renaming and equivariance we thus have that,

$$(m-k, S' \setminus E(\pi(\phi)), H') \in [\![\Delta; \pi(\phi); \pi(\psi) \vdash \pi(\mathsf{Q}) : \texttt{Prop}]\!](\delta; (E, S'|_{E(\pi(\phi))}); \vartheta)$$

□

Lemma 17. *Rule* (frame) *is sound.*

$$\frac{\Gamma;\Delta;\phi;\psi \vdash \mathsf{R} : \mathit{Prop} \qquad \Gamma;\Delta;\phi;\psi \vdash \{\mathsf{P}\}\mathtt{s}\{\mathsf{Q}\} \triangleleft \mathsf{M} \qquad \mathrm{FVV}(\mathsf{R}) \cap \mathsf{M} = \emptyset}{\Gamma;\Delta;\phi;\psi \vdash \{\mathsf{P} * \mathsf{R}\}\mathtt{s}\{\mathsf{Q} * \mathsf{R}\} \triangleleft \mathsf{M}}$$

Proof. Let $n, m, k \in \mathsf{N}$ such that $k \leq m \leq n+1$. Assume $n \in [\![\Gamma : \mathtt{Context}]\!]$,

$$(m-1, C_1, H_1) \in [\![\Delta;\phi;\psi \vdash \mathsf{P} : \mathtt{Prop}]\!](\delta;(E,S);\vartheta)$$

$$(m-1, C_2, H_2) \in [\![\Delta;\phi;\psi \vdash \mathsf{R} : \mathtt{Prop}]\!](\delta;(E,S);\vartheta)$$

$C_1 \# C_2$, $H_1 \# H_2$, and $(C_1 \uplus C_2) \# S$.

By assumption,

$$[\![\Delta;\phi;\psi \vdash \{\mathsf{P}\}\mathtt{s}\{\mathsf{Q}\} \triangleleft \mathsf{M}]\!](\delta;(E,S);\vartheta, n+1)$$

and thus,

$$(E, C_1 \uplus S, H_1, \mathtt{s}) : \mathtt{safe}_m$$

and by safety monotonicity,

$$(E, C_1 \uplus C_2 \uplus S, H_1 \uplus H_2, \mathtt{s}) : \mathtt{safe}_m$$

Correctness: If

$$(E, C_1 \uplus C_2 \uplus S, H_1 \uplus H_2, \mathtt{s}) \Downarrow_k (S', H')$$

then by the stack and heap frame properties it follows that there exists a S_1' and H_1' such that,

$$(E, C_1 \uplus S, H_1, \mathtt{s}) \Downarrow_k (S_1', H_1')$$

and $S' = C_2 \uplus S_1'$ and $H' = H_2 \uplus H_1'$. Hence,

$$(m-k, S_1' \setminus E(\phi), H_1') \in [\![\Delta;\phi;\psi \vdash \mathsf{Q} : \mathtt{Prop}]\!](\delta;(E, S'|_{E(\phi)});\vartheta)$$

Furthermore,

$$[\![\Delta;\phi;\psi \vdash \mathsf{R} : \mathtt{Prop}]\!](\delta;(E,S);\vartheta) = [\![\Delta;\phi;\psi \vdash \mathsf{R} : \mathtt{Prop}]\!](\delta;(E, S'|_{E(\phi)});\vartheta)$$

since $\forall \mathsf{x} \in \mathrm{FVV}(\mathsf{R})$. $S(E(\mathsf{x})) = S'(E(\mathsf{x}))$ and thus, by upwards-closure,

$$(m-k, C_2 \setminus E(\phi), H_2) \in [\![\Delta;\phi;\psi \vdash \mathsf{R} : \mathtt{Prop}]\!](\delta;(E, S'|_{E(\phi)});\vartheta)$$

and thus finally,

$$(m-k, (S_1' \uplus C_2) \setminus E(\phi), H_1' \uplus H_2) \in [\![\Delta;\phi;\psi \vdash \mathsf{P} * \mathsf{R} : \mathtt{Prop}]\!](\delta;(E, S'|_{E(\phi)});\vartheta)$$

□

Lemma 18. *Rule* (seq) *is sound.*

$$\frac{\Gamma;\Delta;\phi;\psi \vdash \{\mathsf{P}\}\mathsf{s}_1\{\mathsf{Q}\} \triangleleft \mathsf{M}_1 \qquad \Gamma;\Delta;\phi;\psi \vdash \{\mathsf{Q}\}\mathsf{s}_2\{\mathsf{R}\} \triangleleft \mathsf{M}_2}{\Gamma;\Delta;\phi;\psi \vdash \{\mathsf{P}\}\mathsf{s}_1;\mathsf{s}_2\{\mathsf{R}\} \triangleleft \mathsf{M}_1 \cup \mathsf{M}_2}$$

Proof. Let $n, m, k \in \mathsf{N}$ such that $k \leq m \leq n+1$. Assume $n \in [\![\Gamma : \texttt{Context}]\!]$,

$$(m-1, C, H) \in [\![\Delta;\phi;\psi \vdash \mathsf{P} : \texttt{Prop}]\!](\delta;(E,S);\vartheta)$$

and $C \# S$.

Safety: By assumption,

$$[\![\Delta;\phi;\psi \vdash \{\mathsf{P}\}\mathsf{s}_1\{\mathsf{Q}\} \triangleleft \mathsf{M}_1]\!](\delta;(E,S);\vartheta;n+1)$$

and thus,

$$(E; C \uplus S; H; \mathsf{s}_1) : \texttt{safe}_m$$

Hence, for any $l \leq k$ if $(E; C \uplus S; H; \mathsf{s}_1\ \mathsf{s}_2) \Downarrow_l \text{err}$ then there exists l_1, l_2, S', and H' such that,

$$(E; C \uplus S; H; \mathsf{s}_1) \Downarrow_{l_1} (S', H')$$

$$(E; S'; H'; \mathsf{s}_2) \Downarrow_{l_2} \text{err}$$

and $l = l_1 + l_2$. Hence,

$$(m - l_1; S' \setminus E(\phi); H') \in [\![\Delta;\phi;\psi \vdash \mathsf{Q} : \texttt{Prop}]\!](\delta;(E, S'|_{E(\phi)});\vartheta)$$

and thus

$$(E; (S' \setminus E(\phi)) \uplus S'|_{E(\phi)}; H', \mathsf{s}_2) : \texttt{safe}_{m-l_1+1}$$

Since $l_1 + l_2 = l \leq k \leq m$ it follows that $l_2 \leq m - l_1 + 1$ and thus

$$(E; S'; H', \mathsf{s}_2) \not\Downarrow_{l_2} \text{err}$$

which is a contradiction.

Correctness: If

$$(E; C \uplus S; H; s_1; s_2) \Downarrow_k (S', H')$$

then there exists k_1, k_2, S'', and H'' such that,

$$(E; C \uplus S; H; s_1) \Downarrow_{k_1} (S'', H'')$$

$$(E; S'', H''; s_2) \Downarrow_{k_2} (S', H')$$

and $k = k_1 + k_2$. Hence,

$$(m - k_1; S'' \setminus E(\phi); H'') \in [\![\Delta;\phi;\psi \vdash \mathsf{Q} : \texttt{Prop}]\!](\delta;(E, S''|_{E(\phi)});\vartheta)$$

Furthermore, by assumption,

$$[\![\Delta;\phi;\psi \vdash \{\mathsf{Q}\}\mathsf{s}_2\{\mathsf{R}\} \triangleleft \mathsf{M}_2]\!](\delta;(E, S''|_{E(\phi)});\vartheta;n+1)$$

Since $1 \leq k_1$ we have it follows that $k_2 \leq (m - k_1 + 1) \leq n + 1$ and thus,

$$(m - k_1 + 1 - k_2; S' \setminus E(\phi); H') \in [\![\Delta;\phi;\psi \vdash \mathsf{R} : \texttt{Prop}]\!](\delta;(E, S'|_{E(\phi)});\vartheta)$$

and by upwards-closure:

$$(m - k; S' \setminus E(\phi); H') \in [\![\Delta;\phi;\psi \vdash \mathsf{R} : \texttt{Prop}]\!](\delta;(E, S'|_{E(\phi)});\vartheta)$$

□

Lemma 19. *Rule* (lookup) *is sound.*

$$\frac{\Gamma;\Delta;\phi;\psi \vdash \{\,\mathit{lookup}\ l\ \mathit{as}\ \mathsf{x}\ \mathit{in}\ \mathsf{P}\}\mathsf{s}\{\,\mathit{lookup}\ l\ \mathit{as}\ \mathsf{x}\ \mathit{in}\ \mathsf{Q}\} \triangleleft \mathsf{M}}{\Gamma;\Delta;\phi,\mathsf{x};\psi \vdash \{\&\mathsf{x} = l \wedge \mathsf{P}\}\mathsf{s}\{\mathsf{Q}\} \triangleleft \mathsf{M} \cup \{\mathsf{x}\}}$$

Proof. Let $n, m, k \in \mathsf{N}$ such that $k \leq m \leq n+1$. Assume $n \in [\![\Gamma : \texttt{Context}]\!]$,

$$(m-1; C; H) \in [\![\Delta;\phi,\mathsf{x};\psi \vdash \&\mathsf{x} = l \wedge \mathsf{P} : \texttt{Prop}]\!](\delta;(E,S);\vartheta)$$

and $C \# S$. Then $\vartheta(l) = E(\mathsf{x})$ and,

$$(m-1; C; H) \in [\![\Delta;\phi,\mathsf{x};\psi \vdash \mathsf{P} : \texttt{Prop}]\!](\delta;(E,S);\vartheta)$$

and by splitting,

$$(m-1; C; H) \in [\![\Delta;\phi;\psi,\mathsf{x} \vdash \mathsf{P} : \texttt{Prop}]\!](\delta;(E|_\phi, S \setminus E(\mathsf{x}));\vartheta[\mathsf{x} \mapsto S(E(\mathsf{x}))])$$

By the definition of `lookup` it thus follows that,

$$(m-1; C[E(\mathsf{x}) \mapsto S(E(\mathsf{x}))]; H) \in [\![\Delta;\phi;\psi \vdash \texttt{lookup}\ l\ \texttt{as}\ \mathsf{x}\ \texttt{in}\ \mathsf{P} : \texttt{Prop}]\!](\delta;(E|_\phi, S \setminus E(\mathsf{x}));\vartheta)$$

and by assumption:

$$[\![\Delta;\phi;\psi \vdash \{\texttt{lookup}\ l\ \texttt{as}\ \mathsf{x}\ \texttt{in}\ \mathsf{P}\}\mathsf{s}\{\texttt{lookup}\ l\ \texttt{as}\ \mathsf{x}\ \texttt{in}\ \mathsf{Q}\} \triangleleft \mathsf{M}]\!](\delta;(E|_\phi; S \setminus E(\mathsf{x}));\vartheta;n+1)$$

Safety: Hence,

$$(E|_\phi; (C[E(\mathsf{x}) \mapsto S(E(\mathsf{x}))) \uplus (S \setminus E(\mathsf{x})); H; \mathsf{s}) : \texttt{safe}_k$$

and by weakening,

$$(E; C \uplus S; H; \mathsf{s}) : \texttt{safe}_k$$

Correctness: Furthermore, if

$$(E; C \uplus S; H; \mathsf{s}) \Downarrow_k (S', H')$$

then since $\mathsf{x} \notin \mathrm{FV}(\mathsf{s})$ it follows that,

$$(E|_\phi; C \uplus S; H; \mathsf{s}) \Downarrow_k (S', H')$$

and thus,

$$(m-k; S' \setminus E(\phi); H') \in [\![\Delta;\phi;\psi \vdash \texttt{lookup}\ l\ \texttt{as}\ \mathsf{x}\ \texttt{in}\ \mathsf{Q} : \texttt{Prop}]\!](\delta;(E|_\phi, S'|_{E(\phi)});\vartheta)$$

By the definition of `lookup` we thus have that $E(\mathsf{x}) \in \mathrm{Dom}(S' \setminus E(\phi))$ and

$$(m-k; S' \setminus E(\phi,\mathsf{x}); H') \in [\![\Delta;\phi;\psi,\mathsf{x} \vdash \mathsf{Q} : \texttt{Prop}]\!](\delta;(E|_\phi, S'|_{E(\phi)});\vartheta[\mathsf{x} \mapsto (S' \setminus E(\phi))(E(\mathsf{x}))])$$

and by splitting,

$$(m-k; S' \setminus E(\phi,\mathsf{x}); H') \in [\![\Delta;\phi,\mathsf{x};\psi \vdash \mathsf{Q} : \texttt{Prop}]\!](\delta;(E, S'|_{E(\phi,\mathsf{x})});\vartheta)$$

□

Lemma 20. *Rule* (anondel) *is sound.*

$$\frac{\bar{\mathsf{u}} \cup \bar{\mathsf{y}} = \mathrm{FV}(\mathsf{B}) \qquad \Gamma; \Delta; \bar{\mathsf{y}}, \bar{\mathsf{u}}; \psi \vdash \{\mathsf{P}\}\mathsf{B}\{\mathsf{d}.\mathsf{Q}\} \triangleleft \mathsf{M} \qquad \bar{\mathsf{u}} \notin \mathsf{M} \qquad \Delta; \bar{\mathsf{y}}; \psi, \bar{\mathsf{u}} \vdash \mathsf{P} : \mathit{Prop} \qquad \Delta; \bar{\mathsf{y}}; \psi, \bar{\mathsf{u}}, \mathsf{d} \vdash \mathsf{Q} : \mathit{Prop}}{\begin{array}{l}\Gamma; \Delta; \bar{\mathsf{y}}, \mathsf{x}; \psi, \bar{l} \vdash \{\bar{l} = \&\bar{\mathsf{y}}\} \\ \qquad \mathsf{x} = \lambda\bar{\mathsf{u}}.\ \{\mathsf{B}\}; \\ \qquad \{\mathsf{x} \mapsto \langle(\bar{\mathsf{u}}).\{\mathit{lookup}\ \bar{l}\ \mathit{as}\ \bar{\mathsf{z}}\ \mathit{in}\ \mathsf{P}[\bar{l}/\&\bar{\mathsf{y}}][\bar{\mathsf{z}}/\bar{\mathsf{y}}]\}\_\{\mathsf{d}.\mathit{lookup}\ \&l\ \mathit{as}\ \bar{\mathsf{z}}\ \mathit{in}\ \mathsf{Q}[\bar{l}/\&\bar{\mathsf{y}}][\bar{\mathsf{z}}/\bar{\mathsf{y}}]\}\rangle\} \triangleleft \{\mathsf{x}\}\end{array}}$$

Proof. Let $n, m, k \in \mathsf{N}$ such that $k \leq m \leq n+1$. Assume $n \in [\![\Gamma : \mathtt{Context}]\!]$,

$$(m-1, C, H) \in [\![\Delta; \bar{\mathsf{y}}, \mathsf{x}; \psi, \bar{l} \vdash \bar{l} = \&\bar{\mathsf{y}} : \mathtt{Prop}]\!](\delta; (E, S); \vartheta)$$

and $C \# S$. If,

$$(E; C \uplus S; H; \mathsf{x} = \lambda\bar{\mathsf{u}}.\{\mathsf{B}\};) \Downarrow_k (S', H')$$

then $S' = C \uplus S[E(\mathsf{x}) \mapsto l]$ and $H' = H[l \mapsto (\delta, E_c, \mathsf{B})])$ where $E_c = E|_{\bar{\mathsf{y}}}$, $k = 1$ and $l \notin \mathrm{Dom}(C \uplus S)$.

Let $k' \leq m' \leq m-1$, $C_c \in \mathbb{S}, H_c \in \mathbb{H}, \bar{l}_x, \bar{l}_z \in \mathrm{Loc} \setminus (\mathrm{Dom}(C_c) \cup \mathrm{Rng}(E_c)), \bar{v}_x \in \mathbb{V}$ such that,

$$(m'-1, C_c, H_c) \in [\![\Delta; \bar{\mathsf{y}}, \mathsf{x}; \psi, \bar{l}, \bar{\mathsf{u}} \vdash \mathtt{lookup}\ \bar{l}\ \mathtt{as}\ \bar{\mathsf{z}}\ \mathtt{in}\ \mathsf{P}[\bar{l}/\&\bar{\mathsf{y}}][\bar{\mathsf{z}}/\bar{\mathsf{y}}] : \mathtt{Prop}]\!](\delta; (E, S'); \vartheta[\bar{\mathsf{u}} \mapsto \bar{v}_x])$$

Hence,

$$(m'-1, C_c \backslash E(\bar{\mathsf{y}}), H_c) \in [\![\Delta; \bar{\mathsf{y}}, \mathsf{x}; \psi, \bar{l}, \bar{\mathsf{u}}, \bar{\mathsf{z}} \vdash \mathsf{P}[\bar{l}/\&\bar{\mathsf{y}}][\bar{\mathsf{z}}/\bar{\mathsf{y}}] : \mathtt{Prop}]\!](\delta; (E, S'); \vartheta[\bar{\mathsf{u}} \mapsto \bar{v}_x, \bar{\mathsf{z}} \mapsto C_c(E(\bar{\mathsf{y}}))])$$

and by strengthening and renaming,

$$(m'-1, C_c \setminus E(\bar{\mathsf{y}}), H_c) \in [\![\Delta; -; \psi, \bar{l}, \bar{\mathsf{u}}, \bar{\mathsf{y}} \vdash \mathsf{P}[\bar{l}/\&\bar{\mathsf{y}}] : \mathtt{Prop}]\!](\delta; ([], []); \vartheta[\bar{\mathsf{u}} \mapsto \bar{v}_x, \bar{\mathsf{y}} \mapsto C_c(E(\bar{\mathsf{y}}))])$$

and splitting,

$$(m'-1, C_c \backslash E(\bar{\mathsf{y}}), H_c) \in [\![\Delta; \bar{\mathsf{y}}, \bar{\mathsf{u}}; \psi \vdash \mathsf{P} : \mathtt{Prop}]\!](\delta; ([\bar{\mathsf{y}} \mapsto E(\bar{\mathsf{y}}), \bar{\mathsf{u}} \mapsto \bar{l}_u], [E(\bar{\mathsf{y}}) \mapsto C_c(E(\bar{\mathsf{y}})), \bar{l}_u \mapsto \bar{v}_x]); \vartheta|_\psi)$$

Futhermore, by assumption,

$$[\![\Delta; \bar{\mathsf{y}}, \bar{\mathsf{u}}; \psi \vdash \{\mathsf{P}\}\mathsf{B}\{\mathsf{Q}\} \triangleleft \mathsf{M}]\!](\delta; ([\bar{\mathsf{y}} \mapsto E(\bar{\mathsf{y}}), \bar{\mathsf{u}} \mapsto \bar{l}_u], [E(\bar{\mathsf{y}}) \mapsto C_c(E(\bar{\mathsf{y}})), \bar{l}_u \mapsto \bar{v}_x]); \vartheta|_\psi; n+1)$$

and since $k' \leq m' \leq n+1$,

$$(\delta, E'; C_c \setminus E(\bar{\mathsf{y}}) \uplus [E(\bar{\mathsf{y}}) \mapsto C_c(E(\bar{\mathsf{y}})), \bar{l}_u \mapsto \bar{v}_x, \bar{l}_z \mapsto \mathtt{null}]; H_c, \mathsf{s}) : \mathtt{safe}_{k'}$$

where $E' = [\bar{\mathsf{y}} \mapsto E(\bar{\mathsf{y}}), \bar{\mathsf{u}} \mapsto \bar{l}_u, \bar{\mathsf{z}} \mapsto \bar{l}_z]$. Safety follows by safety monotonicity.

Correctness: If,

$$(\delta, E'; C_c[\bar{l}_u \mapsto \bar{v}_x, \bar{l}_z \mapsto \mathtt{null}]; H_c; \mathsf{s}) \Downarrow_{k'} (S'', H'')$$

then,

$$(m'-k', S'' \backslash E'(\bar{\mathsf{u}}, \bar{\mathsf{y}}), H'') \in [\![\Delta; \bar{\mathsf{y}}, \bar{\mathsf{u}}; \psi, \mathsf{d} \vdash \mathsf{Q} : \mathtt{Prop}]\!](\delta; ([\bar{\mathsf{y}} \mapsto E(\bar{\mathsf{y}}), \bar{\mathsf{u}} \mapsto \bar{l}_u], S''|_{E'(\bar{\mathsf{u}}, \bar{\mathsf{y}})}); \vartheta|_\psi[\mathsf{d} \mapsto S''(E'(\mathsf{r}))])$$

and since $\bar{\mathsf{u}} \notin \mathsf{M}$, $S''(\bar{l}_u) = \bar{v}_x$. By splitting and renaming we thus have that,

$$(m'-k', S'' \setminus E'(\bar{\mathsf{y}}, \bar{\mathsf{u}}), H'') \in [\![\Delta; -; \psi, \bar{l}, \bar{\mathsf{z}}, \bar{\mathsf{u}}, \mathsf{d} \vdash \mathsf{Q}[\bar{l}/\&\bar{\mathsf{y}}][\bar{\mathsf{z}}/\bar{\mathsf{y}}] : \mathtt{Prop}]\!](\delta; ([], []); \vartheta')$$

where $\vartheta' = \vartheta[\bar{\mathsf{z}} \mapsto S''(E(\bar{\mathsf{y}})), \bar{\mathsf{u}} \mapsto \bar{v}_x, \mathsf{d} \mapsto S''(E'(\mathsf{r}))]$. By weakening,

$$(m' - k', S'' \setminus E(\bar{\mathsf{y}}, \bar{\mathsf{u}}), H'') \in [\![\Delta; \bar{\mathsf{y}}, \mathsf{x}; \psi, \bar{l}, \bar{\mathsf{z}}, \bar{\mathsf{u}} \vdash \mathsf{Q}[\bar{l}/\&\bar{\mathsf{y}}][\bar{\mathsf{z}}/\bar{\mathsf{y}}] : \mathtt{Prop}]\!](\delta; (E, S'); \vartheta')$$

and hence,

$$(m' - k', S'' \setminus \bar{l}_u, H'') \in [\![\Delta; \bar{\mathsf{y}}, \mathsf{x}; \psi, \bar{l}, \bar{\mathsf{u}}, \mathsf{d} \vdash \mathtt{lookup}\ \bar{l}\ \mathtt{as}\ \bar{\mathsf{z}}\ \mathtt{in}\ \mathsf{Q}[\bar{l}/\&\bar{\mathsf{y}}][\bar{\mathsf{z}}/\bar{\mathsf{y}}] : \mathtt{Prop}]\!](\delta; (E, S'); \vartheta'')$$

where $\vartheta'' = \vartheta[\bar{\mathsf{u}} \mapsto \bar{v}_x, \mathsf{d} \mapsto S''(E'(\mathsf{r}))]$. □

Lemma 21. *Rule* (delcall) *is sound.*

$$\frac{\mathsf{R} = \mathsf{y} \mapsto \langle(\bar{\mathsf{u}}).\{\mathsf{P}\}\_\{\mathsf{d}.\mathsf{Q}\}\rangle \qquad \mathsf{x} \notin \mathrm{FV}(\mathsf{R}) \qquad \mathsf{y} \in \phi}{\Gamma; \Delta; \phi, \bar{\mathsf{x}}, \mathsf{x}; \psi \vdash \{\mathsf{R} * \mathsf{P}[\bar{\mathsf{x}}/\bar{\mathsf{u}}]\}\mathsf{x} = \mathsf{y}(\bar{\mathsf{x}}); \{\mathsf{R} * \mathsf{Q}[\bar{\mathsf{x}}/\bar{\mathsf{u}}, \mathsf{x}/\mathsf{d}]\} \triangleleft \{\mathsf{x}\}}$$

Proof. Let $n, m, k \in \mathsf{N}$ such that $k \leq m \leq n + 1$. Assume $n \in [\![\Gamma : \mathtt{Context}]\!]$,

$$(m - 1; C_1; H_1) \in [\![\Delta; \phi, \bar{\mathsf{x}}, \mathsf{x}; \psi \vdash \mathsf{y} \mapsto \langle(\bar{\mathsf{u}}).\{\mathsf{P}\}\_\{\mathsf{d}.\mathsf{Q}\}\rangle : \mathtt{Prop}]\!](\delta; (E, S); \vartheta)$$

$$(m - 1; C_2; H_2) \in [\![\Delta; \phi, \bar{\mathsf{x}}, \mathsf{x}; \psi \vdash \mathsf{P}[\bar{\mathsf{x}}/\bar{\mathsf{u}}] : \mathtt{Prop}]\!](\delta; (E, S); \vartheta)$$

$$H_1(E(\mathsf{y})) = (\delta_c, E_c, \bar{\mathsf{u}}, \bar{\mathsf{z}}, \mathsf{s}, \mathsf{r})$$

$C_1 \# C_2$, $H_1 \# H_2$, and $(C_1 \uplus C_2) \# S$.

By strengthening, splitting, renaming and upwards-closure it follows that,

$$(m - 1; C_1; H_1) \in [\![\Delta; \phi, \bar{\mathsf{x}}; \psi \vdash \mathsf{y} \mapsto \langle(\bar{\mathsf{u}}).\{\mathsf{P}\}\_\{\mathsf{d}.\mathsf{Q}\}\rangle : \mathtt{Prop}]\!](\delta; (E|_{\phi,\bar{\mathsf{x}}}, S|_{E(\phi,\bar{\mathsf{x}})}); \vartheta)$$

and

$$(m - 2; C_2; H_2) \in [\![\Delta; \phi, \bar{\mathsf{x}}; \psi, \bar{\mathsf{u}} \vdash \mathsf{P} : \mathtt{Prop}]\!](\delta; (E|_{\phi,\bar{\mathsf{x}}}, S|_{E(\phi,\bar{\mathsf{x}})}); \vartheta[\bar{\mathsf{u}} \mapsto S(E(\bar{\mathsf{x}}))])$$

Hence,

$$(\delta_c, E_c'; C_2[\bar{l}_u \mapsto S(E(\bar{\mathsf{x}})), \bar{l}_z \mapsto \mathtt{null}]; H_2; \mathsf{s}) : \mathtt{safe}_{m-1}$$

where $E_c' = E_c[\bar{\mathsf{u}} \mapsto \bar{l}_u, \bar{\mathsf{z}} \mapsto \bar{l}_z]$.

Correctness: If

$$(\delta_c, E; C \uplus S; H, \mathsf{x} = \mathsf{y}(\bar{\mathsf{x}});) \Downarrow_k (S', H')$$

then

$$(\delta_c, E_c'; C \uplus S[\bar{l}_u \mapsto S(E(\bar{\mathsf{x}})), \bar{l}_z \mapsto \mathtt{null}]; H; \mathsf{s}) \Downarrow_{k-1} (S'', H'')$$

and $S' = S''[E(\mathsf{x}) \mapsto S''(E_c'(\mathsf{r}))]$ and $H' = H''$. By the stack and heap frame property there exists C_2' and H_2' such that,

$$(\delta_c, E_c'; C_2[\bar{l}_u \mapsto S(E(\bar{\mathsf{x}})), \bar{l}_z \mapsto \mathtt{null}]; H_2; \mathsf{s}) \Downarrow_{k-1} (C_2', H_2')$$

and $S'' = C_1 \uplus S \uplus C_2'$ and $H'' = H_1 \uplus H_2'$. Hence,

$$((m - 1) - (k - 1); C_2' \setminus \bar{l}_u; H_2') \in [\![\Delta; \phi, \bar{\mathsf{x}}; \psi, \bar{\mathsf{u}}, \mathsf{d} \vdash \mathsf{Q} : \mathtt{Prop}]\!](\delta; (E|_{\phi,\bar{\mathsf{x}}}, S|_{E(\phi,\bar{\mathsf{x}})}); \vartheta')$$

where $\vartheta' = \vartheta[\bar{\mathsf{u}} \mapsto S(E(\bar{\mathsf{x}})), \mathsf{d} \mapsto C_2'(E_c'(\mathsf{r}))]$. Since $S''(E(\phi, \bar{\mathsf{x}})) = S(E(\phi, \bar{\mathsf{x}}))$, by splitting and renaming it follows that,

$$(m - k; C_2' \setminus \bar{l}_u; H_2') \in [\![\Delta; \phi, \bar{\mathsf{x}}; \psi, \mathsf{x} \vdash \mathsf{Q}[\bar{\mathsf{x}}/\bar{\mathsf{u}}, \mathsf{x}/\mathsf{d}] : \mathtt{Prop}]\!](\delta; (E|_{\phi,\bar{\mathsf{x}}}, S''|_{E(\phi,\bar{\mathsf{x}})}); \vartheta[\mathsf{x} \mapsto C_2'(E_c'(\mathsf{r}))])$$

and by splitting again,

$$(m-k; C_2' \setminus \bar{l}_u; H_2') \in [\![\Delta; \phi, \bar{\mathsf{x}}, \mathsf{x}; \psi \vdash \mathsf{Q}[\bar{\mathsf{x}}/\bar{\mathsf{u}}, \mathsf{x}/\mathsf{d}] : \mathtt{Prop}]\!](\delta; (E, S''|_{E(\phi,\bar{\mathsf{x}})}[E(\mathsf{x}) \mapsto C_2'(E_c'(\mathsf{r}))]); \vartheta)$$

Hence, by upwards-closure,

$$(m-k; (S \uplus C_2') \setminus E(\phi, \bar{\mathsf{x}}, \mathsf{x}); H_2') \in [\![\Delta; \phi, \bar{\mathsf{x}}, \mathsf{x}; \psi \vdash \mathsf{Q}[\bar{\mathsf{x}}/\bar{\mathsf{u}}, \mathsf{x}/\mathsf{d}] : \mathtt{Prop}]\!](\delta; (E, S'|_{E(\phi,\bar{\mathsf{x}},\mathsf{x})}); \vartheta)$$

and since $1 \leq k$ and $\mathsf{x} \notin \mathrm{FV}(\mathsf{R})$ it follows by upwards-closure and strengthening and weakening that,

$$(m-k; C_1; H_1) \in [\![\Delta; \phi, \bar{\mathsf{x}}, \mathsf{x}; \psi \vdash \mathsf{y} \mapsto \langle(\bar{\mathsf{u}}).\{\mathsf{P}\}\_\{\mathsf{d}.\mathsf{Q}\}\rangle : \mathtt{Prop}]\!](\delta; (E, S'|_{E(\phi,\bar{\mathsf{x}},\mathsf{x})}); \vartheta)$$

□

Lemma 22. *Rule* (localvar) *is sound.*

$$\frac{\Gamma; \Delta; \phi, \bar{\mathsf{z}}; \psi \vdash \{\mathsf{P} \wedge \bar{\mathsf{z}} = \mathit{null}\}\mathsf{s}; \mathit{return}\ \mathsf{x}\{\mathsf{d}.\mathsf{Q}\} \triangleleft \mathsf{M}}{\Gamma; \Delta; \phi; \psi \vdash \{\mathsf{P}\}\bar{\mathsf{G}}\bar{\mathsf{z}}; \mathsf{s}; \mathit{return}\ \mathsf{x}\{\mathsf{d}.\exists \bar{l} : \mathit{Var}.\ \mathit{lookup}\ \bar{l}\ \mathit{as}\ \bar{\mathsf{z}}\ \mathit{in}\ \mathsf{Q}[\bar{l}/\&\bar{\mathsf{z}}]\} \triangleleft \mathsf{M} \setminus \bar{\mathsf{z}}}$$

Proof. Let $n, m, k \in \mathsf{N}$ such that $k \leq m \leq n+1$. Assume $n \in [\![\Gamma : \mathtt{Context}]\!]$,

$$(m-1; C; H) \in [\![\Delta; \phi; \psi \vdash \mathsf{P} : \mathtt{Prop}]\!](\delta; (E, S); \vartheta)$$

$C \# S$ and $\bar{l}_z \in \mathbb{L}_s \setminus \mathrm{Dom}(C \uplus S)$. By weakening,

$$(m-1; C; H) \in [\![\Delta; \phi, \bar{\mathsf{z}}; \psi \vdash \mathsf{P} : \mathtt{Prop}]\!](\delta; (E', S'); \vartheta)$$

for $E' = E[\bar{\mathsf{z}} \mapsto \bar{l}_z]$ and $S' = S[\bar{l}_z \mapsto \mathtt{null}]$. Since $C \# S'$,

$$(E', C \uplus S', H, \mathsf{s}) : \mathtt{safe}_k$$

and if,

$$(E', C \uplus S', H, \mathsf{s}) \Downarrow_k (S'', H'')$$

then

$$(m-k; S'' \setminus E'(\phi, \bar{\mathsf{z}}); H') \in [\![\Delta; \phi, \bar{\mathsf{z}}; \psi, \mathsf{d} \vdash \mathsf{Q} : \mathtt{Prop}]\!](\delta; (E', S'_{E'(\phi,\bar{\mathsf{z}})}); \vartheta[\mathsf{d} \mapsto S''(E'(\mathsf{x}))])$$

splitting the zs into their values and locations we get:

$$(m-k; S'' \setminus E'(\phi, \bar{\mathsf{z}}); H') \in [\![\Delta; \phi; \psi, \bar{\mathsf{z}}, \bar{l}, \mathsf{d} \vdash \mathsf{Q}[\bar{l}/\&\bar{\mathsf{z}}] : \mathtt{Prop}]\!](\delta; (E'|_\phi, S'_{E'(\phi)}); \vartheta')$$

where $\vartheta' = \vartheta[\mathsf{d} \mapsto S''(E'(\mathsf{x})), \bar{\mathsf{z}} \mapsto S''(\bar{l}_z), \bar{l} \mapsto \bar{l}_z]$. Thus, by definition,

$$(m-k, S'' \setminus E'(\phi); H') \in [\![\Delta; \phi; \psi, \bar{l}, \mathsf{d} \vdash \mathtt{lookup}\ \bar{l}\ \mathtt{as}\ \bar{\mathsf{z}}\ \mathtt{in}\ \mathsf{Q}[\bar{l}/\&\bar{\mathsf{z}}] : \mathtt{Prop}]\!](\delta; (E'|_\phi, S'_{E'(\phi)}); \vartheta'')$$

where $\vartheta'' = \vartheta[\mathsf{d} \mapsto S''(E'(\mathsf{x})), \bar{l} \mapsto \bar{l}_z]$. □

4 Examples

Proof outline of append

```
public static Node⟨X⟩ append⟨X⟩(Node⟨X⟩ front, Node⟨X⟩ tail) {
  Node⟨X⟩ tmp, tmp2;
  { list(front, xs, P) ∗ list(tail, ys, P) }
  if(front == null) {
    { front = null ∧ list(front, xs, P) ∗ list(tail, ys, P) }
    { xs = [] ∧ list(tail, ys, P) }
    { list(tail, xs@ys, P) }
    return tail;
    { r. list(r, xs@ys, P) }
  } else {
    { front != null ∧ list(front, xs, P) ∗ list(tail, ys, P) }
    { ∃v, xs', n, x. xs = v::xs' ∧ front.next ↦ n ∗ front.item ↦ x
        ∗ P(x, v) ∗ list(n, xs', P) ∗ list(tail, ys, P) }
    tmp2 = front.next;
    { ∃v, xs', x. xs = v::xs' ∧ front.next ↦ tmp2 ∗ front.item ↦ x
        ∗ P(x, v) ∗ list(tmp2, xs', P) ∗ list(tail, ys, P) }
    tmp = append⟨X⟩(tmp2, tail);
    { ∃v, xs', x. xs = v::xs' ∧ front.next ↦ tmp2 ∗ front.item ↦ x
        ∗ P(x, v) ∗ list(tmp, xs'@ys, P) }
    front.next = tmp;
    { ∃v, xs', x. xs = v::xs' ∧ front.next ↦ tmp ∗ front.item ↦ x
        ∗ P(x, v) ∗ list(tmp, xs'@ys, P) }
    { list(front, xs@ys, P) }
    return front;
    { r. list(r, xs@ys, P) }
  }
}
```

Chapter 2

State, sharing and concurrency in C♯

Modular Reasoning about Separation of Concurrent Data Structures

Kasper Svendsen[1], Lars Birkedal[1], and Matthew Parkinson[2]

[1] IT University of Copenhagen, {kasv,birkedal}@itu.dk
[2] Microsoft Research Cambridge, mattpark@microsoft.com

Abstract. In a concurrent setting, the usage protocol of standard separation logic specifications are not refinable by clients, because standard specifications abstract all information about potential interleavings. This breaks modularity, as libraries cannot be verified in isolation, since the appropriate specification depends on how clients intend to use the library.

In this paper we propose a new logic and a new style of specification for thread-safe concurrent data structures. Our specifications allow clients to refine usage protocols and associate ownership of additional resources with instances of these data structures.

1 Introduction

Why? One of the challenges of specifying the abstract behavior of a library is that the appropriate specification depends on the context in which the library is going to be used. Consider the case of simple bag library with operations to push and pop elements from the bag. In a sequential setting the standard separation logic specification is:

$$\{\mathsf{bag}_e(\mathsf{x},\mathsf{X})\}\ \mathsf{x.Push(y)}\ \{\mathsf{bag}_e(\mathsf{x},\mathsf{X}\cup\{\mathsf{y}\})\}$$
$$\{\mathsf{bag}_e(\mathsf{x},\mathsf{X})\}\ \ \mathsf{x.Pop()}\ \ \{\mathsf{ret}.\ (\mathsf{X}=\emptyset\wedge\mathsf{ret}=\mathsf{null}\wedge\mathsf{bag}_e(\mathsf{x},\mathsf{X}))\vee (\exists\mathsf{Y}.\ \mathsf{X}=\mathsf{Y}\cup\{\mathsf{ret}\}\wedge\mathsf{bag}_e(\mathsf{x},\mathsf{Y}))\}$$
$$\mathsf{bag}_e(\mathsf{x},\mathsf{X})*\mathsf{bag}_e(\mathsf{x},\mathsf{Y})\Rightarrow\bot$$

Here bag_e is an abstract predicate, i.e., implicitly existentially quantified, so that clients cannot depend on its definition [2], x is a reference to a bag object, and X and Y range over multisets of elements. The implication in the third line expresses that the bag_e predicate cannot be duplicated. Hence this specification enforces that clients follow a strict usage protocol, with a single exclusive owner of the bag object. On the other hand, this specification allows the owner of the bag to track the exact contents of the bag. In other words, $\mathsf{bag}_e(\mathsf{x},\mathsf{X})$ asserts full ownership of the bag and that the bag contains exactly the objects in the multiset X.

Now consider a client of the bag library and suppose this client wants to implement a bag of independent tasks scheduled for execution. This client might not care about the exact contents of the bag, only that each task in the bag

owns the resources necessary to perform its task. In addition, this client might wish to share the bag to allow multiple users to schedule tasks for execution. Thus this client might prefer the following specification for shared bags:

$$\{\mathsf{bag}_s(\mathsf{x},\mathsf{P}) * \mathsf{P}(\mathsf{y})\}\ \mathsf{x.Push(y)}\ \{\mathsf{bag}_s(\mathsf{x},\mathsf{P})\}$$
$$\{\mathsf{bag}_s(\mathsf{x},\mathsf{P})\}\ \ \mathsf{x.Pop()}\ \ \{\mathsf{ret}.\ \mathsf{bag}_s(\mathsf{x},\mathsf{P}) * (\mathsf{ret} = \mathsf{null}\ \lor\ \mathsf{P}(\mathsf{ret}))\}$$
$$\mathsf{bag}_s(\mathsf{x},\mathsf{P}) \Rightarrow \mathsf{bag}_s(\mathsf{x},\mathsf{P}) * \mathsf{bag}_s(\mathsf{x},\mathsf{P})$$

This specification allows more sharing, but it does not track the exact contents of the bag. Instead, it allows clients to associate additional resources with each element of the bag using the P predicate, and to freely share the bag as expressed by the implication in the third line. Clients thus transfer $\mathsf{P}(\mathsf{y})$ to the bag when pushing y, and receive $\mathsf{P}(\mathsf{ret})$ from the bag, when pop returns a non-null element.

In a sequential first-order setting without reentrancy, the standard separation logic specification suffices. Using techniques from fictional separation logic [11], clients can refine the standard specification to allow the additional sharing of the shared bag specification. However, in a concurrent setting, it is easy to come up with a non-thread-safe implementation (without synchronization), that satisfies the standard specification (as it enforces a single exclusive owner), but not the shared bag specification. Hence, in a higher-order concurrent setting with reentrancy, this type of refinement is unsound!

What? The key challenge is to provide a logic that enables clients to refine the specifications to their requirements in a concurrent setting. In this paper we propose such a logic, called Higher-Order Concurrent Abstract Predicates (HOCAP), and a new style of specification for thread-safe concurrent data structures.[3] This style of specification allows clients to refine the usage protocol and associate ownership of additional resources with instances of the data structure, in a concurrent higher-order setting.

How? Observe first that while it is not sound to refine specifications to allow *more sharing* in a concurrent setting, it is sound to refine specifications to permit *less sharing*. Thus we will start with a weak specification that allows unrestricted sharing of instances of the data structure, and then let clients refine this specification as needed.

To reason about sharing we partition the state into *regions*, with *protocols* governing how the state in each region is allowed to evolve, following earlier work on concurrent abstract predicates [5]. Our new program logic, HOCAP, also uses *phantom fields* – a logical construct akin to auxiliary variables, that only occur in the logic.

To support abstract refinement of library specifications, we propose to verify the implementation using a region to share the concrete state of the implementation, with a fixed protocol that *relates* the concrete state of the implementation

[3] We consider a concurrent data structure thread-safe if each of its methods has one *or more* synchronization points, where the abstract effects of the method appear to take affect. See Related Work for a discussion of the relation to linearizability.

with an abstract description of the state of the data structure. To refine this specification, clients define a region of their own, with a protocol on the *abstract state* of the data structure. For soundness, these two regions must evolve in lock-step and *synchronize* when the abstract state changes (in synchronization points). We do so by giving each region a half permission to a shared phantom field; synchronization can then be enforced since updating a phantom field requires full permission. Half permissions have previously been used to synchronize local and shared state [14]; here we are using it to synchronize two shared regions.

For the bag example, we introduce a phantom field `cont` that contains the abstract state of the bag: a multiset of references to the elements in the bag. The bag constructor also returns a half permission to the phantom field `cont`:

$$\frac{}{\{\mathsf{emp}\}\mathsf{new\ Bag()}\{\mathsf{ret.\ bag(ret)} * \mathsf{ret}_{\mathsf{cont}} \overset{1/2}{\longmapsto} \emptyset\}}$$

Here $\mathsf{ret}_{\mathsf{cont}} \overset{1/2}{\longmapsto} \emptyset$ asserts partial ownership of the phantom `cont` field. Since the client obtains half the `cont` permission upon calling the constructor, the library cannot update the `cont` field on its own. To allow the library to update `cont` in synchronization points, we therefore transfer the library's half-permission to the client and require the client to update the phantom field with the abstract effects of the method, and then transfer a half-permission back to the library. When the client updates the phantom field, the client is forced to prove that the abstract effects of the method is permitted by whatever protocols the client may have imposed on the abstract state.

We express the update to the phantom `cont` field using a *view-shift* [4]. Conceptually, a view-shift corresponds to a step in the instrumented semantics that does not change the concrete machine state. View-shifts, written $\mathsf{P} \sqsubseteq \mathsf{Q}$, thus generalize assertion implication by allowing updates to phantom fields (given sufficient permission) and ownership transfer between the local state and shared regions.

The bag push method thus requires the client to provide a view-shift, to update the abstract state from X to $\mathsf{X} \cup \{\mathsf{y}\}$ in the synchronization point:

$$\frac{\forall \mathsf{X}.\ \mathsf{x}_{\mathsf{cont}} \overset{1/2}{\longmapsto} \mathsf{X} * \mathsf{P} \sqsubseteq \mathsf{x}_{\mathsf{cont}} \overset{1/2}{\longmapsto} \mathsf{X} \cup \{\mathsf{y}\} * \mathsf{Q}}{\{\mathsf{bag(x)} * \mathsf{P}\}\mathsf{x.Push(y)}\{\mathsf{bag(x)} * \mathsf{Q}\}}$$

Here, P and Q are universally quantified and thus picked by the client. Hence, the client can use P and Q to perform further updates of the instrumented state in the synchronization point and relate the new abstract state with its local state. We thus refer to P and Q as synchronization pre- and post-conditions.

Likewise, the bag pop operation requires two view-shifts; one, in case the bag is empty in the synchronization point, and another, in case the bag is non-empty in the synchronization point:

$$\frac{\begin{array}{c}\mathsf{x}_{\mathsf{cont}} \overset{1/2}{\longmapsto} \emptyset * \mathsf{P} \sqsubseteq \mathsf{x}_{\mathsf{cont}} \overset{1/2}{\longmapsto} \emptyset * \mathsf{Q(null)} \\ \forall \mathsf{X}.\ \forall \mathsf{y}.\ \mathsf{x}_{\mathsf{cont}} \overset{1/2}{\longmapsto} \mathsf{X} \cup \{\mathsf{y}\} * \mathsf{P} \sqsubseteq \mathsf{x}_{\mathsf{cont}} \overset{1/2}{\longmapsto} \mathsf{X} * \mathsf{Q(y)}\end{array}}{\{\mathsf{bag(x)} * \mathsf{P}\}\mathsf{x.Pop()}\{\mathsf{bag(x)} * \mathsf{Q(ret)}\}}$$

Finally, the bag predicate is freely duplicable:

$$\mathsf{bag}(x) \Rightarrow \mathsf{bag}(x) * \mathsf{bag}(x)$$

Note that since P and Q are universally quantified — our logic is *higher order* — the client could potentially pick instantiations referring to the library's region, thus introducing self-referential region assertions. To prevent this, we introduce a notion of *region type* and a notion of *support*, as an over-approximation of the types of regions a given assertion refers to. Our formal bag specification (presented in Section 3) thus impose support restrictions on P and Q to ensure the client does not introduce self-referential region assertions.

We also use region types to break a possible circularity in the model caused by higher-order protocols. In particular, instead of assigning protocols to individual regions, we assign parameterized protocols to region types. This allows us to reason about higher-order protocols that refer to the region types – and thus, implicitly, the protocol – of other regions. We show that this well-behaved subset of higher-order protocols, called *state-independent* protocols, suffices for sophisticated libraries, such as the Joins library [16].

To summarize, our new logic and specification methodology allows clients to refine the usage protocol of the bag. It also allows clients to transfer ownership of resources to the bag, by transferring them to a client region synchronized with the abstract state of the bag.

Related work. Jacobs and Piessens introduced the idea of parameterizing the specification of concurrent methods with ghost code, to be executed in synchronization points [10]. Here we build on their idea, but use a much stronger logic based on CAP [5], to address the two main problems with their approach.

Instead of regions with protocols, Jacobs and Piessens use ghost objects – data structures built from ghost variables – with handles that represent partial information about the data structure and permissions to modify it. This approach is problematic since ghost objects are entirely first-order; while they can refer to other ghost objects, they cannot take ownership of state. Hence, they cannot derive the shared bag specification, as it explicitly allows clients to associate additional resources with each element in the bag. Instead, they let the client create the synchronization primitive (locks in their examples) protecting the concurrent data structure. This allows the client to pick a lock invariant containing both the state of the concurrent data structure and any additional resources the client may have associated with the data structure. But this breaks abstraction completely; in particular, it exposes internal implementation details to the client (the synchronization primitive used) and it requires the client to reprove the shared bag specification every time it is needed. We solve this problem using higher-order protocols.

The other main problem with the approach in [10] is a lack of support for reasoning about ghost objects. Jacobs and Piessens provide a single instance – ghost bags – that has been verified directly in the semantics. Any further ghost objects have to be defined in terms of ghost bags or proven directly in the semantics. It is unclear how expressive these ghost objects are; for instance,

whether they can express monotonic state changes. Here, we address this issue by providing a generic logic for reasoning about CAP.

CAP was designed to verify concurrent data structures [5]. However, the original specifications and proofs are non-modular in the sense that implementations have been verified against unrefinable specifications with fixed usage protocols.

Recently, Dodds et. al. introduced a higher-order variant of CAP to give a generic specification for a library for deterministic parallelism [6]. While their proofs make explicit use of nested region assertions and higher-order protocols, the authors failed to recognize the semantic difficulties these features introduce. Consequently, their reasoning is unsound. In particular, their higher-order representation predicates are not stable. (See Appendix E for concrete counterexamples.)

Another approach for achieving modular reasoning is to prove concurrent implementations to be contextual refinements of coarse-grained counterparts – thus taking the coarse-grained counterparts as specifications. Previous efforts for proving such contextual refinements have mostly focused on indirect proofs through a linearizability property on traces of concurrent libraries [9, 7]. So far, this approach lacks support for transfer of ownership of resources between client and library. More recently, there has been work on proving such contextual refinements directly, using logical relations [21]. Unless combined with a program logic, both of these approaches restrict all reasoning to statements about contextual refinement or contextual equivalence. As our approach demonstrates, if a Hoare-style specification is what we are ultimately interested in, then contextual refinement is unnecessary; what we really want is a generic specification that is refinable by clients.

Conceptually, linearizability aims to provide a fiction of atomicity to clients of concurrent libraries. Our approach does not. Instead, we aim to allow clients to reason about changes of the abstract state in synchronization points *inside* concurrent libraries. To illustrate the distinction, consider an extension of the bag library with a `Push2(x, y)` method that takes two elements and pushes them one at a time (i.e., with the implementation `Push(x); Push(y)`). This method is not linearizable, as it has two synchronization points. However, it still has a natural specification expressed in terms of two view-shifts, one for each synchronization point:

$$\frac{\forall \mathsf{X}.\ \mathsf{x}_{\mathsf{cont}} \overset{1/2}{\longmapsto} \mathsf{X} * \mathsf{P} \sqsubseteq \mathsf{x}_{\mathsf{cont}} \overset{1/2}{\longmapsto} \mathsf{X} \cup \{\mathsf{y}\} * \mathsf{Q} \qquad \forall \mathsf{X}.\ \mathsf{x}_{\mathsf{cont}} \overset{1/2}{\longmapsto} \mathsf{X} * \mathsf{Q} \sqsubseteq \mathsf{x}_{\mathsf{cont}} \overset{1/2}{\longmapsto} \mathsf{X} \cup \{\mathsf{z}\} * \mathsf{R}}{\{\mathsf{bag}(\mathsf{x}) * \mathsf{P}\}\mathsf{x}.\mathsf{Push2}(\mathsf{y},\mathsf{z})\{\mathsf{bag}(\mathsf{x}) * \mathsf{R}\}}$$

From this specification, a client can derive a natural shared bag specification:

$$\{\mathsf{bag}_s(\mathsf{x}, \mathsf{P}) * \mathsf{P}(\mathsf{y}) * \mathsf{P}(\mathsf{z})\}\mathsf{x}.\mathsf{Push2}(\mathsf{y},\ \mathsf{z})\{\mathsf{bag}_s(\mathsf{x}, \mathsf{P})\}$$

Contributions. We propose a new style of specification for thread-safe concurrent data structures. Using protocol synchronization, this style of specification

allows clients to refine the usage protocol of concurrent data structures. Moreover, using nested region assertions and state-independent higher-order protocols, our specification style allows clients to associate additional resources with the data structure.

Technically, we realize the ideas by developing HOCAP, a higher-order separation logic for a subset of C♯ featuring named delegates and fork concurrency. The logic allows two or more protocols to be synchronized and evolve in lock-step. In addition, we support nested region assertions, state-independent higher-order protocols, and guarded recursive assertions. We present a step-indexed model of the logic and use it to prove the logic sound. We emphasize that unlike earlier versions of CAP, our logic includes sufficient proof rules for carrying out all proofs (including stability proofs) of examples *in the logic*, i.e., without passing to the semantics.

Lastly, we demonstrate the power and utility of the logic by verifying a library for executing tasks in parallel, based on Doug Lea's Fork/Join framework [12]. We have also used to logic to specify and verify the Joins library [16] and clients thereof, but that will be described in a separate paper.

2 The logic

Our logic is a general program logic for a subset of C♯, featuring delegates referring to named methods[4] and an atomic compare-and-swap statement. New threads are allocated via a fork statement that forks a delegate. Each thread has a private stack, but all threads share a common heap. We use an interleaving semantics.

The specification logic is an intuitionistic higher-order logic over a simply typed term language, and the assertion logic an intutionistic higher-order separation logic over the same simply typed term language. Types are closed under the usual type constructors, $\rightarrow$, $\times$, and $+$. Basic types include the type of assertions, Prop, the type of specifications, Spec, the type of C♯ values, Val, and the type of fractional permissions, Perm.

2.1 Concurrent Abstract Predicates

Recall that the basic idea behind CAP is to provide an abstraction of possible interference from concurrently executing threads, by partitioning the state into regions, with protocols governing how the state in each region is allowed to evolve. Requiring all assertions to be *stable* – i.e., closed under protocols – and proving all specifications with respect to arbitrary stable frames, then achieves thread-local reasoning about shared mutable state.

[4] Anonymous delegates in C♯ may capture the *l*-values of free variables and hence the semantics and logic for anonymous methods is non-trivial, see our earlier paper [19]. Those semantic issues are orthogonal to what we discuss in the present paper and hence we omit anonymous delegates here.

Following earlier work on CAP [5], we use a shared region assertion, written $\boxed{\mathsf{P}}^{\mathsf{r,t,a}}$, which asserts that r is a region and that the resources in region r satisfies the assertion P. Unlike earlier versions, the region assertion is also annotated with a region type t and a protocol argument a, since we assign parameterized protocols to region types instead of regions, as mentioned above. Region assertions are freely duplicable and thus satisfy,

$$\boxed{\mathsf{P}}^{\mathsf{r,t,a}} \Leftrightarrow \boxed{\mathsf{P}}^{\mathsf{r,t,a}} * \boxed{\mathsf{P}}^{\mathsf{r,t,a}} \tag{1}$$

Protocols consist of *named actions* and updates to a shared region requires *ownership* of a named action justifying the update. Protocols are specified using protocol assertions, written $\mathsf{protocol}(\mathsf{t}, \mathsf{I})$. Here t is a region type and I is a parametric protocol. We use the following notation for a parametric protocol I with parameter a and named actions α_1, ..., α_n:

$$\mathsf{I}(\mathsf{a}) = (\alpha_1 : (\Delta_1).\ \mathsf{P}_1 \rightsquigarrow \mathsf{Q}_1; \cdots ; \alpha_n : (\Delta_n).\ \mathsf{P}_n \rightsquigarrow \mathsf{Q}_n)$$

Here Δ_i is a context of logical variables relating the action precondition P_i with the action post-condition Q_i. The action α_i thus allows updates from states satisfying P_i to states satisfying Q_i. We use $\mathsf{I}(\mathsf{a})[\alpha_i]$ to refer to the definition of the α_i action in protocol I applied to argument a. Hence, $\mathsf{I}(\mathsf{a})[\alpha_i] = (\Delta_i).\ \mathsf{P}_i \rightsquigarrow \mathsf{Q}_i$. We use $\boxed{\mathsf{P}}^{\mathsf{r,t,a}}_{\mathsf{I}}$ as shorthand for $\boxed{\mathsf{P}}^{\mathsf{r,t,a}} * \mathsf{protocol}(\mathsf{t}, \mathsf{I})$.

We can distinguish different client roles in protocols through ownership of named actions. An action assertion $[\alpha]^{\mathsf{r}}_{\pi}$ asserts fractional ownership of the named action α on region r with fraction π. Fractions are used to allow multiple clients to use the same action. We can split or reassemble action assertions using the following property,

$$[\alpha]^{\mathsf{r}}_{\mathsf{p+q}} \Leftrightarrow [\alpha]^{\mathsf{r}}_{\mathsf{p}} * [\alpha]^{\mathsf{r}}_{\mathsf{q}} \tag{2}$$

where $\mathsf{p}, \mathsf{q}, \mathsf{p}+\mathsf{q}$ are terms of type Perm – permissions in $(0, 1]$.

An assertion p is *stable* if it is closed under interference from the environment. In the absence of self-referential region assertions and higher-order protocols, the region assertion, $\boxed{\mathsf{P}}^{\mathsf{r,t,a}}_{\mathsf{I}}$ is stable if P is closed under all $\mathsf{I}(\mathsf{a})$ actions:[5]

$$\forall \tilde{\mathsf{y}}.\ \mathsf{valid}(\mathsf{P} \wedge \mathsf{P}_i(\tilde{\mathsf{y}}) \Rightarrow \bot) \vee \mathsf{valid}(\mathsf{Q}_i(\tilde{y}) \Rightarrow \mathsf{P})$$

for all i, where $\mathsf{I}(\mathsf{a})[\alpha_i] = (\tilde{\mathsf{x}}).\ \mathsf{P}_i(\tilde{\mathsf{x}}) \rightsquigarrow \mathsf{Q}_i(\tilde{\mathsf{x}})$.

Example. To illustrate reasoning about sharing, consider a counter with read and increment methods. Since the count can only be increased, this counter satisfies the specification of a monotonic counter [15]:

$$\begin{array}{l}\{\mathsf{counter}(\mathsf{x}, \mathsf{n})\}\ \mathsf{x.Increment}()\ \{\mathsf{counter}(\mathsf{x}, \mathsf{n}+1)\}\\ \{\mathsf{counter}(\mathsf{x}, \mathsf{n})\} \quad \mathsf{x.Read}() \quad \{\mathsf{ret}.\ \mathsf{counter}(\mathsf{x}, \mathsf{ret}) * \mathsf{n} \leq \mathsf{ret}\}\\ \mathsf{counter}(\mathsf{x}, \mathsf{n}) \Rightarrow \mathsf{counter}(\mathsf{x}, \mathsf{n}) * \mathsf{counter}(\mathsf{x}, \mathsf{n})\end{array}$$

[5] This is a formula in the specification logic; P and Q are assertions and for an assertion P, $\mathsf{valid}(\mathsf{P})$ is the specification that expresses that P is valid in the assertion logic.

Here $\mathsf{counter}(\mathsf{x}, \mathsf{n})$ asserts that n is a lower-bound on the current count. Hence we expect that this predicate can be freely duplicated, as expressed by the third line above.

To verify a counter implementation against this specification, we place the current count in a shared region, with a protocol that allows the current count to be increased. Assertions about lower bounds are thus invariant under the protocol. If the counter implementation maintains the current count in field count, then we can specify the counter protocol as follows:

$$\mathsf{counter}(\mathsf{x}, \mathsf{n}) \stackrel{\text{def}}{=} \exists \mathsf{r}, \pi.\ [\textsc{incr}]^{\mathsf{r}}_{\pi} * \boxed{\exists \mathsf{m}.\ \mathsf{n} \leq \mathsf{m} * \mathsf{x}.\mathsf{count} \mapsto \mathsf{m}}^{\mathsf{r},\mathsf{Counter},\mathsf{x}}$$

where I is a parametric protocol with parameter x and a single action INCR, that allows the count field of x to be increased:

$$\mathsf{I}(\mathsf{x}) = (\textsc{incr} : (\mathsf{m}, \mathsf{k} : \mathsf{N}).\ \mathsf{x}.\mathsf{count} \mapsto \mathsf{m} * \mathsf{m} \leq \mathsf{k} \rightsquigarrow \mathsf{x}.\mathsf{count} \mapsto \mathsf{k})$$

Here we have used a fixed region type $\mathsf{Counter}$ for the counter region r. Since fractional permissions can always be split (2), and region assertions always duplicated (1), it follows that $\mathsf{counter}(\mathsf{x}, \mathsf{n}) \Rightarrow \mathsf{counter}(\mathsf{x}, \mathsf{n}) * \mathsf{counter}(\mathsf{x}, \mathsf{n})$, as required by the specification. Since the shared region assertion in $\mathsf{counter}(\mathsf{x}, \mathsf{n})$ contains no self-referential region assertions or higher-order protocols, to prove it stable, it suffices to show that,

$$\begin{aligned}\forall \mathsf{m}, \mathsf{k}.\ \ & \mathsf{valid}((\exists \mathsf{m} : \mathsf{N}.\ \mathsf{n} \leq \mathsf{m} * \mathsf{x}.\mathsf{count} \mapsto \mathsf{m}) \wedge (\mathsf{x}.\mathsf{count} \mapsto \mathsf{m} * \mathsf{m} \leq \mathsf{k}) \Rightarrow \bot) \vee \\ & \mathsf{valid}(\mathsf{x}.\mathsf{count} \mapsto \mathsf{k} \Rightarrow (\exists \mathsf{m} : \mathsf{N}.\ \mathsf{n} \leq \mathsf{m} * \mathsf{x}.\mathsf{count} \mapsto \mathsf{m}))\end{aligned}$$

This follows easily by case analysis on $\mathsf{n} \leq \mathsf{k}$.[6] Lastly, to verify the implementation of $\mathsf{Increment}$ and Read, we have to prove they satisfy the protocol, namely that they do not decrease the current count. This is easy.

2.2 Higher-order Concurrent Abstract Predicates

As the above example illustrates, we can use CAP to reason about a shared counter by imposing a protocol on the shared count field. Since this is a protocol on a primitive resource (the count field), first-order CAP suffices. To reason about examples, such as the shared bag, which associates ownership of general resources – through the P predicate – with a shared bag, we need Higher-Order CAP. In particular, to define the bag_s predicate requires region and protocol assertions containing the predicate variable P.

To support modular reasoning about region and protocol assertions containing predicate and assertion variables, ideally, we want to treat predicate and assertion variables as black boxes. For instance, consider the assertion,

$$\mathsf{Q} \stackrel{\text{def}}{=} \boxed{\mathsf{P}}^{\mathsf{r},\mathsf{t},-} * \mathsf{protocol}(\mathsf{t}, \mathsf{I}) \tag{3}$$

[6] As usual, $\leq$ is a decidable predicate so we are free to do case analysis on $\mathsf{n} \leq \mathsf{k}$.

where I is the parametric protocol $\mathsf{I}(-) = (\tau : \mathsf{P} \rightsquigarrow \mathsf{P})$ expressed in terms of the assertion variable P. Treating P as a black box, Q is clearly stable if P is stable, as Q asserts that P holds of the resources in region r, which is clearly closed under the protocol I. However, in general P could itself be instantiated with region and protocol assertions, introducing the possibility of *self-referential region assertions* and turning I into a *higher-order protocol*. This makes reasoning significantly more challenging. In particular, some self-referential region assertions do not admit modular stability proofs, as there exists instantiations of P with stable assertions for which Q is not stable. Furthermore, higher-order protocols introduce a circularity in the definition of the model.

Self-referential region assertions. To see how self-referential region assertions can break the modularity of stability proofs, consider assertion p below:

$$\mathsf{P} \stackrel{\text{def}}{=} \mathsf{x} \mapsto 0 * \boxed{\mathsf{y} \mapsto 0}^{\mathsf{r}',\mathsf{t}',-} * \mathsf{protocol}(\mathsf{t}', \mathsf{J}),$$

where J is the protocol with a single α action that allows the y variable to be changed from 0 to 1, provided region r owns variable x and x is zero:

$$\mathsf{J}(-) = \left(\alpha : \boxed{\mathsf{x} \mapsto 0}^{\mathsf{r},\mathsf{t},-} * \mathsf{y} \mapsto 0 \rightsquigarrow \boxed{\mathsf{x} \mapsto 0}^{\mathsf{r},\mathsf{t},-} * \mathsf{y} \mapsto 1\right)$$

Then P is stable, because P asserts full ownership of the x variable, ensuring that the environment cannot perform the α action, as x cannot also be owned by region r. However, the region assertion q defined above is not stable when instantiated with this P, as $\boxed{\mathsf{P}}^{\mathsf{r},\mathsf{t},-}$ asserts that region r *does* own x, thus allowing the environment to perform the α action. As this example illustrates, some self-referential region assertions thus do not admit modular stability proofs. A similar problem occurs when reasoning about atomic updates to shared regions.

Support. To ensure modular reasoning about stability and atomic updates to shared regions, we require clients to explicitly prove that their instantiations of predicate variables do not introduce self-referential region assertions. To facilitate these proofs, we introduce a notion of support, which gives an over-approximation of the types of regions a given assertion refers to.

An assertion P is supported by a set of region types A, if P is invariant under arbitrary changes to the state and protocol of any region of a region type not in A. To support modular reasoning about hierarchies of concurrent libraries, instead of reasoning directly in terms of sets of regions, we introduce a partial order on region types and reason in terms of upwards-closed sets of region types. More formally, we introduce a new type, RType, of region types with a partial order $\leq$: RType $\times$ RType $\to$ Spec, with a bottom element $\bot$: RType and finite meets. We say that an assertion P is dependent on region type t if it is supported by the set of region types greater than or equal to t. We introduce two new specification assertions, $\mathsf{dep}, \mathsf{indep}$: RType $\times$ Prop $\to$ Spec for asserting that an assertion is dependent and independent of a given region type, respectively. Figure 2 in Appendix A contains a set of natural inference rules for dep and

indep, for instance, one expressing that if P is dependent on region type $\mathtt{t}_1$, then $\boxed{\mathsf{P}}^{\mathsf{r},\mathtt{t}_2,\mathsf{a}}$ is dependent on the greatest lower bound, of $\mathtt{t}_1$ and $\mathtt{t}_2$.

Whenever we reason about region assertions, $\boxed{\mathsf{P}}^{\mathsf{r},\mathtt{t},\mathsf{a}}$ we thus require that P is independent of the region type $\mathtt{t}$. This excludes self-referential region assertions through protocols (such as in (3)), and through nested region assertions (such as $\boxed{\boxed{\mathsf{P}}^{\mathsf{r},\mathtt{t},\mathsf{a}}}^{\mathsf{r},\mathtt{t},\mathsf{a}}$).

Stability. General higher-order protocols would introduce a circularity in the definition of the model. We break this circularity by exploiting the indirection of region types – i.e., that we assign protocols to region types instead of individual regions. This allows us to support protocols with assertions about the region types of regions, but without assertions about the protocols assigned to those region types. Technically, we enforce this restriction by ignoring protocol assertions in action pre- and post-conditions when interpreting protocols. The parameterized higher-order protocol I,

$$\mathsf{I}(\mathsf{x}) = (\mathsf{x} \mapsto 0 * \mathsf{protocol}(\mathtt{t}, \mathsf{J}) \rightsquigarrow \mathsf{x} \mapsto 1 * \mathsf{protocol}(\mathtt{t}, \mathsf{J}))$$

is thus interpreted as $\mathsf{I}(\mathsf{x}) = (\mathsf{x} \mapsto 0 \rightsquigarrow \mathsf{x} \mapsto 1)$. The interpretation simply ignores the $\mathsf{protocol}(\mathtt{t}, \mathsf{J})$ assertion (See definition of *act* in the technical report).

In the absence of self-referential region assertions, a region assertion $\boxed{\mathsf{P}}_{\mathsf{I}}^{\mathsf{r},\mathtt{t},\mathsf{a}}$ is stable under the α action, if P is closed under the action pre- and post-condition of the α action of $\mathsf{I}(\mathsf{a})$ and I is a first-order protocol. If I is a higher-order protocol, then the assertion $\boxed{\mathsf{P}}_{\mathsf{I}}^{\mathsf{r},\mathtt{t},\mathsf{a}}$ is stable under the α action, if P is closed under the action pre- and post-condition of the α action of $\mathsf{I}(\mathsf{a})$ and p is also *protocol-pure*. We thus have the following proof-rule for stability:

$$\frac{\begin{array}{c}\mathsf{I}(\mathsf{a})[\alpha] = (\tilde{\mathsf{x}}).\mathsf{I}_p(\tilde{\mathsf{x}}) \rightsquigarrow \mathsf{I}_q(\tilde{\mathsf{x}}) \qquad \forall \tilde{\mathsf{x}}.\ \mathsf{valid}(\mathsf{P} \wedge \mathsf{I}_p(\tilde{\mathsf{x}}) \Rightarrow \bot) \vee \mathsf{valid}(\mathsf{I}_q(\tilde{\mathsf{x}}) \Rightarrow \mathsf{P}) \\ \mathsf{indep}_{\mathtt{t}}(\mathsf{P}) \qquad \mathsf{indep}_{\mathtt{t}}(\mathsf{Q}) \qquad \mathsf{stable}(\mathsf{P} * \mathsf{Q}) \qquad \mathsf{pure}_{\mathsf{protocol}}(\mathsf{P}) \qquad \mathsf{pure}_{\mathsf{state}}(\mathsf{Q})\end{array}}{\mathsf{stable}^{\mathsf{r}}_{\alpha}\left(\boxed{\mathsf{P}}_{\mathsf{I}}^{\mathsf{r},\mathtt{t},\mathsf{a}} * \mathsf{Q}\right)}\ \mathrm{SA}$$

Here $\mathsf{pure}_{\mathsf{protocol}}$ and $\mathsf{pure}_{\mathsf{state}}$ are propositions in the specification logic; $\mathsf{pure}_{\mathsf{protocol}}(\mathsf{P})$ expresses that P is invariant under any changes to protocols and $\mathsf{pure}_{\mathsf{state}}(\mathsf{P})$ expresses that P is invariant under any change to the local or shared state. Figures 3 and 4 in Appendix A give inference rules for proving $\mathsf{pure}_{\mathsf{protocol}}$ and $\mathsf{pure}_{\mathsf{state}}$ assertions. The SA proof-rule thus allows us to prove stability of region assertions, by first "pulling out" any protocol assertions, Q, from the region assertion. We say that an assertion is *expressible using state-independent protocols* if the protocol assertions can be "pulled out" in this sense. Formally,

$$\mathsf{sip} \stackrel{\text{def}}{=} \lambda \mathsf{P} : \mathsf{Prop}.\ \exists \mathsf{Q}, \mathsf{R} : \mathsf{Prop}.\ \mathsf{valid}(\mathsf{P} \Leftrightarrow \mathsf{Q} * \mathsf{R}) \wedge \mathsf{pure}_{\mathsf{protocol}}(\mathsf{Q}) \wedge \mathsf{pure}_{\mathsf{state}}(\mathsf{R})$$

In particular, if $\mathsf{P} \Leftrightarrow \mathsf{Q} * \mathsf{R}$ and $\mathsf{pure}_{\mathsf{state}}(\mathsf{R})$, then $\boxed{\mathsf{P}}_{\mathsf{I}}^{\mathsf{r},\mathtt{t},\mathsf{a}} \Leftrightarrow \boxed{\mathsf{Q}}_{\mathsf{I}}^{\mathsf{r},\mathtt{t},\mathsf{a}} * \mathsf{R}$. Thus, if $\mathsf{sip}(\mathsf{P})$, then $\boxed{\mathsf{P}}_{\mathsf{I}}^{\mathsf{r},\mathtt{t},\mathsf{a}}$ can be rewritten to a form that satisfies the $\mathsf{pure}_{\mathsf{protocol}}$ premise of the SA rule. Expressibility using state-independent protocols is closed under

conjunction and separating conjunction, but in general not under disjunction or existential quantification. To achieve closure under existential quantification, $\exists \mathsf{x} : \mathsf{X}.\ \mathsf{P}(\mathsf{x})$, we have to impose a stronger restriction on the predicate family P. Namely, P has to be uniformly expressible using state-independent protocols:

$$\mathsf{usip}_{\mathsf{X}} \stackrel{\text{def}}{=} \lambda \mathsf{P} : \mathsf{X} \to \mathsf{Prop}.\ \exists \mathsf{R} : \mathsf{Prop}.\ \exists \mathsf{Q} : \mathsf{X} \to \mathsf{Prop}.\ \mathsf{pure}_{\mathsf{state}}(\mathsf{R}) \wedge \\ \forall \mathsf{x} \in \mathsf{X}.\ (\mathsf{P}(\mathsf{x}) \Leftrightarrow \mathsf{Q}(\mathsf{x}) * \mathsf{R}) \wedge \mathsf{pure}_{\mathsf{protocol}}(\mathsf{Q}(x))$$

Then we have that $\mathsf{usip}_{\mathsf{X}}(\mathsf{P}) \Rightarrow \mathsf{sip}(\exists \mathsf{x} \in \mathsf{X}.\ \mathsf{P}(\mathsf{x}))$.

2.3 View-shifts.

Phantom state. Proofs in Hoare logic often employ auxiliary variables [13], as an abstraction of the history of execution and state. To support this style of reasoning, without changing the formal operational semantics, we instrument our abstract semantics with phantom fields.

We thus extend our logic with a phantom points-to assertion, written $\mathsf{x}_{\mathsf{f}} \overset{\mathsf{p}}{\mapsto} \mathsf{v}$, which asserts partial ownership, with fraction p, of the phantom field f on object x, and that the current value of the phantom field is v.

Phantom fields live in the instrumented state and are thus updated through view-shifts. Updating a phantom field requires full ownership of the field ($\mathsf{x}_{\mathsf{f}} \overset{1}{\mapsto} \mathsf{v}_1 \sqsubseteq_{\perp} \mathsf{x}_{\mathsf{f}} \overset{1}{\mapsto} \mathsf{v}_2$).[7] A fractional phantom field permission can be split and reassembled arbitrarily. As a partial fraction only confers read-only ownership, two partial fractional assertions must agree on the current value of a given phantom field ($\mathsf{x}_{\mathsf{f}} \overset{\mathsf{p}_1}{\mapsto} \mathsf{v}_1 * \mathsf{x}_{\mathsf{f}} \overset{\mathsf{p}_2}{\mapsto} \mathsf{v}_2 \Rightarrow \mathsf{v}_1 = \mathsf{v}_2$). To create a phantom field f we require that the field does not already exist, so that we can take full ownership of the field. We thus require all phantom fields of an object o to be created simultaneously when o is first constructed (in the proof rule for contructors, see the technical report [20]).

Simultaneous updates. To support synchronization of two regions by splitting ownership of a common phantom field, we need to update the value of the phantom field in both regions *simultaneously*. Previous versions of CAP have only supported sequences of *independent* updates to *single* regions. To support synchronization of protocols we thus extend CAP with support for simultaneous updates of *multiple* regions.

We have chosen a semantics that requires that updates of regions have the same action granularity (you cannot have one simultaneous update of two regions, where the update of one region is justified by one action, and the update of the other region is justified by two actions). This is a choice; it simplifies stability proofs, but it means that we must explicitly track which regions that may have been updated by a view-shift. (See Section B in Appendix for examples illustrating this choice.) We thus index the view-shift relation with a region type t. The indexed view-shift relation, $\sqsubseteq_{\mathsf{t}}$, thus describes a *single* update that, in

[7] The view-shift is annotated with the $\perp$ region type; we explain the reason for such annotations on view-shifts in the following.

$$\frac{\mathsf{pure}_{\mathsf{perm}}(\mathsf{P}_1) \qquad \mathsf{indep}_{\mathsf{t}_1 \sqcap \mathsf{t}_2}(\mathsf{P}_1, \mathsf{P}_2, \mathsf{Q}_1, \mathsf{Q}_2) \qquad \mathsf{t}_2 \not\leq \mathsf{t}_1}{\boxed{\mathsf{P}_1}^{\mathsf{r},\mathsf{t}_1,\mathsf{a}}_{\shortparallel} * \mathsf{P}_2 \rightsquigarrow^{\mathsf{r},\mathsf{t}_2} \boxed{\mathsf{Q}_1}^{\mathsf{r},\mathsf{t}_1,\mathsf{a}}_{\shortparallel} * \mathsf{Q}_2 \qquad \mathsf{P}_1 * \mathsf{P}_2 \sqsubseteq_{\mathsf{t}_1 \sqcap \mathsf{t}_2} \mathsf{Q}_1 * \mathsf{Q}_2} \text{ VSNOpen}$$
$$\boxed{\mathsf{P}_1}^{\mathsf{r},\mathsf{t}_1,\mathsf{a}}_{\shortparallel} * \mathsf{P}_2 \sqsubseteq_{\mathsf{t}_2} \boxed{\mathsf{Q}_1}^{\mathsf{r},\mathsf{t}_1,\mathsf{a}}_{\shortparallel} * \mathsf{Q}_2$$

$$\frac{\mathsf{indep}_{\mathsf{t}_1 \sqcap \mathsf{t}_2}(\mathsf{P}_1, \mathsf{P}_2, \mathsf{Q}_1, \mathsf{Q}_2) \qquad \mathsf{t}_2 \not\leq \mathsf{t}_1}{\boxed{\mathsf{P}_1}^{\mathsf{r},\mathsf{t}_1,\mathsf{a}}_{\shortparallel} * \mathsf{P}_2 \rightsquigarrow^{\mathsf{r},\mathsf{t}_2} \boxed{\mathsf{Q}_1}^{\mathsf{r},\mathsf{t}_1,\mathsf{a}}_{\shortparallel} * \mathsf{Q}_2 \qquad \mathsf{P}_1 * \mathsf{P}_2 \sqsubseteq_{\bot} \mathsf{Q}_1 * \mathsf{Q}_2} \text{ VSOpen}$$
$$\boxed{\mathsf{P}_1}^{\mathsf{r},\mathsf{t}_1,\mathsf{a}}_{\shortparallel} * \mathsf{P}_2 \sqsubseteq_{\mathsf{t}_2} \boxed{\mathsf{Q}_1}^{\mathsf{r},\mathsf{t}_1,\mathsf{a}}_{\shortparallel} * \mathsf{Q}_2$$

$$\frac{\mathsf{P} \sqsubseteq_{\mathsf{t}} \mathsf{Q} \qquad \mathsf{stable}(\mathsf{R})}{\mathsf{P} * \mathsf{R} \sqsubseteq_{\mathsf{t}} \mathsf{Q} * \mathsf{R}} \text{ VSFrame} \qquad \frac{\mathsf{P} \sqsubseteq_{\mathsf{t}_1} \mathsf{Q} \qquad \mathsf{t}_1 \leq \mathsf{t}_2}{\mathsf{P} \sqsubseteq_{\mathsf{t}_2} \mathsf{Q}} \text{ VSWeaken}$$

Fig. 1. Selected view-shift proof rules

addition to updating the local state, may update *multiple* shared regions with region types not greater than or equal to t, where each update must be justified by a *single* action. The indexed view-shift relation is thus *not* transitive.

Figure 1 contains a selection of proof rules for view-shifts. The two main rules, VSNOpen and VSOpen, are used to open a region, to allow access to the resources in that shared region. Both rules allow us to open a region and perform a nested view-shift on the contents of that region. This is how we reason about simultaneous updates to multiple regions in the logic). Rule VSNOpen allows the nested view-shift to modify further regions, while VSOpen does not (note the use of region type $\bot$ on the nested view shift in VSOpen). Both rules require a proof the update is possible –

$$\mathsf{P}_1 * \mathsf{P}_2 \sqsubseteq_{\mathsf{t}_1 \sqcap \mathsf{t}_2} \mathsf{Q}_1 * \mathsf{Q}_2 \qquad \text{and} \qquad \mathsf{P}_1 * \mathsf{P}_2 \sqsubseteq_{\bot} \mathsf{Q}_1 * \mathsf{Q}_2,$$

respectively – and a proof that the update is allowed by the protocol, denoted

$$\boxed{\mathsf{P}_1}^{\mathsf{r},\mathsf{t}_1,\mathsf{a}}_{\shortparallel} * \mathsf{P}_2 \rightsquigarrow^{\mathsf{r},\mathsf{t}_2} \boxed{\mathsf{Q}_1}^{\mathsf{r},\mathsf{t}_1,\mathsf{a}}_{\shortparallel} * \mathsf{Q}_2$$

and explained below.

Since actions owned by shared regions cannot be used to perform updates to shared regions, the VSNOpen rule further requires that P_1 does not assert ownership of any local action permissions ($\mathsf{pure}_{\mathsf{perm}}(\mathsf{P}_1)$). This ensures that no local action permissions from P_1 were used to justify any actions performed in the nested view-shift. Since VSOpen does not allow the nested view-shift to update any regions, this restriction is unnecessary for the VSOpen rule.

Update allowed. The update allowed relation, $\mathsf{P} \rightsquigarrow^{\mathsf{r},\mathsf{t}} \mathsf{Q}$, asserts that the update described by P and Q to region r is justified by an action owned by P.

Thus the basic proof rule for the update allowed relation is:

$$\frac{\mathsf{indep}_{\mathsf{t}_2}(\mathsf{P}(\tilde{\mathsf{v}}), \mathsf{Q}(\tilde{\mathsf{v}})) \qquad \mathsf{t}_2 \not\leq \mathsf{t}_1 \qquad \mathsf{I}(\mathsf{a})[\alpha] = (\tilde{\mathsf{x}}).\ \mathsf{P}(\tilde{\mathsf{x}}) \rightsquigarrow \mathsf{Q}(\tilde{\mathsf{x}})}{\boxed{\mathsf{P}(\tilde{\mathsf{v}})}^{\mathsf{r},\mathsf{t}_1,\mathsf{a}}_{\shortparallel} * [\alpha]^{\mathsf{r}}_{\pi} \rightsquigarrow^{\mathsf{r},\mathsf{t}_2} \boxed{\mathsf{Q}(\tilde{\mathsf{v}})}^{\mathsf{r},\mathsf{t}_1,\mathsf{a}}_{\shortparallel} * [\alpha]^{\mathsf{r}}_{\pi}} \text{ UAAct}$$

Since the update allowed relation simply asserts that any update described by P and Q is allowed, it satisfies a slightly non-standard rule of consequence, that allows strengthening of both the pre- and post-condition. From this non-standard rule-of-consequence, it follows that the update allowed relation satisfies the a frame rules that allows arbitrary changes to the context:

$$\frac{\mathsf{P} \Rightarrow \mathsf{P}' \qquad \mathsf{P}' \rightsquigarrow^{\mathsf{r},\mathsf{t}} \mathsf{Q}' \qquad \mathsf{Q} \Rightarrow \mathsf{Q}'}{\mathsf{P} \rightsquigarrow^{\mathsf{r},\mathsf{t}} \mathsf{Q}}\ \textsc{UAConseq} \qquad \frac{\mathsf{P} \rightsquigarrow^{\mathsf{s},\mathsf{t}} \mathsf{Q}}{\mathsf{P} * \mathsf{R}_1 \rightsquigarrow^{\mathsf{s},\mathsf{t}} \mathsf{Q} * \mathsf{R}_2}\ \textsc{UAF}$$

3 Concurrent Bag

We now return to the concurrent bag from the introduction, and show how to formalize the informal specification from the introduction. Next, we show how to derive the two bag specifications from the introduction, using protocol synchronization, nested region assertions, and higher-order protocols.

Specification. In the introduction we proposed a refineable bag specification with phantom variables to force protocol synchronization and with view-shifts to synchronize client and library in synchronization points. In the formal specification we restrict the synchronization pre- and post-conditions, P and Q, using region types, to ensure that the client's instantiation does not introduce self-referential region assertions. Upon creation of new bag instances, the client picks a region type t for that bag instance and the client is then required to prove that all its synchronization pre- and post-conditions are independent of region type t. The formal refinable bag specification is:

$$\frac{}{\{\mathsf{emp}\}\mathsf{new\ Bag}()\{\mathsf{ret}.\ \mathsf{bag}(\mathsf{t},\mathsf{ret}) * \mathsf{ret}_{\mathsf{cont}} \overset{1/2}{\longmapsto} \emptyset\}}$$

$$\frac{\begin{array}{c}\mathsf{stable}(\mathsf{P}) \qquad \mathsf{stable}(\mathsf{Q}) \qquad \mathsf{indep}_{\mathsf{t}}(\mathsf{P}) \qquad \mathsf{indep}_{\mathsf{t}}(\mathsf{Q}) \\ \forall \mathsf{x}.\ \mathsf{x}_{\mathsf{cont}} \overset{1/2}{\longmapsto} \emptyset * \mathsf{P}(\mathsf{x}) \sqsubseteq_{\mathsf{t}} \mathsf{x}_{\mathsf{cont}} \overset{1/2}{\longmapsto} \emptyset * \mathsf{Q}(\mathsf{x},\mathsf{null}) \\ \forall \mathsf{X}.\ \forall \mathsf{x},\mathsf{y}.\ \mathsf{x}_{\mathsf{cont}} \overset{1/2}{\longmapsto} \mathsf{X} \cup \{\mathsf{y}\} * \mathsf{P}(\mathsf{x}) \sqsubseteq_{\mathsf{t}} \mathsf{x}_{\mathsf{cont}} \overset{1/2}{\longmapsto} \mathsf{X} * \mathsf{Q}(\mathsf{x},\mathsf{y})\end{array}}{\{\mathsf{bag}(\mathsf{t},\mathsf{x}) * \mathsf{P}(\mathsf{x})\}\mathsf{x.Pop}()\{\mathsf{bag}(\mathsf{t},\mathsf{x}) * \mathsf{Q}(\mathsf{x},\mathsf{ret})\}}$$

$$\frac{\begin{array}{c}\mathsf{stable}(\mathsf{P}) \qquad \mathsf{stable}(\mathsf{Q}) \qquad \mathsf{indep}_{\mathsf{t}}(\mathsf{P}) \qquad \mathsf{indep}_{\mathsf{t}}(\mathsf{Q}) \\ \forall \mathsf{X}.\ \forall \mathsf{x},\mathsf{y}.\ \mathsf{x}_{\mathsf{cont}} \overset{1/2}{\longmapsto} \mathsf{X} * \mathsf{P}(\mathsf{x},\mathsf{y}) \sqsubseteq_{\mathsf{t}} \mathsf{x}_{\mathsf{cont}} \overset{1/2}{\longmapsto} \mathsf{X} \cup \{\mathsf{y}\} * \mathsf{Q}(\mathsf{x},\mathsf{y})\end{array}}{\{\mathsf{bag}(\mathsf{t},\mathsf{x}) * \mathsf{P}(\mathsf{x},\mathsf{y})\}\mathsf{x.Push}(\mathsf{y})\{\mathsf{bag}(\mathsf{t},\mathsf{x}) * \mathsf{Q}(\mathsf{x},\mathsf{y})\}}$$

$$\frac{}{\mathsf{bag}(\mathsf{t},\mathsf{x}) \Leftrightarrow \mathsf{bag}(\mathsf{t},\mathsf{x}) * \mathsf{bag}(\mathsf{t},\mathsf{x})} \qquad \frac{}{\mathsf{dep}_{\mathsf{t}}(\mathsf{bag}(\mathsf{t},\mathsf{x}))}$$

The $\mathsf{indep}_{\mathsf{t}}$ assumptions on the synchronization pre- and post-conditions ensure that P and Q do not introduce self-referential region assertions. Furthermore, the index on the view-shifts, $\sqsubseteq_{\mathsf{t}}$, ensures that the granularity of actions match between the library and any client protocols.

See Appendix C for a proof outline of a fine-grained bag implementation against this refineable bag specification.

Exclusive owner. We now show how to derive the standard specification with a single exclusive owner. This specification is very simple to derive; we simply let the exclusive owner of the bag keep the $1/2$ permission of the phantom field containing the abstract state of the bag: $\mathsf{bag}_e(\mathsf{t},\mathsf{x},\mathsf{X}) \stackrel{\text{def}}{=} \mathsf{bag}(\mathsf{t},\mathsf{x}) * \mathsf{x}_{\mathsf{cont}} \overset{1/2}{\longmapsto} \mathsf{X}$.

Shared bag. The derivation of the shared bag specification is more interesting, as it uses both protocol synchronization and higher-order protocols. We begin by formalizing the shared bag specification in our logic:

$$\frac{\mathsf{dep}_{\mathsf{r}}(\mathsf{P})}{\mathsf{dep}_{\mathsf{r}\sqcap\mathsf{t}}(\mathsf{bag}(\mathsf{t},\mathsf{x},\mathsf{P}))} \qquad \frac{\mathsf{stable}(\mathsf{P}) \qquad \mathsf{indep}_{\mathsf{t}}(\mathsf{P}) \qquad \mathsf{usip}_{\mathrm{Val}}(\mathsf{P})}{\{\mathsf{emp}\}\mathsf{new\ Bag}()\{\mathsf{ret}.\ \mathsf{bag}(\mathsf{t},\mathsf{ret},\mathsf{P})\}}$$

$$\frac{}{\{\mathsf{bag}_s(\mathsf{t},\mathsf{x},\mathsf{P}) * \mathsf{P}(\mathsf{y})\}\mathsf{x.Push(y)}\{\mathsf{bag}_s(\mathsf{t},\mathsf{x},\mathsf{P})\}}$$

$$\frac{}{\{\mathsf{bag}_s(\mathsf{t},\mathsf{x},\mathsf{P})\}\mathsf{x.Pop()}\{\mathsf{ret}.\ \mathsf{bag}_s(\mathsf{t},\mathsf{x},\mathsf{P}) * (\mathsf{ret} = \mathsf{null}\ \vee\ \mathsf{P}(\mathsf{ret}))\}}$$

$$\frac{}{\mathsf{bag}_s(\mathsf{t},\mathsf{x},\mathsf{P}) \Leftrightarrow \mathsf{bag}_s(\mathsf{t},\mathsf{x},\mathsf{P}) * \mathsf{bag}_s(\mathsf{t},\mathsf{x},\mathsf{P})}$$

This corresponds to the specification from the introduction, except with restrictions on predicate P to ensure it is expressible using state-independent protocols and does not introduce self-referential protocol or region assertions.

With these restrictions on P we can now derive the shared bag specification from our generic specification. The idea is to introduce a new region containing the state associated with each element currently in the bag:

$$\begin{aligned}
\mathsf{bag}_s(\mathsf{t},\mathsf{x},\mathsf{P}) \stackrel{\text{def}}{=}\ & \exists \mathsf{r} : \mathrm{RId}.\ \exists \pi : \mathrm{Perm}.\ \exists \mathsf{t}_1,\mathsf{t}_2 : \mathrm{RType}. \\
& \mathsf{t} \le \mathsf{t}_1 \wedge \mathsf{t} \le \mathsf{t}_2 \wedge \mathsf{t}_1 \not\le \mathsf{t}_2 \wedge \mathsf{t}_2 \not\le \mathsf{t}_1 \wedge \mathsf{indep}_{\mathsf{t}}(\mathsf{P}) \wedge \mathsf{usip}(\mathsf{P}) \wedge \\
& \mathsf{bag}(\mathsf{t}_1,\mathsf{x}) * \boxed{\mathsf{q}(\mathsf{x},\mathsf{P})}^{\mathsf{r},\mathsf{t}_2,\mathsf{x}}_{\mathsf{I}(\mathsf{P})} * [\textsc{Upd}]^{\mathsf{r}}_{\pi} \\
\mathsf{q}(\mathsf{x},\mathsf{P}) \stackrel{\text{def}}{=}\ & \exists \mathsf{X} : \mathcal{P}_m(\mathrm{Val}).\ \mathsf{x}_{\mathsf{cont}} \overset{1/2}{\longmapsto} \mathsf{X} * \circledast_{\mathsf{y}\in\mathsf{X}} \mathsf{P}(\mathsf{y}) \\
\mathsf{I}(\mathsf{P})(\mathsf{x}) \stackrel{\text{def}}{=}\ & (\textsc{Upd} : \mathsf{q}(\mathsf{x},\mathsf{P}) \rightsquigarrow \mathsf{q}(\mathsf{x},\mathsf{P}))
\end{aligned}$$

The parametric protocol $I(\mathsf{P})$ allows the bag to be changed arbitrarily, provided the region still contains the state associated with each element currently in the bag. From the assumption that each $\mathsf{P}(\mathsf{x})$ is stable and that $\mathsf{usip}_{\mathrm{Val}}(\mathsf{P})$ it follows that $\mathsf{q}(\mathsf{x},\mathsf{P})$ is stable and $\mathsf{sip}(\mathsf{q}(\mathsf{x},\mathsf{P}))$. Hence, there exists $\mathsf{R},\mathsf{S} : \mathsf{Prop}$ such that $\mathsf{q}(\mathsf{x},\mathsf{P}) \Leftrightarrow \mathsf{R} * \mathsf{S}$, $\mathsf{pure}_{\mathsf{protocol}}(\mathsf{S})$ and $\mathsf{pure}_{\mathsf{state}}(\mathsf{R})$. Thus, $\mathsf{bag}_s(\mathsf{t},\mathsf{x},\mathsf{P})$ is equivalent to the following assertion:

$$\exists r,\pi,\mathsf{t}_1,\mathsf{t}_2.\ \mathsf{t} \le \mathsf{t}_1 \wedge \mathsf{t} \le \mathsf{t}_2 \wedge \mathsf{t}_1 \not\le \mathsf{t}_2 \wedge \mathsf{t}_2 \not\le \mathsf{t}_1 \wedge \mathsf{bag}(\mathsf{t}_1,\mathsf{x}) * \boxed{\mathsf{S}}^{\mathsf{r},\mathsf{t}_2,\mathsf{x}}_{\mathsf{I}(\mathsf{P})} * \mathsf{R} * [\textsc{Upd}]^{\mathsf{r}}_{\pi}$$

Hence, to prove $\mathsf{bag}_s(\mathsf{t},\mathsf{x},\mathsf{P})$ stable, it suffices to prove stability of $\boxed{\mathsf{S}}^{\mathsf{r},\mathsf{t}_2,\mathsf{x}}_{\mathsf{I}(\mathsf{P})} * \mathsf{R}$. Applying rule SA, it thus suffices to prove,

$$\mathsf{valid}(\mathsf{q}(\mathsf{x},\mathsf{P}) \wedge \mathsf{S} \Rightarrow \bot) \vee \mathsf{valid}(\mathsf{q}(\mathsf{x},\mathsf{P}) \Rightarrow \mathsf{S})$$

and the right disjunct follows easily from the assumption that $\mathsf{q}(\mathsf{x},\mathsf{P}) \Leftrightarrow \mathsf{R} * \mathsf{S}$.

To derive the shared bag specification for `push`, we thus have to transfer the resources associated with the element being pushed, $\mathsf{P}(\mathsf{y})$, to the client region containing the element resources. We thus instantiate P and Q in the generic bag specification with $\mathsf{P}(\mathsf{y}) * \boxed{\mathsf{q}(\mathsf{x},\mathsf{P})}^{\mathsf{r},\mathsf{t}_2,\mathsf{x}}_{\mathsf{I}(\mathsf{P})} * [\textsc{Upd}]^{\mathsf{r}}_{\pi}$ and $\boxed{\mathsf{q}(\mathsf{x},\mathsf{P})}^{\mathsf{r},\mathsf{t}_2,\mathsf{x}}_{\mathsf{I}(\mathsf{P})} * [\textsc{Upd}]^{\mathsf{r}}_{\pi}$, respectively.

We thus have to provide a view-shift to synchronize the abstract state of the library protocol with our client protocol r:

$$\forall \mathsf{X} : \mathcal{P}_m(\mathrm{Val}).\ \mathsf{x}_{\mathsf{cont}} \overset{1/2}{\longmapsto} \mathsf{X} * \mathsf{P}(\mathsf{y}) * \boxed{\mathsf{q}(\mathsf{x},\mathsf{P})}^{\mathsf{r},\mathsf{t}_2,\mathsf{x}}_{\mathsf{I}(\mathsf{P})} * [\textsc{Upd}]^{\mathsf{r}}_{\pi} \ \sqsubseteq_{\mathsf{t}_1}$$
$$\mathsf{x}_{\mathsf{cont}} \overset{1/2}{\longmapsto} (\mathsf{X} \cup \{\mathsf{y}\}) * \boxed{\mathsf{q}(\mathsf{x},\mathsf{P})}^{\mathsf{r},\mathsf{t}_2,\mathsf{x}}_{\mathsf{I}(\mathsf{P})} * [\textsc{Upd}]^{\mathsf{r}}_{\pi}$$

Since $\mathsf{x}_{\mathsf{cont}} \overset{1/2}{\longmapsto} \mathsf{X} * \mathsf{P}(\mathsf{y}) * [\textsc{Upd}]^{\mathsf{r}}_{\pi}$ and $\mathsf{q}(\mathsf{x},\mathsf{P})$ are all independent of region type t, by rule VSOpen it suffices to prove that the change to region r is allowed and possible. The update is easily shown to be allowed by the Upd action, using the UAAct rule and update action frame rule (UAF). To show the possibility of the view shift it suffices to prove that:

$$\mathsf{x}_{\mathsf{cont}} \overset{1/2}{\longmapsto} \mathsf{X} * \mathsf{P}(\mathsf{y}) * \exists \mathsf{Z} : \mathcal{P}_m(\mathrm{Val}).\ \mathsf{x}_{\mathsf{cont}} \overset{1/2}{\longmapsto} \mathsf{Z} * \circledast_{\mathsf{z}\in\mathsf{Z}} \mathsf{P}(\mathsf{z}) * [\textsc{Upd}]^{\mathsf{r}}_{\pi} \ \sqsubseteq_{\bot}$$
$$\mathsf{x}_{\mathsf{cont}} \overset{1/2}{\longmapsto} (\mathsf{X} \cup \{\mathsf{y}\}) * \exists \mathsf{Z} : \mathcal{P}_m(\mathrm{Val}).\ \mathsf{x}_{\mathsf{cont}} \overset{1/2}{\longmapsto} \mathsf{Z} * \circledast_{\mathsf{z}\in\mathsf{Z}} \mathsf{P}(\mathsf{z}) * [\textsc{Upd}]^{\mathsf{r}}_{\pi}$$

which follows easily, as $\mathsf{x}_{\mathsf{cont}} \overset{1/2}{\longmapsto} \mathsf{X} * \mathsf{x}_{\mathsf{cont}} \overset{1/2}{\longmapsto} \mathsf{Z} \Rightarrow \mathsf{X} = \mathsf{Z}$.

Note that to provide a view-shift to synchronize the abstract state of the library protocol with the client protocol, we were essentially forced to update the phantom field `cont` in the client region, which in turn forced us to transfer ownership of $\mathsf{P}(\mathsf{y})$ to the client region.

4 Concurrent Runner

The concurrent runner is a small library for parallelizing divide-and-conquer algorithms, inspired by Doug Lea's Fork/Join framework [12]. Clients interact with the library by registering a delegate for parallelization. The library then provides methods for scheduling executions of the registered delegate using a set of worker threads. In general, this delegate could have side-effects. The registered delegate is also allowed to schedule itself for execution, leading to recursion through the store through the library. The concurrent runner thus makes for an interesting specification challenge, as it requires a specification that supports higher-order code with effects and recursion through the store through the library. It also makes for an interesting verification challenge, as the library is implemented using shared mutable state. In particular, the implementation uses a shared bag to share tasks scheduled for execution between the worker threads.

To demonstrate that our approach scales, we have verified the concurrent runner. The proof uses the shared bag specification derived in the previous section.

We use nested Hoare triples and guarded recursion to reason about delegates and recursion. See Appendix D for a detailed explanation of the concurrent runner and a proof outline of the implementation.

5 Semantics

In this section we sketch the model and the interpretation of our logic. Due to lack of space, we focus on parts presented in Section 2. The full model, interpretation and accompanying soundness proof can be found in the technical report [20].

The presentation of the model is strongly inspired by the Views framework presentation [4]. The model is an instance of the Views framework extended with step-indexing to model guarded recursion, and thread local state to model dynamic allocation of threads.

The basic structure of the model is defined below. Assertions are modeled as step-indexed predicates on instrumented states ($\mathcal{M}$). Instrumented states consist of three components, a local state, a shared state and an action model. The local state specifies the current local resources. The shared state is further partitioned into regions and each region consists of a local state, a region type and a protocol parameter. The action model maps region types to parameterized protocols, which are functions from a tuple containing a protocol argument, a region identifier and an action identifier to an action. Lastly, actions are modeled as certain step-indexed relations on shared states. In particular, actions are *not* relations on shared states *and* action models, and thus do not support general higher-order protocols. Actions do however support state-independent protocols, through the region type indirection.

$$\text{LState} \stackrel{\text{def}}{=} \text{Heap} \times \text{PHeap} \times \text{Cap} \qquad \text{SState} \stackrel{\text{def}}{=} \text{RId} \rightharpoonup (\text{LState} \times \text{RType} \times \text{Val})$$

$$\mathcal{M} \stackrel{\text{def}}{=} \text{LState} \times \text{SState} \times \text{AMod} \qquad \text{AMod} \stackrel{\text{def}}{=} \text{RType} \rightharpoonup ((\text{Val} \times \text{RId} \times \text{AId}) \to \text{Act})$$

$$\text{Cap} \stackrel{\text{def}}{=} \{f \in \text{RId} \times \text{AId} \to [0,1] \mid \exists R \subseteq_{fin} \text{RId}.\ \forall r \in \text{RId} \setminus R.\ \forall \alpha \in \text{AId}.\ f(r,\alpha) = 0\}$$

$$\begin{aligned}\text{Act} \stackrel{\text{def}}{=} \{R \in \mathcal{P}(\mathsf{N} \times \text{SState} \times \text{SState}) \mid \\ \forall (i, s_1, s_2) \in R.\ \forall j \leq i.\ \forall r \in \text{RId} \setminus \mathit{dom}(s_2).\ \forall n \in \text{RType}.\ \forall l, l' \in \text{LState}. \\ s_1 \leq s_2 \wedge (j, s_1, s_2[r \mapsto (l', n)]) \in R \wedge \\ (j, s_1[r \mapsto (l, n)], s_2[r \mapsto (l', n)]) \in R\}\end{aligned}$$

$$\begin{aligned}\mathsf{Prop} \stackrel{\text{def}}{=} \{U \in \mathcal{P}(\mathsf{N} \times \mathcal{M}) \mid \forall (i, m_1) \in U.\ \forall j \leq i.\ \forall m_2 \in \mathcal{M}. \\ (m_1 =_j m_2 \vee m_1 \leq m_2) \Rightarrow (j, m_2) \in U\}\end{aligned}$$

$$\text{Spec} \stackrel{\text{def}}{=} \{U \in \mathcal{P}(\mathsf{N}) \mid \forall i \in U.\ \forall j \leq i.\ j \in U\}$$

The semantics of both the assertion logic and specification logic is step-indexed. The specification logic is step-indexed to allow reasoning about mutual recursion. The assertion logic is step-indexed to support nested triples (which embed specifications in the assertion logic) [18] and guarded recursive predicates [1, 3]. Specifications are thus modeled as downwards closed subsets of numbers, and assertions are modeled as step-indexed predicates on instrumented states, that are downwards closed in the step-index and upwards closed in $\mathcal{M}$. The upwards closure in $\mathcal{M}$ ensures that assertions are closed under allocation of new regions

and protocols (the ordering $\leq$ on $\mathcal{M}$ is defined as expected). To define guarded recursive functions and predicates, the types of our logic are modeled as sets with a step-indexed equivalence relation, $=_i$, and terms and predicates are modeled as non-expansive functions. However, as this part of the model is mostly orthogonal to CAP, we will elide the details, which can be found in the technical report [20].

Comparison with previous models of CAP. The original model of (first-order) CAP [5] employed a syntactic treatment of actions to break a circularity in the definition of worlds. Our model follows the previous model of higher-order CAP (without higher-order protocols) [6] in treating actions semantically. However, to support higher-order protocols we introduce a new indirection, in the form of region types. Actions are thus relations on shared states, which include the region types of allocated regions. Actions can thus implicitly refer to the protocol on regions through the region type indirection. While previous work has only considered CAP for a *first-order programming language*, our HOCAP is for a *higher-order programming language*. We thus step-index both the specification and assertion logic, instead of just the specification logic.

Model operations. Separating conjunction is interpreted as the lifting of the partial commutative $\bullet_{\mathcal{M}}$ function to Prop (point-wise in the step-index). The $\bullet_{\mathcal{M}}$ function expresses how to compose two instrumented states. Two instrumented states are combinable if they agree on the shared state and action model, by combining their local states, using $\bullet_{\mathrm{LState}}$. Local states are combined using the standard combination function, $\bullet_{\uplus}$, on disjoint partial functions, on the heap and phantom heap component, and by point-wise summing up the action permissions.

While assertions are modeled as step-indexed predicates on instrumented states, which include phantom fields, protocols, and regions, the operational semantics operates on concrete states, which are simply heaps. The main soundness theorem (Theorem 1) expresses that any step in the concrete semantics has a corresponding step in the instrumented semantics. This is expressed in terms of an erasure function, $\lfloor - \rfloor \in \mathcal{M} \rightharpoonup \mathrm{Heap}$, that erases the instrumentation from an instrumented state. The erasure of an instrumented state is simply the combination of the local state and all shared regions.

$$\lceil (l, s) \rceil \stackrel{\text{def}}{=} l \bullet_{\mathrm{LState}} \prod_{r \in dom(s)} s(r).l$$

$$\lfloor (l, s, \varsigma) \rfloor \stackrel{\text{def}}{=} \begin{cases} h & \text{if } (h, ph, c) = \lceil (l, s) \rceil \text{ and } \pi_1(dom(ph)) \subseteq objs(h) \\ \text{undef} & \text{otherwise} \end{cases}$$

Interference. The interference relation $R_i^A \subseteq \mathcal{M} \times \mathcal{M}$ describes possible interference from the environment. It is defined as the reflexive, transitive closure of the single-action interference relation, $\hat{R}_i^A$ (defined below), that describes possible environment interference using at most one action on each region. Defining R_i^A as the reflexive, transitive closure of $\hat{R}_i^A$ forces a common action granularity on updates to multiple regions with protocols referring to each other (see

Appendix B). In addition to the step-index $i \in \mathsf{N}$, the single-action interference relation is also indexed by a set $A \in \mathcal{P}(\text{RType})$ of region types of those regions that are allowed to change and that actions justifying those changes are allowed to depend on.

$$\begin{array}{l}(l_1, s_1, \varsigma_1)\ \hat{R}_i^A\ (l_2, s_2, \varsigma_2) \quad \text{iff} \quad l_1 = l_2 \land s_1 \leq s_2 \land \varsigma_1 \leq \varsigma_2 \land \lceil (l_1, s_1) \rceil \text{ defined} \land \\ \quad (\forall r \in \mathit{dom}(s_1).\ s_1(r) = s_2(r) \lor (\exists \alpha.\ s_1(r).t \in A \land \\ \quad\quad (\lceil (l_1, s_1) \rceil .c)(r, \alpha) < 1 \land (i, s_1|_A, s_2|_A) \in \varsigma_1(s_1(r).t)(s_1(r).a, r, \alpha)))\end{array}$$

$$s|_A \stackrel{\text{def}}{=} \lambda r \in \text{RId}.\ \begin{cases} s(r) & \text{if } r \in \mathit{dom}(s) \text{ and } s(r).t \in A \\ \text{undef} & \text{otherwise} \end{cases}$$

In particular, the $\hat{R}_i^A$ relation expresses that the environment is not allowed to change the local state ($l_1 = l_2$), but it is allowed to allocate new regions and protocols ($s_1 \leq s_2$ and $\varsigma_1 \leq \varsigma_2$). Furthermore, the environment is allowed to update the resources of any region r with a region type in A ($s_1(r).t \in A$), provided the update is justified by an action α that is partially owned by the environment ($\lceil (l_1, s_1) \rceil (r, \alpha) < 1$).

An assertion is stable if it is closed under interference to all region types:

$$\mathit{stable}(p) \stackrel{\text{def}}{=} \{i \in \mathsf{N} \mid \forall j \leq i.\ \forall (m_1, m_2) \in R_j^{\text{RType}}.\ (j, m_1) \in p \Rightarrow (j, m_2) \in p\}$$

Previous models of CAP have only permitted multiple independent updates, whereas our model supports multiple dependent updates. Previous models thus lack the A-index that we use to enforce a common action granularity on updates to multiple dependent regions.

View-shifts. View-shifts describe a step in the instrumented semantics that correspond to a no-op in the concrete semantics. To perform a view-shift from p to q we thus have to prove that for every concrete state c in the erasure of some instrumented state $m \in p$ there exists an instrumented state $m' \in q$ such that c is in the erasure of m'.

$$\begin{array}{l}p \sqsubseteq_t q \stackrel{\text{def}}{=} \{i \in \mathsf{N} \mid\ \forall m \in \mathcal{M}.\ \forall j \in \mathsf{N}.\ 0 \leq j \leq i\ \Rightarrow \\ \qquad\qquad \lfloor p * \{(j, m)\} \rfloor_j \subseteq \lfloor q * \{(j, m') \mid m\ \hat{R}_j^{\{t' \mid t \not\leq t'\}}\ m'\} \rfloor_j\}\end{array}$$

To allow framing on view-shifts (rule VSFRAME in Section 2.3) we bake in framing under certain stable frames. The frames in question depend on the region index $t \in \text{RType}$. In particular, $\sqsubseteq_t$ permits a single simultaneous update of multiple regions with region types not greater than or equal to t, each justified by a single action. Hence, we require that $\sqsubseteq_t$ is closed under arbitrary frames that are stable under a single simultaneous update of multiple regions with region types not greater than or equal to t, each justified by a single action, i.e., $\hat{R}^{\{t' \mid t \not\leq t'\}}$.

Support. In Section 2.2 we introduced specification logic assertions indep and dep, to internalize a notion of region type support in the logic, to allow explicit proofs of the absence of self-referential region assertions. Their meaning is defined in terms of the following supp assertion, which asserts that p is supported by the

set of region types $A \in \mathcal{P}(\text{RType})$. Formally, $supp_A(p)$ asserts that p is closed under arbitrary shared states that agree on all regions of type A ($s|_A = s'|_A$) and arbitrary action models that are A equivalent ($\varsigma \equiv_A \varsigma'$).

$$\mathsf{supp}_A(p) \stackrel{\text{def}}{=} \{i \in \mathsf{N} \mid \forall j \leq i.\ \forall (j,(l,s,\varsigma)) \in p.\ \forall s'.\ \forall \varsigma'.\ s|_A = s'|_A \wedge \varsigma \equiv_A \varsigma' \Rightarrow (j,(l,s',\varsigma')) \in p\}$$

Intuitively, two action models are considered A-equivalent if they agree on the regions of types in A (but they are allowed to differ on regions of types not in A). An assertion p is then dependent on region type $t \in \text{RType}$ if p is supported by the set of region types greater than or equal to t, and independent if it is supported by the set of region types not greater than or equal to t:

$$\mathsf{dep}_t(p) \stackrel{\text{def}}{=} \mathsf{supp}_{\{t' \mid t \leq t'\}}(p) \qquad \mathsf{indep}_t(p) \stackrel{\text{def}}{=} \mathsf{supp}_{\{t' \mid t \not\leq t'\}}(p)$$

Purity. To reason about state-independent protocols and nested view-shifts we have introduced several types of purity; namely, state, protocol and permission purity. Since our assertion logic is intuitionistic, we interpret purity as closure under arbitrary changes to the state, protocols, and permissions, respectively. For instance, $pure_{prot}(p) \stackrel{\text{def}}{=} \{i \in \mathsf{N} \mid \forall j \leq i.\ \forall (j,(l,s,\varsigma)) \in p.\ \forall \varsigma'.\ (j,(l,s,\varsigma')) \in p\}$.

Soundness. The main soundness theorem expresses that for any derivable Hoare triple, $\{p\}\bar{\mathsf{c}}\{q\}$, if $\bar{\mathsf{c}}$ is executed with a local stack s as thread t, with a global heap h that is in the erasure of some instrumented state in $p(s)$, then, if t (and any threads t may have forked) terminates, then the terminal heap h' is in the erasure of some instrumented state in $q(s')$, where s' is the terminal stack of t.

Theorem 1. *If* $\Gamma \vdash (\Delta).\{P\}\bar{\mathsf{c}}\{Q\}$ *then for all* $\vartheta \in [\![\Gamma]\!]$, *thread identifiers* $t \in TId$, *stacks* $s \in [\![\Delta]\!]$, *and heaps* $h \in \lfloor [\![\Gamma; \Delta \vdash P : \mathsf{Prop}]\!](\vartheta, s) \rfloor$, *if*

$$(h, \{(t, s, \bar{\mathsf{c}})\}) \to (h', \{(t, s', skip)\} \uplus T')$$

and T' *is irreducible then* $h' \in \lfloor [\![\Gamma; \Delta \vdash Q : \mathsf{Prop}]\!](\vartheta, s') \rfloor$.

6 Conclusion and Future Work

We have proposed a new style of specification for thread-safe data structures that allows the client to refine the specification with a usage protocol, in a concurrent setting. We have shown how to apply it to the bag and concurrent runner example. To realize this style of specification we have presented a new higher-order separation logic with Concurrent Abstract Predicates, that supports state-independent higher-order protocols and synchronization of multiple regions. We have also used the logic to specify and verify Joins, a sophisticated library implemented using higher-order code and shared mutable state.

We have demonstrated that our logic and style of specification scales to implementations of fine-grained concurrent data structures without helping [8]. Future work includes investigating concurrent data structures that use helping.

References

1. A. Appel, P.-A. Melliès, C. Richards, and J. Vouillon. A very modal model of a modern, major, general type system. In *Proc. of POPL*, 2007.
2. B. Biering, L. Birkedal, and N. Torp-Smith. BI-Hyperdoctrines, Higher-order Separation Logic, and Abstraction. *ACM TOPLAS*, 2007.
3. L. Birkedal, R. Møgelberg, J. Schwinghammer, and K. Støvring. First Steps in Synthetic Guarded Domain Theory: Step-Indexing in the Topos of Trees. In *Proc. of LICS*, 2011.
4. T. Dinsdale-Young, L. Birkedal, P. Gardner, M. Parkinson, and H. Yang. Views: Compositional Reasoning for Concurrent Programs. In *Proceedings of POPL*, 2013.
5. T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis. Concurrent Abstract Predicates. In *Proceedings of ECOOP*, 2010.
6. M. Dodds, S. Jagannathan, and M. J. Parkinson. Modular reasoning for deterministic parallelism. In *Proceedings of POPL*, pages 259–270, 2011.
7. I. Filipović, P. O'Hearn, N. Rinetzky, and H. Yang. Abstraction for concurrent objects. In *Proceedings of ESOP 2009*, pages 252–266, 2009.
8. M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.
9. M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM TOPLAS*, 12:463–492, 1990.
10. B. Jacobs and F. Piessens. Expressive modular fine-grained concurrency specification. In *Proceedings of POPL*, pages 271–282, 2011.
11. J. B. Jensen and L. Birkedal. Fictional Separation Logic. In *Proceedings of ESOP*, pages 377–396, 2012.
12. D. Lea. A java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, JAVA '00, pages 36–43. ACM, 2000.
13. S. S. Owicki. *Axiomatic Proof Techniques for Parallel Programs*. PhD thesis, Cornell, 1975.
14. M. Parkinson, R. Bornat, and P. O'Hearn. Modular verification of a non-blocking stack. *SIGPLAN Not.*, 42(1), 2007.
15. A. Pilkiewicz and F. Pottier. The essence of monotonic state. In *Proceedings of TLDI*, pages 73–86, 2011.
16. C. V. Russo. The Joins Concurrency Library. In *Proceedings of PADL*, pages 260–274, 2007.
17. J. Schwinghammer, L. Birkedal, B. Reus, and H. Yang. Nested Hoare triples and frame rules for higher-order store. In *Proceedings of CSL*, Apr. 2009.
18. J. Schwinghammer, L. Birkedal, B. Reus, and H. Yang. Nested Hoare Triples and Frame Rules for Higher-Order Store. *LMCS*, 7(3:21), 2011.
19. K. Svendsen, L. Birkedal, and M. Parkinson. Verifying Generics and Delegates. In *Proceedings of ECOOP*, pages 175–199, 2010.
20. K. Svendsen, L. Birkedal, and M. Parkinson. Higher-order Concurrent Abstract Predicates. Technical report, IT University of Copenhagen, 2012. Available at `http://www.itu.dk/people/kasv/hocap-tr.pdf`.
21. A. Turon, J. Thamsborg, A. Ahmed, L. Birkedal, and D. Dreyer. Logical Relations for Fine-Grained Concurrency. In *Proceedings of POPL*, 2013.

A Proof rules

$$\frac{}{\mathsf{dep}_{\mathsf{t}}(\bot)} \qquad \frac{}{\mathsf{dep}_{\mathsf{t}}(\top)} \qquad \frac{\mathsf{t}_1 \leq \mathsf{t}_2 \quad \mathsf{dep}_{\mathsf{t}_2}(\mathsf{P})}{\mathsf{dep}_{\mathsf{t}_1}(\mathsf{P})} \qquad \frac{op \in \{\vee, \wedge, *, \Rightarrow\} \quad \mathsf{dep}_{\mathsf{t}}(\mathsf{P}) \quad \mathsf{dep}_{\mathsf{t}}(\mathsf{Q})}{\mathsf{dep}_{\mathsf{t}}(\mathsf{P}\ op\ \mathsf{Q})}$$

$$\frac{\mathsf{dep}_{\mathsf{t}_1}(\mathsf{P})}{\mathsf{dep}_{\mathsf{t}_1 \sqcap \mathsf{t}_2}(\boxed{\mathsf{P}}^{\mathsf{r},\mathsf{t}_2,\mathsf{a}})} \qquad \frac{\mathsf{dep}_{\mathsf{t}_1}(\mathsf{I})}{\mathsf{dep}_{\mathsf{t}_1 \sqcap \mathsf{t}_2}(\mathsf{protocol}(\mathsf{t}_2, \mathsf{I}))} \qquad \frac{}{\mathsf{dep}_{\mathsf{t}}([\alpha]^{\mathsf{r}}_{\pi})}$$

$$\frac{\mathsf{dep}_{\mathsf{t}}(\mathsf{P}) \quad Q \in \{\exists, \forall\}}{\mathsf{dep}_{\mathsf{t}}(Q\mathsf{x}.\ \mathsf{P}(\mathsf{x}))} \qquad \frac{\mathsf{t}_1 \not\leq \mathsf{t}_2 \quad \mathsf{dep}_{\mathsf{t}_1}(\mathsf{P}) \quad \mathsf{t}_2 \not\leq \mathsf{t}_1}{\mathsf{indep}_{\mathsf{t}_2}(\mathsf{P})} \qquad \frac{\mathsf{t}_1 \not\leq \mathsf{t}_2 \quad \mathsf{indep}_{\mathsf{t}_1}(\mathsf{P})}{\mathsf{indep}_{\mathsf{t}_1}(\boxed{\mathsf{P}}^{\mathsf{r},\mathsf{t}_2,\mathsf{a}})}$$

Fig. 2. Proof rules for dependence and independence. To simplify the presentation we also use indep and dep for the point-wise lifting of indep and dep to predicates.

B On the granularity of Actions

As mentioned in Section 2.3, to support synchronization of protocols we extend CAP with simultaneous updates of multiple regions. This raises questions about the granularity of actions. For instance, it seems natural that the following view-shift should hold, by sequencing the α action followed by the β action:

$$\boxed{\mathsf{x_f} \mapsto 0}^{\mathsf{r},\mathsf{t},-}_{I} * [\alpha]^{\mathsf{r}}_1 * [\beta]^{\mathsf{r}}_1 \sqsubseteq \boxed{\mathsf{x_f} \mapsto 2}^{\mathsf{r},\mathsf{t},-}_{I} * [\alpha]^{\mathsf{r}}_1 * [\beta]^{\mathsf{r}}_1$$

$$\mathsf{I}(-) = \begin{pmatrix} \alpha : \mathsf{x_f} \mapsto 0 \rightsquigarrow \mathsf{x_f} \mapsto 1 \\ \beta : \mathsf{x_f} \mapsto 1 \rightsquigarrow \mathsf{x_f} \mapsto 2 \end{pmatrix}$$

However, it is not clear whether the following view-shift should hold,

$$\begin{aligned} &\boxed{\mathsf{x_f} \mapsto 0}^{\mathsf{r},\mathsf{t},-}_{\mathsf{I}} * \boxed{\mathsf{x_g} \mapsto 0}^{\mathsf{r}',\mathsf{t}',-}_{\mathsf{J}} * [\alpha]^{\mathsf{r}}_1 * [\beta]^{\mathsf{r}}_1 * [\eta]^{\mathsf{r}'}_1 \\ &\quad \sqsubseteq \boxed{\mathsf{x_f} \mapsto 2}^{\mathsf{r},\mathsf{t},-}_{\mathsf{I}} * \boxed{\mathsf{x_g} \mapsto 2}^{\mathsf{r}',\mathsf{t}',-}_{\mathsf{J}} * [\alpha]^{\mathsf{r}}_1 * [\beta]^{\mathsf{r}}_1 * [\eta]^{\mathsf{r}'}_1 \end{aligned} \tag{4}$$

$$\mathsf{J}(-) = \left(\eta : \mathsf{x_g} \mapsto 0 * \boxed{\mathsf{x_f} \mapsto 0}^{\mathsf{r},\mathsf{t},-} \rightsquigarrow \mathsf{x_g} \mapsto 2 * \boxed{\mathsf{x_f} \mapsto 2}^{\mathsf{r},\mathsf{t},-}\right)$$

as it requires a *simultaneous* update of two regions, r and r', using two actions on r and one action on r'. We have chosen a semantics that does not allow the

$$\frac{}{\mathsf{pure}_{\mathsf{protocol}}(\bot)} \qquad \frac{}{\mathsf{pure}_{\mathsf{protocol}}(\top)}$$

$$\frac{\mathsf{pure}_{\mathsf{protocol}}(\mathsf{P}) \qquad \mathsf{pure}_{\mathsf{protocol}}(\mathsf{Q}) \qquad op \in \{\vee, \wedge, *, \Rightarrow\}}{\mathsf{pure}_{\mathsf{protocol}}(\mathsf{P}\ op\ \mathsf{Q})}$$

$$\frac{\forall \mathsf{x} : \mathsf{X}.\ \mathsf{pure}_{\mathsf{protocol}}(\mathsf{P}(\mathsf{x}))}{\mathsf{pure}_{\mathsf{protocol}}(\exists \mathsf{x} : \mathsf{X}.\ \mathsf{P}(\mathsf{x}))} \qquad \frac{\forall \mathsf{x} : \mathsf{X}.\ \mathsf{pure}_{\mathsf{protocol}}(\mathsf{P}(\mathsf{x}))}{\mathsf{pure}_{\mathsf{protocol}}(\forall \mathsf{x} : \mathsf{X}.\ \mathsf{P}(\mathsf{x}))}$$

$$\frac{\mathsf{pure}_{\mathsf{protocol}}(\mathsf{P})}{\mathsf{pure}_{\mathsf{protocol}}(\boxed{\mathsf{P}}^{\mathsf{r},\mathsf{t},\mathsf{a}})} \qquad \frac{}{\mathsf{pure}_{\mathsf{protocol}}([\alpha]^{\mathsf{r}}_{\pi})} \qquad \frac{}{\mathsf{pure}_{\mathsf{protocol}}(\mathsf{x} \mapsto (\Delta).\{\mathsf{P}\}\{\mathsf{Q}\})}$$

$$\frac{}{\mathsf{pure}_{\mathsf{protocol}}(\mathsf{x}.\mathsf{f} \overset{\pi}{\mapsto} \mathsf{v})} \qquad \frac{}{\mathsf{pure}_{\mathsf{protocol}}(\mathsf{x}_{\mathsf{f}} \overset{\pi}{\mapsto} \mathsf{v})}$$

Fig. 3. $\mathsf{pure}_{\mathsf{protocol}}$ proof rules

$$\frac{}{\mathsf{pure}_{\mathsf{state}}(\bot)} \qquad \frac{}{\mathsf{pure}_{\mathsf{state}}(\top)}$$

$$\frac{\mathsf{pure}_{\mathsf{state}}(\mathsf{P}) \qquad \mathsf{pure}_{\mathsf{state}}(\mathsf{Q}) \qquad op \in \{\vee, \wedge, *, \Rightarrow\}}{\mathsf{pure}_{\mathsf{state}}(\mathsf{P}\ op\ \mathsf{Q})}$$

$$\frac{\forall \mathsf{x} : \mathsf{X}.\ \mathsf{pure}_{\mathsf{state}}(\mathsf{P}(\mathsf{x}))}{\mathsf{pure}_{\mathsf{state}}(\exists \mathsf{x} : \mathsf{X}.\ \mathsf{P}(\mathsf{x}))} \qquad \frac{\forall \mathsf{x} : \mathsf{X}.\ \mathsf{pure}_{\mathsf{state}}(\mathsf{P}(\mathsf{x}))}{\mathsf{pure}_{\mathsf{state}}(\forall \mathsf{x} : \mathsf{X}.\ \mathsf{P}(\mathsf{x}))}$$

$$\frac{}{\mathsf{pure}_{\mathsf{state}}(\mathsf{protocol}(\mathsf{t}, \mathsf{l}))}$$

Fig. 4. $\mathsf{pure}_{\mathsf{state}}$ proof rules

granularity of actions to differ when updating multiple regions. Thus, in our logic, (4) does not hold. This is a choice; it simplifies stability proofs, but it means that we must explicitly track which regions that may have been updated by a view-shift. We thus index the view-shift relation with a region type t. The indexed view-shift relation, $\sqsubseteq_{\mathsf{t}}$, thus describes a *single* update that, in addition to updating the local state, may update *multiple* shared regions with region types not greater than or equal to t, where each update must be justified by a *single* action. The indexed view-shift relation is thus *not* transitive.

C Proof of Bag specification

In the main text we showed how to derive the exclusive owner and shared bag specification from the refineable bag specification. In this section we sketch how to prove that a concurrent bag implementation satisfies the refinable specification. In particular, we sketch a proof of a non-locking concurrent bag, implemented using compare-and-swap.

Representation predicates. As mentioned in the introduction, the idea is to share the *concrete state* of the concurrent bag using a shared region. To allow the client to refine the specification, we store the *abstract state* of the concurrent bag in a phantom field and let the client keep a half-permission to the phantom field. The library protocol thus has to relate the concrete state of the concurrent bag with its abstract state (i.e., a multiset of elements).

In our implementation, the bag is represented as a singly-linked list. An abstract state given by a multiset of elements X is thus related to a concrete state containing a singly-linked list with the elements of X. We can express this relation between the concrete and abstract state as follows:

$$\mathsf{q}(\mathsf{x}) \stackrel{\text{def}}{=} \exists \mathsf{h} : \text{Val}.\ \exists \mathsf{l} : \mathit{seq}\ \text{Val}.\ \mathsf{x.head} \mapsto \mathsf{h} * \mathsf{lst}_r(\mathsf{h}, \mathsf{l}) * \mathsf{x}_{\mathsf{cont}} \overset{1/2}{\longmapsto} \mathsf{mem}(\mathsf{l})$$

where lst_r is the following list-representation predicate:

$$\mathsf{lst}_r(\mathsf{x}, \varepsilon) \stackrel{\text{def}}{=} \mathsf{x} = \mathsf{null}$$
$$\mathsf{lst}_r(\mathsf{x}, \mathsf{v} :: \mathsf{l}) \stackrel{\text{def}}{=} \exists \mathsf{y} : \text{Val}.\ \mathsf{x.value} \overset{-}{\mapsto} \mathsf{v} * \mathsf{x.next} \overset{-}{\mapsto} \mathsf{y} * \mathsf{lst}_r(\mathsf{y}, \mathsf{l})$$

and $\mathsf{mem} : \mathit{seq}\ \text{Val} \to \mathcal{P}_m(\text{Val})$ returns the multiset of elements of the given sequence. The bag predicate thus asserts that there exists a shared region containing the concrete and abstract state of the bag, which is currently related by q, and a protocol that enforces that the concrete and abstract state is always related by q:

$$\mathsf{bag}(\mathsf{t}, \mathsf{x}) \stackrel{\text{def}}{=} \exists \mathsf{r} : \text{RId}.\ \exists \pi \in \text{Perm}.\ \exists \mathsf{s} : \text{RType}.$$
$$\mathsf{t} \leq \mathsf{s} \wedge \boxed{\mathsf{q}(\mathsf{x})}^{\mathsf{r},\mathsf{s},\mathsf{x}} * \mathsf{protocol}(\mathsf{s}, \mathsf{l}) * [\textsc{Upd}]^{\mathsf{r}}_{\pi}$$

where I is the following parametric protocol:

$$\mathsf{I}(\mathsf{x}) = (\textsc{Upd} : \mathsf{q}(\mathsf{x}) \rightsquigarrow \mathsf{q}(\mathsf{x}))$$

Note that the lst_r representation predicate only asserts read-only ownership of the underlying singly-linked list[8]. By only asserting read-only ownership of the underlying singly-linked list, we ensure that once an element has been added to the list, its next-field never changes; this is used in the correctness proof of `Pop`. The bag predicate is trivially stable using the SA rule, and freely duplicable.

Proof outline.

```
using Interlocked;

internal class Node⟨A⟩ {
  internal Node⟨A⟩ next;
  internal A value;

  public Node(A value) {
{this.next ↦ null ∗ this.value ↦ null}
    this.value = value;
{this.next ↦ null ∗ this.value ↦ value}
  }
}

public class Bag⟨A⟩ where A : class {
  internal Node⟨A⟩ head;

  public Bag() {
{this.head ↦ null ∗ this_cont ↦ ∅}
{∃l, x. this.head ↦ x ∗ this_cont ↦^1/2 mem(l) ∗ lst_r(x, l) ∗ this_cont ↦^1/2 ∅}
{bag(t, this) ∗ this_cont ↦^1/2 ∅}
  }
```

In the bag constructor, we introduce a phantom field cont, a region type s, and a region with region type s, initialized with ownership of the head field and a half-permission to the cont field.

```
  public void Push(A x) {
    Node⟨A⟩ nHead; Node⟨A⟩ oHead; Node⟨A⟩ tmp;
{bag(t, this) ∗ P}
{t ≤ s ∗ |q(this)|^r,s,this ∗ [Upd]^r_π ∗ P}
    nHead = new Node⟨A⟩(x);
{t ≤ s ∗ |q(this)|^r,s,this ∗ [Upd]^r_π ∗ P ∗ nHead.next ↦ _ ∗ nHead.value ↦ x}
```

[8] We use $\mathsf{x.f} \mapsto \mathsf{v}$ as shorthand for $\exists \mathsf{p} \in \text{Perm}.\ \mathsf{x.f} \overset{\mathsf{p}}{\mapsto} \mathsf{v}$.

```
    do {
      oHead = this.head;
      nHead.next = oHead;
{t ≤ s ∗ [q(this)]^{r,s,this}_{‖} ∗ [UPD]^r_π ∗ P ∗ nHead.next ↦ oHead ∗ nHead.value ↦ x}
      tmp = CompareExchange⟨Node⟨A⟩⟩(ref head, nHead, oHead);
{t ≤ s ∗ [q(this)]^{r,s,this}_{‖} ∗ [UPD]^r_π ∗ ((tmp = oHead ∗ Q) ∨
   (tmp ≠ oHead ∗ P ∗ nHead.next ↦ oHead ∗ nHead.value ↦ x))}
    } while (tmp != oHead);
{t ≤ s ∗ [q(this)]^{r,s,this}_{‖} ∗ [UPD]^r_π ∗ Q}
{bag(t, this) ∗ Q}
  }
```

The `Push` method allocates a new node and inserts it at the front of the list, using compare-and-swap to atomically update the head field. The synchronization point of this method, if it terminates, occurs when the compare-and-swap succeeds. At this point, we transfer ownership of the newly allocated node to the shared region, and use the client-provided view-shift to update the phantom cont field (thus synchronizing with any protocols the client may have imposed on the abstract state). To verify the atomic compare-and-swap, we use a proof rule corresponding to VSNOPEN, for "opening" the shared r region to perform a nested *atomic update* (rule OPENA in the accompanying technical report).

```
  public A Pop() {
    Node⟨A⟩ oHead; Node⟨A⟩ nHead; Node⟨A⟩ tmp;
    A res; bool done;
{bag(t, this) ∗ P}
    done = false;
{bag(t, this) ∗ ((done = false ∗ P) ∨ (done = true ∗ Q(res)))}
    while(!done) {
{bag(t, this) ∗ done = false ∗ P}
{t ≤ s ∗ [q(this)]^{r,s,this}_{‖} ∗ [UPD]^r_π ∗ done = false ∗ P}
      oHead = this.head;
{t ≤ s ∗ [q(this)]^{r,s,this}_{‖} ∗ [UPD]^r_π ∗ done = false
   ∗ ((oHead = null ∗ Q(null)) ∨ (∃l. oHead ≠ null ∗ lst_r(oHead, l) ∗ P))}
      if (oHead == null) {
{t ≤ s ∗ [q(this)]^{r,s,this}_{‖} ∗ [UPD]^r_π ∗ done = false ∗ Q(null)}
        res = null;
        done = true;
{bag(t, this) ∗ done = true ∗ Q(res)}
      } else {
{∃l. t ≤ s ∗ [q(this)]^{r,s,this}_{‖} ∗ [UPD]^r_π ∗ done = false ∗ oHead ≠ null ∗ lst_r(oHead, l) ∗ P}
```

$\{\exists l.\ t \leq s * \boxed{q(\textbf{this})}^{r,s,\textbf{this}}_{\parallel} * [\textsc{Upd}]^r_\pi * done = false$
$* \ oHead.next \mapsto y * oHead.value \mapsto v * lst_r(y, l) * P\}$
nHead = oHead.next;
$\{\exists l.\ t \leq s * \boxed{q(\textbf{this})}^{r,s,\textbf{this}}_{\parallel} * [\textsc{Upd}]^r_\pi * done = false$
$* \ oHead.next \mapsto nHead * oHead.value \mapsto v * lst_r(nHead, l) * P\}$
tmp = CompareExchange⟨Node⟨A⟩⟩(ref head, nHead, oHead);
$\{t \leq s * \boxed{q(\textbf{this})}^{r,s,\textbf{this}}_{\parallel} * [\textsc{Upd}]^r_\pi * done = false$
$* \ ((tmp = oHead * Q(v) * oHead.value \mapsto v) \vee (tmp \neq oHead * P))\}$
if (tmp == oHead) {
$\{t \leq s * \boxed{q(\textbf{this})}^{r,s,\textbf{this}}_{\parallel} * [\textsc{Upd}]^r_\pi * done = false * Q(v) * oHead.value \mapsto v\}$
res = oHead.value;
done = true;
$\{bag(t, \textbf{this}) * done = true * Q(res)\}$
} **else** {
$\{t \leq s * \boxed{q(\textbf{this})}^{r,s,\textbf{this}}_{\parallel} * [\textsc{Upd}]^r_\pi * done = false * P\}$
$\{bag(t, \textbf{this}) * done = false * P\}$
}
}
}
$\{bag(t, \textbf{this}) * Q(res)\}$
return res;
$\{ret.\ bag(t, \textbf{this}) * Q(ret)\}$
}
}

D Concurrent Runner

In this section we consider a more advanced example, which uses the concurrent bag library to implement a simple library for parallelizing divide-and-conquer algorithms. The library is inspired by Doug Lea's Fork/Join framework [12].

The API for the library is given in Figure 5. The library consists of a runner class for parallelizing executions of a delegate. The constructor takes as argument a delegate and the number of worker threads to create. Calling `Fork` on a concurrent runner schedules the registered delegate for execution with the given argument. `Fork` further returns a task representing the scheduled computation. Calling `Join` on the returned task causes the caller to wait until the given task has completed. While the caller is waiting, it turns into a worker thread, helping to execute tasks from the underlying pool. Once the given task has completed, `Join` returns the return-value of the terminated delegate.

Note that the runner constructor takes as argument a delegate, which itself takes as argument a runner. This is to allow the delegate to schedule parallel

```
public class Runner⟨A,B⟩
  public Runner(Func⟨Runner⟨A,B⟩,A,B⟩ body, int n)
  public Task⟨A,B⟩ Fork(A a)

public class Task⟨A,B⟩
  public B Join()
```

Fig. 5. Concurrent Runner API

executions of itself on smaller sub-problems. In particular, when a scheduled task with argument `a` on a runner `r` is chosen for execution, the delegate is called with runner `r` and argument `a`. The delegate can thus schedule itself for execution by calling `Fork` on its runner argument. We can thus implement a concurrent version of a naive fibonacci divide-and-conquer computation as follows:

```
public int parFib(Runner⟨int,int⟩ f, int a) {
  if (a < 25) return seqFib(a);
  else {
    Task⟨int, int⟩ t1 = f.Fork(a − 1);
    Task⟨int, int⟩ t2 = f.Fork(a − 2);
    return t1.Join() + t2.Join();
  }
}

public int fib(int n, int a) {
  return (new Runner⟨int, int⟩(parFib, n)).Fork(a).Join();
}
```

Here `fib` creates a new concurrent runner with `n` worker threads and schedules `parFib` for execution with argument `a`. `parFib` itself computes the fibonacci number of its argument `a`. When `a` is below a certain threshold, it uses a sequential fibonacci implementation, `seqFib`, to avoid the overhead of scheduling a task. Otherwise, it schedules itself recursively to compute the desired result.

The reason we have chosen this API, which requires the client to choose a fixed delegate upon constructing the runner, instead of letting `Fork` take the delegate as argument, is that this API better illustrates the challenges of verifying the Joins library.

Specification. To verify the concurrent runner implementation, we first need a specification. We want a specification that supports concurrent execution of effectful delegates. The idea behind the concurrent runner specification is to let the client pick a specification for the delegate, and require that the client proves the delegate satisfies the given specification. To allow the delegate to schedule new executions of itself for execution, we explicitly allow the delegate to assume it is called with a concurrent runner with a delegate that satisfies the chosen

specification. The delegate specification simply consists of a pre-condition P, indexed by the argument, and a post-condition Q, indexed by the argument and return value. The pre-condition describes the resources the delegate needs to perform its task, while the post-condition describes the resources the delegate transfers back to the owner of the task. We can express this formally as follows:

$$\frac{\mathsf{indep_t(P)} \quad \mathsf{indep_t(Q)} \quad \mathsf{stable(P)} \quad \mathsf{stable(Q)} \quad \mathsf{usip(P)} \quad \mathsf{usip(Q)}}{\left\{\begin{array}{l}\mathsf{b} \mapsto (\mathsf{r},\mathsf{a}).\ \{\mathsf{runner(t,r,P,Q)} * \mathsf{P(a)}\} \\ \qquad\qquad \{\mathsf{ret.\ runner(t,r,P,Q)} * \mathsf{Q(a,ret)}\}\end{array}\right\}}$$
$$\mathsf{new\ Runner(b,n)}$$
$$\{\mathsf{ret.\ runner(t,ret,P,Q)}\}$$

Upon creation of a new concurrent runner the client thus chooses a delegate pre-condition P and post-condition Q. The client is then required to prove that the supplied delegate satisfies the chosen specification. This is expressed using a *nested Hoare triple* [17], which asserts that `b` refers to a delegate satisfying the given specification. In the case of the concurrent runner, the delegate is allowed to use the resources described by its pre-condition, $\mathsf{P(a)}$, and is required to transfer back the resources described by its post-condition, $\mathsf{Q(a,ret)}$. The $\mathsf{runner(t,r,P,Q)}$ assertion further allows the delegate to assume it is called with a concurrent runner `r` with a delegate that satisfies the specification chosen by the client.

To schedule the registered delegate for execution with argument `a`, the client thus has to transfer the state needed by the delegate, $\mathsf{P(a)}$. When scheduling a delegate, the client is given exclusive ownership of the returned task.

$$\overline{\{\mathsf{runner(t,x,P,Q)} * \mathsf{P(arg)}\}\mathsf{x.Fork(arg)}\{\mathsf{ret.\ task(ret,x,P,Q,arg)}\}}$$

Ownership of a task gives the client the right to join the task, and take ownership of the state returned by the scheduled computation.

$$\overline{\{\mathsf{runner(t,y,P,Q)} * \mathsf{task(x,y,P,Q,a)}\}\mathsf{x.Join()}\{\mathsf{ret.\ Q(a,ret)}\}}$$

While tasks have exclusive owners, the concurrent runner itself can be freely shared, to allow multiple threads to schedule tasks concurrently.

$$\overline{\mathsf{runner(t,x,P,Q)} \Leftrightarrow \mathsf{runner(t,x,P,Q)} * \mathsf{runner(t,x,P,Q)}}$$

With this specification it is easy to show that `parFib` implements fibonacci.

Representation predicates. The concurrent runner library is implemented using shared mutable state. In particular, each concurrent runner maintains a bag of scheduled tasks, which is shared between each worker thread. Upon creation of the concurrent runner, each worker thread enters an infinite loop, in which they keep popping and running tasks from the bag. Tasks can be in

one of two states, pending and executed. They start in the pending state and transition to the executed state once they have been popped and executed.

To verify the implementation, we use the shared bag specification to share the bag of tasks between each worker thread. We let each task own the resources needed by the delegate to perform the given task (i.e., the resources described by the delegate pre-condition P). We can thus define the runner representation predicate as follows:

$$
\begin{aligned}
&\mathsf{runner}(\mathsf{t},\mathsf{x},\mathsf{P},\mathsf{Q}) \stackrel{\text{def}}{=} \exists \mathsf{y},\mathsf{z} : \text{Val}.\ \exists \mathsf{t}_1,\mathsf{t}_2 : \text{RType}.\\
&\quad \mathsf{t} \leq \mathsf{t}_1 \wedge \mathsf{t} \leq \mathsf{t}_2 \wedge \mathsf{t}_1 \not\leq \mathsf{t}_2 \wedge \mathsf{t}_2 \not\leq \mathsf{t}_1 \wedge \mathsf{usip}(\mathsf{P}) \wedge \mathsf{usip}(\mathsf{Q})\\
&\quad \wedge\ \mathsf{stable}(\mathsf{P}) \wedge \mathsf{stable}(\mathsf{Q}) \wedge \mathsf{indep}_{\mathsf{t}}(\mathsf{P}) \wedge \mathsf{indep}_{\mathsf{t}}(\mathsf{Q})\\
&\quad *\ \mathsf{x.bag} \mapsto\!\!\!\!\rightarrow \mathsf{y} * \mathsf{x.body} \mapsto\!\!\!\!\rightarrow \mathsf{z} * \mathsf{protocol}(\mathsf{t}_2, \mathsf{I}(\mathsf{Q}))\\
&\quad *\ \mathsf{bag}_s(\mathsf{t}_1, \mathsf{y}, \lambda \mathsf{y} : \text{Val}.\ \mathsf{isTask}(\mathsf{t}_2, \mathsf{y}, \mathsf{x}, \mathsf{P}, \mathsf{Q}))\\
&\quad *\ \triangleright \left(\begin{array}{l} \mathsf{z} \mapsto (\mathsf{r},\mathsf{a}).\ \{\mathsf{runner}(\mathsf{t},\mathsf{r},\mathsf{P},\mathsf{Q}) * \mathsf{P}(\mathsf{a})\} \\ \qquad\qquad \{\mathsf{ret}.\ \mathsf{runner}(\mathsf{t},\mathsf{r},\mathsf{P},\mathsf{Q}) * \mathsf{Q}(\mathsf{a},\mathsf{ret})\} \end{array} \right)
\end{aligned}
$$

where $\mathsf{I}(\mathsf{Q})$ is the parametric protocol defined after the isTask predicate below. Note that, since we explicitly allow the delegate to assume it is called with a concurrent runner satisfying the chosen specification, the runner representation predicate needs to refer to itself to specify the registered delegate. Hence, we define runner by guarded recursion, as signaled by the use of the so-called later operator, $\triangleright$.

The $\mathsf{isTask}(\mathsf{t},\mathsf{y},\mathsf{x},\mathsf{P},\mathsf{Q})$ representation predicate asserts that y refers to a task in the pending state, and asserts ownership of the resources required to perform the given task. Thus, when a worker thread pops a task from the shared bag, it takes ownership of the resources needed to execute the delegate. Upon termination of the delegate, the resources produced by the delegate must be transferred back to the owner of the task, upon joining the task. To achieve this, we associate a shared region with each task, that owns the field containing the state of the task. This allows us to impose a protocol on the task state, that forces the worker thread to transfer the resources produced by the delegate to this shared region, when changing the state of the task from pending to executed. Lastly, to ensure only the owner of the task can take ownership of the resources, we give the owner of the task exclusive ownership of the action required to do so. We thus define isTask as follows:

$$
\begin{aligned}
&\mathsf{isTask}(\mathsf{t},\mathsf{x},\mathsf{y},\mathsf{P},\mathsf{Q}) \stackrel{\text{def}}{=} \exists \mathsf{r} : \text{RId}.\ \exists \mathsf{a} : \text{Val}.\\
&\quad \mathsf{x.runner} \mapsto\!\!\!\!\rightarrow \mathsf{y} * \mathsf{x.arg} \mapsto\!\!\!\!\rightarrow \mathsf{a} * \mathsf{P}(\mathsf{a}) * \mathsf{protocol}(\mathsf{t}, \mathsf{I}(\mathsf{Q}))\\
&\quad *\ \boxed{\mathsf{x.state} \mapsto 0 * \mathsf{x.res} \mapsto \_}^{\mathsf{r},\mathsf{t},(\mathsf{x},\mathsf{a})} * [\textsc{Exec}]_1^{\mathsf{r}} * [\textsc{SetRes}]_1^{\mathsf{r}}
\end{aligned}
$$

where $\mathsf{I}(\mathsf{Q})$ is the parametric protocol,

$$\mathsf{I}(\mathsf{Q})(\mathsf{x},\mathsf{a}) = \left(\begin{array}{rl} \textsc{SetRes}: & \mathsf{x.state} \mapsto 0 * \mathsf{x.res} \mapsto \mathsf{null} \ \rightsquigarrow \\ & \quad \exists \mathsf{r} : \mathrm{Val}.\ \mathsf{x.state} \mapsto 0 * \mathsf{x.res} \mapsto \mathsf{r} \\ \textsc{Exec}: & \exists \mathsf{r} : \mathrm{Val}.\ \mathsf{x.state} \mapsto 0 * \mathsf{x.res} \mapsto \mathsf{r} \ \rightsquigarrow \\ & \quad \exists \mathsf{r} : \mathrm{Val}.\ \mathsf{x.state} \mapsto 1 * \mathsf{x.res} \mapsto \mathsf{r} * \mathsf{Q}(\mathsf{a},\mathsf{r}) \\ \textsc{Join}: & \exists \mathsf{r} : \mathrm{Val}.\ \mathsf{x.state} \mapsto 1 * \mathsf{x.res} \mapsto \mathsf{r} * \mathsf{Q}(\mathsf{a},\mathsf{r}) \ \rightsquigarrow \\ & \quad \exists \mathsf{r} : \mathrm{Val}.\ \mathsf{x.state} \mapsto 1 * \mathsf{x.res} \mapsto \mathsf{r} \\ \tau_1: & \exists \mathsf{r} : \mathrm{Val}.\ \mathsf{x.state} \mapsto 0 * \mathsf{x.res} \mapsto \mathsf{r} \ \rightsquigarrow \\ & \quad \exists \mathsf{r} : \mathrm{Val}.\ \mathsf{x.state} \mapsto 0 * \mathsf{x.res} \mapsto \mathsf{r} \\ \tau_2: & \exists \mathsf{r} : \mathrm{Val}.\ \mathsf{x.state} \mapsto 1 * \mathsf{x.res} \mapsto \mathsf{r} * \mathsf{Q}(\mathsf{a},\mathsf{r}) \ \rightsquigarrow \\ & \quad \exists \mathsf{r} : \mathrm{Val}.\ \mathsf{x.state} \mapsto 1 * \mathsf{x.res} \mapsto \mathsf{r} * \mathsf{Q}(\mathsf{a},\mathsf{r}) \end{array}\right)$$

The owner of the isTask assertion thus owns $\mathsf{P}(\mathsf{a})$ and the full Exec action, to change the state of the task from pending (0) to executed (1), by transferring $\mathsf{Q}(\mathsf{a},\mathsf{r})$ to the shared region. Conversely, the owner of the task asserts full ownership of the Join action, allowing only the owner to grab $\mathsf{Q}(\mathsf{a},\mathsf{r})$, once the task has been executed:

$$\begin{array}{l} \mathsf{task}(\mathsf{x},\mathsf{y},\mathsf{P},\mathsf{Q},\mathsf{a}) \stackrel{\text{def}}{=} \exists \mathsf{r} : \mathrm{RId}.\ \exists \mathsf{t} : \mathrm{RType}. \\ \quad \mathsf{x.runner} \mapsto \vec{\ } \mathsf{y} * \mathsf{x.arg} \mapsto \vec{\ } \mathsf{a} * \mathsf{protocol}(\mathsf{t},\mathsf{I}(\mathsf{Q})) * [\textsc{Join}]^{\mathsf{r}}_1 * [\tau_1]^{\mathsf{r}}_1 * [\tau_2]^{\mathsf{r}}_1 \\ \quad * \boxed{\begin{array}{l} (\mathsf{x.state} \mapsto 0 * \mathsf{x.res} \mapsto \_) \vee \\ (\exists \mathsf{r} : \mathrm{Val}.\ \mathsf{x.state} \mapsto 1 * \mathsf{x.res} \mapsto \mathsf{r} * \mathsf{Q}(\mathsf{a},\mathsf{r})) \end{array}}^{\mathsf{r},\mathsf{t},(\mathsf{x},\mathsf{a})} \end{array}$$

To ensure that the isTask predicate is expressible using state-independent protocols, we use a parametric task protocol $\mathsf{I}(\mathsf{Q})$ and existentially quantify over the task region type t_2 in the runner predicate instead of the isTask predicate.

Stability. The isTask predicate is trivially stable under the Exec action, as it asserts full ownership of Exec. Its region assertion is also trivially expressible using state-independent protocols. It is thus also stable under the Join action, by rule SA, as $(\mathsf{x.state} \mapsto 0 * ...) \wedge (\mathsf{x.state} \mapsto 1 * ...) \Rightarrow \bot$. As isTask makes no assertions about the value of the res field, it is also easily shown to be stable under the SetRes, τ_1 and τ_2 actions, using rule SA.

Furthermore, the predicate $\lambda \mathsf{y} : \mathrm{Val}.\ \mathsf{isTask}(\mathsf{t},\mathsf{x},\mathsf{y},\mathsf{P},\mathsf{Q})$ is uniformly expressible using state-independent protocols, for any fixed t, y, P and Q, as P is uniformly expressible using state-independent protocols. Stability of runner thus follows easily from the stability of bag_s.

Lastly, task is trivially stable under the Join action, and by rewriting task using the $\mathsf{usip}(\mathsf{Q})$ assumption, it is easily proven to be stable under the Exec, SetRes, τ_1, and τ_2 actions, using the SA rule.

Proof outline. In this section we sketch the proof of concurrent runner implementation, in the form of a proof outline. To keep the size of the outline manageable, we will not write out the assumptions about stability, independence or expressibility using state-independent protocols on P and Q; nor will we write out the assumptions about the ordering on region types t, t_1 and t_2, when working with the runner predicate. Define S_{del} as follows:

$$\mathsf{S}_{del}(\mathsf{t},\mathsf{b},\mathsf{P},\mathsf{Q}) \stackrel{\text{def}}{=} \triangleright \left(\begin{array}{l} \mathsf{b} \mapsto (\mathsf{r},\mathsf{a}).\ \{\mathsf{runner}(\mathsf{t},\mathsf{r},\mathsf{P},\mathsf{Q}) * \mathsf{P}(\mathsf{a})\} \\ \qquad\qquad \{\mathsf{ret}.\ \mathsf{runner}(\mathsf{t},\mathsf{r},\mathsf{P},\mathsf{Q}) * \mathsf{Q}(\mathsf{a},\mathsf{ret})\} \end{array} \right)$$

```
public class Runner⟨A,B⟩ {
  internal readonly Func⟨Runner⟨A,B⟩,A,B⟩ body;
  internal readonly Bag⟨Task⟨A,B⟩⟩ bag;

  public Runner(Func⟨Runner⟨A,B⟩, A, B⟩ body, int n) {
    int i;
{body ↦ (r, a). {runner(t, r, P, Q) ∗ P(a)}{ret. runner(t, r, P, Q) ∗ Q(a, ret)}
    ∗ this.bag ↦ null ∗ this.body ↦ null}
    this.bag = new Bag⟨Task⟨A,B⟩⟩();
    this.body = body;
{S_del(t, body, P, Q) ∗ this.bag ↦̇ b ∗ this.body ↦̇ body
    ∗ bag_s(t_1, b, λy : Val. isTask(t_2, y, x, P, Q))}
{S_del(t, body, P, Q) ∗ this.bag ↦̇ b ∗ this.body ↦̇ body
    ∗ bag_s(t_1, b, λy : Val. isTask(t_2, y, x, P, Q)) ∗ protocol(t_2, I(Q))}
{runner(t, ret, P, Q)}
    i = 0;
    while (i < n) {
{runner(t, ret, P, Q)}
{runner(t, ret, P, Q) ∗ runner(t, ret, P, Q)}
      new Thread(runTasks).Start();
      i++;
{runner(t, ret, P, Q)}
    }
{runner(t, ret, P, Q)}
  }
```

The proof of the runner constructor is fairly straightforward; since assertions are downwards-closed in the step-index, we have $\mathsf{P} \Rightarrow \triangleright\mathsf{P}$, allowing us to "forget a step". The body and bag fields are never modified are their assignment in the constructor. We can thus freely share read-only access to these fields. Furthermore, nested Hoare triples are freely duplicable, allowing us to duplicate the runner assertion and transfer a runner assertion to each new worker thread. Note that the runner constructor also allocates a new task region type t_2 with the parametric protocol $\mathsf{I}(\mathsf{Q})$.

```
  public Task⟨A,B⟩ Fork(A arg) {
```

```
{runner(t, this, P, Q) ∗ P(arg)}
{S_del(t, body, P, Q) ∗ this.bag ↦→ b ∗ this.body ↦→ body
    ∗ bag_s(t_1, b, λy : Val. isTask(t_2, y, this, P, Q))
    ∗ protocol(t_2, I(Q)) ∗ protocol(t_2, I(Q))}
    Task⟨A,B⟩ task = new Task⟨A, B⟩(this, arg);
{S_del(t, body, P, Q) ∗ this.bag ↦→ b ∗ this.body ↦→ body
    ∗ bag_s(t_1, b, λy : Val. isTask(t_2, y, this, P, Q)) ∗ protocol(t_2, I(Q))
    ∗ isTask(t_2, task, this, P, Q) ∗ task(task, this, P, Q, arg)}
    tasks.Push(task);
{S_del(t, body, P, Q) ∗ this.bag ↦→ b ∗ this.body ↦→ body
    ∗ bag_s(t_1, b, λy : Val. isTask(t_2, y, this, P, Q))
    ∗ protocol(t_2, I(Q)) ∗ task(task, this, P, Q, arg)}
    return task;
{ret. task(ret, this, P, Q, arg)}
  }
```

The `Fork` method constructs a new task, resulting in an isTask assertion – asserting the task is concurrently pending and the resources and permission needed to execute it – and a task assertion – asserting ownership of the task. The fork method transfers the isTask assertion to the shared bag by calling `Push` and returns the task assertion to the caller.

```
  internal void runTasks() {
{runner(t, this, P, Q)}
    while(true) {
{runner(t, this, P, Q)}
      runTask();
{runner(t, this, P, Q)}
    }
{runner(t, this, P, Q)}
  }

  internal void runTask() {
{runner(s, this, P, Q)}
    Task⟨A,B⟩ t = tasks.Pop();
{s ≤ s_1 ∗ runner(s, this, P, Q) ∗ (t = null ∨ isTask(s_1, t, this, P, Q))}
    if (t != null) {
{s ≤ s_1 ∗ runner(s, this, P, Q) ∗ isTask(s_1, t, this, P, Q)}
      t.Run();
{runner(s, this, P, Q)}
    }
{runner(s, this, P, Q)}
  }
}
```

The `runTasks` and `runTask` methods are internal methods used by the worker threads to pop and execute tasks. Their proofs are fairly obvious.

```
public class Task⟨A, B⟩ {
  internal readonly Runner⟨A,B⟩ runner;
  internal readonly A arg;
  internal int state = 0;
  internal B res;

  internal Task(Runner<A,B> runner, A arg) {
{P(arg) ∗ protocol(t, I(Q)) ∗ this.runner ↦ null
    ∗ this.arg ↦ null ∗ this.state = 0 ∗ this.res ↦ null}
    this.runner = runner;
    this.arg = arg;
{P(arg) ∗ protocol(t, I(Q)) ∗ this.runner ↦ runner
    ∗ this.arg ↦ arg ∗ this.state = 0 ∗ this.res ↦ null}
{isTask(t, this, runner, P, Q) ∗ task(this, runner, P, Q, arg)}
  }
```

In the task constructor, we allocate a new region and transfer ownership of the status and res field to this new region. We explicitly *do not* allocate a new region type, but use the existing task region type t, allocated by the runner constructor.

```
  internal void Run() {
    Func⟨Runner⟨A,B⟩,A,B⟩ body;
    Runner⟨A,B⟩ runner;
    A arg; B tmp;
{t ≤ t2 ∗ runner(t, x, P, Q) ∗ isTask(t2, this, x, P, Q)}
    runner = this.runner;
    body = this.body;
    arg = this.arg;
{t ≤ t2 ∗ runner(t, x, P, Q) ∗ P(arg) ∗ [SetRes]^r_1 ∗ [Exec]^r_1
    ∗ this.state ↦ 0 ∗ this.res ↦ null  ^{r,t2,(this,arg)}  ∗ protocol(t2, I(Q))}
{t ≤ t2 ∗ runner(t, x, P, Q) ∗ S_del(t, body, P, Q) ∗ P(arg) ∗ [SetRes]^r_1 ∗ [Exec]^r_1
    ∗ this.state ↦ 0 ∗ this.res ↦ null  ^{r,t2,(this,arg)}  ∗ protocol(t2, I(Q))}
    tmp = body(runner, arg);
{t ≤ t2 ∗ runner(t, x, P, Q) ∗ S_del(t, body, P, Q) ∗ Q(arg, tmp) ∗ [SetRes]^r_1 ∗ [Exec]^r_1
    ∗ this.state ↦ 0 ∗ this.res ↦ null  ^{r,t2,(this,arg)}  ∗ protocol(t2, I(Q))}
    this.res = tmp;
{t ≤ t2 ∗ runner(t, x, P, Q) ∗ S_del(t, body, P, Q) ∗ Q(arg, tmp) ∗ [SetRes]^r_1 ∗ [Exec]^r_1
    ∗ this.state ↦ 0 ∗ this.res ↦ tmp  ^{r,t2,(this,arg)}  ∗ protocol(t2, I(Q))}
    state = 1;
{runner(t, x, P, Q)}
  }
```

To run a task, we first duplicate the nested Hoare triple assertion about the runner delegate, $S_{del}(...)$. From the isTask assertion, we know the task is pending, we own the resources necessary to execute the delegate and permission to change

its status afterwards. Each of the atomic updates, **this**.res = tmp and status = 1, requires a proof that the update satisfies the protocol. See the technical report for proof rules and examples of how we prove such atomic updates.

```
  public B Join() {
    Runner⟨A,B⟩ runner; B tmp;
{runner(t, y, P, Q) ∗ task(this, y, P, Q, a)}
    runner = this.runner;
{runner(t, runner, P, Q) ∗ task(this, runner, P, Q, a)}
    while (state != 1) {
{runner(t, runner, P, Q) ∗ task(this, runner, P, Q, a)}
      runner.runTask();
{runner(t, runner, P, Q) ∗ task(this, runner, P, Q, a)}
    }
{∃r : Val. this.state ↦ 1 ∗ this.res ↦ r ∗ Q(a, r)}^{r,t2,(this,a)}
    ∗ [JOIN]^r_1 ∗ [τ1]^r_1 ∗ [τ2]^r_1 ∗ protocol(t2, I(Q))}
    tmp = this.res;
{Q(a, tmp)}
    return tmp;
{ret. Q(a, ret)}
  }
}
```

Finally, the `Join` method uses the τ action to continually test whether the task has executed. If this test succeeds, from the definition of the task() predicate, the client knows the shared region contains the resources produced by the task. Using the JOIN action, the client can thus transfer these resources from the shared region to its local state.

E Modular Reasoning for Deterministic Parallelism

As mentioned in the introduction, Dodds et. al., recently proposed a higher-order variant of Concurrent Abstract Predicates [6] in their paper on "Modular Reasoning for Deterministic Parallelism". In their paper, the authors define a model for a higher-order variant of CAP and give proof rules for the specification logic, but not the CAP parts. Stability and atomic updates are left as semantic proof obligations in the model. The authors use this logic to verify a library for deterministic parallelism. The proof explicitly uses higher-order protocols and higher-order region assertions, without any restrictions to ensure the absence of self-referential region or protocol assertions or that the higher-order protocols are expressible using state-independent protocols. Instead of proving semantic proofs of stability and atomic updates in their model, the authors give informal proofs. This style of informal proof is unsound in the model proposed in [6]

in the presence of some self-referential region or protocol assertions and some state-dependent higher-order protocols.

To illustrate one stability counterexample in the presence of state-dependent protocols, define P as follows:

$$\mathsf{P} \stackrel{\text{def}}{=} (\mathsf{x} \mapsto 0 * (\boxed{\mathsf{y} \mapsto 0}^{\mathsf{r}}_{\mathsf{I}} \vee \boxed{\mathsf{y} \mapsto 0}^{\mathsf{r}}_{\mathsf{J}})) \vee (\mathsf{x} \mapsto 1 * \boxed{\mathsf{y} \mapsto 0}^{\mathsf{r}}_{\mathsf{J}})$$

where I and J are protocols with a single α action:

$$\mathsf{I} \stackrel{\text{def}}{=} (\alpha : \mathsf{y} \mapsto 1 \rightsquigarrow \mathsf{y} \mapsto 2) \qquad\qquad \mathsf{J} \stackrel{\text{def}}{=} (\alpha : \mathsf{y} \mapsto 1 \rightsquigarrow \mathsf{y} \mapsto 3)$$

Then P is stable, because both α actions require that $\mathsf{y} \mapsto 1$, which is impossible when P holds. Note that this P is not expressible using state-independent protocols, as the protocol assertions cannot be "pulled outside" the disjunctions. This P does not support modular stability proofs, due to the interpretation of protocols, which ignores protocol assertions in protocols. In particular, define Q as follows, where K is the protocol with a single α action:

$$\mathsf{Q} \stackrel{\text{def}}{=} \boxed{\mathsf{emp} \vee \mathsf{P}}^{\mathsf{r}'}_{\mathsf{K}} \qquad\qquad \mathsf{K} \stackrel{\text{def}}{=} (\alpha : \mathsf{emp} \rightsquigarrow \mathsf{P})$$

Then Q is not stable in the model proposed in [6], as the interpretation of K ignores any assertions P makes about protocols on r. Thus for P to be stable, it would have to be closed under the action $\mathsf{emp} \rightsquigarrow \mathsf{x} \mapsto 1$, which it is not, if the protocol on region r is I.

In the presence of state-dependent higher-order protocols it is thus unsound to treat propositional variables (referring to stable assertions) in protocols as black boxes. This renders the informal stability argument for the box and fut representation predicates defined in Figure 8 of [6] unsound.

Higher-order Concurrent Abstract Predicates

Kasper Svendsen
IT University of Copenhagen
kasv@itu.dk

Lars Birkedal
IT University of Copenhagen
birkedal@itu.dk

Matthew Parkinson
Microsoft Research, Cambridge
mattpark@microsoft.com

Contents

1 Mini C♯

In this section we define the syntax and semantics of a subset of C♯, dubbed mini C♯. In addition to the basic object-oriented and imperative features of C♯, this subset includes named delegates, fork concurrency, and compare-and-swap. We use a restricted syntax to simplify the presentation of the proof system.

1.1 Syntax

The syntax of the language is given below. In the syntax we use the following metavariables: f ranges over field names, m over method names, C over class names, x, y, z, o, and n over program variables. We denote the set of field names by *FName*, the set of method names by *MName*, the set of class names by *CName*, and the set of statements by *Stm*. We use an overbar for sequences.

L	::=	`class` C $\{\overline{\mathsf{Cf}}; \bar{\mathsf{M}}\}$	Class definition
M	::=	C m$(\overline{\mathsf{Cx}})$ $\{\overline{\mathsf{Cy}}; \bar{\mathsf{s}}; \texttt{return}\ \mathsf{z}\}$	Method definition
s	::=		Statement
		x = y	assignment
		x = null	initialization
		x = y.f	field access
		x.f = y	field update
		`if` (x == y) $\{\bar{\mathsf{s}}_1\}$ `else` $\{\bar{\mathsf{s}}_2\}$	conditional
		x = `new` C()	object creation
		x = `delegate` y.m	named delegate
		x = y.m$(\bar{\mathsf{z}})$	method invocation
		x = y$(\bar{\mathsf{z}})$	delegate application
		`fork`(x)	fork process
		x = `CAS`(y.f, o, n)	compare-and-swap

A new thread is forked by calling `fork` with a reference to a named delegate referring to a method with no parameters. The language does not feature a join statement[1], and the return value of a forked delegate is simply ignored. Each thread has a private stack, but all threads share a common heap.

The compare-and-swap statement, x = `CAS`(y.f, o, n), atomically compares the value of field f of object y with the value of o, and updates it with the value of n, if they match. In this case `CAS` also sets x to y; otherwise, `CAS` sets x to null.

The statement syntax does not include sequential composition. Instead we take the body of a method to be a sequence of statements. We use ; for concatenation of statement sequences. Hence, when we write $s_1; s_2$ for $s_1, s_2 \in \mathit{seq}\ \mathit{Stm}$, it does not implicitly follow that s_1 is non-empty.

[1]However, using compare-and-swap it is possible to code join-like functionality, and the logic is sufficiently strong to reason about such an encoding.

1.2 Operational Semantics

The operational semantics is a small-step interleaving semantics, giving a strong memory model. The semantic domains used in the definition of the operational semantics are defined below. We assume disjoint countably infinite sets of thread identifiers (*TId*), object identifiers (*OId*), and closure identifiers (*CId*).

$$
\begin{array}{rll}
t \in \mathit{TId} & & \text{thread identifiers} \\
o \in \mathit{OId} & & \text{object identifiers} \\
c \in \mathit{CId} & & \text{closure identifiers} \\
v \in \mathit{CVal} ::= \mathit{null} \mid o \mid c & & \text{C}^\sharp \text{ values} \\
\mathit{OHeap} \stackrel{\text{def}}{=} \mathit{OId} \times \mathit{FName} \xrightarrow{\text{fin}} \mathit{CVal} & & \text{object heap} \\
\mathit{THeap} \stackrel{\text{def}}{=} \mathit{OId} \xrightarrow{\text{fin}} \mathit{CName} & & \text{type heap} \\
\mathit{CHeap} \stackrel{\text{def}}{=} \mathit{CId} \xrightarrow{\text{fin}} \mathit{OId} \times \mathit{MName} & & \text{closure heap} \\
h \in \mathit{Heap} \stackrel{\text{def}}{=} \mathit{OHeap} \times \mathit{THeap} \times \mathit{CHeap} & & \text{heap} \\
l \in \mathit{Stack} \stackrel{\text{def}}{=} \mathit{Var} \xrightarrow{\text{fin}} \mathit{CVal} & & \text{stack} \\
\mathit{TCStack} \stackrel{\text{def}}{=} \mathit{seq}\ (\mathit{Stm} + (\mathit{Stack} \times \mathit{Var} \times \mathit{Var})) & & \text{thread call stack} \\
x, y, z \in \mathit{Thread} \stackrel{\text{def}}{=} \mathit{TId} \times \mathit{Stack} \times \mathit{TCStack} & & \text{thread} \\
T \in \mathit{TPool} \stackrel{\text{def}}{=} \{U \in \mathcal{P}_{fin}(\mathit{Thread}) \mid \mathit{utid}(U)\} & & \text{thread pool} \\
\mathit{Prg} \stackrel{\text{def}}{=} (\mathit{Heap} \uplus \{\lightning\}) \times \mathit{TPool} & & \text{program}
\end{array}
$$

Here *utid* expresses that thread identifiers are unique in a given thread pool:

$$\mathit{utid}(U) = \forall x, y \in U.\ x.t = y.t \Rightarrow x = y \qquad \text{for } U \in \mathcal{P}_{fin}(\mathit{Thread})$$

We use $x.t$, $x.l$, and $x.s$ to refer to the first, second and third component of a thread $x \in \mathit{Thread}$. We use $h.o$, $h.t$ and $h.c$ to refer to the object, type and closure heap of a heap $h \in \mathit{Heap}$.

A normal machine configuration consists of a heap and a pool of threads. The heap is global and shared between every thread. Each thread consists of a thread identifier, a private stack, and a call stack. Since we are using a small-step semantics, the call stack includes explicit returns to restore the stack at the end of method calls. The call stack is thus a sequence of programming language statements and method returns. Method returns consists of the stack prior to the method call, the variable the method return value should be assigned to, and the variable containing the method return value. We use $\mathit{return}\ (l, x, y)$ as notation for method returns. Formally, $\mathit{return}\ (l, x, y)$ is notation for $\mathit{inr}(l, x, y) \in \mathit{Stm} + (\mathit{Stack} \times \mathit{Var} \times \mathit{Var})$. We use $\mathit{stm}(s)$ as notation for $\mathit{map}(\mathit{inl})(s) \in \mathit{TCStack}$ when $s \in \mathit{Stm}$. We use $\cdot$ for cons-ing on call stacks and ; for concatenation of call stacks.

A faulty machine configuration consists of a special fault heap, $\lightning$, and a thread pool. We use this faulty configuration to indicate memory errors, such as an attempt to access a field that does not exist.

The presentation of the operational semantics is inspired by the Views framework [3], and factors the small-step program evaluation relation, $\rightarrow$, into a labelled thread pool evaluation relation, $\xrightarrow{a}$, and an action semantics, $[\![-]\!]$. The labelled thread pool evaluation relation, $\xrightarrow{a}$, defines the local effects (i.e., stack effects) of executing a single thread for one step of execution. The action semantics defines the global effects (i.e., heap effects) of executing an atomic action. These are related through the action label (a) of the thread pool evaluation relation. This factorization simplifies the definition of safety in Section 4. Due to this factorization, the labelled thread pool evaluation relation non-deterministically "guesses" the right values when reading from and allocating on the heap. For instance, the READ rule "guesses" the current value v of the given field in the heap. The semantics of the $read(-)$ action then enforces that the "guess" was correct, through the STEP rule.

Actions $\boxed{Act \in \text{Sets}}$

$$\begin{array}{rcl} a \in Act & ::= & id \mid read(o, f, v) \mid write(o, f, v) \mid cas(o, f, v_o, v_n, r) \\ & \mid & oalloc(T, o) \mid calloc(o, m, c) \mid otype(o, T) \mid ctype(c, T, o, m) \mid \lightning \end{array}$$

where $c \in CId$, $o \in OId$, $f \in FName$, $v, v_o, v_n, r \in CVal$, $T \in CName$, $m \in MName$.

Single-step semantics of non-forking statements

$$\boxed{\rightarrow \subseteq (Stack \times (Stm + (Stack \times Var \times Var))) \times Act \times (Stack \times TCStack)}$$

We use $\mathsf{body}(T, \mathsf{m})$ to refer to the body of method m from the T class. We use $\mathsf{fields}(T)$ to refer to the names of the fields defined by class T. Naturally, both body and fields are partial functions.

$$\frac{\mathsf{x}, \mathsf{y} \in dom(l)}{(l, inl(\mathsf{x} = \mathsf{y})) \xrightarrow{id} (l[\mathsf{x} \mapsto l(\mathsf{y})], \varepsilon)} \textsc{Assign}$$

$$\frac{\mathsf{x}, \mathsf{y} \in dom(l) \qquad l(\mathsf{x}) \in OId \qquad v \in CVal}{(l, inl(\mathsf{y} = \mathsf{x.f})) \xrightarrow{read(l(\mathsf{x}), \mathsf{f}, v)} (l[\mathsf{y} \mapsto v], \varepsilon)} \textsc{Read}$$

$$\frac{\mathsf{x}, \mathsf{y} \in dom(l) \qquad l(\mathsf{x}) \in OId}{(l, inl(\mathsf{x.f} = \mathsf{y})) \xrightarrow{write(l(\mathsf{x}), \mathsf{f}, l(\mathsf{y}))} (l, \varepsilon)} \textsc{Write}$$

$$\frac{\mathsf{x},\mathsf{y},\mathsf{o},\mathsf{n} \in dom(l) \qquad l(\mathsf{y}) \in OId \qquad v \in CVal}{(l, inl(\mathsf{x} = \mathtt{CAS}(\mathsf{y}.\mathsf{f},\mathsf{o},\mathsf{n}))) \xrightarrow{cas(l(\mathsf{y}),\mathsf{f},l(\mathsf{o}),l(\mathsf{n}),v)} (l[\mathsf{x} \mapsto v], \varepsilon)}\ \textsc{CAS}$$

$$\frac{\mathsf{x},\mathsf{y} \in dom(l) \qquad l(\mathsf{x}) = l(\mathsf{y})}{(l, inl(\mathtt{if}\ (\mathsf{x}\ \mathtt{==}\ \mathsf{y})\ \{\mathsf{s}_1\}\ \mathtt{else}\ \{\mathsf{s}_2\})) \xrightarrow{id} (l, stm(\mathsf{s}_1))}\ \textsc{IfT}$$

$$\frac{\mathsf{x},\mathsf{y} \in dom(l) \qquad l(\mathsf{x}) \neq l(\mathsf{y})}{(l, inl(\mathtt{if}\ (\mathsf{x}\ \mathtt{==}\ \mathsf{y})\ \{\mathsf{s}_1\}\ \mathtt{else}\ \{\mathsf{s}_2\})) \xrightarrow{id} (l, stm(\mathsf{s}_2))}\ \textsc{IfF}$$

$$\frac{\mathsf{x} \in dom(l) \qquad o \in OId}{(l, inl(\mathsf{x} = \mathtt{new}\ T)) \xrightarrow{oalloc(T,o)} (l[\mathsf{x} \mapsto o], \varepsilon)}\ \textsc{CAlloc}$$

$$\frac{\mathsf{x},\mathsf{y} \in dom(l) \qquad l(\mathsf{y}) \in OId \qquad c \in CId}{(l, inl(\mathsf{x} = \mathtt{delegate}\ \mathsf{y}.\mathsf{m})) \xrightarrow{calloc(l(\mathsf{y}),\mathsf{m},c)} (l[\mathsf{x} \mapsto c], \varepsilon)}\ \textsc{DAlloc}$$

$$\frac{\begin{array}{c}\mathsf{x},\mathsf{y},\bar{\mathsf{z}} \in dom(l) \qquad l(\mathsf{y}) \in OId \qquad T \in CName \\ \mathsf{body}(T,\mathsf{m}) = \mathsf{C}\ \mathsf{m}(\overline{\mathsf{C}\mathsf{x}})\{\overline{C\mathsf{y}}; \bar{\mathsf{s}}_2; \mathtt{return}\ \mathsf{r}\}\end{array}}{(l, inl(\mathsf{x} = \mathsf{y}.\mathsf{m}(\bar{\mathsf{z}}))) \xrightarrow{otype(l(\mathsf{y}),T)} ([\mathtt{this} \mapsto l(\mathsf{y}), \bar{\mathsf{x}} \mapsto l(\bar{\mathsf{z}}), \bar{\mathsf{y}} \mapsto null], stm(\bar{\mathsf{s}}_2); return\ (l,\mathsf{x},\mathsf{r}))}\ \textsc{MCall}$$

$$\frac{\begin{array}{c}\mathsf{x},\mathsf{y},\bar{\mathsf{z}} \in dom(l) \qquad l(\mathsf{y}) \in CId \qquad T \in CName \qquad o \in OId \\ \mathsf{body}(T,\mathsf{m}) = \mathsf{C}\ \mathsf{m}(\overline{\mathsf{C}\mathsf{x}})\{\overline{C\mathsf{y}}; \bar{\mathsf{s}}_2; \mathtt{return}\ \mathsf{r}\}\end{array}}{(l, inl(\mathsf{x} = \mathsf{y}(\bar{\mathsf{z}}))) \xrightarrow{ctype(l(\mathsf{y}),T,o,\mathsf{m})} ([\mathtt{this} \mapsto o, \bar{\mathsf{x}} \mapsto l(\bar{\mathsf{z}}), \bar{\mathsf{y}} \mapsto null], stm(\bar{\mathsf{s}}_2); return\ (l,\mathsf{x},\mathsf{r}))}\ \textsc{DCall}$$

$$\frac{\mathsf{x} \in dom(l_2) \qquad \mathsf{y} \in dom(l_1)}{(l_1, return\ (l_2,\mathsf{x},\mathsf{y})) \xrightarrow{id} (l_2[\mathsf{x} \mapsto l_1(\mathsf{y})], \varepsilon)}\ \textsc{Return}$$

All of the above rules require that the local stack variables involved exist on the stack and contain values in the right semantic domains. For instance, the Read rule requires that x and y exist on the stack and that x contains an object identifier. If these conditions are not met, read should fault, as expressed by the following rule.

$$\frac{\mathsf{x} \notin dom(l) \vee \mathsf{y} \notin dom(l) \vee l(\mathsf{x}) \notin OId}{(l, inl(\mathsf{y} = \mathsf{x}.\mathsf{f})) \xrightarrow{\lightning} (l, \varepsilon)}\ \textsc{ReadF}$$

The other fault cases are similar and have been omitted.

Single-step semantics $\boxed{\rightarrow \subseteq \mathit{Thread} \times \mathit{Act} \times \mathit{TPool}}$

$$\frac{(l, s) \xrightarrow{a} (l', s_1)}{(t, l, s \cdot s_2) \xrightarrow{a} \{(t, l', s_1; s_2)\}}\ \textsc{seq}$$

$$\frac{\begin{array}{c}\mathsf{x} \in \mathit{dom}(l) \qquad l(\mathsf{x}) \in \mathit{CId} \qquad T \in \mathit{CName} \qquad o \in \mathit{OId} \\ \mathsf{body}(T, \mathsf{m}) = \texttt{void m()}\{\overline{\mathsf{C}\mathsf{y}}; \bar{\mathsf{s}}_2; \texttt{return r}\} \qquad t_1 \neq t_2\end{array}}{(t_1, l, \mathit{inl}(\texttt{fork}(\mathsf{x})) \cdot s_1) \xrightarrow{\mathit{ctype}(l(\mathsf{x}), T, o, \mathsf{m})} \{(t_1, l, s_1), (t_2, [\texttt{this} \mapsto o, \bar{\mathsf{y}} \mapsto \texttt{null}], \mathit{stm}(\bar{\mathsf{s}}_2))\}}\ \textsc{Fork}$$

There is a corresponding fault rule for `fork`, in case the stack variables involved are not defined or contain values in the wrong semantic domains or the delegate refers to a method that does not exist.

Thread pool evaluation $\boxed{\rightarrow \subseteq \mathit{TPool} \times \mathit{Act} \times \mathit{TPool}}$

$$\frac{x \in T \qquad x \xrightarrow{a} T' \qquad \mathit{utid}((T \setminus \{x\}) \cup T')}{T \xrightarrow{a} (T \setminus \{x\}) \cup T'}$$

Single-step program evaluation $\boxed{\rightarrow \subseteq \mathit{Prg} \times \mathit{Prg}}$

$$\frac{T \xrightarrow{a} T' \qquad h' \in [\![a]\!](h)}{(h, T) \rightarrow (h', T')}\ \textsc{Step}$$

Multi-step program evaluation $\boxed{\rightarrow^n \subseteq \mathit{Prg} \times \mathit{Prg}}$

$$\frac{}{(h, T) \rightarrow^0 (h, T)} \qquad \frac{(h, T) \rightarrow (h'', T'') \qquad (h'', T'') \rightarrow^n (h', T')}{(h, T) \rightarrow^{n+1} (h', T')}$$

Irreducibility $\boxed{\mathit{irr} \in \mathit{Thread} \rightarrow 2, \mathit{irr} \in \mathit{TPool} \rightarrow 2}$

$$\begin{array}{ll} \mathit{irr}(x) \stackrel{\text{def}}{=} \forall a \in \mathit{Act}.\ \forall T \in \mathit{TPool}.\ x \not\xrightarrow{a} T & \text{for } x \in \mathit{Thread} \\ \mathit{irr}(T) \stackrel{\text{def}}{=} \forall x \in T.\ \mathit{irr}(x) & \text{for } T \in \mathit{TPool} \end{array}$$

Action semantics $\boxed{[\![-]\!] : Act \times Heap \to \mathcal{P}(Heap \uplus \{\lightning\})}$

$$[\![id]\!](h) = \{h\}$$
$$[\![\lightning]\!](h) = \{\lightning\}$$
$$[\![read(o,f,v)]\!](h) = \begin{cases} \{h\} & \text{if } (o,f) \in dom(h) \text{ and } h(o,f) = v \\ \emptyset & \text{if } (o,f) \in dom(h) \text{ and } h(o,f) \neq v \\ \{\lightning\} & \text{if } (o,f) \notin dom(h) \end{cases}$$
$$[\![write(o,f,v)]\!](h) = \begin{cases} \{h[(o,\mathsf{f}) \mapsto v]\} & \text{if } (o,f) \in dom(h) \\ \{\lightning\} & \text{if } (o,f) \notin dom(h) \end{cases}$$
$$[\![cas(o,f,v_o,v_n,r)]\!](h) = \begin{cases} \{h\} & \text{if } (o,f) \in dom(h),\ h(o,f) \neq v_o \text{ and } r = null \\ \{h[(o,f) \mapsto v_n]\} & \text{if } (o,f) \in dom(h),\ h(o,f) = v_o \text{ and } r = o \\ \{\lightning\} & \text{if } (o,f) \notin dom(h) \end{cases}$$
$$[\![oalloc(T,o)]\!](h) = \begin{cases} \{h[(o,\bar{\mathsf{f}}) \mapsto null, o \mapsto T]\} & \text{if } o \notin dom(h) \text{ and } \mathsf{fields}(T) = \bar{\mathsf{f}} \\ \emptyset & \text{if } o \in dom(h) \\ \{\lightning\} & \text{if } \mathsf{fields}(T) \text{ is undefined} \end{cases}$$
$$[\![calloc(o,m,c)]\!](h) = \begin{cases} \{h[c \mapsto (o,m)]\} & \text{if } c \notin dom(h) \\ \emptyset & \text{if } c \in dom(h) \end{cases}$$
$$[\![otype(o,T)]\!](h) = \begin{cases} \{h\} & \text{if } o \in dom(h_t) \text{ and } h_t(o) = T \\ \emptyset & \text{if } o \in dom(h_t) \text{ and } h_t(o) \neq T \\ \{\lightning\} & \text{if } o \notin dom(h_t) \end{cases}$$
$$[\![ctype(c,T,o,m)]\!](h) = \begin{cases} \{h\} & \text{if } c \in dom(h),\ o \in dom(h_t),\ h(c) = (o,m) \text{ and } h_t(o) = T \\ \emptyset & \text{if } c \in dom(h),\ o \in dom(h_t), \text{ and } h(c) \neq (o,m) \text{ or } h_t(o) \neq T \\ \{\lightning\} & \text{if } c \notin dom(h) \text{ or } o \notin dom(h_t) \end{cases}$$

Action semantics $\boxed{[\![-]\!] : Act \times (Heap \uplus \{\lightning\}) \to \mathcal{P}(Heap \uplus \{\lightning\})}$

$$[\![a]\!](x) = \begin{cases} [\![a]\!](x) & \text{if } x \in Heap \\ \{\lightning\} & \text{if } x = \lightning \end{cases}$$

Modifies set

$$\boxed{\begin{array}{l}\mathsf{mod}_1 : \mathit{Stm} \to \mathcal{P}(\mathit{Var}) \\ \mathsf{mod} : \mathit{seq}\ \mathit{Stm} \to \mathcal{P}(\mathit{Var})\end{array}}$$

$$\begin{aligned}
\mathsf{mod}_1(\mathsf{x} = \mathsf{y}) &= \{\mathsf{x}\} \\
\mathsf{mod}_1(\mathsf{x} = \mathsf{null}) &= \{\mathsf{x}\} \\
\mathsf{mod}_1(\mathsf{x} = \mathsf{y.f}) &= \{\mathsf{x}\} \\
\mathsf{mod}_1(\mathsf{x} = \texttt{new C}) &= \{\mathsf{x}\} \\
\mathsf{mod}_1(\mathsf{x} = \texttt{delegate y.m}) &= \{\mathsf{x}\} \\
\mathsf{mod}_1(\mathsf{x} = \texttt{CAS}(\mathsf{y.f}, \mathsf{o}, \mathsf{n})) &= \{\mathsf{x}\} \\
\mathsf{mod}_1(\mathsf{x} = \mathsf{y.m}(\bar{\mathsf{z}})) &= \{\mathsf{x}\} \\
\mathsf{mod}_1(\mathsf{x} = \mathsf{y}(\bar{\mathsf{z}})) &= \{\mathsf{x}\} \\
\mathsf{mod}_1(\mathsf{x.f} = \mathsf{y}) &= \emptyset \\
\mathsf{mod}_1(\texttt{fork}(\mathsf{x})) &= \emptyset \\
\mathsf{mod}_1(\texttt{if } (\mathsf{x} == \mathsf{y})\ \{\bar{\mathsf{s}}_1\}\ \texttt{ else }\ \{\bar{\mathsf{s}}_2\}) &= \mathsf{mod}(\bar{\mathsf{s}}_1) \cup \mathsf{mod}(\bar{\mathsf{s}}_2)
\end{aligned}$$

$$\begin{aligned}
\mathsf{mod}(\varepsilon) &= \emptyset \\
\mathsf{mod}(\mathsf{s}_1; \bar{\mathsf{s}}_2) &= \mathsf{mod}_1(\mathsf{s}_1) \cup \mathsf{mod}(\bar{\mathsf{s}}_2)
\end{aligned}$$

2 Logic

In this section we define a proof system for reasoning about mini C♯ programs. The proof system consists of several components.

- A higher-order separation logic for reasoning about mutable data structures. The main ingredient is a separating conjunction connective, written $p * q$, which asserts that p and q holds disjointly.
- A higher-order variant of concurrent abstract predicates for reasoning about shared mutable data structures. Concurrent abstract predicates partitions the state into regions with protocols to describe how the state in each region is allowed to evolve. This allows local reasoning about shared mutable data structures, by providing an abstraction of possible environment interference.
- A higher-order specification logic for modular reasoning about libraries. This allows libraries to be specified abstractly, by existentially quantifying over representation predicates in library specifications. Since representation predicates describe both the state and possible interference, this allows specifications to abstract both the internal data-representation and any internal parallelism.
- An embedding from assertions into specifications, to allow specifications to expose axioms about representation predicates to clients. An embedding from specifications into assertions, to allow reasoning about delegates.
- A later modality and guarded recursion for reasoning about mutually recursive methods and recursion through the store. The later modality internalizes a notion of an execution step into the proof system. This allows reasoning about mutually recursive methods, by induction on execution steps.
- Phantom fields to record an abstraction of the state and history of execution.

Combined into one proof system, these features allow us to reason abstractly about libraries and clients that combine shared mutable data structures, concurrency, and recursion through the store. For a realistic example that combines all these features, see our work on the Joins library [6, 7].

2.1 Syntax

Conceptually, the program logic consists of two layers: an assertion logic for reasoning about program states and a specification logic for reasoning about programs. Since the language features delegates, we allow specifications to be embedded in assertions. Likewise, to allow library specifications to expose axioms about representation predicates to clients, we also allow assertions to be embedded in specifications. Formally, the proof system thus contains a single term language, defined below, containing both specifications and assertions.

$$
\begin{array}{rrcl}
\text{Terms} & \mathsf{M},\mathsf{N},\mathsf{P},\mathsf{Q},\mathsf{S},\mathsf{T},\mathsf{F} & ::= & \lambda \mathsf{x} : \tau.\ \mathsf{M} \mid \mathsf{M}\ \mathsf{N} \mid \mathsf{x} \\
& & \mid & \bot \mid \top \mid \mathsf{M} \vee \mathsf{N} \mid \mathsf{M} \wedge \mathsf{M} \mid \mathsf{M} \Rightarrow \mathsf{N} \\
& & \mid & \forall \mathsf{x} : \tau.\ \mathsf{P} \mid \exists \mathsf{x} : \tau.\ \mathsf{P} \mid \mathsf{M} =_\tau \mathsf{N} \\
& & \mid & \mathsf{P} * \mathsf{Q} \mid \mathsf{P} \mathrel{-\!\!*} \mathsf{Q} \mid \mathsf{emp} \\
& & \mid & \mathsf{M}.\mathsf{F} \mapsto \mathsf{N} \mid \mathsf{M}_\mathsf{F} \mapsto \mathsf{N} \mid \mathsf{M} \mapsto \mathsf{N}.\mathsf{M} \mid \mathsf{M} : \mathsf{N} \\
& & \mid & C \mid \mathsf{m} \mid \mathsf{f} \mid \mathsf{null} \\
& & \mid & \bot \mid \mathsf{M} \leq \mathsf{N} \mid \mathsf{M} \sqcap \mathsf{N} \\
& & \mid & \boxed{\mathsf{P}}^{\mathsf{R},\mathsf{T},\mathsf{M}} \mid \mathsf{protocol}(\mathsf{R},\mathsf{M},\mathsf{N}) \mid [\mathsf{M}]^{\mathsf{R}}_{\mathsf{N}} \mid p \\
& & \mid & \mathsf{pure}_{\mathsf{protocol}}(\mathsf{P}) \mid \mathsf{pure}_{\mathsf{state}}(\mathsf{P}) \mid \mathsf{pure}_{\mathsf{perm}}(\mathsf{P}) \\
& & \mid & \mathsf{dep}_{\mathsf{T}}(\mathsf{P}) \mid \mathsf{indep}_{\mathsf{T}}(\mathsf{P}) \mid \mathsf{stable}(\mathsf{P}) \mid \mathsf{stable}^{\mathsf{R}}_{\mathsf{A}}(\mathsf{P}) \\
& & \mid & \mathsf{P} \sqsubseteq \mathsf{Q} \mid \mathsf{P} \sqsubseteq^{\mathsf{T}} \mathsf{Q} \mid (\Delta).\{\mathsf{P}\}\langle \mathsf{s}\rangle\{\mathsf{Q}\} \mid (\Delta).\{\mathsf{P}\}\langle \mathsf{s}\rangle^{\mathsf{T}}\{\mathsf{Q}\} \\
& & \mid & \mathsf{P} \rightsquigarrow^{\mathsf{N},\mathsf{M}} \mathsf{Q} \mid \mathsf{P} \rightsquigarrow^{\mathsf{N},\mathsf{M}}_{(\Delta).\langle \mathsf{s}\rangle} \mathsf{Q} \\
& & \mid & \triangleright \mathsf{M} \mid \mathsf{fix}_\tau(\mathsf{M}) \mid \mathsf{guarded}_\tau(\mathsf{M}) \\
& & \mid & \mathsf{valid}(\mathsf{P}) \mid \mathsf{asn}(\mathsf{S}) \\
& & \mid & \mathsf{M}.\mathsf{N} : (\Delta).\{\mathsf{P}\}\{\mathsf{x}.\mathsf{Q}\} \mid \mathsf{M} : (\Delta).\{\mathsf{P}\}\{\mathsf{x}.\mathsf{Q}\} \\
& & \mid & (\Delta).\{\mathsf{P}\}\bar{\mathsf{s}}\{\mathsf{Q}\}
\end{array}
$$

where p is a fraction in $(0, 1]$.

We use a type system to carve out specifications and assertions from this common term language. The set of types is generated by the following grammar:

$$
\begin{array}{rrcl}
\text{Types} & \tau,\sigma & ::= & 1 \mid \tau \to \sigma \mid \tau \times \sigma \mid \mathsf{Prop} \mid \mathsf{Spec} \\
& & \mid & \mathsf{Val} \mid \mathsf{Class} \mid \mathsf{Method} \mid \mathsf{Field} \\
& & \mid & \mathsf{Perm} \mid \mathsf{Action} \mid \mathsf{Region} \mid \mathsf{RType}
\end{array}
$$

and includes the standard type constructors for a simply-typed lambda calculus. Basic types include the type of assertions, Prop, the type of specifications, Spec, and the type of mathematical values, Val. The Val type includes all C♯ values and strings, and is closed under formation of pairs, such that mathematical sequences and other mathematical objects can be conveniently represented.[2] In addition, basic types include Class, Method, and Field, the types of C♯ classes, methods and fields, respectively. Finally, basic types include the type of permissions, Perm, the type of action identifiers, Action, the type of region identifiers, Region, and the type of region types, RType.

As a convention, we mostly use meta-variables P and Q for terms of type Prop, S and T for terms of type Spec, C for terms of type Class, and F for terms of type Field. We also use P and T for terms of type Perm and RType, respectively.

[2]We use a single universe Val for the universe of mathematical values to avoid also having to quantify over types in the logic. We omit explicit encodings of pairs and write $(v_1, ..., v_n)$ for tuples coded as elements of Val.

2.2 Typing

Terms are typed with the typing judgment $\Gamma; \Delta \vdash \mathsf{M} : \tau$, where Γ is a logical variable context and Δ a program variable context, and τ is a well-formed type. Well-formed types are given by the $\vdash \tau : \mathsf{Type}$ judgment. The type system thus consists of the following judgments:

$\vdash \tau : \mathsf{Type}$	τ is a well-formed type
$\Gamma; \Delta \vdash \mathsf{M} : \tau$	M is a well-formed term of type τ

where

$$\begin{array}{rcll} \Gamma & ::= & \Gamma, \mathsf{x} : \tau \mid \varepsilon & \text{logical variable context} \\ \Delta & ::= & \Delta, \mathsf{x} : \mathsf{Val} \mid \varepsilon & \text{program variable context} \end{array}$$

Specifications do not have any free program variables and are thus typed with an empty program variable context, Δ. Hence, we use $\Gamma \vdash \mathsf{M} : \tau$ as shorthand for $\Gamma; - \vdash \mathsf{M} : \tau$. We write Γ, Δ for the concatenation of Γ and Δ. Thus, $\Gamma, \Delta \vdash \mathsf{M} : \tau$ is shorthand for $\Gamma, \Delta; - \vdash \mathsf{M} : \tau$.

Well-formed types $\boxed{\vdash \tau : \mathsf{Type}}$

$$\frac{\vdash \tau : \mathsf{Type} \qquad \vdash \sigma : \mathsf{Type}}{\vdash \tau \times \sigma : \mathsf{Type}} \qquad \frac{\vdash \tau : \mathsf{Type} \qquad \vdash \sigma : \mathsf{Type}}{\vdash \tau \to \sigma : \mathsf{Type}}$$

$$\frac{}{\vdash 1 : \mathsf{Type}} \qquad \frac{}{\vdash \mathsf{Prop} : \mathsf{Type}} \qquad \frac{}{\vdash \mathsf{Spec} : \mathsf{Type}}$$

$$\frac{}{\vdash \mathsf{Class} : \mathsf{Type}} \qquad \frac{}{\vdash \mathsf{Method} : \mathsf{Type}} \qquad \frac{}{\vdash \mathsf{Field} : \mathsf{Type}} \qquad \frac{}{\vdash \mathsf{Val} : \mathsf{Type}}$$

$$\frac{}{\vdash \mathsf{Perm} : \mathsf{Type}} \qquad \frac{}{\vdash \mathsf{Action} : \mathsf{Type}} \qquad \frac{}{\vdash \mathsf{Region} : \mathsf{Type}} \qquad \frac{}{\vdash \mathsf{RType} : \mathsf{Type}}$$

Well-formed terms $\boxed{\Gamma; \Delta \vdash \mathsf{M} : \tau}$

Lambda calculus typing

$$\frac{\vdash \tau : \mathsf{Type} \qquad (\mathsf{x} : \tau) \in \Gamma}{\Gamma; \Delta \vdash \mathsf{x} : \tau} \qquad \frac{(\mathsf{x} : \mathsf{Val}) \in \Delta}{\Gamma; \Delta \vdash \mathsf{x} : \mathsf{Val}}$$

$$\frac{\Gamma, \mathsf{x} : \tau; \Delta \vdash \mathsf{M} : \sigma}{\Gamma; \Delta \vdash \lambda \mathsf{x} : \tau.\ \mathsf{M} : \tau \to \sigma} \qquad \frac{\Gamma; \Delta \vdash \mathsf{M} : \tau \to \sigma \qquad \Gamma; \Delta \vdash \mathsf{M} : \tau}{\Gamma; \Delta \vdash \mathsf{M}\ \mathsf{N} : \sigma}$$

To distinguish variables in the logic and meta-logic, we mostly use sans-serif identifiers, such as x in the logic, and italic identifiers, such as x in the meta-logic.

Assertion typing

$$\frac{}{\Gamma; \Delta \vdash \bot : \mathsf{Prop}} \qquad \frac{}{\Gamma; \Delta \vdash \top : \mathsf{Prop}}$$

$$\frac{\Gamma; \Delta \vdash \mathsf{P} : \mathsf{Prop} \qquad \Gamma; \Delta \vdash \mathsf{Q} : \mathsf{Prop}}{\Gamma; \Delta \vdash \mathsf{P} \wedge \mathsf{Q} : \mathsf{Prop}} \qquad \frac{\Gamma; \Delta \vdash \mathsf{P} : \mathsf{Prop} \qquad \Gamma; \Delta \vdash \mathsf{Q} : \mathsf{Prop}}{\Gamma; \Delta \vdash \mathsf{P} \vee \mathsf{Q} : \mathsf{Prop}}$$

$$\frac{\Gamma; \Delta \vdash \mathsf{P} : \mathsf{Prop} \qquad \Gamma; \Delta \vdash \mathsf{Q} : \mathsf{Prop}}{\Gamma; \Delta \vdash \mathsf{P} \Rightarrow \mathsf{Q} : \mathsf{Prop}} \qquad \frac{\Gamma; \Delta \vdash \mathsf{M} : \tau \qquad \Gamma; \Delta \vdash \mathsf{N} : \tau}{\Gamma; \Delta \vdash \mathsf{M} =_\tau \mathsf{N} : \mathsf{Prop}}$$

$$\frac{\Gamma, \mathsf{x} : \tau; \Delta \vdash \mathsf{P} : \mathsf{Prop}}{\Gamma; \Delta \vdash \forall \mathsf{x} : \tau.\ \mathsf{P} : \mathsf{Prop}} \qquad \frac{\Gamma, \mathsf{x} : \tau; \Delta \vdash \mathsf{P} : \mathsf{Prop}}{\Gamma; \Delta \vdash \exists \mathsf{x} : \tau.\ \mathsf{P} : \mathsf{Prop}}$$

Separation logic typing

$$\frac{}{\Gamma; \Delta \vdash \mathsf{emp} : \mathsf{Prop}} \qquad \frac{\Gamma; \Delta \vdash \mathsf{P} : \mathsf{Prop} \qquad \Gamma; \Delta \vdash \mathsf{Q} : \mathsf{Prop} \qquad op \in \{*, -\!\!*\}}{\Gamma; \Delta \vdash \mathsf{P}\ op\ \mathsf{Q} : \mathsf{Prop}}$$

C♯ typings

$$\frac{\mathsf{x} \in \Delta}{\Gamma; \Delta \vdash \mathsf{x} : \mathsf{Val}} \qquad \frac{}{\Gamma; \Delta \vdash \mathsf{null} : \mathsf{Val}}$$

$$\frac{}{\Gamma; \Delta \vdash \mathsf{C} : \mathsf{Class}} \qquad \frac{}{\Gamma; \Delta \vdash \mathsf{m} : \mathsf{Method}} \qquad \frac{}{\Gamma; \Delta \vdash \mathsf{f} : \mathsf{Field}}$$

$$\frac{\Gamma; \Delta \vdash \mathsf{M} : \mathsf{Val} \qquad \Gamma; \Delta \vdash \mathsf{C} : \mathsf{Class}}{\Gamma; \Delta \vdash \mathsf{M} : \mathsf{C} : \mathsf{Prop}}$$

$$\frac{\Gamma; \Delta \vdash \mathsf{M} : \mathsf{Val} \qquad \Gamma; \Delta \vdash \mathsf{F} : \mathsf{Field} \qquad \Gamma; \Delta \vdash \mathsf{N} : \mathsf{Val}}{\Gamma; \Delta \vdash \mathsf{M.F} \mapsto \mathsf{N} : \mathsf{Prop}}$$

$$\frac{\Gamma; \Delta \vdash \mathsf{M} : \mathsf{Val} \qquad \Gamma; \Delta \vdash \mathsf{F} : \mathsf{Field} \qquad \Gamma; \Delta \vdash \mathsf{N} : \mathsf{Val}}{\Gamma; \Delta \vdash \mathsf{M}_\mathsf{F} \mapsto \mathsf{N} : \mathsf{Prop}}$$

$$\frac{\Gamma; \Delta \vdash \mathsf{M}_1 : \mathsf{Val} \qquad \Gamma; \Delta \vdash \mathsf{N} : \mathsf{Val} \qquad \Gamma; \Delta \vdash \mathsf{M}_2 : \mathsf{Method}}{\Gamma; \Delta \vdash \mathsf{M}_1 \mapsto \mathsf{N}.\mathsf{M}_2 : \mathsf{Prop}}$$

Specification & assertion embedding

$$\frac{\Gamma \vdash \mathsf{S} : \mathsf{Spec}}{\Gamma; \Delta \vdash \mathsf{asn}(\mathsf{S}) : \mathsf{Prop}} \qquad \frac{\Gamma; - \vdash \mathsf{P} : \mathsf{Prop}}{\Gamma \vdash \mathsf{valid}(\mathsf{P}) : \mathsf{Spec}}$$

The embedding of specifications into the assertion logic allows us to *define* nested Hoare triples [4] for delegates, by embedding a specification of the named method the delegate refers to.

$$\begin{aligned}\mathsf{x} \mapsto (\Delta).\{\mathsf{P}\}\{\mathsf{r}.\mathsf{Q}\} \stackrel{\text{def}}{=} \exists \mathsf{y} : \mathsf{Val}.\ \exists \mathsf{m} : \mathsf{Method}.\ \exists \mathsf{C} : \mathsf{Class}.& \\ \mathsf{x} \mapsto \mathsf{y}.\mathsf{m} * \mathsf{y} : \mathsf{C} * \mathsf{asn}(\mathsf{C}.\mathsf{m} : (\Delta).\{\mathsf{P}\}\{\mathsf{r}.\mathsf{Q}\})&\end{aligned}$$

This asserts that x refers to a named method m on object y, that object y has dynamic type C, and that the method m from the C class satisfies the given Hoare specification.

Guarded recursion typing

$$\frac{\Gamma; \Delta \vdash \mathsf{M} : (\tau \to \mathsf{Prop}) \to (\tau \to \mathsf{Prop})}{\Gamma; \Delta \vdash \mathsf{fix}_\tau(\mathsf{M}) : \tau \to \mathsf{Prop}} \qquad \frac{\Gamma; \Delta \vdash \mathsf{P} : \mathsf{Prop}}{\Gamma; \Delta \vdash \triangleright \mathsf{P} : \mathsf{Prop}}$$

$$\frac{\Gamma; \Delta \vdash \mathsf{M} : (\tau \to \mathsf{Prop}) \to (\tau \to \mathsf{Prop})}{\Gamma; \Delta \vdash \mathsf{guarded}_\tau(\mathsf{M}) : \mathsf{Spec}}$$

We omit the type subscript from fix and guarded when it is clear from the context.

Region types typing

$$\frac{}{\Gamma; \Delta \vdash \bot : \mathsf{RType}} \qquad \frac{\Gamma; \Delta \vdash \mathsf{M} : \mathsf{RType} \qquad \Gamma; \Delta \vdash \mathsf{N} : \mathsf{RType}}{\Gamma; \Delta \vdash \mathsf{M} \leq \mathsf{N} : \mathsf{Prop}}$$

$$\frac{\Gamma; \Delta \vdash \mathsf{M} : \mathsf{RType} \qquad \Gamma; \Delta \vdash \mathsf{N} : \mathsf{RType}}{\Gamma; \Delta \vdash \mathsf{M} \sqcap \mathsf{N} : \mathsf{RType}}$$

Concurrent abstract predicates typing

$$\frac{\Gamma; \Delta \vdash \mathsf{P} : \mathsf{Prop} \qquad \Gamma; \Delta \vdash \mathsf{R} : \mathsf{Region} \qquad \Gamma; \Delta \vdash \mathsf{T} : \mathsf{RType} \qquad \Gamma; \Delta \vdash \mathsf{A} : \mathsf{Val}}{\Gamma; \Delta \vdash \boxed{\mathsf{P}}^{\mathsf{R},\mathsf{T},\mathsf{A}} : \mathsf{Prop}}$$

$$\frac{\Gamma; \Delta \vdash \mathsf{A} : \mathsf{Action} \qquad \Gamma; \Delta \vdash \mathsf{R} : \mathsf{Region} \qquad \Gamma; \Delta \vdash \mathsf{P} : \mathsf{Perm}}{\Gamma; \Delta \vdash [\mathsf{A}]^{\mathsf{R}}_{\mathsf{P}} : \mathsf{Prop}}$$

$$\frac{\Gamma;\Delta \vdash \mathsf{R} : \mathsf{Region} \qquad \Gamma;\Delta \vdash \mathsf{I}_p, \mathsf{I}_q : \mathsf{Val} \times \mathsf{Action} \times \mathsf{Val} \to \mathsf{Prop}}{\Gamma;\Delta \vdash \mathsf{protocol}(\mathsf{R}, \mathsf{I}_p, \mathsf{I}_q) : \mathsf{Prop}}$$

Formally, protocols are given by a pair of parameterized action pre- and post-conditions I_p and I_q. Formally, these are given as predicates on a protocol argument of type Val, an action identifier of type Action and a action parameter of type Val.

We use the following informal notation for a parametric protocol I with parameter a and actions α_1, ..., α_n:

$$\mathsf{I}(\mathsf{a}) = \begin{pmatrix} \alpha_1 : (\mathsf{x}_1, ..., \mathsf{x}_k).\ p_1(\mathsf{a}, \mathsf{x}_1, ..., \mathsf{x}_k) \rightsquigarrow q_1(\mathsf{a}, \mathsf{x}_1, ..., \mathsf{x}_k) \\ \vdots \\ \alpha_n : (\mathsf{x}_1, ..., \mathsf{x}_k).\ p_n(\mathsf{a}, \mathsf{x}_1, ..., \mathsf{x}_k) \rightsquigarrow q_n(\mathsf{a}, \mathsf{x}_1, ..., \mathsf{x}_k) \end{pmatrix}$$

Here $(\mathsf{a}, \mathsf{x}_1, ..., \mathsf{x}_k)$ is a context of logical variables of type Val, relating the action pre-condition p_i with the action post-condition q_i. Formally, this corresponds to the protocol given by the following action pre- and post-conditions:

$$\mathsf{I}_p(a, \alpha, y) = \begin{cases} p_i(a, \pi_1(y), ..., \pi_k(y)) & \text{if } \alpha = \alpha_i \\ \bot & \text{if } \forall i.\ \alpha \neq \alpha_i \end{cases}$$

$$\mathsf{I}_q(a, \alpha, y) = \begin{cases} q_i(a, \pi_1(y), ..., \pi_k(y)) & \text{if } \alpha = \alpha_i \\ \bot & \text{if } \forall i.\ \alpha \neq \alpha_i \end{cases}$$

where π denotes the assumed projection operation on Val. Furthermore, we use $\boxed{\mathsf{P}}_{\mathsf{I}}^{r,t,(a_1,...,a_k)}$ as shorthand for

$$\boxed{\mathsf{P}}_{\mathsf{I}_p,\mathsf{I}_q}^{r,t,\langle a_1,...,a_k\rangle}$$

where $\langle - \rangle$ denotes the assumed pairing operator on Val and I_p and I_q are the formal action pre- and post-conditions induced by the parametric protocol I.

$$\frac{0 < p \leq 1}{\Gamma;\Delta \vdash p : \mathsf{Perm}} \qquad \frac{}{\Gamma;\Delta \vdash a : \mathsf{Action}}$$

$$\frac{\Gamma \vdash \mathsf{P} : \mathsf{Prop} \qquad \Gamma \vdash \mathsf{Q} : \mathsf{Prop}}{\Gamma \vdash \mathsf{P} \sqsubseteq \mathsf{Q} : \mathsf{Spec}} \qquad \frac{\Gamma \vdash \mathsf{T} : \mathsf{RType} \quad \Gamma \vdash \mathsf{P} : \mathsf{Prop} \qquad \Gamma \vdash \mathsf{Q} : \mathsf{Prop}}{\Gamma \vdash \mathsf{P} \sqsubseteq^{\mathsf{T}} \mathsf{Q} : \mathsf{Spec}}$$

$$\frac{\Gamma;\Delta \vdash \mathsf{P} : \mathsf{Prop} \qquad \Gamma;\Delta \vdash \mathsf{Q} : \mathsf{Prop} \quad \Gamma;\Delta \vdash \mathsf{T} : \mathsf{RType} \qquad \Gamma;\Delta \vdash \mathsf{R} : \mathsf{Region}}{\Gamma \vdash \mathsf{P} \rightsquigarrow^{\mathsf{R},\mathsf{T}}_{(\Delta).\langle \mathsf{s} \rangle} \mathsf{Q} : \mathsf{Spec}} \qquad \frac{\Gamma \vdash \mathsf{P} : \mathsf{Prop} \qquad \Gamma \vdash \mathsf{Q} : \mathsf{Prop} \quad \Gamma \vdash \mathsf{T} : \mathsf{RType} \qquad \Gamma \vdash \mathsf{R} : \mathsf{Region}}{\Gamma \vdash \mathsf{P} \rightsquigarrow^{\mathsf{R},\mathsf{T}} \mathsf{Q} : \mathsf{Spec}}$$

$$\frac{\Gamma \vdash \mathsf{P} : \mathsf{Prop}}{\Gamma \vdash \mathsf{stable}(\mathsf{P}) : \mathsf{Spec}} \qquad \frac{\Gamma \vdash \mathsf{P} : \mathsf{Prop} \qquad \Gamma \vdash \mathsf{R} : \mathsf{Region} \qquad \Gamma \vdash \mathsf{A} : \mathsf{Action}}{\Gamma \vdash \mathsf{stable}^{\mathsf{R}}_{\mathsf{A}}(\mathsf{P}) : \mathsf{Spec}}$$

$$\frac{\Gamma \vdash \mathsf{P} : \mathsf{Prop} \qquad \Gamma \vdash \mathsf{T} : \mathsf{RType}}{\Gamma \vdash \mathsf{dep}_{\mathsf{T}}(\mathsf{P}) : \mathsf{Spec}} \qquad \frac{\Gamma \vdash \mathsf{P} : \mathsf{Prop} \qquad \Gamma \vdash \mathsf{T} : \mathsf{RType}}{\Gamma \vdash \mathsf{indep}_{\mathsf{T}}(\mathsf{P}) : \mathsf{Spec}}$$

$$\frac{\Gamma \vdash \mathsf{P} : \mathsf{Prop}}{\Gamma \vdash \mathsf{pure}_{\mathsf{perm}}(\mathsf{P}) : \mathsf{Spec}} \qquad \frac{\Gamma \vdash \mathsf{P} : \mathsf{Prop}}{\Gamma \vdash \mathsf{pure}_{\mathsf{state}}(\mathsf{P}) : \mathsf{Spec}} \qquad \frac{\Gamma \vdash \mathsf{P} : \mathsf{Prop}}{\Gamma \vdash \mathsf{pure}_{\mathsf{protocol}}(\mathsf{P}) : \mathsf{Spec}}$$

Specification typings

$$\frac{}{\Gamma \vdash \bot : \mathsf{Spec}} \qquad \frac{}{\Gamma \vdash \top : \mathsf{Spec}} \qquad \frac{\Gamma \vdash \mathsf{M} : \tau \qquad \Gamma \vdash \mathsf{N} : \tau}{\Gamma \vdash \mathsf{M} =_{\tau} \mathsf{N} : \mathsf{Spec}}$$

$$\frac{\Gamma \vdash \mathsf{S} : \mathsf{Spec} \qquad \Gamma \vdash \mathsf{T} : \mathsf{Spec}}{\Gamma \vdash \mathsf{S} \wedge \mathsf{T} : \mathsf{Spec}} \qquad \frac{\Gamma \vdash \mathsf{S} : \mathsf{Spec} \qquad \Gamma \vdash \mathsf{T} : \mathsf{Spec}}{\Gamma \vdash \mathsf{S} \vee \mathsf{T} : \mathsf{Spec}}$$

$$\frac{\Gamma \vdash \mathsf{S} : \mathsf{Spec} \qquad \Gamma \vdash \mathsf{T} : \mathsf{Spec}}{\Gamma \vdash \mathsf{S} \Rightarrow \mathsf{T} : \mathsf{Spec}} \qquad \frac{\Gamma \vdash \mathsf{S} : \mathsf{Spec}}{\Gamma \vdash \triangleright \mathsf{T} : \mathsf{Spec}}$$

$$\frac{\Gamma, \mathsf{x} : \tau \vdash \mathsf{S} : \mathsf{Spec}}{\Gamma \vdash \forall \mathsf{x} : \tau.\ \mathsf{S} : \mathsf{Spec}} \qquad \frac{\Gamma, \mathsf{x} : \tau \vdash \mathsf{S} : \mathsf{Spec}}{\Gamma \vdash \exists \mathsf{x} : \tau.\ \mathsf{S} : \mathsf{Spec}}$$

Hoare typings

The specification logic features five atomic propositions for Hoare-style partial correctness reasoning. The three basic Hoare-assertions (HoareS, HoareM and HoareC) are for specifying statements, methods and constructors, respectively. The proof rules for these assertions (See Section 2.3.5) enforce stability of the pre- and post-condition.

$$\frac{\Gamma; \Delta \vdash \mathsf{P} : \mathsf{Prop} \qquad \Gamma; \Delta \vdash \mathsf{Q} : \mathsf{Prop} \qquad \mathsf{FV}(\mathsf{s}) \subseteq \mathsf{vars}(\Delta)}{\Gamma \vdash (\Delta).\{\mathsf{P}\}\mathsf{s}\{\mathsf{Q}\} : \mathsf{Spec}}\ \textsc{HoareS}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{C} : \mathsf{Class} \qquad \Gamma \vdash \mathsf{M} : \mathsf{Method} \\ \Gamma; \Delta, \mathtt{this} \vdash \mathsf{P} : \mathsf{Prop} \qquad \Gamma; \Delta, \mathtt{this}, \mathsf{x} \vdash \mathsf{Q} : \mathsf{Prop}\end{array}}{\Gamma \vdash \mathsf{C.M} : (\Delta).\{\mathsf{P}\}\{\mathsf{x}.\mathsf{Q}\}}\ \textsc{HoareM}$$

$$\frac{\Gamma \vdash \mathsf{C} : \mathsf{Class} \qquad \Gamma \vdash \mathsf{P} : \mathsf{Prop} \qquad \Gamma; \mathsf{x} \vdash \mathsf{Q} : \mathsf{Prop}}{\Gamma \vdash \mathsf{C} : \{\mathsf{P}\}\{\mathsf{x}.\mathsf{Q}\}}\ \textsc{HoareC}$$

The specification logic features another two Hoare-assertions for specifying atomic statements. These rules specifically do *not* enforce stability of the pre- and post-condition.

$$\frac{\Gamma; \Delta \vdash \mathsf{P} : \mathsf{Prop} \qquad \Gamma; \Delta \vdash \mathsf{Q} : \mathsf{Prop}}{\Gamma \vdash (\Delta).\{\mathsf{P}\}\langle \mathsf{s} \rangle\{\mathsf{Q}\} : \mathsf{Spec}}$$

$$\frac{\Gamma; \Delta \vdash \mathsf{P} : \mathsf{Prop} \qquad \Gamma; \Delta \vdash \mathsf{Q} : \mathsf{Prop} \qquad \Gamma \vdash \mathsf{T} : \mathsf{RType}}{\Gamma \vdash (\Delta).\{\mathsf{P}\}\langle \mathsf{s} \rangle^{\mathsf{T}}\{\mathsf{Q}\} : \mathsf{Spec}}$$

2.3 Logics

The specification logic is given by the specification entailment judgment

$$\Gamma \mid \Phi \vdash \mathsf{S},$$

where S is a specification and Φ is a specification context:

$$\Phi ::= \Phi, \mathsf{S} \mid \varepsilon$$

such that $\Gamma \vdash \mathsf{S} : \mathsf{Spec}$ and $\Gamma \vdash \mathsf{T} : \mathsf{Spec}$ for each assumption T in Φ.

The assertion logic is given by the assertion entailment judgment

$$\Gamma; \Delta \mid \Phi \mid \mathsf{P} \vdash \mathsf{Q}$$

where P and Q are assertions, such that $\Gamma; \Delta \vdash \mathsf{P} : \mathsf{Prop}$ and $\Gamma; \Delta \vdash \mathsf{Q} : \mathsf{Prop}$ and $\Gamma \vdash \mathsf{T} : \mathsf{Spec}$ for each assumption T in Φ. The assertion entailment includes the specification context Φ, to allow the use of assertion assumptions embedded in specifications.

2.3.1 Assertion logic

The assertion logic includes a standard intuitionistic higher-order separation logic.

$$\frac{}{\Gamma; \Delta \mid \Phi \mid \bot \vdash \mathsf{P}} \qquad \frac{}{\Gamma; \Delta \mid \Phi \mid \mathsf{P} \vdash \top}$$

$$\frac{}{\Gamma; \Delta \mid \Phi \mid \mathsf{P} * \mathsf{Q} \vdash \mathsf{P}} \qquad \frac{\Gamma; \Delta \mid \Phi \mid \mathsf{P} \vdash \mathsf{Q} \qquad \Gamma; \Delta \mid \Phi \mid \mathsf{Q} \vdash \mathsf{R}}{\Gamma; \Delta \mid \Phi \mid \mathsf{P} \vdash \mathsf{R}}$$

$$\frac{\Gamma;\Delta \mid \Phi \mid \mathsf{P} \vdash \mathsf{R} \qquad \Gamma;\Delta \mid \Phi \mid \mathsf{Q} \vdash \mathsf{R}}{\Gamma;\Delta \mid \Phi \mid \mathsf{P} \vee \mathsf{Q} \vdash \mathsf{R}}$$

$$\frac{\Gamma;\Delta \mid \Phi \mid \mathsf{P} \vee \mathsf{Q} \vdash \mathsf{R}}{\Gamma;\Delta \mid \Phi \mid \mathsf{P} \vdash \mathsf{R}} \qquad \frac{\Gamma;\Delta \mid \Phi \mid \mathsf{P} \vee \mathsf{Q} \vdash \mathsf{R}}{\Gamma;\Delta \mid \Phi \mid \mathsf{Q} \vdash \mathsf{R}}$$

$$\frac{\Gamma;\Delta \mid \Phi \mid \mathsf{P} \wedge \mathsf{Q} \vdash \mathsf{R}}{\Gamma;\Delta \mid \Phi \mid \mathsf{P} \vdash \mathsf{Q} \Rightarrow \mathsf{R}} \qquad \frac{\Gamma;\Delta \mid \Phi \mid \mathsf{P} \vdash \mathsf{Q} \Rightarrow \mathsf{R}}{\Gamma;\Delta \mid \Phi \mid \mathsf{P} \wedge \mathsf{Q} \vdash \mathsf{R}}$$

$$\frac{\Gamma;\Delta \mid \Phi \mid \mathsf{P} \vdash \mathsf{Q} \mathrel{-\!\!*} \mathsf{R}}{\Gamma;\Delta \mid \Phi \mid \mathsf{P} * \mathsf{Q} \vdash \mathsf{R}} \qquad \frac{\Gamma;\Delta \mid \Phi \mid \mathsf{P} * \mathsf{Q} \vdash \mathsf{R}}{\Gamma;\Delta \mid \Phi \mid \mathsf{P} \vdash \mathsf{Q} \mathrel{-\!\!*} \mathsf{R}}$$

$$\frac{\Gamma;- \mid \Phi \mid \top \vdash \mathsf{P}}{\Gamma \mid \Phi \vdash \mathsf{valid}(\mathsf{P})} \qquad \frac{\Gamma \mid \Phi \vdash \mathsf{valid}(\mathsf{P})}{\Gamma;- \mid \Phi \mid \top \vdash \mathsf{P}}$$

2.3.2 Specification logic

The specification logic includes a standard intuitionistic higher-order logic.

$$\frac{}{\Gamma \mid \Phi, \bot \vdash \mathsf{S}} \qquad \frac{}{\Gamma \mid \Phi \vdash \top} \qquad \frac{\Gamma \mid \Phi, \triangleright\mathsf{S} \vdash \mathsf{S}}{\Gamma \mid \Phi \vdash \mathsf{S}}$$

$$\frac{\Gamma \mid \Phi \vdash \mathsf{S} \qquad \Gamma \mid \Phi \vdash \mathsf{T}}{\Gamma \mid \Phi \vdash \mathsf{S} \wedge \mathsf{T}} \qquad \frac{\Gamma \mid \Phi \vdash \mathsf{S}}{\Gamma \mid \Phi \vdash \mathsf{S} \vee \mathsf{T}} \qquad \frac{\Gamma \mid \Phi \vdash \mathsf{T}}{\Gamma \mid \Phi \vdash \mathsf{S} \vee \mathsf{T}}$$

$$\frac{\Gamma \mid \Phi \vdash \mathsf{S} \wedge \mathsf{T}}{\Gamma \mid \Phi \vdash \mathsf{S}} \qquad \frac{\Gamma \mid \Phi \vdash \mathsf{S} \wedge \mathsf{T}}{\Gamma \mid \Phi \vdash \mathsf{T}} \qquad \frac{\Gamma \mid \Phi \vdash \mathsf{S}_1 \vee \mathsf{S}_2 \qquad \Gamma \mid \Phi, \mathsf{S}_i \vdash \mathsf{T}}{\Gamma \mid \Phi \vdash \mathsf{T}}$$

$$\frac{\Gamma \mid \Phi, \mathsf{S} \vdash \mathsf{T}}{\Gamma \mid \Phi \vdash \mathsf{S} \Rightarrow \mathsf{T}} \qquad \frac{\Gamma \mid \Phi \vdash \mathsf{S} \Rightarrow \mathsf{T} \qquad \Gamma \mid \Phi \vdash \mathsf{S}}{\Gamma \mid \Phi \vdash \mathsf{T}}$$

$$\frac{\Gamma \mid \Phi \vdash \forall \mathsf{x} : \tau.\ \mathsf{S} \qquad \Gamma \vdash \mathsf{M} : \tau}{\Gamma \mid \Phi \vdash \mathsf{S}[\mathsf{M}/\mathsf{x}]} \qquad \frac{\Gamma, \mathsf{x} : \tau \mid \Phi \vdash \mathsf{S} \qquad \mathsf{x} \notin \mathsf{FV}(\Phi)}{\Gamma \mid \Phi \vdash \forall \mathsf{x} : \tau.\ \mathsf{S}}$$

$$\frac{\Gamma \mid \Phi \vdash \mathsf{S}[\mathsf{M}/\mathsf{x}] \qquad \Gamma \vdash \mathsf{M} : \tau}{\Gamma \mid \Phi \vdash \exists \mathsf{x} : \tau.\ \mathsf{S}} \qquad \frac{\Gamma, \mathsf{x} : \tau \mid \Phi, \mathsf{S} \vdash \mathsf{T} \qquad \mathsf{x} \notin \mathsf{FV}(\Phi, \mathsf{T})}{\Gamma \mid \Phi, \exists \mathsf{x} : \tau.\ \mathsf{S} \vdash \mathsf{T}}$$

$$\frac{\Gamma \mid \mathsf{M} : \tau}{\Gamma \mid \Phi \vdash \mathsf{M} =_\tau \mathsf{M}} \qquad \frac{\Gamma \mid \Phi \vdash \mathsf{M} =_\tau \mathsf{N} \qquad \Gamma \mid \Phi \vdash \mathsf{S}[\mathsf{M}/x]}{\Gamma \mid \Phi \vdash \mathsf{S}[\mathsf{N}/x]}$$

$$\frac{\Gamma \mid \Phi, \mathsf{S} \vdash (\Delta).\{\mathsf{P}\}\bar{\mathsf{s}}\{\mathsf{Q}\}}{\Gamma \mid \Phi \vdash (\Delta).\{\mathsf{P} * \mathsf{asn}(\mathsf{S})\}\bar{\mathsf{s}}\{\mathsf{Q}\}}\ \textsc{AsnI} \qquad \frac{\Gamma \mid \Phi \vdash (\Delta).\{\mathsf{P} * \mathsf{asn}(\mathsf{S})\}\bar{\mathsf{s}}\{\mathsf{Q}\}}{\Gamma \mid \Phi, \mathsf{S} \vdash (\Delta).\{\mathsf{P}\}\bar{\mathsf{s}}\{\mathsf{Q}\}}\ \textsc{AsnE}$$

2.3.3 Guarded recursion and later modalities

The logic features a fixed-point operator fix on predicate functionals, which defines a fixed-point when applied to guarded predicate functionals.

$$\frac{\Gamma \vdash \mathsf{M} : (\tau \to \mathsf{Prop}) \to (\tau \to \mathsf{Prop}) \qquad \Gamma \mid \Phi \vdash \mathsf{guarded}(\mathsf{M})}{\Gamma \mid \Phi \vdash \forall \mathsf{x} : \tau.\ \mathsf{valid}(\mathsf{fix}(\mathsf{M})(\mathsf{x}) \Leftrightarrow \mathsf{M}(\mathsf{fix}(\mathsf{M}))(\mathsf{x}))}$$

Definitions are guarded using the later modality, $\triangleright$. To support modular guardedness proofs, we build non-expansiveness into the interpretation. A definition is thus guarded, if it can be expressed as a composition of functions where one function is guarded:

$$\frac{\Gamma \vdash \mathsf{M}_1, \mathsf{M}_2, \mathsf{M}_3 : (\tau \to \mathsf{Prop}) \to (\tau \to \mathsf{Prop}) \qquad \Gamma \mid \Phi \vdash \mathsf{guarded}(\mathsf{M}_2)}{\Gamma \mid \Phi \vdash \mathsf{guarded}(\mathsf{M}_3 \circ \mathsf{M}_2 \circ \mathsf{M}_1)}$$

where $\mathsf{M} \circ \mathsf{N} = \lambda P : \tau \to \mathsf{Prop}.\ \mathsf{M}(\mathsf{N}(P))$. Furthermore, recursive predicates are guarded if all occurrences of the recursive predicate appears under $\triangleright$ operators:

$$\frac{\Gamma \vdash \mathsf{M} : (\tau \to \mathsf{Prop}) \to (\tau \to \mathsf{Prop})}{\Gamma \mid \Phi \vdash \mathsf{guarded}(\lambda \mathsf{P} : \tau \to \mathsf{Prop}.\ \mathsf{M} \circ (\lambda \mathsf{x} : \tau.\ \triangleright \mathsf{P}(\mathsf{x})))}\ \textsc{guardI}$$

$$\frac{\Gamma \vdash \mathsf{M} : (\tau \to \mathsf{Prop}) \to (\tau \to \mathsf{Prop})}{\Gamma \mid \Phi \vdash \mathsf{guarded}(\lambda \mathsf{P} : \tau \to \mathsf{Prop}.\ \lambda \mathsf{x} : \tau.\ \triangleright (\mathsf{M} \circ \mathsf{P})(\mathsf{x}))}\ \textsc{guardE}$$

The later modality internalizes a notion of step in the logic. If the specification S holds for i steps of execution, then $\triangleright\mathsf{S}$ holds for $i+1$ steps of execution. This allows us to reason about mutually recursive methods and recursion through the store, using the Loeb rule, which internalizes induction on steps in the model. See Section 3.1 for an example that uses recursion through the store.

$$\frac{\Gamma \mid \Phi, \triangleright \mathsf{S} \vdash \mathsf{S}}{\Gamma \mid \Phi \vdash \mathsf{S}}\ \textsc{Loeb} \qquad\qquad \frac{}{\Gamma \mid \Phi, \mathsf{S} \vdash \triangleright \mathsf{S}}$$

The $\triangleright$-operator commutes with existentials over inhabited types (and every type is currently inhabited), and over separating conjunction.

$$\frac{\Gamma, \mathsf{x} : \tau \vdash \mathsf{P} : \mathsf{Prop} \qquad \mathit{inhabited}(\tau)}{\Gamma \mid \Phi \vdash \mathsf{valid}(\triangleright(\exists \mathsf{x} : \tau.\ \mathsf{P}) \Leftrightarrow (\exists \mathsf{x} : \tau.\ \triangleright \mathsf{P}))} \qquad \frac{\Gamma \vdash \mathsf{P} : \mathsf{Prop} \qquad \Gamma \vdash \mathsf{Q} : \mathsf{Prop}}{\Gamma \mid \Phi \vdash \mathsf{valid}(\triangleright(\mathsf{P} * \mathsf{Q}) \Leftrightarrow (\triangleright\mathsf{P} * \triangleright\mathsf{Q}))}$$

2.3.4 Concurrent abstract predicates

The logic includes a large set of inference rules for reasoning about concurrent abstract predicates in the logic, without resorting to the semantics. This logic suffices for verifying realistic examples such as the spin-lock in Section 3.2 and the joins library (See [6] and accompanying technical report [7]). The logic for reasoning about concurrent abstract predicates consists of several parts:

- A logic for reasoning about stability. The logic allows stability proofs to be decomposed into stability under individual actions. The corner-stone of the stability logic is rule StableA, which allows one to prove stability of a region assertion containing nested (but not self-referential) region assertions.
- A logic for reasoning about accesses and updates to resources in a shared region. The main rule of this logic is OpenA, which allows a shared region to be "opened", moving the shared resources into the local state.
- A logic for reasoning about view-shifts on shared regions. Like the logic for reasoning about accesses and updates, the main rule, OpenV, allows a shared region to be "opened".
- A logic for reasoning about whether an atomic access or update to the resources of a shared region is allowed by the protocol on the given region.
- A logic for reasoning about whether a view-shift on the resources of a shared region is allowed by the protocol on the given region.
- A logic for reasoning about the region support of an assertion, to prove the absence of self-referential region and protocol assertions.
- Logics for reasoning about whether assertions makes assertions about protocols and state, respectively, to prove that assertions are expressible using state-independent protocols.

Stability

Stability is closed under all the standard assertion connectives and quantifiers.

$$\frac{}{\Gamma \mid \Phi \vdash \mathsf{stable}(\bot)} \qquad \frac{}{\Gamma \mid \Phi \vdash \mathsf{stable}(\top)} \qquad \frac{}{\Gamma \mid \Phi \vdash \mathsf{stable}(\mathsf{emp})}$$

$$\frac{\Gamma \mid \Phi \vdash \mathsf{stable}(\mathsf{P}) \qquad \Gamma \mid \Phi \vdash \mathsf{stable}(\mathsf{Q}) \qquad op \in \{\vee, \wedge, \Rightarrow, *\}}{\Gamma \mid \Phi \vdash \mathsf{stable}(\mathsf{P}\ op\ \mathsf{Q})}$$

$$\frac{\Gamma \mid \Phi \vdash \forall \mathsf{x} : \tau.\ \mathsf{stable}(\mathsf{P})}{\Gamma \mid \Phi \vdash \mathsf{stable}(\forall \mathsf{x} : \tau.\ \mathsf{P})} \qquad \frac{\Gamma \mid \Phi \vdash \forall \mathsf{x} : \tau.\ \mathsf{stable}(\mathsf{P})}{\Gamma \mid \Phi \vdash \mathsf{stable}(\exists \mathsf{x} : \tau.\ \mathsf{P})}$$

Stability decomposes into stability under every action, considered individually.

$$\frac{\Gamma \vdash \mathsf{P} : \mathsf{Prop} \qquad \Gamma \vdash \mathsf{R} : \mathsf{Region}}{\Gamma \mid \Phi \vdash (\forall \alpha : \mathsf{Action}.\ \mathsf{stable}^{\mathsf{R}}_{\alpha}(\mathsf{P})) \Leftrightarrow \mathsf{stable}(\mathsf{P})}\ \textsc{StableD}$$

Here the $\mathsf{stable}^{\mathsf{R}}_{\alpha}(\mathsf{P})$ specification expresses that the assertion P is stable under arbitrary actions on any region other than R and the α action on region R (See definition $stable^{r,A}$ on page 182 for the formal semantics of $\mathsf{stable}^{\mathsf{R}}_{\alpha}$). A proposition P is thus stable if there *exists* an R such that $\mathsf{stable}^{\mathsf{R}}_{\alpha}(\mathsf{P})$ holds for all actions α.

An assertion is stable under actions that it owns (StableOwn). Furthermore, a region assertion is stable under an action if it is closed under every instantiation of the action that satisfies the pre-condition (StableClosed).

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{P} : \mathsf{Prop} \qquad \Gamma \vdash \mathsf{R} : \mathsf{Region} \\ \Gamma \vdash \mathsf{T} : \mathsf{RType} \qquad \Gamma \vdash \mathsf{A} : \mathsf{Val} \qquad \Gamma \vdash \mathsf{I}_p, \mathsf{I}_q : \mathsf{Val} \times \mathsf{Action} \times \mathsf{Val} \to \mathsf{Prop}\end{array}}{\Gamma \mid \Phi \vdash \mathsf{stable}^{\mathsf{R}}_{\alpha}\left(\boxed{\mathsf{P}}^{\mathsf{R},\mathsf{T},\mathsf{A}}_{\mathsf{I}_p,\mathsf{I}_q} * [\alpha]^{\mathsf{R}}_{1}\right)} \textsc{StableOwn}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{R} : \mathsf{Region} \qquad \Gamma \vdash \mathsf{T} : \mathsf{RType} \qquad \Gamma \vdash \mathsf{A} : \mathsf{Val} \qquad \Gamma \vdash \alpha : \mathsf{Action} \\ \Gamma \vdash \mathsf{P}, \mathsf{Q} : \mathsf{Prop} \qquad \Gamma \vdash \mathsf{I}_p, \mathsf{I}_q : \mathsf{Val} \times \mathsf{Action} \times \mathsf{Val} \to \mathsf{Prop} \\ \Gamma \mid \Phi \vdash \mathsf{pure}_{\mathsf{protocol}}(\mathsf{P}) \qquad \Gamma \mid \Phi \vdash \mathsf{pure}_{\mathsf{state}}(\mathsf{Q}) \\ \Gamma \mid \Phi \vdash \mathsf{stable}(\mathsf{P} * \mathsf{Q}) \qquad \Gamma \mid \Phi \vdash \mathsf{indep}_{\mathsf{T}}(\mathsf{P}) \\ \Gamma \mid \Phi \vdash \forall \mathsf{x} : \mathsf{Val}.\ \mathsf{valid}((\mathsf{I}_p(\mathsf{A}, \alpha, \mathsf{x}) \wedge \mathsf{P}) \Rightarrow \bot) \vee \mathsf{valid}(\mathsf{I}_q(\mathsf{A}, \alpha, \mathsf{x}) \Rightarrow \mathsf{P})\end{array}}{\Gamma \mid \Phi \vdash \mathsf{stable}^{\mathsf{R}}_{\alpha}\left(\boxed{\mathsf{P}}^{\mathsf{R},\mathsf{T},\mathsf{A}}_{\mathsf{I}_p,\mathsf{I}_q} * \mathsf{Q}\right)} \textsc{StableClosed}$$

Since the underlying model lacks support for general higher-order protocols, the StableClosed rule requires that the region assertion can be expressed using state-independent protocols. We say an assertion R can be expressed using state-independent protocols if it can be written as $\mathsf{S} * \mathsf{P}$ where S makes no assertions about protocols and P makes no assertions about the state:

$$\mathsf{sip} = \lambda \mathsf{R} : \mathsf{Prop}.\ \exists \mathsf{S}, \mathsf{P} : \mathsf{Prop}.\ \mathsf{valid}(\mathsf{R} \Leftrightarrow \mathsf{S} * \mathsf{P}) \wedge \mathsf{pure}_{\mathsf{protocol}}(\mathsf{S}) \wedge \mathsf{pure}_{\mathsf{state}}(\mathsf{P})$$

Here $\mathsf{pure}_{\mathsf{protocol}}(\mathsf{S})$ asserts that S is invariant under arbitrary changes to protocols and $\mathsf{pure}_{\mathsf{state}}(\mathsf{P})$ asserts that P is invariant under arbitrary changes to the local and shared state.

For assertions that are expressible using state-independent protocols, protocol assertions can be "pulled outside" region assertions, to allow the StableClosed rule to be used. In particular, if $\mathsf{valid}(\mathsf{R} \Leftrightarrow (\mathsf{S} * \mathsf{P}))$, $\mathsf{pure}_{\mathsf{protocol}}(\mathsf{S})$ and $\mathsf{pure}_{\mathsf{state}}(\mathsf{P})$, then

$$\boxed{\mathsf{R}}^{\mathsf{R},\mathsf{T},\mathsf{A}}_{\mathsf{I}_p,\mathsf{I}_q} \Leftrightarrow \boxed{\mathsf{S}}^{\mathsf{R},\mathsf{T},\mathsf{A}}_{\mathsf{I}_p,\mathsf{I}_q} * \mathsf{P}$$

The left-hand side is thus stable if and only if the right-hand side is, and we can apply the StableClosed rule to the right-hand side.

$$\frac{\Gamma \mid \Phi, \mathsf{S} \vdash \mathsf{stable}(\mathsf{P}) \qquad \Gamma; - \mid \Phi \mid \mathsf{P} \vdash \mathsf{asn}(\mathsf{S})}{\Gamma \mid \Phi \vdash \mathsf{stable}(\mathsf{P})} \textsc{StableAsn}$$

When defining higher-order representation predicates, we will often use specification assertions to constrain instantiations of assertion and predicate arguments. The above rule allows us to copy such embedded specifications from an assertion P into the specification context, when proving stability of P.

View-shifts and atomic updates

The following rules provide a small logic for reasoning about view-shifts and atomic statements that update or access resources in shared regions. The cornerstone of the logic are the rules for opening a region (rules OpenV and OpenA). Standard separation logic rules for updating and accessing state requires local ownership of a resource justifying this update/access. The OpenV and OpenA rules allow us to open a region, temporarily turning the region's shared resources into local resources, for the duration of a view-shift and an atomic statement, respectively. This allows us to use standard separation logic rules to reason about updates to these temporarily local resources.

To reason about an update to a shared region, both rules require a proof that the update is possible (the last hypothesis of each rule) and a proof that the update is permitted by the protocol of the given region (the second to last hypothesis of each rule). The following subsection contains rules for proving that the update of the region is permitted by the protocol. Both rules further require explicit proofs of the absence of self-referential region and protocol assertions, using the indep assumptions.

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{R} : \mathsf{Region} \qquad \Gamma \vdash \mathsf{T}_1, \mathsf{T}_2 : \mathsf{RType} \qquad \Gamma \vdash \mathsf{A} : \mathsf{Val} \\ \Gamma \vdash \mathsf{P}_1, \mathsf{P}_2, \mathsf{Q}_1, \mathsf{Q}_2 : \tau \to \mathsf{Prop} \qquad \Gamma \mid \Phi \vdash \mathsf{T}_2 \not\leq \mathsf{T}_1 \\ \Gamma \mid \Phi \vdash \mathsf{indep}_{\mathsf{T}_1 \sqcap \mathsf{T}_2}(\mathsf{P}_1) \qquad \Gamma \mid \Phi \vdash \mathsf{indep}_{\mathsf{T}_1 \sqcap \mathsf{T}_2}(\mathsf{P}_2) \\ \Gamma \mid \Phi \vdash \mathsf{indep}_{\mathsf{T}_1 \sqcap \mathsf{T}_2}(\mathsf{Q}_1) \qquad \Gamma \mid \Phi \vdash \mathsf{indep}_{\mathsf{T}_1 \sqcap \mathsf{T}_2}(\mathsf{Q}_2) \\ \Gamma \mid \Phi \vdash \exists \mathsf{x} : \tau.\ \boxed{\mathsf{P}_1(\mathsf{x})}^{\mathsf{R},\mathsf{T}_1,\mathsf{A}} * \mathsf{P}_2(\mathsf{x}) \rightsquigarrow^{\mathsf{R},\mathsf{T}_2} \exists \mathsf{x} : \tau.\ \boxed{\mathsf{Q}_1(\mathsf{x})}^{\mathsf{R},\mathsf{T}_1,\mathsf{A}} * \mathsf{Q}_2(\mathsf{x}) \\ \Gamma; - \mid \Phi \mid \exists \mathsf{x} : \tau.\ \mathsf{P}_1(\mathsf{x}) * \mathsf{P}_2(\mathsf{x}) \vdash \exists \mathsf{x} : \tau.\ \mathsf{Q}_1(\mathsf{x}) * \mathsf{Q}_2(\mathsf{x})\end{array}}{\Gamma \mid \Phi \vdash \exists \mathsf{x} : \tau.\ \boxed{\mathsf{P}_1(\mathsf{x})}^{\mathsf{R},\mathsf{T}_1,\mathsf{A}} * \mathsf{P}_2(\mathsf{x}) \sqsubseteq^{\mathsf{T}_2} \exists \mathsf{x} : \tau.\ \boxed{\mathsf{Q}_1(\mathsf{x})}^{\mathsf{R},\mathsf{T}_1,\mathsf{A}} * \mathsf{Q}_2(\mathsf{x})}\ \textsc{OpenV}$$

To support modular stability proofs, action permissions owned by shared regions cannot be used to justify updates to shared regions. The following rule, which "opens" the shared region R thus explicitly requires that P_1 does not assert local ownership of any permissions. This ensures that no actions from P_1 are used to justify updates in the nested atomic update.

$$\frac{\begin{array}{c}\Gamma, \Delta \vdash \mathsf{R} : \mathsf{Region} \qquad \Gamma, \Delta \vdash \mathsf{T}_1, \mathsf{T}_2 : \mathsf{RType} \qquad \Gamma, \Delta \vdash \mathsf{A} : \mathsf{Val} \\ \Gamma, \Delta \vdash \mathsf{P}_1, \mathsf{P}_2, \mathsf{Q}_1, \mathsf{Q}_2 : \tau \to \mathsf{Prop} \\ \Gamma, \Delta \mid \Phi \vdash \mathsf{T}_2 \not\leq \mathsf{T}_1 \qquad \Gamma, \Delta \mid \Phi \vdash \mathsf{pure}_{\mathsf{perm}}(\mathsf{P}_1) \\ \Gamma, \Delta \mid \Phi \vdash \mathsf{indep}_{\mathsf{T}_1 \sqcap \mathsf{T}_2}(\mathsf{P}_1) \qquad \Gamma, \Delta \mid \Phi \vdash \mathsf{indep}_{\mathsf{T}_1 \sqcap \mathsf{T}_2}(\mathsf{P}_2) \\ \Gamma, \Delta \mid \Phi \vdash \mathsf{indep}_{\mathsf{T}_1 \sqcap \mathsf{T}_2}(\mathsf{Q}_1) \qquad \Gamma, \Delta \mid \Phi \vdash \mathsf{indep}_{\mathsf{T}_1 \sqcap \mathsf{T}_2}(\mathsf{Q}_2) \\ \Gamma \mid \Phi \vdash \exists \mathsf{x} : \tau.\ \boxed{\mathsf{P}_1(\mathsf{x})}^{\mathsf{R},\mathsf{T}_1,\mathsf{A}} * \mathsf{P}_2(\mathsf{x}) \rightsquigarrow^{\mathsf{R},\mathsf{T}_2}_{(\Delta).\langle s \rangle} \exists \mathsf{x} : \tau.\ \boxed{\mathsf{Q}_1(\mathsf{x})}^{\mathsf{R},\mathsf{T}_1,\mathsf{A}} * \mathsf{Q}_2(\mathsf{x}) \\ \Gamma \mid \Phi \vdash (\Delta).\{\exists \mathsf{x} : \tau.\ \mathsf{P}_1(\mathsf{x}) * \mathsf{P}_2(\mathsf{x})\}\langle s \rangle^{\mathsf{T}_1 \sqcap \mathsf{T}_2}\{\exists \mathsf{x} : \tau.\ \mathsf{Q}_1(\mathsf{x}) * \mathsf{Q}_2(\mathsf{x})\}\end{array}}{\Gamma \mid \Phi \vdash (\Delta).\{\exists \mathsf{x} : \tau.\ \boxed{\mathsf{P}_1(\mathsf{x})}^{\mathsf{R},\mathsf{T}_1,\mathsf{A}} * \mathsf{P}_2(\mathsf{x})\}\langle s \rangle^{\mathsf{T}_2}\{\exists \mathsf{x} : \tau.\ \boxed{\mathsf{Q}_1(\mathsf{x})}^{\mathsf{R},\mathsf{T}_1,\mathsf{A}} * \mathsf{Q}_2(\mathsf{x})\}}\ \textsc{OpenA}$$

$$\frac{\begin{array}{c}\Gamma, \Delta \vdash \mathsf{T}_1, \mathsf{T}_2, \mathsf{T}_3 : \mathsf{RType} \\ \Gamma, \Delta \mid \Phi \vdash (\mathsf{T}_1 = \mathsf{T}_2 = \bot) \vee (\mathsf{T}_1 = \mathsf{T}_3 = \bot) \vee (\mathsf{T}_2 = \mathsf{T}_3 = \bot) \\ \Gamma, \Delta \mid \Phi \vdash \mathsf{P} \sqsubseteq^{\mathsf{T}_1} \mathsf{P}' \qquad \Gamma \mid \Phi \vdash (\Delta).\{\mathsf{P}'\}\langle s \rangle^{\mathsf{T}_2}\{\mathsf{Q}'\} \qquad \Gamma, \Delta \mid \Phi \vdash \mathsf{Q}' \sqsubseteq^{\mathsf{T}_3} \mathsf{Q}\end{array}}{\Gamma \mid \Phi \vdash (\Delta).\{\mathsf{P}\}\langle s \rangle^{\mathsf{T}_1 \sqcap \mathsf{T}_2 \sqcap \mathsf{T}_3}\{\mathsf{Q}\}}\ \textsc{ATrans}$$

$$\frac{\begin{array}{c}\Gamma, \Delta \mid \Phi \vdash \mathsf{T}_1 \leq \mathsf{T}_2 \\ \Gamma \mid \Phi \vdash (\Delta).\{\mathsf{P}\}\langle s \rangle^{\mathsf{T}_1}\{\mathsf{Q}\}\end{array}}{\Gamma \mid \Phi \vdash (\Delta).\{\mathsf{P}\}\langle s \rangle^{\mathsf{T}_2}\{\mathsf{Q}\}} \qquad \frac{\begin{array}{c}\Gamma \mid \Phi \vdash \mathsf{T}_1 \leq \mathsf{T}_2 \\ \Gamma \mid \Phi \vdash \mathsf{P} \sqsubseteq^{\mathsf{T}_1} \mathsf{Q}\end{array}}{\Gamma \mid \Phi \vdash \mathsf{P} \sqsubseteq^{\mathsf{T}_2} \mathsf{Q}}$$

The following two rules are used for creating a new region and allocating a new region type, respectively. To allocate a new region type, we pick a region type T_1 and a protocol, and get back a new region type T_2 greater than or equal to T_2 with the given protocol. To allocate a new region, we pick a region type and get back a region identifier and the full action permission for any finite number of action identifiers. The region allocation rule further allows us to transfer some initial resources, P, to the shared region.

$$\frac{\Gamma \vdash \mathsf{P} : \mathsf{Prop} \qquad \Gamma \vdash \alpha_1, ..., \alpha_n : \mathsf{Action} \qquad \Gamma \vdash \mathsf{A} : \mathsf{Val} \qquad \Gamma \vdash \mathsf{T} : \mathsf{RType}}{\Gamma \mid \Phi \vdash \mathsf{P} \sqsubseteq^{\bot} \exists \mathsf{R} : \mathsf{Region}.\ \boxed{\mathsf{P}}^{\mathsf{R},\mathsf{T},\mathsf{A}} * [\alpha_1]^{\mathsf{R}}_1 * \cdots * [\alpha_n]^{\mathsf{R}}_1}$$

$$\frac{\Gamma \vdash \mathsf{T}_1 : \mathsf{RType} \qquad \Gamma \vdash \mathsf{I}_p, \mathsf{I}_q : \mathsf{Val} \times \mathsf{Action} \times \mathsf{Val} \to \mathsf{Prop}}{\Gamma \mid \Phi \vdash \mathsf{emp} \sqsubseteq^{\bot} \exists \mathsf{T}_2 : \mathsf{RType}.\ \mathsf{T}_1 \leq \mathsf{T}_2 * \mathsf{protocol}(\mathsf{T}_2, \mathsf{I}_p, \mathsf{I}_q)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{T} : \mathsf{RType} \qquad \Gamma \vdash \mathsf{P}_1, \mathsf{P}_2, \mathsf{Q} : \mathsf{Prop} \\ \Gamma \mid \Phi \vdash \mathsf{P}_1 \sqsubseteq^{\mathsf{T}} \mathsf{Q} \qquad \Gamma \mid \Phi \vdash \mathsf{P}_2 \sqsubseteq^{\mathsf{T}} \mathsf{Q}\end{array}}{\Gamma \mid \Phi \vdash \mathsf{P}_1 \vee \mathsf{P}_2 \sqsubseteq^{\mathsf{T}} \mathsf{Q}}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{P}, \mathsf{Q}, \mathsf{R} : \mathsf{Prop} \\ \Gamma \mid \Phi \vdash \mathsf{P} \sqsubseteq \mathsf{Q} \qquad \Gamma \mid \Phi \vdash \mathsf{stable}(\mathsf{R})\end{array}}{\Gamma \mid \Phi \vdash \mathsf{P} * \mathsf{R} \sqsubseteq \mathsf{Q} * \mathsf{R}} \qquad \frac{\begin{array}{c}\Gamma \vdash \mathsf{T} : \mathsf{RType} \qquad \Gamma \vdash \mathsf{P}, \mathsf{Q}, \mathsf{R} : \mathsf{Prop} \\ \Gamma \mid \Phi \vdash \mathsf{P} \sqsubseteq^{\mathsf{T}} \mathsf{Q} \qquad \Gamma \mid \Phi \vdash \mathsf{stable}(\mathsf{R})\end{array}}{\Gamma \mid \Phi \vdash \mathsf{P} * \mathsf{R} \sqsubseteq^{\mathsf{T}} \mathsf{Q} * \mathsf{R}}$$

$$\frac{\Gamma; - \mid \Phi \mid \mathsf{P} \vdash \mathsf{Q}}{\Gamma \mid \Phi \vdash \mathsf{P} \sqsubseteq \mathsf{Q}}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{T} : \mathsf{RType} \qquad \Gamma, \Delta \vdash \mathsf{P}, \mathsf{Q} : \mathsf{Prop} \\ \Gamma \mid \Phi \vdash (\Delta).\{\mathsf{P}\}\langle \mathsf{s} \rangle^{\mathsf{T}}\{\mathsf{Q}\} \qquad \mathsf{atomic}(\mathsf{s})\end{array}}{\Gamma \mid \Phi \vdash (\Delta).\{\mathsf{P}\}\mathsf{s}\{\mathsf{Q}\}} \qquad \frac{\begin{array}{c}\Gamma \vdash \mathsf{T} : \mathsf{RType} \\ \Gamma \vdash \mathsf{P}, \mathsf{Q} : \mathsf{Prop} \\ \Gamma \mid \Phi \vdash \mathsf{P} \sqsubseteq^{\mathsf{T}} \mathsf{Q}\end{array}}{\Gamma \mid \Phi \vdash \mathsf{P} \sqsubseteq \mathsf{Q}}$$

$$\frac{\Gamma \vdash \mathsf{T} : \mathsf{RType}}{\begin{array}{l}\Gamma \mid \Phi \vdash (\Delta).\{\mathsf{y.f} \mapsto \mathsf{M}\} \\ \qquad \langle \mathsf{x} = \mathtt{CAS}(\mathsf{y.f}, \mathsf{o}, \mathsf{n}) \rangle^{\mathsf{T}} \\ \qquad \{(\mathsf{x} = \mathsf{y} * \mathsf{o} = \mathsf{M} * \mathsf{y.f} \mapsto \mathsf{n}) \vee (\mathsf{x} = \mathsf{null} * \mathsf{o} \neq \mathsf{M} * \mathsf{y.f} \mapsto \mathsf{M})\}\end{array}} \text{ACAS}$$

Atomic statements

$$\frac{}{\mathsf{atomic}(\mathsf{x} = \mathtt{CAS}(\mathsf{y.f}, \mathsf{o}, \mathsf{n}))} \qquad \frac{}{\mathsf{atomic}(\mathsf{x} = \mathsf{y})} \qquad \frac{}{\mathsf{atomic}(\mathsf{x.f} = \mathsf{y})} \qquad \frac{}{\mathsf{atomic}(\mathsf{x} = \mathsf{y.f})}$$

Atomic update & view-shift allowed

The proof system includes two specifications assertions,

$$\mathsf{P} \rightsquigarrow^{\mathsf{R},\mathsf{T}} \mathsf{Q} \qquad \text{and} \qquad \mathsf{P} \rightsquigarrow^{\mathsf{R},\mathsf{T}}_{(\Delta).\langle \mathsf{s} \rangle} \mathsf{Q},$$

for asserting that a given view-shift/atomic update is permitted according to the protocol on region R. More formally, $\mathsf{P} \rightsquigarrow^{\mathsf{R},\mathsf{T}} \mathsf{Q}$ asserts that for any step at the instrumented level from P to Q that corresponds to a no-op at the concrete level, the update to region R is justified by the protocol on region R using an action owned by P. Furthermore, the action used to justify the update only depends on regions with region types greater than or equal to T. The meaning of $\mathsf{P} \rightsquigarrow^{\mathsf{R},\mathsf{T}}_{(\Delta).\langle \mathsf{s} \rangle} \mathsf{Q}$ is almost the same, as it asserts that any step at the instrumented level from P to Q that corresponds to the atomic statement s at the concrete level is permitted by region R. See the definition of $p \rightsquigarrow^{r,A}_{a} q$ and $p \rightsquigarrow^{r,A} q$ on page 187 for the formal semantics of $\mathsf{P} \rightsquigarrow^{\mathsf{R},\mathsf{T}}_{(\Delta).\langle \mathsf{s} \rangle} \mathsf{Q}$ and $\mathsf{P} \rightsquigarrow^{\mathsf{R},\mathsf{T}} \mathsf{Q}$.

There are several things worth noting about these assertions; first, they explicitly do *not* enforce stability of the pre- and post-condition. In fact, they allow case analysis on disjunctions and existentials in both P and Q – even inside region assertions. These predicates also satisfy a non-standard rule of consequence that allows weakening of *both* the pre- and post-condition. From this non-standard rule of consequence, we can further derive an asymmetric frame rule that allows arbitrary changes to the context.

Since most of the proof rules for these two predicates are the same, we write $\mathsf{P} \rightsquigarrow^{\mathsf{R},\mathsf{T}}_{\mathcal{J}} \mathsf{Q}$ where $\mathcal{J}$ is defined as follows,

$$\mathcal{J} ::= (\Delta).\langle \mathsf{s} \rangle \mid$$

to refer to both variants at the same time.

$$\frac{\begin{array}{cc}\Gamma \vdash \mathsf{R} : \mathsf{Region} & \Gamma \vdash \mathsf{T}_1, \mathsf{T}_2 : \mathsf{RType} \\ \Gamma \mid \Phi \vdash \mathsf{P} \rightsquigarrow^{\mathsf{R},\mathsf{T}_1}_{\mathcal{J}} \mathsf{Q} & \Gamma \mid \Phi \vdash \mathsf{T}_1 \leq \mathsf{T}_2\end{array}}{\Gamma \mid \Phi \vdash \mathsf{P} \rightsquigarrow^{\mathsf{R},\mathsf{T}_2}_{\mathcal{J}} \mathsf{Q}}$$

$$\frac{}{\Gamma \mid \Phi \vdash \bot \rightsquigarrow^{\mathsf{R},\mathsf{T}}_{\mathcal{J}} \mathsf{Q}} \qquad \frac{}{\Gamma \mid \Phi \vdash \mathsf{P} \rightsquigarrow^{\mathsf{R},\mathsf{T}}_{\mathcal{J}} \bot}$$

$$\frac{\Gamma \mid \Phi \vdash \mathsf{P}_1 \rightsquigarrow^{\mathsf{R},\mathsf{T}}_{\mathcal{J}} \mathsf{Q} \quad \Gamma \mid \Phi \vdash \mathsf{P}_2 \rightsquigarrow^{\mathsf{R},\mathsf{T}}_{\mathcal{J}} \mathsf{Q}}{\Gamma \mid \Phi \vdash \mathsf{P}_1 \vee \mathsf{P}_2 \rightsquigarrow^{\mathsf{R},\mathsf{T}}_{\mathcal{J}} \mathsf{Q}} \qquad \frac{\Gamma \mid \Phi \vdash \mathsf{P} \rightsquigarrow^{\mathsf{R},\mathsf{T}}_{\mathcal{J}} \mathsf{Q}_1 \quad \Gamma \mid \Phi \vdash \mathsf{P} \rightsquigarrow^{\mathsf{R},\mathsf{T}}_{\mathcal{J}} \mathsf{Q}_2}{\Gamma \mid \Phi \vdash \mathsf{P} \rightsquigarrow^{\mathsf{R},\mathsf{T}}_{\mathcal{J}} \mathsf{Q}_1 \vee \mathsf{Q}_2}$$

$$\frac{\Gamma \mid \Phi \vdash \boxed{\mathsf{P}_1}^{\mathsf{R}_2,\mathsf{T}_2,\mathsf{A}} * \mathsf{R} \rightsquigarrow^{\mathsf{R}_1,\mathsf{T}_1}_{\mathcal{J}} \mathsf{Q} \quad \Gamma \mid \Phi \vdash \boxed{\mathsf{P}_2}^{\mathsf{R}_2,\mathsf{T}_2,\mathsf{A}} * \mathsf{R} \rightsquigarrow^{\mathsf{R}_1,\mathsf{T}_1}_{\mathcal{J}} \mathsf{Q}}{\Gamma \mid \Phi \vdash \boxed{\mathsf{P}_1 \vee \mathsf{P}_2}^{\mathsf{R}_2,\mathsf{T}_2,\mathsf{A}} * \mathsf{R} \rightsquigarrow^{\mathsf{R}_1,\mathsf{T}_1}_{\mathcal{J}} \mathsf{Q}} \qquad \frac{\Gamma \mid \Phi \vdash \mathsf{P} \rightsquigarrow^{\mathsf{R}_2,\mathsf{A}}_{\mathcal{J}} \boxed{\mathsf{Q}_1}^{\mathsf{R}_1,\mathsf{T}_1} * \mathsf{R} \quad \Gamma \mid \Phi \vdash \mathsf{P} \rightsquigarrow^{\mathsf{R}_2,\mathsf{A}}_{\mathcal{J}} \boxed{\mathsf{Q}_2}^{\mathsf{R}_1,\mathsf{T}_1} * \mathsf{R}}{\Gamma \mid \Phi \vdash \mathsf{P} \rightsquigarrow^{\mathsf{R}_2,\mathsf{A}}_{\mathcal{J}} \boxed{\mathsf{Q}_1 \vee \mathsf{Q}_2}^{\mathsf{R}_1,\mathsf{T}_1} * \mathsf{R}}$$

$$\frac{\Gamma \vdash \mathsf{P}, \mathsf{P}', \mathsf{Q}, \mathsf{Q}' : \mathsf{Prop} \qquad \Gamma \vdash \mathsf{R} : \mathsf{Region} \qquad \Gamma \vdash \mathsf{T} : \mathsf{RType} \quad \Gamma; - \mid \Phi \mid \mathsf{P}' \vdash \mathsf{P} \qquad \Gamma \mid \Phi \vdash \mathsf{P} \rightsquigarrow^{\mathsf{R},\mathsf{T}} \mathsf{Q} \qquad \Gamma; - \mid \Phi \mid \mathsf{Q}' \vdash \mathsf{Q}}{\Gamma \mid \Phi \vdash \mathsf{P}' \rightsquigarrow^{\mathsf{R},\mathsf{T}} \mathsf{Q}'}$$

$$\frac{\Gamma, \Delta \vdash \mathsf{P}, \mathsf{P}', \mathsf{Q}, \mathsf{Q}' : \mathsf{Prop} \qquad \Gamma, \Delta \vdash \mathsf{R} : \mathsf{Region} \qquad \Gamma, \Delta \vdash \mathsf{T} : \mathsf{RType} \quad \Gamma, \Delta \mid \Phi \mid \mathsf{P}' \vdash \mathsf{P} \qquad \Gamma \mid \Phi \vdash \mathsf{P} \rightsquigarrow^{\mathsf{R},\mathsf{T}}_{(\Delta).\langle s \rangle} \mathsf{Q} \qquad \Gamma, \Delta \mid \Phi \mid \mathsf{Q}' \vdash \mathsf{Q}}{\Gamma \mid \Phi \vdash \mathsf{P}' \rightsquigarrow^{\mathsf{R},\mathsf{T}}_{(\Delta).\langle s \rangle} \mathsf{Q}'}$$

Any view-shift and atomic update on a region R corresponding to an instance of an action of the protocol on R is permitted, given partial ownership of said action.

$$\frac{\begin{array}{c}\Gamma, \Delta \vdash \mathsf{I}_p, \mathsf{I}_q : \mathsf{Val} \times \mathsf{Action} \times \mathsf{Val} \to \mathsf{Prop} \qquad \Gamma, \Delta \vdash \alpha : \mathsf{Action} \\ \Gamma, \Delta \vdash \mathsf{A}, \mathsf{M} : \mathsf{Val} \\ \Gamma, \Delta \vdash \mathsf{P} : \mathsf{Perm} \qquad \Gamma, \Delta \vdash \mathsf{R} : \mathsf{Region} \qquad \Gamma, \Delta \vdash \mathsf{T}_1, \mathsf{T}_2 : \mathsf{RType} \\ \Gamma, \Delta \mid \Phi \vdash \mathsf{indep}_{\mathsf{T}_2}(\mathsf{I}_p(\mathsf{A}, \alpha, \mathsf{M}), \mathsf{I}_q(\mathsf{A}, \alpha, \mathsf{M})) \qquad \Gamma, \Delta \mid \Phi \vdash \mathsf{T}_2 \not\leq \mathsf{T}_1\end{array}}{\Gamma \mid \Phi \vdash \boxed{\mathsf{I}_p(\mathsf{A}, \alpha, \mathsf{M})}^{\mathsf{R},\mathsf{T}_1,\mathsf{A}}_{\mathsf{I}_p,\mathsf{I}_q} * [\alpha]^{\mathsf{R}}_{\mathsf{P}} \rightsquigarrow^{\mathsf{R},\mathsf{T}_2}_{(\Delta).\langle s \rangle} \boxed{\mathsf{I}_q(\mathsf{A}, \alpha, \mathsf{M})}^{\mathsf{R},\mathsf{T}_1,\mathsf{A}}_{\mathsf{I}_p,\mathsf{I}_q} * [\alpha]^{\mathsf{R}}_{\mathsf{P}}} \text{AUAct}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{I}_p, \mathsf{I}_q : \mathsf{Val} \times \mathsf{Action} \times \mathsf{Val} \to \mathsf{Prop} \qquad \Gamma \vdash \alpha : \mathsf{Action} \\ \Gamma \vdash \mathsf{A}, \mathsf{M} : \mathsf{Val} \qquad \Gamma \vdash \mathsf{P} : \mathsf{Perm} \qquad \Gamma \vdash \mathsf{R} : \mathsf{Region} \qquad \Gamma \vdash \mathsf{T}_1, \mathsf{T}_2 : \mathsf{RType} \\ \Gamma \mid \Phi \vdash \mathsf{indep}_{\mathsf{T}_2}(\mathsf{I}_p(\mathsf{A}, \alpha, \mathsf{M}), \mathsf{I}_q(\mathsf{A}, \alpha, \mathsf{M})) \qquad \Gamma \mid \Phi \vdash \mathsf{T}_2 \not\leq \mathsf{T}_1\end{array}}{\Gamma \mid \Phi \vdash \boxed{\mathsf{I}_p(\mathsf{A}, \alpha, \mathsf{M})}^{\mathsf{R},\mathsf{T}_1,\mathsf{A}}_{\mathsf{I}_p,\mathsf{I}_q} * [\alpha]^{\mathsf{R}}_{\mathsf{P}} \rightsquigarrow^{\mathsf{R},\mathsf{T}_2} \boxed{\mathsf{I}_q(\mathsf{A}, \alpha, \mathsf{M})}^{\mathsf{R},\mathsf{T}_1,\mathsf{A}}_{\mathsf{I}_p,\mathsf{I}_q} * [\alpha]^{\mathsf{R}}_{\mathsf{P}}} \text{VSAct}$$

Since $\mathsf{P} \rightsquigarrow^{\mathsf{R},\mathsf{T}}_{(\Delta).\langle \mathsf{s} \rangle} \mathsf{Q}$ assert that any step at the instrumented level from P to Q that corresponds to the atomic statement s at the concrete level is permitted by region R, we

can also prove $\mathsf{P} \rightsquigarrow^{\mathsf{R},\mathsf{T}}_{(\Delta).\langle \mathsf{s} \rangle} \mathsf{Q}$ by proving that there exists no such step at the instrumented level. This is useful after performing case analysis to exclude impossible cases, as illustrated by the spin-lock example in Section 3.2. The following rule provides one way of proving that no such step exists at the instrumented level, by providing a concrete field ($\mathsf{x.f}$) whose actual value after the update (M_2) is distinct from the assumed value (M_1).

$$\frac{\begin{array}{cc}\Gamma \mid \Phi \vdash (\Delta).\{\mathsf{P}_1 * \mathsf{P}_2\}\langle s \rangle\{\mathsf{Q}_3\} & \Gamma, \Delta \mid \Phi \vdash \mathsf{M}_1 \neq \mathsf{M}_2 \\ \Gamma, \Delta \mid \Phi \mid \mathsf{Q}_1 * \mathsf{Q}_2 \vdash \mathsf{x.f} \mapsto \mathsf{M}_1 & \Gamma, \Delta \mid \Phi \mid \mathsf{Q}_3 \vdash \mathsf{x.f} \mapsto \mathsf{M}_2\end{array}}{\Gamma \mid \Phi \vdash \boxed{\mathsf{P}_1}^{\mathsf{R},\mathsf{T}_1} * \mathsf{P}_2 \rightsquigarrow^{\mathsf{S},\mathsf{T}_2}_{(\Delta).\langle \mathsf{s} \rangle} \boxed{\mathsf{Q}_1}^{\mathsf{R},\mathsf{T}_1} * \mathsf{Q}_2} \text{ AUFALSE1}$$

Dependence

The $\mathsf{dep}_{\mathsf{T}}$ assertion internalizes a notion of region support, allowing explicit reasoning about the absence of self-referential region and protocol assertions. In particular, $\mathsf{dep}_{\mathsf{T}}(\mathsf{P})$ asserts that P is invariant under arbitrary changes to shared regions and protocols of regions with region types not greater than or equal to T. Conversely, $\mathsf{indep}_{\mathsf{T}}(\mathsf{P})$ asserts that P is invariant under arbitrary changes to shared regions and protocols of regions with region types greater than or equal to T.

$$\frac{\Gamma \vdash \mathsf{T} : \mathsf{RType}}{\Gamma \mid \Phi \vdash \mathsf{dep}_{\mathsf{T}}(\bot)} \qquad \frac{\Gamma \vdash \mathsf{T} : \mathsf{RType}}{\Gamma \mid \Phi \vdash \mathsf{dep}_{\mathsf{T}}(\top)} \qquad \frac{\begin{array}{c}\Gamma \vdash \mathsf{S} : \mathsf{Spec} \\ \Gamma \vdash \mathsf{T} : \mathsf{RType}\end{array}}{\Gamma \mid \Phi \vdash \mathsf{dep}_{\mathsf{T}}(\mathsf{asn}(\mathsf{S}))}$$

$$\frac{\Gamma \mid \Phi \vdash \mathsf{dep}_{\mathsf{T}}(\mathsf{P}) \qquad \Gamma \mid \Phi \vdash \mathsf{dep}_{\mathsf{T}}(\mathsf{Q}) \qquad op \in \{\wedge, \vee, *, \Rightarrow\}}{\Gamma \mid \Phi \vdash \mathsf{dep}_{\mathsf{T}}(\mathsf{P}\ op\ \mathsf{Q})}$$

$$\frac{\Gamma \mid \Phi \vdash \forall x : \tau.\ \mathsf{dep}_{\mathsf{T}}(\mathsf{P}) \qquad Q \in \{\forall, \exists\}}{\Gamma \mid \Phi \vdash \mathsf{dep}_{\mathsf{T}}(Qx : \tau.\ \mathsf{P})} \qquad \frac{\begin{array}{c}\Gamma \mid \Phi \vdash \mathsf{T}_1 \leq \mathsf{T}_2 \\ \Gamma \mid \Phi \vdash \mathsf{dep}_{\mathsf{T}_2}(\mathsf{P})\end{array}}{\Gamma \mid \Phi \vdash \mathsf{dep}_{\mathsf{T}_1}(\mathsf{P})}$$

$$\frac{\Gamma \mid \Phi \vdash \mathsf{dep}_{\mathsf{T}_1}(\mathsf{P})}{\Gamma \mid \Phi \vdash \mathsf{dep}_{\mathsf{T}_1 \sqcap \mathsf{T}_2}(\boxed{\mathsf{P}}^{\mathsf{R},\mathsf{T}_2,\mathsf{A}})} \qquad \frac{\begin{array}{c}\Gamma \mid \Phi \vdash \forall x : \mathsf{Val} \times \mathsf{Action} \times \mathsf{Val}.\ \mathsf{dep}_{\mathsf{T}_1}(\mathsf{I}_p(x)) \\ \Gamma \mid \Phi \vdash \forall x : \mathsf{Val} \times \mathsf{Action} \times \mathsf{Val}.\ \mathsf{dep}_{\mathsf{T}_1}(\mathsf{I}_q(x))\end{array}}{\Gamma \mid \Phi \vdash \mathsf{dep}_{\mathsf{T}_1 \sqcap \mathsf{T}_2}(\mathsf{protocol}(\mathsf{T}_2, \mathsf{I}_p, \mathsf{I}_q))}$$

$$\frac{\Gamma \mid \Phi \vdash \mathsf{dep}_{\mathsf{T}_1}(\mathsf{P}) \qquad \Gamma \mid \Phi \vdash \mathsf{T}_1 \not\leq \mathsf{T}_2 \qquad \Gamma \mid \Phi \vdash \mathsf{T}_2 \not\leq \mathsf{T}_1}{\Gamma \mid \Phi \vdash \mathsf{indep}_{\mathsf{T}_2}(\mathsf{P})}$$

$$\frac{\Gamma \mid \Phi \vdash \mathsf{T}_1 \not\leq \mathsf{T}_2 \qquad \Gamma \mid \Phi \vdash \mathsf{indep}_{\mathsf{T}_1}(\mathsf{P})}{\Gamma \mid \Phi \vdash \mathsf{indep}_{\mathsf{T}_1}(\boxed{\mathsf{P}}^{\mathsf{R},\mathsf{T}_2,\mathsf{A}})}$$

Protocol purity

The $\mathsf{pure}_{\mathsf{protocol}}$ predicate expresses that a given assertion is invariant under arbitrary changes to currently allocated protocols. The $\mathsf{pure}_{\mathsf{protocol}}$ predicate is closed under all the usual connectives.

$$\frac{}{\Gamma \mid \Phi \vdash \mathsf{pure}_{\mathsf{protocol}}(\bot)} \qquad \frac{}{\Gamma \mid \Phi \vdash \mathsf{pure}_{\mathsf{protocol}}(\top)}$$

$$\frac{\Gamma \mid \Phi \vdash \mathsf{pure}_{\mathsf{protocol}}(\mathsf{P}) \qquad \Gamma \mid \Phi \vdash \mathsf{pure}_{\mathsf{protocol}}(\mathsf{Q}) \qquad op \in \{\wedge, \vee, *, \Rightarrow\}}{\Gamma \mid \Phi \vdash \mathsf{pure}_{\mathsf{protocol}}(\mathsf{P}\ op\ \mathsf{Q})}$$

$$\frac{\Gamma \mid \Phi \vdash \forall \mathsf{x} : \tau.\ \mathsf{pure}_{\mathsf{protocol}}(\mathsf{P}) \qquad Q \in \{\exists, \forall\}}{\Gamma \mid \Phi \vdash \mathsf{pure}_{\mathsf{protocol}}(Q\mathsf{x} : \tau.\ \mathsf{P})}$$

The $\mathsf{pure}_{\mathsf{protocol}}$ predicate is also closed under the mini $\mathrm{C}^\sharp$ specific state assertions.

$$\frac{\Gamma \vdash \mathsf{M} : \mathsf{Val} \qquad \dfrac{\Gamma \vdash \mathsf{F} : \mathsf{Field} \qquad \Gamma \vdash \mathsf{N} : \mathsf{Val}}{}}{\Gamma \mid \Phi \vdash \mathsf{pure}_{\mathsf{protocol}}(\mathsf{M}.\mathsf{F} \mapsto \mathsf{N})} \qquad \frac{\Gamma \vdash \mathsf{M} : \mathsf{Val} \qquad \dfrac{\Gamma \vdash \mathsf{F} : \mathsf{Field} \qquad \Gamma \vdash \mathsf{N} : \mathsf{Val}}{}}{\Gamma \mid \Phi \vdash \mathsf{pure}_{\mathsf{protocol}}(\mathsf{M}_{\mathsf{F}} \mapsto \mathsf{N})}$$

$$\frac{\Gamma \vdash \mathsf{N}_1, \mathsf{N}_2 : \mathsf{Val} \qquad \Gamma \vdash \mathsf{M} : \mathsf{Method}}{\Gamma \mid \Phi \vdash \mathsf{pure}_{\mathsf{protocol}}(\mathsf{N}_1 \mapsto \mathsf{N}_2.\mathsf{M})} \qquad \frac{\Gamma \vdash \mathsf{M} : \mathsf{Val} \qquad \Gamma \vdash \mathsf{C} : \mathsf{Class}}{\Gamma \mid \Phi \vdash \mathsf{pure}_{\mathsf{protocol}}(\mathsf{M} : \mathsf{C})}$$

It is also closed under region and action assertions, but not protocol assertions.

$$\frac{\Gamma \vdash \mathsf{P} : \mathsf{Prop} \qquad \Gamma \vdash \mathsf{R} : \mathsf{Region} \qquad \Gamma \vdash \mathsf{T} : \mathsf{RType} \qquad \Gamma \vdash \mathsf{A} : \mathsf{Val} \qquad \Gamma \mid \Phi \vdash \mathsf{pure}_{\mathsf{protocol}}(\mathsf{P})}{\Gamma \mid \Phi \vdash \mathsf{pure}_{\mathsf{protocol}}\left(\boxed{\mathsf{P}}^{\mathsf{R},\mathsf{T},\mathsf{A}}\right)}$$

$$\frac{\Gamma \vdash \mathsf{A} : \mathsf{Action} \qquad \Gamma \vdash \mathsf{R} : \mathsf{Region} \qquad \Gamma \vdash \mathsf{P} : \mathsf{Perm}}{\Gamma \mid \Phi \vdash \mathsf{pure}_{\mathsf{protocol}}([\mathsf{A}]^{\mathsf{R}}_{\mathsf{P}})}$$

Lastly, $\mathsf{pure}_{\mathsf{protocol}}$ is closed under arbitrary embedded specifications.

$$\frac{\Gamma \vdash \mathsf{S} : \mathsf{Spec}}{\Gamma \mid \Phi \vdash \mathsf{pure}_{\mathsf{protocol}}(\mathsf{asn}(\mathsf{S}))}$$

State purity

The $\mathsf{pure}_{\mathsf{state}}$ predicate expresses that a given assertion is invariant under arbitrary changes to the current local and shared state. The $\mathsf{pure}_{\mathsf{state}}$ predicate is closed under all the usual connectives.

$$\frac{}{\Gamma \mid \Phi \vdash \mathsf{pure}_{\mathsf{state}}(\bot)} \qquad \frac{}{\Gamma \mid \Phi \vdash \mathsf{pure}_{\mathsf{state}}(\top)}$$

$$\frac{\Gamma \mid \Phi \vdash \mathsf{pure}_{\mathsf{state}}(\mathsf{P}) \qquad \Gamma \mid \Phi \vdash \mathsf{pure}_{\mathsf{state}}(\mathsf{Q}) \qquad op \in \{\wedge, \vee, *, \Rightarrow\}}{\Gamma \mid \Phi \vdash \mathsf{pure}_{\mathsf{state}}(\mathsf{P}\ op\ \mathsf{Q})}$$

$$\frac{\Gamma \mid \Phi \vdash \forall \mathsf{x} : \tau.\ \mathsf{pure}_{\mathsf{state}}(\mathsf{P}) \qquad Q \in \{\exists, \forall\}}{\Gamma \mid \Phi \vdash \mathsf{pure}_{\mathsf{state}}(Q\mathsf{x} : \tau.\ \mathsf{P})}$$

It is also closed under protocol assertions, but not region or action assertions.

$$\frac{\Gamma \vdash \mathsf{T} : \mathsf{RType} \qquad \Gamma \vdash \mathsf{I}_p, \mathsf{I}_q : \mathsf{Val} \times \mathsf{Action} \times \mathsf{Val} \to \mathsf{Prop}}{\Gamma \mid \Phi \vdash \mathsf{pure}_{\mathsf{state}}(\mathsf{protocol}(\mathsf{T}, \mathsf{I}_p, \mathsf{I}_q))}$$

The $\mathsf{pure}_{\mathsf{state}}$ predicate is also closed under embeddings of arbitrary specifications.

$$\frac{\Gamma \vdash \mathsf{S} : \mathsf{Spec}}{\Gamma \mid \Phi \vdash \mathsf{pure}_{\mathsf{state}}(\mathsf{asn}(\mathsf{S}))}$$

Permission purity

The $\mathsf{pure}_{\mathsf{perm}}$ predicate asserts that a given assertion is invariant under arbitrary ownership transfer of action permissions. It is closed under all the usual connectives.

$$\frac{}{\Gamma \mid \Phi \vdash \mathsf{pure}_{\mathsf{perm}}(\bot)} \qquad \frac{}{\Gamma \mid \Phi \vdash \mathsf{pure}_{\mathsf{perm}}(\top)}$$

$$\frac{\Gamma \mid \Phi \vdash \mathsf{pure}_{\mathsf{perm}}(\mathsf{P}) \qquad \Gamma \mid \Phi \vdash \mathsf{pure}_{\mathsf{perm}}(\mathsf{Q}) \qquad op \in \{\wedge, \vee, *, \Rightarrow\}}{\Gamma \mid \Phi \vdash \mathsf{pure}_{\mathsf{perm}}(\mathsf{P}\ op\ \mathsf{Q})}$$

$$\frac{\Gamma \mid \Phi \vdash \forall \mathsf{x} : \tau.\ \mathsf{pure}_{\mathsf{perm}}(\mathsf{P}) \qquad Q \in \{\exists, \forall\}}{\Gamma \mid \Phi \vdash \mathsf{pure}_{\mathsf{perm}}(Q\mathsf{x} : \tau.\ \mathsf{P})}$$

The $\mathsf{pure}_{\mathsf{perm}}$ predicate is also closed under the mini C$^\sharp$ specific state assertions.

$$\frac{\Gamma \vdash \mathsf{M} : \mathsf{Val} \qquad \Gamma \vdash \mathsf{F} : \mathsf{Field} \qquad \Gamma \vdash \mathsf{N} : \mathsf{Val}}{\Gamma \mid \Phi \vdash \mathsf{pure}_{\mathsf{perm}}(\mathsf{M}.\mathsf{F} \mapsto \mathsf{N})} \qquad \frac{\Gamma \vdash \mathsf{M} : \mathsf{Val} \qquad \Gamma \vdash \mathsf{F} : \mathsf{Field} \qquad \Gamma \vdash \mathsf{N} : \mathsf{Val}}{\Gamma \mid \Phi \vdash \mathsf{pure}_{\mathsf{perm}}(\mathsf{M}_{\mathsf{F}} \mapsto \mathsf{N})}$$

$$\frac{\Gamma \vdash \mathsf{N}_1, \mathsf{N}_2 : \mathsf{Val} \qquad \Gamma \vdash \mathsf{M} : \mathsf{Method}}{\Gamma \mid \Phi \vdash \mathsf{pure}_{\mathsf{perm}}(\mathsf{N}_1 \mapsto \mathsf{N}_2.\mathsf{M})} \qquad \frac{\Gamma \vdash \mathsf{M} : \mathsf{Val} \qquad \Gamma \vdash \mathsf{C} : \mathsf{Class}}{\Gamma \mid \Phi \vdash \mathsf{pure}_{\mathsf{perm}}(\mathsf{M} : \mathsf{C})}$$

It is also closed under region and protocol assertions, but not action assertions.

$$\frac{\Gamma \vdash \mathsf{P} : \mathsf{Prop} \qquad \Gamma \vdash \mathsf{R} : \mathsf{Region} \qquad \Gamma \vdash \mathsf{T} : \mathsf{RType} \qquad \Gamma \vdash \mathsf{A} : \mathsf{Val} \qquad \Gamma \mid \Phi \vdash \mathsf{pure}_{\mathsf{perm}}(\mathsf{P})}{\Gamma \mid \Phi \vdash \mathsf{pure}_{\mathsf{perm}}\left(\boxed{\mathsf{P}}^{\mathsf{R},\mathsf{T},\mathsf{A}}\right)}$$

$$\frac{\Gamma \vdash \mathsf{T} : \mathsf{RType} \qquad \Gamma \vdash \mathsf{I}_p, \mathsf{I}_q : \mathsf{Val} \times \mathsf{Action} \times \mathsf{Val} \to \mathsf{Prop}}{\Gamma \mid \Phi \vdash \mathsf{pure}_{\mathsf{perm}}(\mathsf{protocol}(\mathsf{T}, \mathsf{I}_p, \mathsf{I}_q))}$$

Lastly, $\mathsf{pure}_{\mathsf{perm}}$ is closed under embeddings of arbitrary specifications.

$$\frac{\Gamma \vdash \mathsf{S} : \mathsf{Spec}}{\Gamma \mid \Phi \vdash \mathsf{pure}_{\mathsf{perm}}(\mathsf{asn}(\mathsf{S}))}$$

Phantom state

Hoare logics commonly feature auxiliary variables to record an abstraction of the state and history of execution. Auxiliary variables are updated through standard assignments, but they are not allowed to affect the flow of execution. Phantom fields provide a purely logical notion of an auxiliary variable. Phantom fields are updated through view-shifts in the logic, instead of assignments in code. Updating a phantom fields requires full ownership of said field.

$$\frac{\Gamma \vdash \mathsf{M} : \mathsf{Val} \qquad \Gamma \vdash \mathsf{F} : \mathsf{Field} \qquad \Gamma \vdash \mathsf{N} : \mathsf{Val}}{\Gamma \mid \Phi \vdash \mathsf{M}_\mathsf{F} \mapsto \_ \sqsubseteq \mathsf{M}_\mathsf{F} \mapsto \mathsf{N}}$$

Like ordinary fields, phantom fields have a fixed value at any given point in time.

$$\frac{\Gamma \vdash \mathsf{M} : \mathsf{Val} \qquad \Gamma \vdash \mathsf{F} : \mathsf{Field} \qquad \Gamma \vdash \mathsf{N}_1, \mathsf{N}_2 : \mathsf{Val}}{\Gamma \mid \Phi \vdash \mathsf{M}_\mathsf{F} \mapsto \mathsf{N}_1 \land \mathsf{M}_\mathsf{F} \mapsto \mathsf{N}_2 \Rightarrow \mathsf{N}_1 =_{\mathsf{Val}} \mathsf{N}_2}$$

Phantom fields are allocated in the proof rule for constructor verification.

2.3.5 Hoare logic

Structural rules

$$\frac{\Gamma \mid \Phi \vdash (\Delta).\{\mathsf{P}\}\mathsf{s}\{\mathsf{Q}\} \qquad \Gamma, \Delta \mid \Phi \vdash \mathsf{stable}(\mathsf{R}) \qquad \mathsf{mod}(\mathsf{s}) \cap \mathsf{FV}(\mathsf{R}) = \emptyset}{\Gamma \mid \Phi \vdash (\Delta).\{\mathsf{P} * \mathsf{R}\}\mathsf{s}\{\mathsf{Q} * \mathsf{R}\}}$$

$$\frac{\Gamma, \Delta \mid \Phi \vdash \mathsf{stable}(P) \wedge \mathsf{stable}(Q) \quad \Gamma, \Delta \mid \Phi \vdash P \sqsubseteq P' \qquad \Gamma \mid \Phi \vdash (\Delta).\{P'\}s\{Q'\} \qquad \Gamma, \Delta \mid \Phi \vdash Q' \sqsubseteq Q}{\Gamma \mid \Phi \vdash (\Delta).\{P\}s\{Q\}}$$

$$\frac{\Gamma \mid \Phi \vdash (\Delta).\{P\}s_1\{Q\} \qquad \Gamma \mid \Phi \vdash (\Delta).\{Q\}s_2\{R\}}{\Gamma \mid \Phi \vdash (\Delta).\{P\}s_1; s_2\{R\}}$$

Primitive statements

$$\frac{\Gamma; \Delta \vdash P : \mathsf{Prop} \qquad \Gamma, \Delta \mid \Phi \vdash \mathsf{stable}(P) \qquad x, y \in \mathsf{vars}(\Delta)}{\Gamma \mid \Phi \vdash (\Delta).\{P[y/x]\}x = y\{P\}}$$

$$\frac{x, y \in \mathsf{vars}(\Delta)}{\Gamma \mid \Phi \vdash (\Delta).\{x.f \mapsto \_\}x.f = y\{x.f \mapsto y\}}$$

$$\frac{\Gamma; - \vdash M : \mathsf{Val} \qquad x, y \in \mathsf{vars}(\Delta)}{\Gamma \mid \Phi \vdash (\Delta).\{x.f \mapsto M\}y = x.f\{x.f \mapsto M \wedge y =_{\mathsf{Val}} M\}}$$

$$\frac{\Gamma, \bar{z}, \mathtt{this}, \mathsf{ret} \mid \Phi \vdash \mathsf{stable}(P) \wedge \mathsf{stable}(Q) \quad \Gamma \mid \Phi \vdash \triangleright(C.m : (\bar{z}).\{P\}\{\mathsf{ret}.Q\}) \qquad \Gamma \vdash C : \mathsf{Class}}{\Gamma \mid \Phi \vdash (\Delta).\{P[\bar{u}/\bar{z}, y/\mathtt{this}] * y : C\}x = y.m(\bar{u})\{Q[\bar{u}/\bar{z}, y/\mathtt{this}, x/\mathsf{ret}]\}}$$

$$\frac{\Gamma, \bar{z}, \mathtt{this}, \mathsf{ret} \mid \Phi \vdash \mathsf{stable}(P) \wedge \mathsf{stable}(Q) \quad \Gamma \mid \Phi \vdash \triangleright(C.m : (\bar{z}).\{P\}\{\mathsf{ret}.Q\}) \qquad \Gamma \vdash C : \mathsf{Class}}{\Gamma \mid \Phi \vdash (\Delta).\{P[\bar{u}/\bar{z}, y/\mathtt{this}] * y \mapsto z.m * z : C\}x = y(\bar{u})\{Q[\bar{u}/\bar{z}, y/\mathtt{this}, x/\mathsf{ret}]\}}$$

$$\frac{\Gamma, \mathtt{this}, \mathsf{ret} \mid \Phi \vdash \mathsf{stable}(P) \wedge \mathsf{stable}(Q) \quad \Gamma \mid \Phi \vdash \triangleright(C : \{P\}\{\mathsf{ret}.Q\})}{\Gamma \mid \Phi \vdash (\Delta).\{P\}x = \mathtt{new}\ C()\{Q[x/\mathsf{ret}]\}}$$

$$\frac{\Gamma \vdash C : \mathsf{Class} \qquad x, y \in \mathsf{vars}(\Delta)}{\Gamma \mid \Phi \vdash (\Delta).\{\mathsf{emp}\}x = y.m\{x \mapsto y.m\}}$$

$$\frac{\Gamma, \mathtt{this}, \mathsf{ret} \mid \Phi \vdash \mathsf{stable}(P) \wedge \mathsf{stable}(Q) \quad \Gamma \mid \Phi \vdash \triangleright(C.m : (-).\{P\}\{\mathsf{ret}.Q\}) \qquad \Gamma \vdash C : \mathsf{Class} \qquad \Gamma \vdash m : \mathsf{Method}}{\Gamma \mid \Phi \vdash (\Delta).\{P[y/\mathtt{this}] * x \mapsto y.m * y : C\}\mathtt{fork}(x)\{\mathsf{emp}\}}$$

$$\frac{\Gamma \mid \Phi \vdash (\Delta).\{P * x =_{\mathsf{Val}} y\}\bar{s}_1\{Q\} \qquad \Gamma \mid \Phi \vdash (\Delta).\{P * x \neq_{\mathsf{Val}} y\}\bar{s}_2\{Q\}}{\Gamma \mid \Phi \vdash (\Delta).\{P\}\mathtt{if}\ (x = y)\ \mathtt{then}\ \bar{s}_1\ \mathtt{else}\ \bar{s}_2\{Q\}}$$

Method verification

To verify a method, we verify the method body.

$$\frac{\begin{array}{c}\mathsf{C} \in \mathit{CName} \qquad \mathsf{m} \in \mathit{MName} \\ \mathsf{body}(\mathsf{C},\mathsf{m}) = \mathsf{C}\ \mathsf{m}(\Delta)\{\overline{\mathsf{C}\mathsf{y}};\mathsf{s};\mathtt{return}\ \mathsf{r}\} \qquad \mathsf{vars}(\Delta,\mathtt{this}) \cap \mathsf{mod}(\mathsf{s}) = \emptyset \\ \Gamma;\Delta,\mathtt{this} \vdash \mathsf{P} : \mathsf{Prop} \qquad \Gamma;\Delta,\mathtt{this},\mathsf{ret} \vdash \mathsf{Q} : \mathsf{Prop} \\ \Gamma \mid \Phi \vdash (\Delta,\bar{\mathsf{y}},\mathtt{this}).\{\mathsf{P} * \mathtt{this} : \mathsf{C}\}\mathsf{s}\{\mathsf{Q}[\mathsf{r}/\mathsf{ret}]\}\end{array}}{\Gamma \mid \Phi \vdash \mathsf{C}.\mathsf{m} : (\Delta).\{\mathsf{P}\}\{\mathsf{ret}.\mathsf{Q}\}}$$

Mini C$^\sharp$ lacks constructor bodies. To allocate a phantom field we require that the given field does not already exist. Constructors are thus mainly useful for allocating phantom fields, as objects cannot have phantom fields before they have been allocated (See definition of erasure on page 169 for the formal semantics). The constructor verification rule thus allows the user to introduce an arbitrary (finite) number of phantom fields on the newly created object.

$$\frac{\begin{array}{c}\mathsf{C} \in \mathit{CName} \qquad \mathsf{fields}(\mathsf{C}) = \bar{\mathsf{f}} \qquad \Gamma;\mathtt{this} \vdash \mathsf{P} : \mathsf{Prop} \qquad \Gamma;\mathtt{this},\mathsf{ret} \vdash \mathsf{Q} : \mathsf{Prop} \\ \Gamma \mid \Phi \vdash (\bar{\mathsf{y}},\mathtt{this}).\{\mathsf{P} * \mathtt{this}.\bar{\mathsf{f}} \mapsto \mathsf{null} * \overline{\mathtt{this}\_ \mapsto \_} * \mathtt{this} : \mathsf{C}\}\mathtt{skip}\{\mathsf{Q}[\mathtt{this}/\mathsf{ret}]\}\end{array}}{\Gamma \mid \Phi \vdash \mathsf{C} : \{\mathsf{P}\}\{\mathsf{ret}.\mathsf{Q}\}}$$

3 Examples

In this section we present two examples to illustrate how to use the proof system. The first example illustrates reasoning about recursion through the store using guarded recursion. The second example illustrates higher-order concurrent abstract predicates using a spin-lock.

3.1 Recursion through the store

To illustrate how the embedding of specifications in assertions combined with guarded recursion allows us to reason about recursion through the store, consider the following program.

```
using System;

class NatRec {
  private Func<NatRec, int, int> r = null;

  public NatRec(Func<NatRec, int, int> r) {
    this.r = r;
  }

  public int comp(int n) {
    return r(this, n);
  }
}

class Factorial {
  private int f(NatRec r, int n) {
    if (n == 0) return 1; else return n * r.comp (n-1);
  }

  public static void fac(int m) {
    NatRec fac = new NatRec(f);
    return fac.comp(m);
  }
}
```

This program implements a factorial computation using recursion through the store. In particular, the NatRec constructor takes as argument a delegate that itself takes as argument a NatRec instance and an integer, and returns an integer. When this delegate is invoked in the comp method, NatRec calls the delegate with a reference to itself, allowing the delegate to recursively call itself through the comp method. Since the factorial client explicitly exploits this behavior of the NatRec class, we thus have to give a NatRec specification that explicitly allows the client delegate to call the comp method on its NatRec argument, and that this recursive call returns the factorial of its argument.

To specify NatRec, we require the client to choose a function on natural numbers ($f : \mathbb{N} \to \mathbb{N}$), representing the intended computation. The NatRec constructor then

requires the client to provide a delegate that implements f, assuming that calling comp on its NatRec argument, implements f. In this case, comp implements f. We can thus specify NatRec as follows.

$$\begin{aligned} S_{\mathsf{NatRec}} \stackrel{\text{def}}{=} \forall \mathsf{f} \in \mathbb{N} \to \mathbb{N}.\ \exists \mathsf{rec} : \mathsf{Val} \to \mathsf{Prop}. \\ & \texttt{NatRec} : (\mathsf{r}).\{I_{del}(\mathsf{rec}, \mathsf{f}, \mathsf{r})\}\{\mathsf{rec}(\texttt{this})\} \wedge \\ & \texttt{NatRec.comp} : (\mathsf{n}).\{\mathsf{rec}(\texttt{this})\}\{\mathsf{ret}.\ \mathsf{rec}(\texttt{this}) \wedge \mathsf{ret} = \mathsf{f}(\mathsf{n})\} \end{aligned}$$

where $I_{del} : (\mathsf{Val} \to \mathsf{Prop}) \times (\mathbb{N} \to \mathbb{N}) \times \mathsf{Val} \to \mathsf{Prop}$ is defined as

$$I_{del}(\mathsf{rec}, \mathsf{f}, \mathsf{y}) \stackrel{\text{def}}{=} \mathsf{y} \mapsto (\mathsf{x}, \mathsf{n}).\{\mathsf{rec}(\mathsf{x})\}\{\mathsf{ret}.\ \mathsf{rec}(\mathsf{x}) * \mathsf{ret} = \mathsf{f}(\mathsf{n})\}$$

Lemma 1.

$$- \mid - \vdash S_{\mathit{NatRec}}$$

Proof.

- suppose $\mathsf{f} : \mathbb{N} \to \mathbb{N}$
- define $\mathsf{rec}' : (\mathsf{Val} \to \mathsf{Prop}) \to (\mathsf{Val} \to \mathsf{Prop})$ as the following functional

$$\mathsf{rec}'(\mathsf{p})(\mathsf{x}) \stackrel{\text{def}}{=} \exists \mathsf{y} : \mathsf{Val}.\ \mathsf{x.r} \mapsto \mathsf{y} * \triangleright I_{del}(\mathsf{p}, \mathsf{f}, \mathsf{y})$$

- define rec as the fixed-point of rec'

$$\mathsf{rec} \stackrel{\text{def}}{=} \mathsf{fix}(\mathsf{rec}')$$

- since p is guarded in rec', we have that

$$\mathsf{rec}(\mathsf{x}) \Leftrightarrow \exists \mathsf{y} : \mathsf{Val}.\ \mathsf{x.r} \mapsto \mathsf{y} * \triangleright I_{del}(\mathsf{p}, \mathsf{f}, \mathsf{y})$$

- to prove the constructor we thus need to "forget a step"

```
public NatRec(Func<NatRec, int, int> r) {
     {this.r ↦ null ∗ I_del(rec, f, r)}
     {this.r ↦ null ∗ ▷I_del(rec, f, r)}
  this.r = r;
     {this.r ↦ r ∗ ▷I_del(rec, f, r)}
     {rec(this)}
}
```

- and the proof of comp is similarly straight-forward

```
public int comp(int n) {
     {rec(this)}
  Func<NatRec, int, int> y; int z;
     {rec(this)}
     {∃y : Val. this.r ↦ y ∗ ▷I_del(rec, f, y)}
```

```
  y = this.r;
    {this.r ↦ y ∗ ▷I_del(rec, f, y)}
    {this.r ↦ y ∗ ▷I_del(rec, f, y) ∗ ▷I_del(rec, f, y)}
    {rec(this) ∗ ▷I_del(rec, f, y)}
    {rec(this) ∗ ▷f ↦ (x, n).{rec(x)}{ret. rec(x) ∗ ret =ℕ f(n)}}
  z = f(this, n);
    {rec(this) ∗ z =ℕ f(n)}
  return z;
    {rec(this) ∗ ret =ℕ f(n)}
}
```

□

Lemma 2.

$$- \mid S_{\mathsf{NatRec}} \vdash \mathit{Factorial.fac} : (\mathsf{n}).\{\mathsf{emp}\}\{\mathsf{ret} = \mathsf{n}!\}$$

Proof.

- Pick f to be $! : \mathbb{N} \to \mathbb{N}$ in S_{NatRec}
- Then we first need to show that

  $$\mathsf{Factorial.f} : (\mathsf{x}, \mathsf{n}).\{\mathsf{rec}(\mathsf{x})\}\{\mathsf{rec}(\mathsf{x}) * \mathsf{ret} = \mathsf{n}!\}$$

  which follows by a straight-foward proof:

```
private int f(NatRec x, int n) {
  int m;
    {rec(x)}
  if (n == 0)
      {rec(x) ∗ n = 0}
    m = 1;
      {rec(x) ∗ m = n!}
  else {
      {rec(x)}
    m = x.comp(n-1);
      {rec(x) ∗ m = (n − 1)!}
    m = n * m;
      {rec(x) ∗ m = n!}
  }
    {rec(x) ∗ m = n!}
  return m;
    {ret. rec(x) ∗ ret = n!}
}
```

- Now, we can embed this specification for f to derive the desired fac specification:

```
public static void fac(int n) {
  Func<NatRec, int, int> g; NatRec fac; int m;
    {emp}
  g = this.f;
    {g ↦ (x, n).{rec(x)}{rec(x) ∗ ret = n!}
  fac = new NatRec(g);
    {rec(fac)}
  m = fac.comp(n);
    {rec(fac) ∗ m = n!}
  return m;
    {ret. ret = n!}
}
```

□

3.2 Spin lock

To illustrate the use of higher-order concurrent abstract predicates, consider the following spin-lock.

```
class Lock {
  private int locked = 0;

  public void Acquire() {
    x = CAS(this.locked, 0, 1);
    if(x != null)
      Acquire();
  }

  public void Release() {
    locked = 0;
  }
}
```

It uses the private field locked to maintain the current state of the lock (0 for unlocked, 1 for locked) and a compare-and-swap to atomically acquire the lock.

The standard separation logic specification of a lock, associates a resource invariant R with each lock, which describes the resources protected by the lock. The resource invariant is thus transferred to the client upon acquiring the lock, and transferred back upon releasing the lock. Since the resource invariant R might itself assert ownership of shared resources using CAP, we require that R is stable, that it is independent of the lock region type (picked by the client), and that it is expressible using state-independent

protocols:

$$
\begin{aligned}
S_{\mathsf{Lock}} = \; & \exists \mathsf{islock}, \mathsf{locked} : \mathsf{RType} \times \mathsf{Prop} \times \mathsf{Val} \to \mathsf{Prop}. \\
& \quad \forall \mathsf{t} \in \mathsf{RType}.\ \forall \mathsf{R} : \mathsf{Prop}.\ \mathsf{indep}_{\mathsf{t}}(\mathsf{R}) \wedge \mathsf{sip}(\mathsf{R}) \wedge \mathsf{stable}(\mathsf{R}) \;\Rightarrow \\
& \qquad \mathsf{new}\ \mathsf{Lock}() : (\text{-}).\ \{\mathsf{R}\}\ \{\mathsf{ret}.\ \exists \mathsf{s} : \mathsf{RType}.\ \mathsf{isLock}(\mathsf{s}, \mathsf{R}, \mathsf{ret}) * \mathsf{t} \leq \mathsf{s}\} \\
& \qquad \mathsf{Lock.Acquire} : (\text{-}).\ \{\mathsf{isLock}(\mathsf{t}, \mathsf{R}, \mathtt{this})\}\ \{\mathsf{locked}(\mathsf{t}, \mathsf{R}, \mathtt{this}) * \mathsf{R}\} \\
& \qquad \mathsf{Lock.Release} : (\text{-}).\ \{\mathsf{locked}(\mathsf{t}, \mathsf{R}, \mathtt{this}) * \mathsf{R}\}\ \{\mathsf{isLock}(\mathsf{t}, \mathsf{R}, \mathtt{this})\} \\
& \qquad \forall \mathsf{x} : \mathsf{Val}.\ \mathsf{valid}(\mathsf{isLock}(\mathsf{t}, \mathsf{R}, \mathsf{x}) \Leftrightarrow \mathsf{isLock}(\mathsf{t}, \mathsf{R}, \mathsf{x}) * \mathsf{isLock}(\mathsf{t}, \mathsf{R}, \mathsf{x})) \wedge \\
& \qquad\qquad \mathsf{stable}(\mathsf{isLock}(\mathsf{t}, \mathsf{R}, \mathsf{x})) \wedge \mathsf{stable}(\mathsf{locked}(\mathsf{t}, \mathsf{R}, \mathsf{x}))
\end{aligned}
$$

The $\mathsf{isLock}(\mathsf{t}, \mathsf{R}, \mathsf{x})$ predicate asserts that x refers to a lock with resource invariant R. The $\mathsf{isLock}(-)$ predicate is freely duplicable, allowing multiple clients to use the same lock. Likewise, the $\mathsf{locked}(\mathsf{t}, \mathsf{R}, \mathsf{x})$ predicate asserts that x refers to a locked lock with resource invariant R, and the permission to unlock it; $\mathsf{locked}(-)$ is thus not duplicable.

Lemma 3.

$$- \mid - \vdash S_{\mathit{Lock}}$$

Proof.

- Define the representation predicates $\mathsf{isLock}(-)$ and $\mathsf{locked}(-)$ as follows

  $$
  \begin{aligned}
  & \mathsf{isLock}(\mathsf{t}, \mathsf{R}, \mathsf{x}) \stackrel{\text{def}}{=} \\
  & \quad \exists \mathsf{r} : \mathsf{Region}.\ \exists \pi : \mathsf{Perm}.\ \mathsf{asn}(\mathsf{stable}(\mathsf{R}) \wedge \mathsf{indep}_{\mathsf{t}}(\mathsf{R}) \wedge \mathsf{sip}(\mathsf{R})) \\
  & \qquad * \boxed{(\mathsf{x.locked} \mapsto 0 * \mathsf{R} * [\textsc{Unlock}]^{\mathsf{r}}_1) \vee \mathsf{x.locked} \mapsto 1}^{\mathsf{r},\mathsf{t},(\mathsf{x},\mathsf{r})}_{\mathsf{I}(\mathsf{R})} * [\textsc{Lock}]^{\mathsf{r}}_{\pi} * [\tau_1]^{\mathsf{r}}_{\pi} * [\tau_2]^{\mathsf{r}}_{\pi}
  \end{aligned}
  $$

  $$
  \begin{aligned}
  & \mathsf{locked}(\mathsf{t}, \mathsf{R}, \mathsf{x}) \stackrel{\text{def}}{=} \\
  & \quad \exists \mathsf{r} : \mathsf{Region}.\ \exists \pi : \mathsf{Perm}.\ \mathsf{asn}(\mathsf{stable}(\mathsf{R}) \wedge \mathsf{indep}_{\mathsf{t}}(\mathsf{R}) \wedge \mathsf{sip}(\mathsf{R})) \\
  & \qquad * \boxed{\mathsf{x.locked} \mapsto 1}^{\mathsf{r},\mathsf{t},(\mathsf{x},\mathsf{r})}_{\mathsf{I}(\mathsf{R})} * [\textsc{Lock}]^{\mathsf{r}}_{\pi} * [\tau_1]^{\mathsf{r}}_{\pi} * [\tau_2]^{\mathsf{r}}_{\pi} * [\textsc{Unlock}]^{\mathsf{r}}_1
  \end{aligned}
  $$

  where I is the parametric protocol:

  $$
  \mathsf{I}(\mathsf{R})(\mathsf{x}, \mathsf{r}) \stackrel{\text{def}}{=} \left(\begin{array}{rl}
  \textsc{Lock} : & \mathsf{x.locked} \mapsto 0 * \mathsf{R} * [\textsc{Unlock}]^{\mathsf{r}}_1 \rightsquigarrow \mathsf{x.locked} \mapsto 1 \\
  \textsc{Unlock} : & \mathsf{x.locked} \mapsto 1 \rightsquigarrow \mathsf{x.locked} \mapsto 0 * \mathsf{R} * [\textsc{Unlock}]^{\mathsf{r}}_1 \\
  [\tau_1] : & \mathsf{x.locked} \mapsto 0 * \mathsf{R} * [\textsc{Unlock}]^{\mathsf{r}}_1 \rightsquigarrow \mathsf{x.locked} \mapsto 0 * \mathsf{R} * [\textsc{Unlock}]^{\mathsf{r}}_1 \\
  [\tau_2] : & \mathsf{x.locked} \mapsto 1 \rightsquigarrow \mathsf{x.locked} \mapsto 1
  \end{array} \right)
  $$

- Next, we have to prove the representation predicates stable; we sketch the proof for $\mathsf{isLock}(-)$; the proof for $\mathsf{locked}(-)$ is simpler. To prove $\mathsf{isLock}(-)$ stable, first we apply rule StableAsn to extract the embedded specification assumptions from $\mathsf{isLock}(-)$ about the resource invariant. It thus suffices to prove,

  $$\mathsf{t} : \mathsf{RType}, \mathsf{R} : \mathsf{Prop}, \mathsf{x} : \mathsf{Val} \mid \mathsf{stable}(\mathsf{R}), \mathsf{indep}_{\mathsf{t}}(\mathsf{R}), \mathsf{sip}(\mathsf{R}) \vdash \mathsf{stable}(\mathsf{isLock}(\mathsf{t}, \mathsf{R}, \mathsf{x}))$$

From the $\mathsf{sip}(\mathsf{R})$ assumption, there exists $\mathsf{S}, \mathsf{P} : \mathsf{Prop}$ such that $\mathsf{pure}_{\mathsf{protocol}}(\mathsf{S})$, $\mathsf{pure}_{\mathsf{state}}(\mathsf{P})$ and $\mathsf{valid}(\mathsf{R} \Leftrightarrow (\mathsf{S} * \mathsf{P}))$. We can use this decomposition of R into S and P to pull out the region-assertions of R from the shared region in $\mathsf{isLock}(-)$. That is, in the given context,

$$\boxed{(\mathsf{x.locked} \mapsto 0 * \mathsf{R} * [\textsc{Unlock}]^{\mathsf{r}}_1) \vee \mathsf{x.locked} \mapsto 1}^{\mathsf{r},\mathsf{t},(\mathsf{x},\mathsf{r})}_{\mathsf{I}(\mathsf{R})} \Leftrightarrow$$
$$\boxed{(\mathsf{x.locked} \mapsto 0 * \mathsf{S} * [\textsc{Unlock}]^{\mathsf{r}}_1) \vee \mathsf{x.locked} \mapsto 1}^{\mathsf{r},\mathsf{t},(\mathsf{x},\mathsf{r})}_{\mathsf{I}(\mathsf{R})} * \mathsf{P}$$

Stability of $\mathsf{isLock}(-)$ thus reduces to,

$$\Gamma \mid \Phi \vdash \mathsf{stable}(\boxed{(\mathsf{x.locked} \mapsto 0 * \mathsf{S} * [\textsc{Unlock}]^{\mathsf{r}}_1) \vee \mathsf{x.locked} \mapsto 1}^{\mathsf{r},\mathsf{t},(\mathsf{x},\mathsf{r})}_{\mathsf{I}(\mathsf{R})} * \mathsf{P})$$

where

$$\Gamma = \mathsf{t} : \mathsf{RType}, \mathsf{R}, \mathsf{S}, \mathsf{P} : \mathsf{Prop}, \mathsf{x} : \mathsf{Val}, \mathsf{r} : \mathsf{Region}, \pi : \mathsf{Perm}$$
$$\Phi = \mathsf{stable}(\mathsf{S} * \mathsf{P}), \mathsf{indep}_{\mathsf{t}}(\mathsf{S} * \mathsf{P}), \mathsf{pure}_{\mathsf{protocol}}(\mathsf{S}), \mathsf{pure}_{\mathsf{state}}(\mathsf{P}), \mathsf{valid}(\mathsf{R} \Leftrightarrow \mathsf{S} * \mathsf{P})$$

By StableD this decomposes into stability under every action. We prove stability under the Lock and Unlock actions, the remaining actions are similar. We thus have to show that,

$$\Gamma \mid \Phi \vdash \mathsf{stable}^{\mathsf{r}}_{\textsc{Lock}}(\boxed{(\mathsf{x.locked} \mapsto 0 * \mathsf{S} * [\textsc{Unlock}]^{\mathsf{r}}_1) \vee \mathsf{x.locked} \mapsto 1}^{\mathsf{r},\mathsf{t},(\mathsf{x},\mathsf{r})}_{\mathsf{I}(\mathsf{R})} * \mathsf{P})$$

and

$$\Gamma \mid \Phi \vdash \mathsf{stable}^{\mathsf{r}}_{\textsc{Unlock}}(\boxed{(\mathsf{x.locked} \mapsto 0 * \mathsf{S} * [\textsc{Unlock}]^{\mathsf{r}}_1) \vee \mathsf{x.locked} \mapsto 1}^{\mathsf{r},\mathsf{t},(\mathsf{x},\mathsf{r})}_{\mathsf{I}(\mathsf{R})} * \mathsf{P})$$

To apply the StableClosed rule we need to prove that,

$$\Gamma \mid \Phi \vdash \mathsf{pure}_{\mathsf{protocol}}((\mathsf{x.locked} \mapsto 0 * \mathsf{S} * [\textsc{Unlock}]^{\mathsf{r}}_1) \vee \mathsf{x.locked} \mapsto 1)$$
$$\Gamma \mid \Phi \vdash \mathsf{pure}_{\mathsf{state}}(\mathsf{P})$$
$$\Gamma \mid \Phi \vdash \mathsf{stable}(((\mathsf{x.locked} \mapsto 0 * \mathsf{S} * [\textsc{Unlock}]^{\mathsf{r}}_1) \vee \mathsf{x.locked} \mapsto 1) * \mathsf{P})$$
$$\Gamma \mid \Phi \vdash \mathsf{indep}_{\mathsf{t}}(((\mathsf{x.locked} \mapsto 0 * \mathsf{S} * [\textsc{Unlock}]^{\mathsf{r}}_1) \vee \mathsf{x.locked} \mapsto 1) * \mathsf{P})$$

All of these follow fairly easily from the assumptions in Φ. Stability under the Lock action thus reduces to

$$\begin{aligned}\Gamma \mid \Phi \vdash\ & \mathsf{valid}(((\mathsf{x.locked} \mapsto 0 * \mathsf{R} * [\textsc{Unlock}]^{r}_1) \wedge \\ & \qquad ((\mathsf{x.locked} \mapsto 0 * \mathsf{S} * [\textsc{Unlock}]^{\mathsf{r}}_1) \vee \mathsf{x.locked} \mapsto 1)) \Rightarrow \bot) \vee \\ & \mathsf{valid}(\mathsf{x.locked} \mapsto 1 \Rightarrow ((\mathsf{x.locked} \mapsto 0 * \mathsf{S} * [\textsc{Unlock}]^{\mathsf{r}}_1) \vee \mathsf{x.locked} \mapsto 1))\end{aligned}$$

and stability under the UNLOCK action reduces to

$$\begin{aligned}\Gamma \mid \Phi \vdash\ & \mathsf{valid}(((\mathsf{x.locked} \mapsto 0 * \mathsf{R} * [\textsc{Unlock}]^{\mathsf{r}}_1) \wedge \\ & \qquad ((\mathsf{x.locked} \mapsto 0 * \mathsf{S} * [\textsc{Unlock}]^{\mathsf{r}}_1) \vee \mathsf{x.locked} \mapsto 1)) \Rightarrow \bot) \vee \\ & \mathsf{valid}((\mathsf{x.locked} \mapsto 0 * \mathsf{R} * [\textsc{Unlock}]^{\mathsf{r}}_1) \ \Rightarrow \\ & \qquad ((\mathsf{x.locked} \mapsto 0 * \mathsf{S} * [\textsc{Unlock}]^{\mathsf{r}}_1) \vee \mathsf{x.locked} \mapsto 1))\end{aligned}$$

In both cases we prove the second disjunct. The first obligation is trivial. The second obligation follows from the assumption that $\mathsf{valid}(\mathsf{R} \Leftrightarrow \mathsf{S} * \mathsf{P})$.

- Next, we have to verify the lock methods. Below we give a rough proof-sketch of Lock.Acquire. Formally, we first use the LOEB rule to perform Löb induction, to reason about the recursive call.

```
public void Acquire() {
```
$\{\mathsf{isLock}(\mathsf{t}, \mathsf{R}, \mathtt{this})\}$

$\{\boxed{(\mathtt{this.locked} \mapsto 0 * \mathsf{R} * [\textsc{Unlock}]^{\mathsf{r}}_1) \vee \mathtt{this.locked} \mapsto 1}^{\mathsf{r},\mathsf{t},(\mathtt{this},\mathsf{r})}_{\mathsf{I}(\mathsf{R})} * [\textsc{Lock}]^{\mathsf{r}}_\pi * [\tau_1]^{\mathsf{r}}_\pi * [\tau_2]^{\mathsf{r}}_\pi\}$

```
  x = CAS(this.locked, 0, 1);
```
$\{(\mathsf{x} = \mathtt{this} * \boxed{\mathtt{this.locked} \mapsto 1}^{\mathsf{r},\mathsf{t},(\mathtt{this},\mathsf{r})}_{\mathsf{I}(\mathsf{R})} * [\textsc{Lock}]^{\mathsf{r}}_\pi * [\tau_1]^{\mathsf{r}}_\pi * [\tau_2]^{\mathsf{r}}_\pi * [\textsc{Unlock}]^{\mathsf{r}}_1 * \mathsf{R}) \ \vee$
$(\mathsf{x} = \mathsf{null} * \boxed{\mathtt{this.locked} \mapsto 0 * \mathsf{R} * [\textsc{Unlock}]^{\mathsf{r}}_1}^{\mathsf{r},\mathsf{t},(\mathtt{this},\mathsf{r})}_{\mathsf{I}(\mathsf{R})} * [\textsc{Lock}]^{\mathsf{r}}_\pi * [\tau_1]^{\mathsf{r}}_\pi * [\tau_2]^{\mathsf{r}}_\pi)\}$

$\{(\mathsf{x} = \mathtt{this} * \boxed{\mathtt{this.locked} \mapsto 1}^{\mathsf{r},\mathsf{t},(\mathtt{this},\mathsf{r})}_{\mathsf{I}(\mathsf{R})} * [\textsc{Lock}]^{\mathsf{r}}_\pi * [\tau_1]^{\mathsf{r}}_\pi * [\tau_2]^{\mathsf{r}}_\pi * [\textsc{Unlock}]^{\mathsf{r}}_1 * \mathsf{R}) \ \vee$
$(\mathsf{x} = \mathsf{null} * \boxed{(\mathtt{this.locked} \mapsto 0 * \mathsf{R} * [\textsc{Unlock}]^{\mathsf{r}}_1) \vee \mathtt{this.locked} \mapsto 1}^{\mathsf{r},\mathsf{t},(\mathtt{this},\mathsf{r})}_{\mathsf{I}(\mathsf{R})} * [\textsc{Lock}]^{\mathsf{r}}_\pi * [\tau_1]^{\mathsf{r}}_\pi * [\tau_2]^{\mathsf{r}}_\pi)\}$

```
  if(x == null) {
```
$\{\boxed{(\mathtt{this.locked} \mapsto 0 * \mathsf{R} * [\textsc{Unlock}]^{\mathsf{r}}_1) \vee \mathtt{this.locked} \mapsto 1}^{\mathsf{r},\mathsf{t},(\mathtt{this},\mathsf{r})}_{\mathsf{I}(\mathsf{R})} * [\textsc{Lock}]^{\mathsf{r}}_\pi * [\tau_1]^{\mathsf{r}}_\pi * [\tau_2]^{\mathsf{r}}_\pi\}$

$\{\mathsf{isLock}(\mathsf{t}, \mathsf{R}, \mathtt{this})\}$

```
    Acquire();
```
$\{\mathsf{locked}(\mathsf{t}, \mathsf{R}, \mathtt{this}) * \mathsf{R}\}$

```
  } else {
```
$\{\boxed{\mathtt{this.locked} \mapsto 1}^{\mathsf{r},\mathsf{t},(\mathtt{this},\mathsf{r})}_{\mathsf{I}(\mathsf{R})} * [\textsc{Lock}]^{\mathsf{r}}_\pi * [\tau_1]^{\mathsf{r}}_\pi * [\tau_2]^{\mathsf{r}}_\pi * [\textsc{Unlock}]^{\mathsf{r}}_1 * \mathsf{R}\}$

$\{\mathsf{locked}(\mathsf{t}, \mathsf{R}, \mathtt{this}) * \mathsf{R}\}$

```
  }
```
$\{\mathsf{locked}(\mathsf{t}, \mathsf{R}, \mathtt{this}) * \mathsf{R}\}$

```
}
```

The main step is the atomic compare-and-swap which accesses the locked field, which is owned by the shared lock region. Note that the immediate post-condition of the compare-and-swap is *not* stable. In particular, if the compare-and-swap fails, then in the instant the compare-and-swap fails, the lock is already locked. However, the owner of the lock is free to unlock the lock, hence it is not stable to assert the

lock is locked. We thus immediately apply the rule of consequence, to weaken the post-condition to something that *is* stable.

The next step is thus to verify the atomic update of the shared region:

$$(\mathsf{x}).\ \{\boxed{(\mathtt{this.locked} \mapsto 0 * \mathsf{R} * [\textsc{Unlock}]^{\mathsf{r}}_1) \vee \mathtt{this.locked} \mapsto 1}^{\mathsf{r},\mathsf{t},(\mathtt{this},\mathsf{r})}_{\mathsf{I}(\mathsf{R})} * [\textsc{Lock}]^{\mathsf{r}}_\pi * [\tau_1]^{\mathsf{r}}_\pi * [\tau_2]^{\mathsf{r}}_\pi\}$$
$$\langle \mathsf{x} = \mathtt{CAS}(\mathtt{this.locked}, 0, 1)\rangle$$
$$\left\{\begin{array}{l}(\mathsf{x} = \mathtt{this} * \boxed{\mathtt{this.locked} \mapsto 1}^{\mathsf{r},\mathsf{t},(\mathtt{this},\mathsf{r})}_{\mathsf{I}(\mathsf{R})} * [\textsc{Lock}]^{\mathsf{r}}_\pi * [\tau_1]^{\mathsf{r}}_\pi * [\tau_2]^{\mathsf{r}}_\pi * [\textsc{Unlock}]^{\mathsf{r}}_1 * \mathsf{R}) \vee \\ (\mathsf{x} = \mathsf{null} * \boxed{\mathtt{this.locked} \mapsto 0 * \mathsf{R} * [\textsc{Unlock}]^{\mathsf{r}}_1}^{\mathsf{r},\mathsf{t},(\mathtt{this},\mathsf{r})}_{\mathsf{I}(\mathsf{R})} * [\textsc{Lock}]^{\mathsf{r}}_\pi * [\tau_1]^{\mathsf{r}}_\pi * [\tau_2]^{\mathsf{r}}_\pi)\end{array}\right\}$$

By rule OpenA, this reduces to proving that the compare-and-swap performs the state update on the shared region:

$$(\mathsf{x}).\ \{((\mathtt{this.locked} \mapsto 0 * \mathsf{R} * [\textsc{Unlock}]^{\mathsf{r}}_1) \vee \mathtt{this.locked} \mapsto 1) * [\textsc{Lock}]^{\mathsf{r}}_\pi * [\tau_1]^{\mathsf{r}}_\pi * [\tau_2]^{\mathsf{r}}_\pi\}$$
$$\langle \mathsf{x} = \mathtt{CAS}(\mathtt{this.locked}, 0, 1)\rangle^{\mathsf{t}}$$
$$\left\{\begin{array}{l}(\mathsf{x} = \mathtt{this} * \mathtt{this.locked} \mapsto 1 * [\textsc{Lock}]^{\mathsf{r}}_\pi * [\tau_1]^{\mathsf{r}}_\pi * [\tau_2]^{\mathsf{r}}_\pi * [\textsc{Unlock}]^{\mathsf{r}}_1 * \mathsf{R}) \vee \\ (\mathsf{x} = \mathsf{null} * \mathtt{this.locked} \mapsto 0 * \mathsf{R} * [\textsc{Unlock}]^{\mathsf{r}}_1 * [\textsc{Lock}]^{\mathsf{r}}_\pi * [\tau_1]^{\mathsf{r}}_\pi * [\tau_2]^{\mathsf{r}}_\pi)\end{array}\right\}$$

and that this update is allowed:

$$\boxed{(\mathtt{this.locked} \mapsto 0 * \mathsf{R} * [\textsc{Unlock}]^{\mathsf{r}}_1) \vee \mathtt{this.locked} \mapsto 1}^{\mathsf{r},\mathsf{t},(\mathtt{this},\mathsf{r})}_{\mathsf{I}(\mathsf{R})} * [\textsc{Lock}]^{\mathsf{r}}_\pi * [\tau_1]^{\mathsf{r}}_\pi * [\tau_2]^{\mathsf{r}}_\pi$$
$$\rightsquigarrow^{\mathsf{r},\mathsf{t}}_{(\mathsf{x}).\langle \mathsf{x}=\mathtt{CAS}(\mathtt{this.locked},0,1)\rangle}$$
$$(\mathsf{x} = \mathtt{this} * \boxed{\mathtt{this.locked} \mapsto 1}^{\mathsf{r},\mathsf{t},(\mathtt{this},\mathsf{r})}_{\mathsf{I}(\mathsf{R})} * [\textsc{Lock}]^{\mathsf{r}}_\pi * [\tau_1]^{\mathsf{r}}_\pi * [\tau_2]^{\mathsf{r}}_\pi * [\textsc{Unlock}]^{\mathsf{r}}_1 * \mathsf{R}) \vee$$
$$(\mathsf{x} = \mathsf{null} * \boxed{\mathtt{this.locked} \mapsto 0 * \mathsf{R} * [\textsc{Unlock}]^{\mathsf{r}}_1}^{\mathsf{r},\mathsf{t},(\mathtt{this},\mathsf{r})}_{\mathsf{I}(\mathsf{R})} * [\textsc{Lock}]^{\mathsf{r}}_\pi * [\tau_1]^{\mathsf{r}}_\pi * [\tau_2]^{\mathsf{r}}_\pi)$$

The first proof obligation follows by a fairly standard separation logic proof (using rule ACas). To prove that the update is allowed, it suffices (by the rule of consequence) to prove,

$$\boxed{(\mathtt{this.locked} \mapsto 0 * \mathsf{R} * [\textsc{Unlock}]^{\mathsf{r}}_1) \vee \mathtt{this.locked} \mapsto 1}^{\mathsf{r},\mathsf{t},(\mathtt{this},\mathsf{r})}_{\mathsf{I}(\mathsf{R})} * [\textsc{Lock}]^{\mathsf{r}}_\pi * [\tau_1]^{\mathsf{r}}_\pi * [\tau_2]^{\mathsf{r}}_\pi$$
$$\rightsquigarrow^{\mathsf{r},\mathsf{t}}_{(\mathsf{x}).\langle \mathsf{x}=\mathtt{CAS}(\mathtt{this.locked},0,1)\rangle}$$
$$(\mathsf{x} = \mathtt{this} * \boxed{\mathtt{this.locked} \mapsto 1}^{\mathsf{r},\mathsf{t},(\mathtt{this},\mathsf{r})}_{\mathsf{I}(\mathsf{R})}) \vee$$
$$(\mathsf{x} = \mathsf{null} * \boxed{\mathtt{this.locked} \mapsto 0 * \mathsf{R} * [\textsc{Unlock}]^{\mathsf{r}}_1}^{\mathsf{r},\mathsf{t},(\mathtt{this},\mathsf{r})}_{\mathsf{I}(\mathsf{R})})$$

- From the update allowed rules, we can perform case analysis on each disjunction,

obtaining the following four proof obligations:

$$\boxed{\mathtt{this.locked} \mapsto 0 * \mathsf{R} * [\textsc{Unlock}]_1^{\mathsf{r}}}^{\mathsf{r},\mathsf{t},(\mathtt{this},\mathsf{r})}_{\mathsf{I}(\mathsf{R})} * [\textsc{Lock}]_\pi^{\mathsf{r}} * [\tau_1]_\pi^{\mathsf{r}} * [\tau_2]_\pi^{\mathsf{r}}$$
$$\rightsquigarrow^{\mathsf{r},\mathsf{t}}_{(\mathsf{x}).\langle \mathsf{x}=\mathtt{CAS}(\mathtt{this.locked},0,1)\rangle} (\mathsf{x} = \mathtt{this} * \boxed{\mathtt{this.locked} \mapsto 1}^{\mathsf{r},\mathsf{t},(\mathtt{this},\mathsf{r})}_{\mathsf{I}(\mathsf{R})})$$

$$\boxed{\mathtt{this.locked} \mapsto 0 * \mathsf{R} * [\textsc{Unlock}]_1^{\mathsf{r}}}^{\mathsf{r},\mathsf{t},(\mathtt{this},\mathsf{r})}_{\mathsf{I}(\mathsf{R})} * [\textsc{Lock}]_\pi^{\mathsf{r}} * [\tau_1]_\pi^{\mathsf{r}} * [\tau_2]_\pi^{\mathsf{r}}$$
$$\rightsquigarrow^{\mathsf{r},\mathsf{t}}_{(\mathsf{x}).\langle \mathsf{x}=\mathtt{CAS}(\mathtt{this.locked},0,1)\rangle} (\mathsf{x} = \mathsf{null} * \boxed{\mathtt{this.locked} \mapsto 0 * \mathsf{R} * [\textsc{Unlock}]_1^{\mathsf{r}}}^{\mathsf{r},\mathsf{t},(\mathtt{this},\mathsf{r})}_{\mathsf{I}(\mathsf{R})})$$
$$\boxed{\mathtt{this.locked} \mapsto 1}^{\mathsf{r},\mathsf{t},(\mathtt{this},\mathsf{r})}_{\mathsf{I}(\mathsf{R})} * [\textsc{Lock}]_\pi^{\mathsf{r}} * [\tau_1]_\pi^{\mathsf{r}} * [\tau_2]_\pi^{\mathsf{r}}$$
$$\rightsquigarrow^{\mathsf{r},\mathsf{t}}_{(\mathsf{x}).\langle \mathsf{x}=\mathtt{CAS}(\mathtt{this.locked},0,1)\rangle} (\mathsf{x} = \mathtt{this} * \boxed{\mathtt{this.locked} \mapsto 1}^{\mathsf{r},\mathsf{t},(\mathtt{this},\mathsf{r})}_{\mathsf{I}(\mathsf{R})})$$

$$\boxed{\mathtt{this.locked} \mapsto 1}^{\mathsf{r},\mathsf{t},(\mathtt{this},\mathsf{r})}_{\mathsf{I}(\mathsf{R})} * [\textsc{Lock}]_\pi^{\mathsf{r}} * [\tau_1]_\pi^{\mathsf{r}} * [\tau_2]_\pi^{\mathsf{r}}$$
$$\rightsquigarrow^{\mathsf{r},\mathsf{t}}_{(\mathsf{x}).\langle \mathsf{x}=\mathtt{CAS}(\mathtt{this.locked},0,1)\rangle} (\mathsf{x} = \mathsf{null} * \boxed{\mathtt{this.locked} \mapsto 0 * \mathsf{R} * [\textsc{Unlock}]_1^{\mathsf{r}}}^{\mathsf{r},\mathsf{t},\mathtt{this}}_{\mathsf{I}(\mathsf{R})})$$

The first three proof obligations follow directly from rule AUAct, using the Lock, τ_1 and τ_2 action, respectively. The last obligation follows by rule AUFalse1, as a compare-and-swap from 0 to 1 cannot update locked from 1 to 0. In particular, by rule ACas it follows that

$$(\mathsf{x}).\{\mathtt{this.locked} \mapsto 1\}\langle \mathsf{x} = \mathtt{CAS}(\mathtt{this}.\mathrm{locked}, 0, 1)\rangle\{\mathtt{this.locked} \mapsto 1\}.$$

□

4 Model

In this section we define a model for our proof system and prove some meta-theoretic results about the model. The concrete interpretation of the logic is given in Section 5. The model and its meta-theory is defined using a standard classical meta logic.

The presentation of the model is strongly inspired by the Views framework [3] presentation. In the Views framework, the operational semantics of the underlying programming language operates on concrete machine states, while assertions are predicates on instrumented (abstract) machine states. The soundness of the logic ensures that any step at the concrete level has a corresponding step at the instrumented level. Our model can be seen as a concrete instance of the Views framework, instrumented with actions and protocols for modeling the CAP-part of the logic and a phantom heap for modeling phantom fields. In addition, everything is step-indexed to model the later modality, guarded recursion, and the embeddings between the logics. The Views framework features shared variable concurrency. Since mini C$^\sharp$ features fork-concurrency and threads with local stacks and a shared heap, we have generalized the Views framework with threads and thread-local state. To model guardedness, we model types as sets equipped with a step-indexed equivalence relation. Furthermore, to ensure modular reasoning about guardedness, we build non-expansiveness into the interpretation. The interpretation is thus given in the category of step-indexed equivalence relations and non-expansive functions, instead of the category of sets. We begin by defining the category of step-indexed equivalence relations and non-expansive functions. This category is equivalent to the category of bisected ultrametric spaces. Ultrametric spaces have previously been used to model guarded recursion [5, 2, 1]. We prefer this equivalent but more concrete presentation in terms of step-indexed equivalences.

Category of step-indexed equivalence relations ASets ∈ Cats

A step-indexed equivalence is a pair, $(X, (=_i)_{i\in\mathbb{N}})$, consisting of a set X and a set of equivalence relations, $=_i \subseteq X \times X$, indexed by natural numbers $i \in \mathbb{N}$. We require that the step-indexed equivalence relations become coarser at lower step-indicies:

$$\forall i \in \mathbb{N}.\ \forall x, y \in X.\ x =_{i+1} y \Rightarrow x =_i y$$

and that equality at every step-index corresponds to identity:

$$\forall x, y \in X.\ (\forall i \in \mathbb{N}.\ x =_i y) \Rightarrow x = y$$

Given two step-indexed equivalence relations

$$\mathcal{X} = (X, =_i^X) \in \text{ASets} \qquad\qquad \mathcal{Y} = (Y, =_i^Y) \in \text{ASets}$$

let $\mathcal{X} \to_{ne} \mathcal{Y}$ denote the set of non-expansive functions:

$$\mathcal{X} \to_{ne} \mathcal{Y} \stackrel{\text{def}}{=} \{f : X \to Y \mid \forall i \in \mathbb{N}.\ \forall x, y \in X.\ x =_i^X y \Rightarrow f(x) =_i^Y f(y)\}$$

Define ASets as the category of step-indexed equivalences and non-expansive functions.

Embedding

$$\boxed{\begin{array}{l}\Delta : \text{Sets} \to \text{ASets} \in \text{Cats} \\ U : \text{ASets} \to \text{Sets} \in \text{Cats}\end{array}}$$

The category of sets embeds into ASets via Δ. Furthermore, Δ is a left-adjoint to the forgetful functor, U.

$$\Delta(X) \stackrel{\text{def}}{=} (X, (\{(x,x) \mid x \in X\})_{i \in \mathbb{N}}) \qquad U(X, (=_i)_{i \in \mathbb{N}}) \stackrel{\text{def}}{=} X$$
$$\Delta(f) \stackrel{\text{def}}{=} f \qquad U(f) \stackrel{\text{def}}{=} f$$

We will always write out Δ explicitly, when embedding sets into ASets; however, we will leave U implicit.

Cartesian closed structure

$$\boxed{1, \mathcal{X} \times \mathcal{Y}, \mathcal{X} \to \mathcal{Y} \in \text{ASets}}$$

$$1 \stackrel{\text{def}}{=} (\{*\}, =_i^1)$$
$$\mathcal{X} \times \mathcal{Y} \stackrel{\text{def}}{=} (X \times Y, =_i^{X \times Y})$$
$$\mathcal{X} \to \mathcal{Y} \stackrel{\text{def}}{=} (\mathcal{X} \to_{ne} \mathcal{Y}, =_i^{X \to Y})$$

where

$$\begin{array}{rcl} * =_i^1 * & \text{iff} & \top \\ (x_1, y_1) =_i^{X \times Y} (x_2, y_2) & \text{iff} & x_1 =_i^X x_2 \wedge y_1 =_i^Y y_2 \\ f =_i^{X \to Y} g & \text{iff} & \forall x \in X.\ f(x) =_i^Y g(x) \end{array}$$

for $\mathcal{X} = (X, =_i^X) \in \text{ASets}$ and $\mathcal{Y} = (Y, =_i^Y) \in \text{ASets}$.

Semantic domains

The basic structure of the model is given by the following semantic domains. We assume countably infinite and disjoint sets of region identifiers (*RId*) action identifiers (*AId*) and region types (*RId*).

Instrumented states consist of three components: a local state, a shared state and an action model. The local state specifies the current local resources. Local resources consists of concrete fields, phantom fields and action permissions. The shared state is partitioned into regions. Each region consists of a local state, a region type and a protocol argument (of type *Val*). The local state component of a shared region specifies the local resources currently owned by that region. The action model associates parameterized protocols with region types. Parameterized protocols are represented as functions from action argument to actions. An action argument consists of a protocol argument, a region identifier and an action identifier. Lastly, actions are modeled as certain step-indexed relations on shared states. This avoids the circularity that would arise from defining actions as relations on shared states and action models, but it also means the model

lacks support for general higher-order protocols. The model does however support state-independent protocols through the region type indirection. In particular, as shared states include region types and protocol arguments, actions *can* constrain the region types and protocol arguments of other regions.

$r \in \mathit{RId}$	region identifier
$a \in \mathit{AId}$	action identifier
$t \in \mathit{RType}$	region type
$\mathit{PHeap} \stackrel{\text{def}}{=} \mathit{OId} \times \mathit{FName} \stackrel{\text{fin}}{\rightharpoonup} \mathit{Val}$	phantom heap
$l \in \mathit{LState} \stackrel{\text{def}}{=} \mathit{Heap} \times \mathit{PHeap} \times \mathit{Cap}$	local state
$s \in \mathit{SState} \stackrel{\text{def}}{=} \mathit{RId} \stackrel{\text{fin}}{\rightharpoonup} (\mathit{LState} \times \mathit{RType} \times \mathit{Val})$	shared state
$t \in \mathit{Token} \stackrel{\text{def}}{=} \mathit{RId} \times \mathit{AId}$	token
$c \in \mathit{Cap} \stackrel{\text{def}}{=} \{f \in \mathit{Token} \to [0,1] \mid \mathit{dom}_r(f) \text{ finite}\}$	capability map
$a \in \mathit{Action} \stackrel{\text{def}}{=} \{R \in \mathcal{P}(\mathbb{N} \times \mathit{SState} \times \mathit{SState}) \mid \mathit{good}(R)\}$	action
$\mathit{AArg} \stackrel{\text{def}}{=} \mathit{Val} \times \mathit{RId} \times \mathit{AId}$	action argument
$\varsigma \in \mathit{AMod} \stackrel{\text{def}}{=} \mathit{RType} \rightharpoonup (\mathit{AArg} \to \mathit{Act})$	action model
$m \in \mathcal{M} \stackrel{\text{def}}{=} \mathit{LState} \times \mathit{SState} \times \mathit{AMod}$	instrumented states

We require that actions are good, meaning downwards-closed in the step-index, closed under allocation of new regions and closed under arbitrary changes to additional regions.

$$\begin{aligned}\mathit{good}(R) \stackrel{\text{def}}{=}\ & \forall (i, s_1, s_2) \in R.\ \forall j \le i.\ \forall r \in \mathit{RId} \setminus \mathit{dom}(s_2).\\ & \forall t \in \mathit{RType}.\ \forall a \in \mathit{Val}.\ \forall l, l' \in \mathit{LState}.\\ & \quad s_1 \le s_2 \wedge (j, s_1, s_2) \in R \wedge (j, s_1, s_2[r \mapsto (l', t, a)]) \in R \wedge\\ & \quad (j, s_1[r \mapsto (l, t, a)], s_2[r \mapsto (l', t, a)]) \in R\end{aligned}$$

$$s_1 \le s_2 \stackrel{\text{def}}{=} \mathit{dom}(s_1) \subseteq \mathit{dom}(s_2) \wedge (\forall r \in \mathit{dom}(s_1).\ s_1(r).t = s_2(r).t \wedge s_1(r).a = s_2(r).a)$$

$$\mathit{dom}_r(f) \stackrel{\text{def}}{=} \{r \in \mathit{RId} \mid \exists \alpha \in \mathit{AId}.\ f(r, \alpha) > 0\}$$

We use $m.l$, $m.s$ and $m.a$ as shorthand to refer to the local state, shared state and action model component of an instrumented state m, respectively. We use $s(r).l$, $s(r).t$ and $s(r).a$ to refer to the local state, region type and protocol parameter component of a shared region r from the shared state s. We use $l.h$, $l.p$, and $l.c$ to refer to the heap, the phantom heap and the capability map of a local state $l \in \mathit{LState}$.

Values $\boxed{\mathit{Val} \in \text{Sets}}$

Let *Val* denote the least set such that

$$\mathit{Val} \cong \mathit{CVal} \uplus \mathit{Strings} \uplus (\mathit{Val} \times \mathit{Val})$$

Region types $\boxed{RType \in \text{Sets}}$

Region types are simply modeled as strings, with the prefix-ordering. Values thus include region types.

$$RType = Strings$$

We use t^* as notation for $\{s \in RType \mid t \text{ is a prefix of } s\}$ when $t \in RType$.

Composition operators

$$\boxed{\begin{array}{c} \bullet_{LState} : LState \times LState \rightharpoonup LState \\ \bullet_{\mathcal{M}} : \mathcal{M} \times \mathcal{M} \rightharpoonup \mathcal{M} \end{array}}$$

Instrumented and local states compose using the partial $\bullet_{LState}$ and $\bullet_{\mathcal{M}}$ operators. Undefinedness indicates the two states are incompatible and do not compose. For instance, two local states that assert ownership of the same field are incompatible and do not compose.

$$X \bullet_{\uplus} Y \stackrel{\text{def}}{=} \begin{cases} X \cup Y & \text{if } X \cap Y = \emptyset \\ \text{undef} & \text{otherwise} \end{cases}$$

$$x \bullet_{=} y \stackrel{\text{def}}{=} \begin{cases} x & \text{if } x = y \\ \text{undef} & \text{otherwise} \end{cases}$$

$$f \bullet_{+} g \stackrel{\text{def}}{=} \begin{cases} \lambda x.\ f(x) + g(x) & \text{if } \forall x.\ f(x) + g(x) \leq 1 \\ \text{undef} & \text{otherwise} \end{cases}$$

$$(oh_1, th_1, ch_1) \bullet_{Heap} (oh_2, th_2, ch_2) \stackrel{\text{def}}{=} (oh_1 \bullet_{\uplus} oh_2, th_1 \bullet_{=} th_2, ch_1 \bullet_{=} ch_2)$$

$$(h_1, ph_1, c_1) \bullet_{LState} (h_2, ph_2, c_2) \stackrel{\text{def}}{=} (h_1 \bullet_{Heap} h_2, ph_1 \bullet_{\uplus} ph_2, c_1 \bullet_{+} c_2)$$

$$(l_1, s_1, a_1) \bullet_{\mathcal{M}} (l_2, s_2, a_2) \stackrel{\text{def}}{=} (l_1 \bullet_{LState} l_2, s_1 \bullet_{=} s_2, a_1 \bullet_{=} a_2)$$

Extension ordering $\boxed{\leq_{\mathcal{M}} : \mathcal{M} \times \mathcal{M} \to 2}$

$$m_1 \leq_{\mathcal{M}} m_2 \quad \text{iff} \quad \exists m \in \mathcal{M}.\ m_2 = m_1 \bullet m$$

The extension ordering induced by $\bullet$ defines a partial order.

View $\boxed{\mathcal{V} \in \text{Sets}}$

We interpret propositions in the assertion logic as certain step-indexed predicates on instrumented states, called Views. In particular, we require that views are downwards-closed in the step-index, closed under allocation of new regions and protocols, and

upwards-closed in the local state and the local states of shared regions.

$$
\begin{aligned}
p, q \in \mathcal{V} \stackrel{\text{def}}{=} \{ & p \in \mathcal{P}(\mathbb{N} \times \mathcal{M}) \mid \\
& (\forall (i, m) \in p.\ \forall j \leq i.\ \forall n \geq_{\mathcal{M}} m.\ (j, n) \in p) \wedge \\
& (\forall (i, m_1) \in p.\ \forall m_2 \in \mathcal{M}.\ m_1 =_i^{\mathcal{M}} m_2 \Rightarrow (i, m_2) \in p) \wedge \\
& (\forall (i, (l, s, \varsigma)) \in p.\ \forall r \in (\mathit{RId} \setminus \mathit{dom}(s)).\ \forall t \in \mathit{RType}.\ \forall l_r \in \mathit{LState}.\ \forall a \in \mathit{Val}. \\
& \quad (i, (l, s[r \mapsto (l_r, t, a)], \varsigma)) \in p) \wedge \\
& (\forall (i, (l, s, \varsigma)) \in p.\ \forall r \in (\mathit{RType} \setminus \mathit{dom}(\varsigma)).\ \forall I \in \mathit{AArg} \to \mathit{Act}. \\
& \quad (i, (l, s, \varsigma[r \mapsto I])) \in p)\}
\end{aligned}
$$

where $=_i^{\mathcal{M}} \subseteq \mathcal{M} \times \mathcal{M}$ is given by

$$
\begin{aligned}
(l_1, s_1, \varsigma_1) =_i^{\mathcal{M}} (l_2, s_2, \varsigma_2) \quad \text{iff} \quad & l_1 = l_2 \wedge s_1 = s_2 \wedge \mathit{dom}(\varsigma_1) = \mathit{dom}(\varsigma_2) \wedge \\
& (\forall r \in \mathit{dom}(\varsigma_1).\ \forall \alpha \in \mathit{AArg}.\ \varsigma_1(r)(\alpha)|_i = \varsigma_2(r)(\alpha)|_i)
\end{aligned}
$$

and $R|_i$ is given by

$$R|_i = \{(j, s_1, s_2) \in \mathbb{N} \times \mathit{SState} \times \mathit{SState} \mid j \leq i \wedge (j, s_1, s_2) \in R\}$$

Assertions $\boxed{\mathit{Prop} \in \text{ASets}}$

We consider two semantic assertion propositions i-equivalent if they agree on all instrumented states for step-indexes strictly smaller than i.

$$\mathit{Prop} \stackrel{\text{def}}{=} (\mathcal{V}, (=_i^{\mathcal{V}})_{i \in \mathbb{N}})$$

where $=_i^{\mathcal{V}} \subseteq \mathcal{V} \times \mathcal{V}$ is given by

$$p =_i^{\mathcal{V}} q \quad \text{iff} \quad \forall j < i.\ \forall m \in \mathcal{M}.\ (j, m) \in p \Leftrightarrow (j, m) \in q$$

Specification $\boxed{\mathit{Spec} \in \text{ASets}}$

Likewise, we consider two semantic specification propositions i-equivalent if they agree at step-indexes strictly smaller than i.

$$\mathit{Spec} \stackrel{\text{def}}{=} (\mathsf{P}^{\downarrow}(\mathbb{N}), (=_i^{\mathsf{P}^{\downarrow}(\mathbb{N})})_{i \in \mathbb{N}})$$

where $=_i^{\mathcal{P}^{\downarrow}(\mathbb{N})} \subseteq \mathcal{P}^{\downarrow}(\mathbb{N}) \times \mathcal{P}^{\downarrow}(\mathbb{N})$ is given by

$$p =_i^{\mathcal{P}^{\downarrow}(\mathbb{N})} q \quad \text{iff} \quad \forall j < i.\ j \in p \Leftrightarrow j \in q$$

Region collapse $\boxed{\ulcorner - \urcorner : \mathit{LState} \times \mathit{SState} \rightharpoonup \mathit{LState}}$

$$\ulcorner (l, s) \urcorner \stackrel{\text{def}}{=} l \bullet_{\mathit{LState}} \Pi_{r \in \mathit{dom}(s)} \pi_1(s(r))$$

Action model extension $\boxed{(-) \leq (=) : \mathcal{P}(AMod \times AMod)}$

$$\varsigma_1 \leq \varsigma_2 \stackrel{\text{def}}{=} dom(\varsigma_1) \subseteq dom(\varsigma_2) \wedge \forall r \in dom(\varsigma_1).\ \varsigma_1(r) = \varsigma_2(r)$$

Shared state restriction $\boxed{(-)|_{(=)} : SState \times \mathcal{P}(RType) \to SState}$

$$s|_A(r) \stackrel{\text{def}}{=} \begin{cases} s(r) & \text{if } r \in dom(s) \text{ and } s(r).t \in A \\ \text{undef} & \text{otherwise} \end{cases}$$

Interference relation $\boxed{\hat{R}^{(-)}_{(=)}, R^{(-)}_{(=)} : \mathcal{P}(RType) \times \mathbb{N} \to \mathcal{P}(\mathcal{M} \times \mathcal{M})}$

The interference relation R_i^A describes possible interference from the environment. It is defined as the reflexive, transitive closure of the single-action interference relation, $\hat{R}_i^A$, that describes possible environment interference using at most one action on each region. This thus forces a common granularity on synchronized actions. In addition to the step-index i, the single-action interference relation is also indexed by a set of region types A. This is the types of regions that are allowed to change, and of regions that actions justifying those changes are allowed to depend upon.

$$R_i^A \stackrel{\text{def}}{=} (\hat{R}_i^A)^*$$

where $(-)^*$ denotes the reflexive, transitive closure operator on binary relations, and

$$\begin{array}{l} (l_1, s_1, \varsigma_1)\ \hat{R}_i^A\ (l_2, s_2, \varsigma_2) \quad \text{iff} \\ \quad l_1 = l_2 \wedge s_1 \leq s_2 \wedge \varsigma_1 \leq \varsigma_2 \wedge \\ \quad\quad \exists c \in Cap.\ c = \pi_c(\ulcorner (l_1, s_1) \urcorner). \\ \quad\quad\quad (\forall r \in dom(s_1).\ s_1(r) = s_2(r) \vee \\ \quad\quad\quad\quad (\exists \alpha \in AId.\ s_1(r).t \in A \wedge c(r, \alpha) < 1 \\ \quad\quad\quad\quad\quad \wedge\ (i, s_1|_A, s_2|_A) \in \varsigma_1(s_1(r).t)(s_1(r).a, r, \alpha))) \end{array}$$

Stability $\boxed{stable : Prop \to Spec \in \text{ASets}}$

An assertion is stable at step-index i if it is closed under R_i^{RType}.

$$stable(p) \stackrel{\text{def}}{=} \{i \in \mathbb{N} \mid \forall j \leq i.\ \forall (m_1, m_2) \in R_j^{RType}.\ (j, m_1) \in p \Rightarrow (j, m_2) \in p\}$$

We also use *stable* as notation for the point-wise lifting of *stable* to predicates. Thus, $stable(p)$ is notation for $\{i \in \mathbb{N} \mid \forall x \in X.\ i \in stable(p(x))\}$ when $p \in X \to \mathcal{V}$.

For *stable* to be a well-defined morphism in ASets, we have to prove that *stable* is non-expansive. To illustrate, we write out the proof of non-expansiveness of *stable*; however, in general we will omit trivial non-expansiveness proofs.

Lemma 4. *stable is non-expansive:*

$$\forall p, q \in \mathcal{V}.\ \forall i \in \mathbb{N}.\ p =_i^{\mathcal{V}} q \Rightarrow \mathit{stable}(p) =_i^{\mathcal{P}^{\downarrow}(\mathbb{N})} \mathit{stable}(q)$$

Proof.

- let $j < i$ and assume $j \in \mathit{stable}(p)$
- let $k \leq j$, $(m_1, m_2) \in R_k^{\mathit{RType}}$, and $(k, m_1) \in q$
- then $(k, m_1) \in p$ and thus, as $j \in \mathit{stable}(p)$, $(k, m_2) \in p$
- thus, $(k, m_2) \in q$

□

Erasure $\boxed{\lfloor - \rfloor : \mathcal{M} \rightharpoonup \mathit{Heap}}$

The erasure function, $\lfloor - \rfloor$, erases the instrumentation from instrumented states, yielding a concrete state. Since CAP partitions the state into local and shared states, the erasure first collapses the local and shared states. The erasure is then the heap component of the collapsed local states. The erasure function further requires that the phantom heap does not contain any phantom fields of objects that have not been allocated yet. This allows us to allocate new phantom fields in constructors.

$$\lfloor (l, s, \varsigma) \rfloor \stackrel{\text{def}}{=} \begin{cases} h & \text{if } (h, ph, c) = \ulcorner (l, s) \urcorner \text{ and } \pi_1(\mathit{dom}(ph)) \subseteq \mathit{objs}(h) \\ \text{undef} & \text{otherwise} \end{cases}$$

where $\mathit{objs}(oh, th, ch) = \mathit{dom}(th)$.

Step-indexed erasure $\boxed{\lfloor - \rfloor_{(=)} : \mathcal{V} \times \mathbb{N} \to \mathcal{P}(\mathit{Heap})}$

$$\lfloor p \rfloor_i \stackrel{\text{def}}{=} \lfloor \{m \in \mathcal{M} \mid (i, m) \in p\} \rfloor$$

View-shift $\boxed{\sqsubseteq : \mathit{Prop} \times \mathit{Prop} \to \mathit{Spec} \in \text{ASets}}$

Intuitively, a view-shift from p to q expresses that there exists a step at the instrumented level from p to q, corresponding to a no-op at the concrete level.

$$p \sqsubseteq q \stackrel{\text{def}}{=} \{i \in \mathbb{N} \mid \forall r \in \mathcal{V}.\ \forall j \in \mathbb{N}.\ 0 \leq j \leq i \wedge j \in \mathit{stable}(r) \Rightarrow \lfloor p * r \rfloor_j \subseteq \lfloor q * r \rfloor_j\}$$

We also use $\sqsubseteq$ as notation for the point-wise lifting of $\sqsubseteq$ to predicates. Thus, $p \sqsubseteq q$ is notation for $\{i \in \mathbb{N} \mid \forall x \in X.\ i \in (p(x) \sqsubseteq q(x))\}$ when $p, q \in X \to \mathcal{V}$.

Atomic action $\boxed{sat : \Delta(Act) \times Prop \times Prop \to Spec \in \text{ASets}}$

Likewise, an atomic action a from p to q expresses that there exists a step at the instrumented level from p to q, corresponding to the atomic action a at the concrete level.

$$a\ sat\ \{p\}\{q\} \stackrel{\text{def}}{=} \{i \in \mathbb{N} \mid \forall r \in \mathcal{V}.\ \forall j \in \mathbb{N}.\ 1 \leq j \leq i \wedge j \in stable(r) \Rightarrow [\![a]\!](\lfloor p * r \rfloor_j) \subseteq \lfloor q * r \rfloor_{j-1}\}$$

Thread safety $\boxed{safe_i : Thread \times \mathcal{V} \times (Stack \to \mathcal{V}) \to 2}$

Informally, safety is supposed to establish a simulation between the concrete level and the instrumented level, such that every step at the concrete level has a corresponding step at the instrumented level.

Safety is expressed in terms of a single thread, as possible interference from the environment is implicitly given by the pre-condition and post-condition through the interference relation. Formally, this is captured by Theorem 1, which relates the execution of an entire thread pool, to the safety of the individual threads.

$$\begin{aligned}
safe_0(x, p, q) &\stackrel{\text{def}}{=} \top \\
safe_{i+1}(x, p, q) &\stackrel{\text{def}}{=} (irr(x) \wedge i + 1 \in (p \sqsubseteq q(x.l))) \vee \\
&\quad (\forall T \in TPool.\ \forall a \in Act.\ \forall y \in Thread.\ x \xrightarrow{a} \{y\} \uplus T \wedge x.t = y.t \ \Rightarrow \\
&\qquad \exists p' \in \{y\} \uplus T \to \mathcal{V}. \\
&\qquad\quad (\forall z \in \{y\} \uplus T.\ i + 1 \in stable(p'(z))) \wedge \\
&\qquad\quad i + 1 \in (a\ sat\ \{p\}\{p'(y) * \circledast_{z \in T} p'(z)\}) \wedge \\
&\qquad\quad safe_i(y, p'(y), q) \wedge \\
&\qquad\quad \forall z \in T.\ safe_i(z, p'(z), \lambda l'.\ \top))
\end{aligned}$$

If a thread x is irreducible (i.e., x has terminate), then x is safe relative to p and q if there exists a step at the instrumented level from p to $q(x.l)$, corresponding to a no-op at the concrete level.

If a thread x can take a single step to y ($x.t = y.t$ ensures x and y have the same thread id) with action a by spawning threads T, then x is safe relative to p and q if:

- there exists stable predicates $p'(z)$ describing the intermediate state (at the instrumented level) of each thread $z \in \{y\} \uplus T$
- such that it is possible to take a step at the instrumented level from p to $\circledast_{z \in \{y\} \uplus T} p'(z)$ (thus splitting the combined intermediate resources between each thread), corresponding to the atomic action a at the concrete level
- and y is safe relative to $p'(y)$ and q
- and all spawned threads $z \in T$ are safe relative to $p'(z)$ and any post-condition

Since the post-condition q only describes the the terminal state of the initial thread x, only y (the thread x after one step of execution) is required to be safe relative to q.

Thread safety $\boxed{\mathit{safe} : \Delta(\mathit{Thread}) \times \mathit{Prop} \times (\Delta(\mathit{Stack}) \to \mathit{Prop}) \to \mathit{Spec} \in \mathrm{ASets}}$

$$\mathit{safe}(x, p, q) \stackrel{\text{def}}{=} \{i \in \mathbb{N} \mid \mathit{safe}_i(x, p, q)\}$$

Lemma 5. *safe is non-expansive.*

$$\forall i \in \mathbb{N}.\ \forall x \in \mathit{Thread}.\ \forall p_1, p_2 \in \mathit{Prop}.\ \forall q_1, q_2 \in \mathit{Stack} \to \mathit{Prop}.$$
$$p_1 =_i^{\mathcal{V}} p_2 \wedge (\forall l \in \mathit{Stack}.\ q_1(l) =_i^{\mathcal{V}} q_2(l)) \Rightarrow \mathit{safe}(x, p_1, q_1) =_i^{\mathcal{P}^{\downarrow}(\mathbb{N})} \mathit{safe}(x, p_2, q_2)$$

Proof. Follows by induction on i from the non-expansiveness of $\sqsubseteq$ and sat. □

Statement safety $\boxed{\mathit{safe} : \Delta(\mathit{seq}\ \mathit{Stm}) \times (\Delta(\mathit{Stack}) \to \mathit{Prop})^2 \to \mathit{Spec} \in \mathrm{ASets}}$

$$\mathit{safe}(s, p, q) \stackrel{\text{def}}{=} \{i \in \mathbb{N} \mid \forall t \in \mathit{TId}.\ \forall l \in \mathit{Stack}.\ \mathit{safe}_i((t, l, \mathit{stm}(s)), p(l), q)\}$$

4.1 Views meta-theory

In this section we develop some of the standard Views meta-theory for our step-indexed multithreaded safety predicate, including the rule of consequence (Lemma 14), the frame rule (Lemma 15), and sequential composition (Lemma 17). The main result is Theorem 1, which relates the execution of a thread pool with the safety of the individual threads.

Separation logic connectives
$$\boxed{\begin{array}{l}\mathit{emp} : \mathit{Prop} \\ \quad * : \mathit{Prop} \times \mathit{Prop} \to \mathit{Prop} \in \mathrm{ASets}\end{array}}$$

$$\mathit{emp} \stackrel{\text{def}}{=} \mathbb{N} \times \mathcal{M}$$
$$p * q \stackrel{\text{def}}{=} \{(i, m) \in \mathbb{N} \times \mathcal{M} \mid \exists m_1, m_2 \in \mathcal{M}.\ m = m_1 \bullet_{\mathcal{M}} m_2 \wedge (i, m_1) \in p \wedge (i, m_2) \in q\}$$

Image of an interference relation $\boxed{(-)(=) : \mathcal{P}(\mathcal{V} \times \mathcal{V}) \times \mathcal{V} \to \mathcal{V}}$

$$R(p) \stackrel{\text{def}}{=} \{(i, m') \in \mathbb{N} \times \mathcal{M} \mid (i, m) \in p \wedge m\ R\ m'\}$$

Lemma 6.

$$\forall i \in \mathbb{N}.\ \forall A \in \mathcal{P}(\mathit{RType}).$$
$$(\forall p, q \in \mathcal{V}.\ R_i^A(p * q) \subseteq R_i^A(p) * R_i^A(q)) \wedge (R_i^A(\mathit{emp}) \subseteq \mathit{emp})$$

Lemma 7.

$$\forall i \in \mathbb{N}.\ \forall j \leq i.\ \forall A \in \mathcal{P}(RType).\ R_i^A \subseteq R_j^A$$

Lemma 8.

$$\forall i \in \mathbb{N}.\ \forall A_1, A_2 \in \mathcal{P}(RType).\ A_1 \subseteq A_2 \Rightarrow \hat{R}_i^{A_1} \subseteq \hat{R}_i^{A_2}$$

Lemma 9.

$$\forall i \in \mathbb{N}.\ \forall A \in \mathcal{P}(RType).\ \forall p, q \in \mathcal{V}.\ p \subseteq q \Rightarrow i \in (p \sqsubseteq^A q)$$

Lemma 10.

$$\forall p, q, r \in \mathcal{V}.\ (p \sqsubseteq q) \cap stable(r) \subseteq (p * r \sqsubseteq q * r)$$

Lemma 11.

$$\forall a \in Act.\ \forall p, q, r \in \mathcal{V}.\ (a\ sat\ \{p\}\{q\}) \cap stable(r) \subseteq (a\ sat\ \{p * r\}\{q * r\})$$

Lemma 12.

$$\begin{aligned}&\forall A \in \mathcal{P}(RType).\ \forall a \in Act.\ \forall p, q, r \in \mathcal{V}.\\ &\quad (a\ sat^A\ \{p\}\{q\}) \cap stable(r) \subseteq (a\ sat^A\ \{p * r\}\{q * r\})\end{aligned}$$

Lemma 13 (Basic $\sqsubseteq$-closure)**.**

$$\begin{aligned}&\forall i \in \mathbb{N}.\ \forall a \in Act.\ \forall p, p', q', q \in \mathcal{V}.\\ &\quad i + 1 \in (p \sqsubseteq p') \wedge i + 1 \in (a\ sat\ \{p'\}\{q'\}) \wedge i \in (q' \sqsubseteq q) \Rightarrow i + 1 \in (a\ sat\ \{p\}\{q\})\end{aligned}$$

Proof.

- let $r \in \mathcal{V}$ and $1 \leq j \leq i + 1$
- then

$$\lfloor p * r \rfloor_j \subseteq \lfloor p' * r \rfloor_j \qquad [\![a]\!](\lfloor p' * r \rfloor_j) \subseteq \lfloor q' * r \rfloor_{j-1} \qquad \lfloor q' * r \rfloor_{j-1} \subseteq \lfloor q * r \rfloor_{j-1}$$

- and thus

$$[\![a]\!](\lfloor p * r \rfloor_j) \subseteq \lfloor q * r \rfloor_{j-1}$$

□

Lemma 14 (Consequence)**.**

$$\begin{aligned}&\forall i \in \mathbb{N}.\ \forall x \in Thread.\ \forall p, p' \in \mathcal{V}.\ \forall q', q \in Stack \to \mathcal{V}.\\ &\quad i \in (p \sqsubseteq p') \wedge i \in safe(x, p', q') \wedge i \in (q' \sqsubseteq q) \Rightarrow i \in safe(x, p, q)\end{aligned}$$

Proof. By induction on i.

- Case $i = 0$: trivial.
- Case $i = j + 1$:
  - if $irr(x)$ then $j + 1 \in (p \sqsubseteq p') \cap (p' \sqsubseteq q'(x_l)) \cap (q'(x_l) \sqsubseteq q(x_l))$
  - otherwise, suppose $x \xrightarrow{a} \{y\} \uplus T$
  - by safety there thus exists

    $$p'' : \{y\} \uplus T \to \mathcal{V}$$

    such that

    $$\begin{array}{l} \forall z \in \{y\} \uplus T.\ j + 1 \in stable(p''(z)) \\ j + 1 \in (a\ sat\ \{p'\}\{p''(y) * (\circledast_{z \in T} p''(z))\}) \\ safe_j(y, p''(y), q') \\ \forall z \in T.\ safe_j(z, p''(z), \lambda\_.\ \top) \end{array}$$

  - hence, by Lemma 13,

    $$j + 1 \in (a\ sat\ \{p\}\{p''(y) * (\circledast_{x \in T} p''(x))\})$$

  - and by the induction hypothesis,

    $$safe_j(y, p''(y), q)$$

□

Lemma 15 (Frame)**.**

$$\begin{array}{l} \forall x \in Thread.\ \forall p, r \in \mathcal{V}.\ \forall q \in Stack \to \mathcal{V}. \\ \quad safe(x, p, q) \cap stable(r) \subseteq safe(x, p * r, \lambda l'.\ q(l') * r) \end{array}$$

Proof. By induction on the step-index.

- Case $i = 0$: trivial.
- Case $i = j + 1$:
  - if $irr(x)$ then $i \in (p \sqsubseteq q(x_l))$ and thus $i \in (p * r \sqsubseteq q(x_l) * r)$
  - otherwise, suppose $x \xrightarrow{a} \{y\} \uplus T$

– by safety there thus exists

$$p' : \{y\} \uplus T \to \mathcal{V}$$

such that

$$\begin{array}{l} \forall z \in \{y\} \uplus T.\ i \in \mathit{stable}(p''(z)) \\ i \in (a\ \mathit{sat}\ \{p\}\{p'(y) * (\circledast_{z \in T} p'(z))\}) \\ \mathit{safe}_j(y, p'(y), q) \\ \forall z \in T.\ \mathit{safe}_j(z, p'(z), \lambda\_.\ \top) \end{array}$$

– hence, by Lemma 11,

$$i \in (a\ \mathit{sat}\ \{p * r\}\{p'(y) * r * (\circledast_{z \in T} p'(z))\})$$

– and by the induction hypothesis,

$$\mathit{safe}_j(y, p'(y) * r, \lambda l'.\ q(l') * r)$$

□

Lemma 16.

$$\begin{array}{l} \forall t \in \mathit{TId}.\ \forall l, l' \in \mathit{LState}.\ \forall T \in \mathit{TPool}.\ \forall s_1, s_2, s_3 \in \mathit{TCStack}.\ \forall a \in \mathit{Act}. \\ \quad s_1 \neq \varepsilon \wedge (t, l, s_1; s_2) \xrightarrow{a} \{(t, l', s_3)\} \uplus T\ \Rightarrow \\ \quad\quad (\exists s_4 \in \mathit{TCStack}.\ (t, l, s_1) \xrightarrow{a} \{(t, l', s_4)\} \uplus T \wedge s_3 = s_4; s_2) \end{array}$$

Proof.

- Case SEQ:
  - by definition,

$$(l, \mathit{hd}(s_1)) \xrightarrow{a} (l', s_1'), \qquad s_3 = s_1'; \mathit{tail}(s_1; s_2), \qquad T = \emptyset$$

  - hence, by SEQ,

$$(t, l, s_1) \xrightarrow{a} \{(t, l', s_1'; \mathit{tail}(s_1))\}$$

  - take $s_4 = s_1; \mathit{tail}(s_1)$
- Case FORK:
  - by definition there exists an l_d and s_d such that

$$l' = l, \qquad s_3 = \mathit{tail}(s_1; s_2), \qquad T = \{(t', l_d, s_d)\}$$

– hence, by FORK,

$$(t, l, s_1) \xrightarrow{a} \{(t, l', tail(s_1))\} \uplus T$$

– take $s_4 = tail(s_1)$

□

Lemma 17 (Sequential composition)**.**

$$\forall i \in \mathbb{N}.\ \forall t \in TId.\ \forall l \in Stack.\ \forall s_1, s_2 \in Stm.\ \forall p \in \mathcal{V}.\ \forall q, r \in Stack \to \mathcal{V}.$$
$$i \in safe((t, l, s_1), p, q) \wedge i \in safe(s_2, q, r) \Rightarrow i \in safe((t, l, s_1; s_2), p, r)$$

Proof. By induction on i.

- Case $i = 0$: trivial.
- Case $i = j + 1$:
  - if $irr(s_1; s_2)$ then $irr(s_1)$ and $irr(s_2)$ and thus
    $$i \in (p \sqsubseteq q(l)) \cap (q(l) \sqsubseteq r(l))$$
  - otherwise, $(t, l, s_1; s_2) \xrightarrow{a} \{(t, l', s_3)\} \uplus T$
  - Case $s_1 = \varepsilon$:
    - from $s_1 = \varepsilon$ it follows that $irr(s_1)$ and thus $i \in (p \sqsubseteq q(l))$
    - hence, by Lemma 14,
      $$safe_i((t, l, s_2), p, r)$$
  - Case $s_1 \neq \varepsilon$:
    - by Lemma 16 there exists an s_4 such that,
      $$(t, l, s_1) \xrightarrow{a} \{(t, l', s_4)\} \uplus T, \qquad s_3 = s_4; s_2$$
    - let y denote (t, l', s_4)
    - by safety of s_1 there thus exists
      $$p' \in \{y\} \uplus T \to \mathcal{V}$$
      such that
      $$\forall z \in \{y\} \uplus T.\ i \in stable(p'(z))$$
      $$a\ sat_i\ \{p\}\ \{p'(y) * \circledast_{z \in T} p'(z)\}$$
      $$safe_j((t, l', s_4), p'(y), q)$$
      $$\forall z \in T.\ safe_j(z, p'(z), \lambda\_.\ \top)$$

∗ by downwards-closure it follows that

$$safe_j(s_2, q, r)$$

∗ and thus, by the induction hypothesis,

$$safe_j((t, l', s_4; s_2), p'(y), r)$$

□

Theorem 1 (Evaluation safety)**.** *For any $i, j \in \mathbb{N}$, $T \in TPool$,*

$$p : T \to \mathcal{V}, \qquad q : T \to Stack \to \mathcal{V}, \qquad h \in \lfloor \circledast_{x \in T} p(x) \rfloor_j$$

if

$$i < j, \qquad \forall x \in T.\ safe_j(x, p(x), q(x)), \qquad (T, h) \to_i (T', h'), \qquad irr(T')$$

$$j \in stable(p) \qquad\qquad j \in stable(q)$$

then

$$h' \in \lfloor \circledast_{x \in T} q(x)(l_x) \rfloor_{j-i}$$

where

$$l_x = y.l, \qquad \text{if } y \in T' \text{ and } x.t = y.t$$

Proof. By induction on i.

- Case $i = 0$:
  - since $i = 0$ it follows that $T' = T$ and $h' = h$
  - hence $irr(x)$ for every $x \in T = T'$
  - from safety it thus follows that $j \in (p(x) \sqsubseteq q(x)(x.l))$ for every $x \in T$
  - hence, by stability of p and q,

    $$j \in (\circledast_{x \in T} p(x) \sqsubseteq \circledast_{x \in T} q(x)(x.l))$$

    and thus

    $$h' = h \in \lfloor \circledast_{x \in T} p(x) \rfloor_j \subseteq \lfloor \circledast_{x \in T} q(x)(x.l) \rfloor_j = \lfloor \circledast_{x \in T} q(x)(l_x) \rfloor_j$$

- Case $i = k + 1$:
  - suppose

    $$(T, h) \to (T'', h'') \to_k (T', h')$$

– there thus exists an $x \in T$ and $y \in T''$ such that $x.t = y.t$,

$$x \xrightarrow{a} \{y\} \uplus T''', \qquad T'' = (T \setminus \{x\}) \cup (\{y\} \uplus T'''), \qquad h'' \in [\![a]\!](h)$$

– from the safety of x there thus exists

$$p' \in \{y\} \uplus T''' \to \mathcal{V}$$

such that

$$\begin{array}{l} \forall z \in \{y\} \uplus T'''.\ j \in \mathit{stable}(p'(z)) \\ a\ \mathit{sat}_j\ \left\{p(x)\right\} \left\{p'(y) * (\circledast_{z \in T'''} p'(z))\right\} \\ \mathit{safe}_{j-1}(y, p'(y), q(x)) \\ \forall z \in T'''.\ \mathit{safe}_{j-1}(z, p'(z), \lambda l'.\ \top) \end{array}$$

– let

$$\begin{array}{ll} p'' = [z \in (T \setminus \{x\}) \mapsto p(z), z \in \{y\} \uplus T''' \mapsto p'(z)] & : T'' \to \mathcal{V} \\ q'' = [y \mapsto q(x), z \in (T \setminus \{x\}) \mapsto q(z), z \in T''' \mapsto \lambda\_.\ \top] & : T'' \to \mathit{Stack} \to \mathcal{V} \end{array}$$

– then $\forall z \in T''.\ \mathit{safe}_{j-1}(z, p''(z), q''(z))$

– from Lemma 11 it follows that

$$a\ \mathit{sat}_j\ \left\{\ \circledast_{x \in T} p(x)\right\} \left\{p'(y) * (\circledast_{z \in T'''} p'(z)) * (\circledast_{z \in (T \setminus \{x\})} p(z))\right\}$$

and thus

$$h'' \in \lfloor p'(y) * (\circledast_{z \in T'''} p'(z)) * (\circledast_{z \in (T \setminus \{x\})} p(z)) \rfloor_{j-1} = \lfloor \circledast_{x \in T''} p''(x) \rfloor_{j-1}$$

– by the induction hypothesis it thus follows that

$$\begin{array}{l} h' \in \lfloor \circledast_{z \in T''} q''(z)(l_z) \rfloor_{j-1-k} \\ \quad = \lfloor q(x)(l_y) * (\circledast_{z \in (T \setminus \{x\})} q(z)(l_z)) * (\circledast_{z \in T'''} \top) \rfloor_{j-i} \\ \quad = \lfloor q(x)(l_y) * (\circledast_{z \in (T \setminus \{x\})} q(z)(l_z)) \rfloor_{j-i} \end{array}$$

where $l_z = y.l$ if $y \in T'$ and $z.t = y.t$.

– lastly, since $x.t = y.t$, $l_x = l_y$ and thus,

$$h' \in \lfloor (\circledast_{z \in T} q(z)(l_z)) \rfloor_{j-i}$$

□

4.2 Guarded recursion meta-theory

In this section we develop the meta-theory for modeling guarded recursion. We define a concrete fixed-point operator on predicates on views, and prove that it defines a fixed-point when applied to guarded definitions (Corollary 2). We also develop a small theory of guardedness, and prove that the fixed-point operator is non-expansive, even on non-guarded definitions (Lemma 19).

$$\boxed{(-)_{(=)} : \forall \mathcal{X} : \text{ASets}.\ ((\mathcal{X} \to_{ne} \mathit{Prop}) \to_{ne} (\mathcal{X} \to_{ne} \mathit{Prop})) \times \mathbb{N} \to (\mathcal{X} \to_{ne} \mathit{Prop}) \in \text{Sets}}$$

$$f_0 \stackrel{\text{def}}{=} \lambda x \in \mathcal{X}.\ \mathbb{N} \times \mathcal{M}$$
$$f_{i+1} \stackrel{\text{def}}{=} f(f_i)$$

Lemma 18.

$$\forall \mathcal{X} \in \mathit{ASets}.\ \forall f, g : (\mathcal{X} \to \mathit{Prop}) \to (\mathcal{X} \to \mathit{Prop}) \in \mathit{ASets}.$$
$$\forall i \in \mathbb{N}.\ f =_i g \Rightarrow \forall j \in \mathbb{N}.\ f_j =_i g_j$$

Proof. By induction on j.

- Case $j = 0$: trivial, as $f_0 = (\lambda x \in \mathcal{X}.\ \mathbb{N} \times \mathcal{M}) = g_0$.
- Case $j = k + 1$:
  - by the induction hypothesis, $f_k =_i g_k$
  - hence, by $f =_i g$, and non-expansiveness of g,

$$f_j = f(f_k) =_i g(f_k) =_i g(g_k) = g_j$$

□

Guarded recursion

$$\boxed{\mathit{fix} : \forall \mathcal{X} : \text{ASets}.\ ((\mathcal{X} \to \mathit{Prop}) \to (\mathcal{X} \to \mathit{Prop})) \to (\mathcal{X} \to \mathit{Prop}) \in \text{ASets}}$$

$$\mathit{fix}(f) \stackrel{\text{def}}{=} \lambda x \in \mathcal{X}.\ \bigcap_{i \in \mathbb{N}} [f_i(x)]_i$$

where $[-]_i : \mathit{Prop} \to \mathit{Prop} \in \text{ASets}$ is defined as,

$$[p]_i = \{(j, m) \in \mathbb{N} \times \mathcal{M} \mid (j, m) \in p \vee i \leq j\}$$

Lemma 19. *fix is non-expansive.*

$$\forall \mathcal{X} \in \mathit{ASets}.\ \forall f, g : (\mathcal{X} \to \mathit{Prop}) \to (\mathcal{X} \to \mathit{Prop}) \in \mathit{ASets}.$$
$$\forall i \in \mathbb{N}.\ f =_i g \Rightarrow \mathit{fix}(f) =_i \mathit{fix}(g)$$

Proof.

- assume $x \in \mathcal{X}$, $j \in \mathbb{N}$ and $m \in \mathcal{M}$ such that

$$j < i \qquad\qquad (j, m) \in \mathit{fix}(f)(x) = \bigcap_{k \in \mathbb{N}} [f_k(x)]_k$$

- hence, for every $k \in \mathbb{N}$

$$(j, m) \in [f_k(x)]_k \Leftrightarrow (j, m) \in f_k(x) \vee k \leq j$$

- hence, for every $k > j$, $(j, m) \in f_k(x)$
- furthermore, by Lemma 18, $f_k =_i g_k$
- hence, for every $k > j$,

$$(j, m) \in g_k(x)$$

  and thus $(j, m) \in \bigcap_{k \in \mathbb{N}} [g_k(x)]_k = \mathit{fix}(g)(x)$

□

Lemma 20. *$\mathit{fix}(f)$ is non-expansive.*

$$\forall \mathcal{X} \in \mathit{ASets}.\ \forall f : (\mathcal{X} \to \mathit{Prop}) \to (\mathcal{X} \to \mathit{Prop}) \in \mathit{ASets}.\ \forall x, y \in \mathcal{X}.\\ x =_i y \Rightarrow \mathit{fix}(f)(x) =_i \mathit{fix}(f)(y)$$

Proof.

- let $j \in \mathbb{N}$ and $m \in \mathcal{M}$ such that

$$j < i \qquad\qquad (j, m) \in \mathit{fix}(f)(x) = \bigcap_{k \in \mathbb{N}} [f_k(x)]_k$$

- hence, for every $k > j$, $(j, m) \in f_k(x)$
- furthermore, for every $k \in \mathbb{N}$, f_k is non-expansive and thus $f_k(x) =_i f_k(y)$
- hence, for every $k > j$, $(j, m) \in f_k(y)$
- and thus $(j, m) \in \bigcap_{k \in \mathbb{N}} [f_k(y)]_k = \mathit{fix}(f)(y)$

□

Guarded $\boxed{\mathit{guarded} : \forall \mathcal{X} : \mathrm{ASets}.((\mathcal{X} \to \mathit{Prop}) \to (\mathcal{X} \to \mathit{Prop})) \to \mathit{Spec} \in \mathrm{ASets}}$

$$\mathit{guarded}(f) \stackrel{\text{def}}{=} \{i \in \mathbb{N} \mid \forall j \leq i.\ \forall p, q \in \mathcal{X} \to \mathit{Prop}.\ p =_j q \Rightarrow f(p) =_{j+1} f(q)\}$$

Lemma 21. *guarded is non-expansive.*

$$\forall \mathcal{X} \in \mathit{ASets}.\ \forall f, g : (\mathcal{X} \to \mathit{Prop}) \to (\mathcal{X} \to \mathit{Prop}) \in \mathit{ASets}.\\ \forall i \in \mathbb{N}.\ f =_i g \Rightarrow \mathit{guarded}(f) =_i \mathit{guarded}(g)$$

Proof.

- assume $j < i$ such that $j \in \mathit{guarded}(f)$
- assume $p, q \in \mathcal{X} \to \mathit{Prop}$ such that $p =_k q$ for $k \leq j$
- since $j \in \mathit{guarded}(f)$, $f(p) =_{k+1} f(q)$
- since $k \leq j < i$, $k+1 \leq i$ and thus

$$g(p) =_{k+1} f(p) =_{k+1} f(q) =_{k+1} g(q)$$

- hence, $j \in \mathit{guarded}(g)$

□

Lemma 22.

$$\forall \mathcal{X} \in \mathit{ASets}.\ \forall f : (\mathcal{X} \to \mathit{Prop}) \to (\mathcal{X} \to \mathit{Prop}) \in \mathit{ASets}.$$
$$\forall i \in \mathbb{N}.\ i \in \mathit{guarded}(f) \Rightarrow f_i =_i f_{i+1}$$

Proof. By induction on $i \in \mathbb{N}$.

- Case $i = 0$: trivial, as there exists no $j < 0$
- Case $i = j + 1$:
  - by the induction hypothesis, $f_i =_i f_{i+1}$
  - hence, since $i \in \mathit{guarded}(f)$

$$f_{i+1} = f(f_i) =_{i+1} f(f_{i+1}) = f_{i+2}$$

□

Corollary 1.

$$\forall \mathcal{X} \in \mathit{ASets}.\ \forall f : (\mathcal{X} \to \mathit{Prop}) \to (\mathcal{X} \to \mathit{Prop}) \in \mathit{ASets}.$$
$$\forall i \in \mathbb{N}.\ i \in \mathit{guarded}(f) \Rightarrow \forall j \in \mathbb{N}.\ f_i =_i f_{i+j}$$

Proof. By induction on $j \in \mathbb{N}$.

- case $j = 0$: trivial, as $=_i$ is an equivalence relation
- case $j = k + 1$:
  - by the induction hypothesis, $f_i =_i f_{i+k}$
  - hence, as $i \in \mathit{guarded}(f)$,

$$f_i =_i f_{i+1} = f(f_i) =_{i+1} f(f_{i+k}) = f_{i+j}$$

  - and thus, by downwards-closure of $=$, $f_i =_i f_{i+j}$

□

Lemma 23.

$$\forall \mathcal{X} \in \mathit{ASets}.\ \forall f : (\mathcal{X} \to \mathit{Prop}) \to (\mathcal{X} \to \mathit{Prop}) \in \mathit{ASets}.$$
$$\forall i \in \mathbb{N}.\ i \in \mathit{guarded}(f) \Rightarrow f_i =_i \mathit{fix}(f)$$

Proof.

- suppose $x \in \mathcal{X}$, $j < i$, and $m \in \mathcal{M}$, then we need to show that

$$(j, m) \in f_i(x) \Leftrightarrow (j, m) \in \bigcap_{k \in \mathbb{N}} [f_k(x)]_k$$

- assume $(j, m) \in \bigcap_{k \in \mathbb{N}} [f_k(x)]_k$
  - then, in particular,

$$(j, m) \in [f_i(x)]_i = \{(j, m) \mid (j, m) \in f_i(x) \vee i \leq j\}$$

  - and thus $(j, m) \in f_i(x)$, as $j < i$
- assume $(j, m) \in f_i(x)$
  - clearly

$$\bigcap_{k \in \mathbb{N}} [f_k(x)]_k = \left(\bigcap_{k \leq j} [f_k(x)]_k \right) \cap \left(\bigcap_{k > j} [f_k(x)]_k \right)$$

  - and

$$(j, m) \in \bigcap_{k \leq j} [f_k(x)]_k$$

    by definition of $[-]_i$
  - to prove $(j, m) \in \bigcap_{k > j} [f_k(x)]_k$, assume $k \in \mathbb{N}$ such that $j < k$
  - case $i \leq k$:
    - then, by Corollary 1, $f_i =_i f_k$
    - and thus $(j, m) \in f_k(x)$ as $j < i$
  - case $k \leq i$:
    - then, by Corollary 1, $f_k =_k f_i$
    - and thus $(j, m) \in f_k(x)$ as $j < k$

□

Corollary 2.

$$\forall \mathcal{X} \in \mathit{ASets}.\ \forall f : (\mathcal{X} \to \mathit{Prop}) \to (\mathcal{X} \to \mathit{Prop}) \in \mathit{ASets}.$$
$$\forall i \in \mathbb{N}.\ i \in \mathit{guarded}(f) \Rightarrow \mathit{fix}(f) =_i f(\mathit{fix}(f))$$

Proof. By case analysis on $i \in \mathbb{N}$.

- Case $i = 0$: trivial, as there is no $j < 0$
- Case $i = j + 1$:
  - by Lemma 23

$$\mathit{fix}(f) =_{j+1} f_{j+1}, \qquad f_j =_j \mathit{fix}(f)$$

  - by guardedness of f we thus have that

$$\mathit{fix}(f) =_{j+1} f_{j+1} = f(f_j) =_{j+1} f(\mathit{fix}(f))$$

□

Later

$$\boxed{\begin{array}{l} \triangleright : \mathit{Prop} \to \mathit{Prop} \in \mathrm{ASets} \\ \triangleright : \mathit{Spec} \to \mathit{Spec} \in \mathrm{ASets} \end{array}}$$

$$\triangleright p \stackrel{\text{def}}{=} \{(i+1, m) \in \mathbb{N} \times \mathcal{M} \mid (i, m) \in p\} \cup (\{0\} \times \mathcal{M})$$
$$\triangleright s \stackrel{\text{def}}{=} \{i + 1 \in \mathbb{N} \mid i \in s\} \cup \{0\}$$

Lemma 24.

$$\forall i \in \mathbb{N}.\ i \in \mathit{stable}(p) \Rightarrow i + 1 \in \mathit{stable}(\triangleright p)$$

Proof.

- let $j \leq i + 1$, $(m_1, m_2) \in R_j^{\mathit{RType}}$ and $(j, m_1) \in \triangleright p$
- case $j = 0$:
  - then $(j, m_2) \in \triangleright p$ holds trivially
- case $j = k + 1$:
  - then $(k, m_1) \in p$
  - by Lemma 7, $(m_1, m_2) \in R_k^{\mathit{RType}}$
  - since $k \leq i$ it thus follows by stability of p that $(k, m_2) \in p$
  - and thus $(j, m_2) \in \triangleright p$

□

Lemma 25.

$$\forall p, q \in \mathit{Prop}.\ p =_i q \Rightarrow \triangleright p =_{i+1} \triangleright q$$

Lemma 26.

$$\forall \mathcal{X} \in \mathit{ASets}.\ \forall f, g, h \in (\mathcal{X} \to \mathit{Prop}) \to (\mathcal{X} \to \mathit{Prop}).\ \mathit{guarded}(g) \subseteq \mathit{guarded}(h \circ g \circ f)$$

Proof.

- let $p, q \in \mathcal{X} \to \mathit{Prop}$ such that $p =_i q$
- then by non-expansiveness of f, $f(p) =_i f(q)$
- hence, by guardedness of g, $g(f(p)) =_{i+1} g(f(q))$
- and thus, by non-expansiveness of h, $g(h(f(p))) =_{i+1} h(g(f(q)))$

□

Lemma 27.

$$\forall \mathcal{X} \in \mathit{ASets}.\ \mathit{guarded}(\lambda p \in \mathcal{X} \to \mathit{Prop}.\ \lambda x \in \mathcal{X}.\ \triangleright p(x))$$

Proof.

- let $j \leq i$ and $p, q \in \mathcal{X} \to \mathit{Prop}$ such that $p =_j q$
- let $x \in \mathcal{X}$
- then $p(x) =_j q(x)$ and thus, by Lemma 25, $\triangleright p(x) =_{j+1} \triangleright q(x)$

□

Lemma 28.

$$\forall \mathcal{X} \in \mathit{ASets}.\ \forall f : \mathit{Prop} \to \mathit{Prop} \in \mathit{ASets}.\ \mathit{guarded}(\lambda p \in \mathcal{X} \to \mathit{Prop}.\ \lambda x \in \mathcal{X}.\ \triangleright f(p(x)))$$

Proof.

- let $j \leq i$ and $p, q \in \mathcal{X} \to \mathit{Prop}$ such that $p =_j q$
- let $x \in \mathcal{X}$
- then $p(x) =_j q(x)$
- hence, by non-expansiveness of f, $f(p(x)) =_j f(q(x))$
- and thus, by Lemma 25, $\triangleright f(p(x)) =_{j+1} \triangleright f(q(x))$

□

4.3 Embedding meta-theory

In this section we develop the meta-theory related to the embedding of specifications into assertions. The main result is the soundness of the AsnI and AsnE rules from Section 2.3.2 (Lemma 37 and Corollary 3).

The main difficulty is proving the soundness of the AsnI rule (Lemma 37), which allows us to move an embedded specification from the pre-condition into the specification context. Very informally, if

$$\Gamma \mid \mathsf{S} \vdash \{\mathsf{P}\}\bar{\mathsf{s}}\{\mathsf{Q}\}$$

holds, then for every $i \in [\![\mathsf{S}]\!]$, $\bar{\mathsf{s}}$ is safe for i steps relative to $[\![\mathsf{P}]\!]$ and $[\![\mathsf{Q}]\!]$. Hence, if we run $\bar{\mathsf{s}}$ for one step, then for every $i \in [\![\mathsf{S}]\!]$ there exists a stable assertion p'_i describing the intermediate state, such that the continuation is safe for $i-1$ steps relative to p'_i and $[\![\mathsf{Q}]\!]$. To prove $\bar{\mathsf{s}}$ safe for i steps relative to $[\![\mathsf{P} * \mathsf{asn}(\mathsf{S})]\!]$ and $[\![\mathsf{Q}]\!]$ we need a *single* stable assertion to describe the intermediate state. The idea is to take this assertion to be the disjunction of all the p'_i assertions, only with each p'_i cut-off at step-index $i-1$, as the continuation is only safe for $i-1$ steps relative to p'_i. To formalize this we define a cutoff operator, written $\lceil p \rceil^i$, which is false at every step-index $j > i$. Next, we prove that given an $\mathbb{N}$ indexed family of pre-conditions p, such that for any $n \in \mathbb{N}$, s is safe for n steps relative to $p(n)$ and q, then s is safe for any number of steps relative to $\bigvee_n \lceil p(n) \rceil^n$ and q (see Lemma 36).

Specification embedding $\boxed{asn : Spec \to Prop}$

$$asn(X) \stackrel{\text{def}}{=} \{(i, m) \in \mathbb{N} \times \mathcal{M} \mid i \in X\}$$

Validity embedding $\boxed{valid : Prop \to Spec}$

$$valid(p) \stackrel{\text{def}}{=} \{i \in \mathbb{N} \mid \forall m \in \mathcal{M}.\ (i, m) \in p\}$$

View cutoff $\boxed{\lceil - \rceil^{=} : \mathcal{V} \times \mathbb{N} \to \mathcal{V}}$

$$\lceil p \rceil^i \stackrel{\text{def}}{=} \{(j, m) \in \mathbb{N} \times \mathcal{M} \mid (j, m) \in p \wedge j \leq i\}$$

Lemma 29.

$$\forall i \in \mathbb{N}.\ \forall p, q \in \mathcal{V}.\ i \in (p \sqsubseteq q) \Rightarrow \forall j \in \mathbb{N}.\ j \in (\lceil p \rceil^i \sqsubseteq q)$$

Proof.

- let $k \in \mathbb{N}$, $r \in \mathcal{V}$, $m \in \mathcal{M}$ and $h \in Heap$ such that

$$0 \leq k \leq j \qquad k \in stable(r) \qquad (k, m) \in \lceil p \rceil^i * r \qquad h \in \lfloor m \rfloor$$

- then $k \leq i$ and $(k, m) \in p * r$
- hence, there exists an $m' \in \mathcal{M}$ such that $(k, m') \in q * r$ and $h \in \lfloor m' \rfloor$

□

Lemma 30.

$$\forall i \in \mathbb{N}.\ \forall p, q \in \mathcal{V}.\ \lceil p * q \rceil^i = \lceil p \rceil^i * \lceil q \rceil^i$$

Lemma 31.

$$\forall i, j \in \mathbb{N}.\ \forall p \in \mathcal{V}.\ i \in \mathit{stable}(p) \Rightarrow j \in \mathit{stable}(\lceil p \rceil^i)$$

Proof.

- let $k \leq j$, $(m_1, m_2) \in R_k^{RType}$ and $(k, m_1) \in \lceil p \rceil^i$
- then $k \leq i$ and thus by i-stability of p, $(k, m_2) \in p$

□

Lemma 32.

$$\begin{aligned}&\forall i \in \mathbb{N}.\ \forall a \in \mathit{Act}.\ \forall p, q \in \mathcal{V}.\\ &\quad i \in (a\ \mathit{sat}\ \{p\}\{q\}) \Rightarrow \forall j \in \mathbb{N}.\ j \in (a\ \mathit{sat}\ \{\lceil p \rceil^i\}\{\lceil q \rceil^{i-1}\})\end{aligned}$$

Proof.

- suppose

$$\begin{array}{llll} k \in \mathbb{N}, & r \in \mathcal{V}, & 1 \leq k \leq j, & k \in \mathit{stable}(r) \\ m \in \mathcal{M}, & s \in \lfloor m \rfloor, & (k, m) \in \lceil p \rceil^i * r, & s' \in [\![a]\!](s) \end{array}$$

- then $k \leq i$ and $(k, m) \in p * r$ and thus $(k, m) \in p * \lceil r \rceil^k$
- by assumption there thus exists an $m' \in \mathcal{M}$ such that

$$(k-1, m') \in q * \lceil r \rceil^k, \qquad s' \in \lfloor m' \rfloor$$

- hence, $(k-1, m') \in \lceil q \rceil^{i-1} * r$, as $1 \leq k \leq i$

□

Lemma 33.

$$\forall i \in \mathbb{N}.\ \forall a \in \mathit{Act}.\ \forall p, q \in \mathcal{V}.\ i \in (a\ \mathit{sat}\ \{\lceil p \rceil^0\}\{q\})$$

Lemma 34.

$$\forall p_1, p_2, q_1, q_2 \in \mathcal{V}.\ (p_1 * q_1) \cup (p_2 * q_2) \subseteq (p_1 \cup p_2) * (q_1 \cup q_2)$$

Lemma 35.

$$\forall i \in \mathbb{N}.\ \forall a \in \mathit{Act}.\ \forall p_1, p_2, q_1, q_2 \in \mathcal{V}.$$
$$(a\ \mathit{sat}\ \{p_1\}\{q_1\}) \cap a\ \mathit{sat}\ \{p_2\}\{q_2\}) \subseteq (a\ \mathit{sat}\ \{p_1 \cup p_2\}\{q_1 \cup q_2\})$$

Proof.

- suppose

$$1 \leq j \leq i, \qquad r \in \mathcal{V}, \qquad j \in \mathit{stable}(r)$$
$$(j, m) \in (p_1 \cup p_2) * r, \qquad s \in \lfloor m \rfloor, \qquad s' \in [\![a]\!](s)$$

- by definition there exists $m_1, m_2 \in \mathcal{M}$ such that

$$m \in m_1 \bullet m_2, \qquad (j, m_1) \in p_1 \cup p_2, \qquad (j, m_2) \in r$$

- Case $(j, m_1) \in p_1$:
  - since $(j, m) \in p_1 * r$ there exists an $m' \in \mathcal{M}$ such that

$$(j, m') \in q_1 * r, \qquad s' \in \lfloor m' \rfloor$$

  - hence $(j, m') \in (q_1 \cup q_2) * r$
- Case $(j, m_1) \in p_2$: as above

□

Lemma 36.

$$\forall i \in \mathbb{N}.\ \forall x \in \mathit{Thread}.\ \forall p \in \mathbb{N} \to \mathcal{V}.\ \forall q \in \mathit{Stack} \to \mathcal{V}.$$
$$(\forall j \in \mathbb{N}.\ j \in \mathit{safe}(x, p_j, q)) \Rightarrow i \in \mathit{safe}(x, \bigcup_{j \in \mathbb{N}} \lceil p_j \rceil^j, q)$$

Proof. By induction on i. As the base case is trivial, assume $i > 0$.

- Case $\mathit{irr}(x)$:
  - by assumption

$$\forall i \in \mathbb{N}.\ i \in (p_i \sqsubseteq q(x.l))$$

  - hence, by Lemma 29

$$\forall i, j \in \mathbb{N}.\ i \in (\lceil p_j \rceil^j \sqsubseteq q(x.l))$$

– and thus

$$\forall i \in \mathbb{N}.\ i \in (\bigcup_j \lceil p_j \rceil^j \sqsubseteq q(x.l))$$

- Case $x \xrightarrow{a} \{y\} \uplus T$:
  - by assumption, for each $j \in \mathbb{N}_+$ there exists a

$$p'_j : \{y\} \uplus T \to \mathcal{V}$$

such that

$$\begin{array}{l} \forall z \in \{y\} \uplus T.\ j \in \mathit{stable}(p'_j(z)) \\ j \in (a\ \mathit{sat}\ \{p_j\}\{p'_j(y) * (\circledast_{z \in T} p'_j(z))\}) \\ \mathit{safe}_{j-1}(y, p'_j(y), q) \\ \forall z \in T.\ \mathit{safe}_{j-1}(z, p'_j(z), \lambda\_.\ \top) \end{array}$$

  - hence, Lemmas 32 and 30,

$$\forall k \in \mathbb{N}.\ \forall j \in \mathbb{N}_+.\ k \in (a\ \mathit{sat}\ \{\lceil p_j \rceil^j\}\{\lceil p'_j(y) \rceil^{j-1} * (\circledast_{z \in T} \lceil p'_j(z) \rceil^{j-1})\})$$

  - by Lemmas 34 and 35 we thus have that

$$\forall k \in \mathbb{N}.\ k \in (a\ \mathit{sat}\ \{\bigcup_{j \in \mathbb{N}_+} \lceil p_j \rceil^j\}\{\left(\bigcup_{j \in \mathbb{N}_+} \lceil p'_j(y) \rceil^{j-1}\right) * \left(\circledast_{z \in T} \bigcup_{j \in \mathbb{N}_+} \lceil j'_i(z) \rceil^{j-1}\right)\})$$

  - and thus, by Lemmas 33 and 35,

$$\forall k \in \mathbb{N}.\ k \in (a\ \mathit{sat}\ \{\bigcup_{j \in \mathbb{N}} \lceil p_j \rceil^j\}\{\left(\bigcup_{j \in \mathbb{N}_+} \lceil p'_j(y) \rceil^{j-1}\right) * \left(\circledast_{z \in T} \bigcup_{j \in \mathbb{N}_+} \lceil p'_j(z) \rceil^{j-1}\right)\})$$

  - and by the induction hypothesis

$$\begin{array}{l} \mathit{safe}_{i-1}(y, \bigcup_{j \in \mathbb{N}} \lceil p'_{j+1}(y) \rceil^j, q) \\ \forall z \in T.\ \mathit{safe}_{i-1}(z, \bigcup_{j \in \mathbb{N}} \lceil p'_{j+1}(z) \rceil^j, \lambda\_.\ \top) \end{array}$$

  - furthermore, by Lemma 31, it follows that,

$$\forall z \in \{y\} \uplus T.\ \forall i \in \mathbb{N}.\ i \in \mathit{stable}(\bigcup_{j \in \mathbb{N}} \lceil p'_{j+1}(z) \rceil^j)$$

– lastly

$$\forall z \in \{y\} \uplus T.\ \bigcup_{j \in \mathbb{N}} \lceil p'_{j+1}(z) \rceil^j = \bigcup_{j \in \mathbb{N}_+} \lceil p'_j(z) \rceil^{j-1}$$

□

Lemma 37.

$$\begin{aligned}&\forall p, q \in \mathit{Stack} \to \mathcal{V}.\ \forall X \in \mathcal{P}^{\downarrow}(\mathbb{N}).\ \forall s \in \mathit{seq}\ \mathit{Stm}.\\ &\quad X \subseteq \mathit{safe}(s, p, q) \Rightarrow (\forall i \in \mathbb{N}.\ i \in \mathit{safe}(s, \lambda l.\ p(l) * \mathit{asn}(X), q))\end{aligned}$$

Proof.

- suppose $i \in \mathbb{N}$; if $i = 0$ then s is trivially safe; assume $i > 0$
- let $t \in \mathit{TId}$ and $l \in \mathit{Stack}$
- Case $\mathit{irr}((t, l, s))$:
  - then it suffices to show that $i \in (p(l) * \mathit{asn}(X)) \sqsubseteq q(l)$
    - let $j \in \mathbb{N}$, $r \in \mathcal{V}$, $m \in \mathcal{M}$ and $h \in \mathit{Heap}$ such that

      $$0 \leq j \leq i \qquad j \in \mathit{stable}(r) \qquad (j, m) \in p(l) * \mathit{asn}(X) * r \qquad h \in \lfloor m \rfloor$$

    - then $j \in X$ and thus $j \in (p(l) \sqsubseteq q(l))$
    - hence, there exists an $m' \in \mathcal{M}$ such that $(j, m') \in q(l) * r$ and $h \in \lfloor m' \rfloor$
- Case $(t, l, s) \xrightarrow{a} \{y\} \uplus T$:
  - by assumption, for every $j \in X$ there exists a $p'_j \in \{y\} \uplus T \to \mathcal{V}$ such that

    $$\begin{aligned}&\forall z \in \{y\} \uplus T.\ j \in \mathit{stable}(p'_j(z))\\ &j \in (a\ \mathit{sat}\ \{p\}\{p'_j(y) * (\circledast_{z \in T} p'_j(z))\})\\ &\mathit{safe}_{j-1}(y, p'_j(y), q)\\ &\forall z \in T.\ \mathit{safe}_{j-1}(z, p'_j(z), \lambda\_.\ \top)\end{aligned}$$

  - for each $j \in \mathbb{N}$, take $p''_j \in \{y\} \uplus T \to \mathcal{V}$ to be

    $$p''_j = \begin{cases} p'_j & \text{if } j \in X \\ \lambda z.\ \bot & \text{otherwise} \end{cases}$$

  - then for every $j \in \mathbb{N}$

    $$\begin{aligned}&\mathit{safe}_j(y, p''_{j+1}(y), q)\\ &\forall z \in T.\ \mathit{safe}_j(z, p''_{j+1}(z), \lambda\_.\ \top)\end{aligned}$$

– hence, by Lemma 36,

$$\begin{aligned} &\mathit{safe}_i(y, \bigcup_j \lceil p''_{j+1}(y) \rceil^j, q) \\ &\forall z \in T.\ \mathit{safe}_i(z, \bigcup_j \lceil p''_{j+1}(z) \rceil^j, \lambda\_.\ \top) \end{aligned}$$

– take $p' \in \{y\} \uplus T \to \mathcal{V}$ to be

$$p'(z) = \bigcup_j \lceil p''_{j+1}(z) \rceil^j$$

– then $\forall z \in \{y\} \uplus T.\ \forall i \in \mathbb{N}.\ i \in \mathit{stable}(p'(z))$

– it thus suffices to prove that,

$$i \in (a\ \mathit{sat}\ \{p * \mathit{asn}(X)\}\{p'(y) * (\circledast_{z \in T} p'(z))\})$$

∗ suppose $j \in \mathbb{N}$, $1 \leq j \leq i$, $r \in \mathcal{V}$, $m \in \mathcal{M}$ and $h, h' \in \mathit{Heap}$ such that

$$j \in \mathit{stable}(r) \qquad (j, m) \in p * \mathit{asn}(X) * r \qquad h \in \lfloor m \rfloor \qquad h' \in [\![a]\!](h)$$

∗ then $j \in X$ and $(j, m) \in p * r$

∗ hence $j \in (a\ \mathit{sat}\ \{p\}\{p'_j(y) * (\circledast_{z \in T} p'_j(z))\})$

∗ there thus exists an $m' \in \mathcal{M}$ such that $h' \in \lfloor m' \rfloor$ and

$$(j - 1, m') \in p'_j(y) * (\circledast_{z \in T} p'_j(z)) * r$$

∗ hence

$$(j - 1, m') \in p'(y) * (\circledast_{z \in T} p'(z)) * r$$

□

Lemma 38.

$$\begin{aligned} &\forall p \in \mathcal{V}.\ \forall X \in \mathcal{P}^{\downarrow}(\mathbb{N}). \\ &\quad X \subseteq (p \sqsubseteq p * \mathit{asn}(X)) \end{aligned}$$

Proof.

- let $i \in X$
- suppose $j \in \mathbb{N}$ such that $0 \leq j \leq i$ and $r \in \mathcal{V}$ such that $j \in \mathit{stable}(r)$
- by downwards-closure of X, $j \in X$
- hence

$$\forall m \in \mathcal{M}.\ (j, m) \in p * r \Rightarrow (j, m) \in p * \mathit{asn}(X) * r$$

- and thus

$$\lfloor p * r \rfloor_j \subseteq \lfloor p * asn(X) * r \rfloor_j$$

□

Corollary 3.

$$\begin{array}{l}\forall p, q \in Stack \to \mathcal{V}.\ \forall X \in \mathcal{P}^{\downarrow}(\mathbb{N}).\\ \quad (\forall i \in \mathbb{N}.\ i \in safe(s, p * asn(X), q)) \Rightarrow (X \subseteq safe(s, p, q))\end{array}$$

4.4 CAP meta-theory

In this section we develop the CAP meta-theory of the model. The main results are the soundness of rule StableClosed (Lemma 60) and OpenA (Lemma 68).

Region assertion $\boxed{region : \Delta(RId) \times \Delta(RType) \times \Delta(Val) \times Prop \to Prop \in \text{ASets}}$

$$\begin{array}{l}region(r, t, a, p) \stackrel{\text{def}}{=} \{(i, (l, s, \varsigma)) \in \mathbb{N} \times \mathcal{M} \mid\\ \qquad r \in dom(s) \wedge s(r).t = t \wedge s(r).a = a \wedge (i, (s(r).l, s, \varsigma)) \in p\}\end{array}$$

We use $\boxed{p}^{r,t,a}$ as shorthand for $region(r, t, a, p)$.

Action interpretation $\boxed{act : RType \times (Val \times AId \times Val \to \mathcal{V})^2 \to AArg \to Action}$

The *act* function takes as argument a region type and an action pre- and post-condition and interprets them as an action. Since actions are step-indexed relations on shared states, the interpretation ignores the action model component of the action pre- and post-condition, by existentially quantifying over the action model.

$$\begin{array}{l}act(t, p, q) \stackrel{\text{def}}{=} \lambda(a, r, \alpha) \in Val \times RId \times AId.\\ \qquad \{(i, s_1, s_2) \in \mathbb{N} \times SState \times SState \mid\\ \qquad\ \exists l_1, l_2 \in LState.\ \exists \varsigma_1, \varsigma_2 \in AMod.\ \exists v \in Val.\\ \qquad\quad (i, (l_1, s_1, \varsigma_1)) \in p(a, \alpha, v) \wedge (i, (l_2, s_2, \varsigma_2)) \in q(a, \alpha, v) \wedge\\ \qquad\quad s_1(r) = (l_1, t, a) \wedge s_2(r) = (l_2, t, a) \wedge s_1 \leq s_2 \wedge \varsigma_1 \leq \varsigma_2\}\end{array}$$

Lemma 39.

$$\begin{array}{l}\forall r \in RId.\ \forall t \in RType.\ \forall a \in Val.\ \forall \alpha \in AId.\\ \forall i \in \mathbb{N}.\ \forall p_1, p_2, q_1, q_2 \in Val \times AId \times Val \to Prop.\\ \quad (\forall x \in Val \times AId \times Val.\ p_1(x) =^{\mathcal{V}}_{i+1} p_2(x) \wedge q_1(x) =^{\mathcal{V}}_{i+1} q_2(x)) \ \Rightarrow\\ \qquad act(t, p_1, q_1)(a, r, \alpha)|_i \subseteq act(t, p_2, q_2)(a, r, \alpha)|_i\end{array}$$

Proof.

- assume $(j, s_1, s_2) \in act(t, p_1, q_1)(a, r, \alpha)|_i$
- then $j \leq i$ and there exists $l_1, l_2 \in LState$, $\varsigma_1, \varsigma_2 \in AMod$, and $v \in Val$ such that

$$(j, (l_1, s_1, \varsigma_1)) \in p_1(a, \alpha, v) \qquad (j, (l_2, s_2, \varsigma_2)) \in q_1(a, \alpha, v) \qquad s_i(r) = (l_i, t, a)$$

- hence,

$$(j, (l_1, s_1, a)) \in p_2(a, \alpha, v) \qquad (j, (l_2, s_2, a)) \in q_2(a, \alpha, v)$$

- and thus, $(j, s_1, s_2) \in act(t, p_2, q_2)(a, r, \alpha)|_i$

□

Protocol assertion

$$\boxed{protocol : \Delta(RType) \times (\Delta(Val) \times \Delta(AId) \times \Delta(Val) \to Prop)^2 \to Prop \in \text{ASets}}$$

The *protocol* function corresponds to the protocol assertion in the logic.

$$protocol(t, p, q) \stackrel{\text{def}}{=} \{(i, (l, s, \varsigma)) \in \mathbb{N} \times \mathcal{M} \mid \forall x \in AArg.\ \varsigma(t)(x)|_i = act(t, p, q)(x)|_i\}$$

At index i, it asserts that the protocol on the given region type is i-equal to the protocol given by the action pre- and post-conditions p and q. We specifically do not require strict equality, to ensure that assertion is closed under i-equality, as required. In particular, this ensures that,

$$\forall (i, m_1) \in protocol(t, p, q).\ \forall m_2.\ m_1 =_i m_2 \Rightarrow (i, m_2) \in protocol(t, p, q)$$

Lemma 40. *protocol is non-expansive.*

$$\forall t \in RId.\ \forall i \in \mathbb{N}.\ \forall p_1, p_2, q_1, q_2 \in Val \times AId \times Val \to Prop.$$
$$(\forall x \in Val \times AId \times Val.\ p_1(x) =_i^{\mathcal{V}} p_2(x) \wedge q_1(x) =_i^{\mathcal{V}} q_2(x)) \Rightarrow$$
$$protocol(t, p_1, q_1) =_i^{\mathcal{V}} protocol(t, p_2, q_2)$$

Proof.

- assume $j < i$ and $(j, m) \in protocol(t, p_1, q_1)$
- then

$$\forall x \in AArg.\ \pi_a(m)(t)(x)|_j = act(t, p_1, q_1)(x)|_j$$

- and by Lemma 39 and downwards-closure of $=_i^{\mathcal{V}}$,

$$\forall x \in Val \times AId \times Val.\ act(t, p_1, q_1)(x)|_j = act(t, p_2, q_2)(x)|_j$$

- hence $(j, m) \in protocol(t, p_2, q_2)$

□

Action assertion $\boxed{action : \Delta(AId) \times \Delta(RId) \times \Delta(Perm) \to Prop \in \text{ASets}}$

$$action(\alpha, r, p) \stackrel{\text{def}}{=} \{(i, m) \in \mathbb{N} \times \mathcal{M} \mid p \leq (m.l.c)(r, \alpha)\}$$

Shared state equivalence $\boxed{\equiv_{(-)} : \mathcal{P}(RType) \to \mathcal{P}(SState \times SState)}$

$$s_1 \equiv_A s_2 \stackrel{\text{def}}{=} dom(s_1) = dom(s_2) \wedge (\forall r \in dom(s_1).\ s_1(r).t = s_2(r).t \wedge s_1(r).a = s_2(r).a) \wedge (\forall r \in dom(s_1).\ s_1(r).a \in A \Rightarrow s_1(r).l = s_2(r).l)$$

Action model restriction $\boxed{(-)|_{(=)} : AMod \times \mathcal{P}(RType) \to AMod}$

$$\varsigma|_A(t) \stackrel{\text{def}}{=} \begin{cases} \lambda\alpha \in AArg. & \\ \quad \{(i, s_1, s_2) \in SState \times SState \mid \exists s_1', s_2' \in SState. & \text{if } t \in dom(\varsigma) \\ \qquad s_1 \equiv_A s_1' \wedge s_2 \equiv_A s_2' \wedge (i, s_1', s_2') \in \varsigma(t)(\alpha)\} & \\ \text{undef} & \text{otherwise} \end{cases}$$

Action model extension $\boxed{\leq_{(-)} : \mathcal{P}(RType) \to \mathcal{P}(AMod \times AMod)}$

$$\varsigma_1 \leq_A \varsigma_2 \stackrel{\text{def}}{=} \forall t \in dom(\varsigma_1).\ \forall \alpha \in AArg.\ \forall (i, s_1, s_2) \in \varsigma_1(t)(\alpha).\ \exists s_1', s_2' \in SState.\ t \in dom(\varsigma_2) \wedge s_1 \equiv_A s_1' \wedge s_2 \equiv_A s_2' \wedge (i, s_1', s_2') \in \varsigma_2(t)(\alpha)$$

Action model equivalence $\boxed{\equiv_{(-)} : \mathcal{P}(RType) \to \mathcal{P}(AMod \times AMod)}$

$$\varsigma_1 \equiv_A \varsigma_2 \stackrel{\text{def}}{=} \varsigma_1 \leq_A \varsigma_2 \wedge \varsigma_2 \leq_A \varsigma_1$$

Protocol purity $\boxed{pure_{protocol} : Prop \to Spec \in \text{ASets}}$

$$pure_{protocol}(p) \stackrel{\text{def}}{=} \{i \in \mathbb{N} \mid \forall j \leq i.\ \forall (j, (l, s, \varsigma)) \in p.\ \forall \varsigma' \in AMod.\ (j, (l, s, \varsigma')) \in p\}$$

Permission purity $\boxed{pure_{perm} : Prop \to Spec \in \text{ASets}}$

$$pure_{perm}(p) = \{i \in \mathbb{N} \mid \forall j \leq i.\ \forall (j, ((h, c), s, \varsigma)) \in p.\ \forall c' \in Cap.\ (j, ((h, c'), s, \varsigma)) \in p\}$$

State purity $\boxed{pure_{state} : Prop \to Spec \in \text{ASets}}$

$$pure_{state}(p) \stackrel{\text{def}}{=} \{i \in \mathbb{N} \mid \forall j \leq i.\ \forall (l, s, \varsigma) \in p.\ \forall l' \in LState.\ \forall s' \in SState.\ (l', s', \varsigma) \in p\}$$

Single-step view-shift $\boxed{\sqsubseteq^{(-)} : \Delta(\mathcal{P}(RType)) \to Prop \times Prop \to Spec \in \mathrm{ASets}}$

$$p \sqsubseteq^A q \stackrel{\text{def}}{=} \{i \in \mathbb{N} \mid \forall m \in \mathcal{M}.\ \forall j \in \mathbb{N}.\ 0 \leq j \leq i \ \Rightarrow \lfloor p * \{(j,m)\}\rfloor_j \subseteq \lfloor q * \{(j,m') \mid m\ \hat{R}^A_j\ m'\}\rfloor_j\}$$

Single-step atomic action

$$\boxed{sat_{(-)} : \Delta(\mathcal{P}(RType)) \to \Delta(Action) \times Prop \times Prop \to Spec \in \mathrm{ASets}}$$

$$a\ \mathrm{sat}^A\ \{p\}\{q\} \stackrel{\text{def}}{=} \{i \in \mathbb{N} \mid \forall m \in \mathcal{M}.\ \forall j \in \mathbb{N}.\ 1 \leq j \leq i \ \Rightarrow [\![a]\!](\lfloor p * \{(j,m)\}\rfloor_j) \subseteq \lfloor q * \{(j-1,m') \mid m\ \hat{R}^A_{j-1}\ m'\}\rfloor_{j-1}\}$$

Lemma 41.

$$\forall A, B \in \mathcal{P}(RType).\ \forall p, q \in \mathcal{V}.\ \forall a \in Act.\ a\ sat^A\ \{p\}\{q\} \subseteq a\ sat^{A \cup B}\ \{p\}\{q\}$$

Lemma 42.

$$\forall A \in \mathcal{P}(RType).\ \forall p, q \in \mathcal{V}.\ \forall a \in Act.\ a\ sat^A\ \{p\}\{q\} \subseteq a\ sat\ \{p\}\{q\}$$

Lemma 43.

$$\forall A, B \in \mathcal{P}(RType).\ \forall s_1, s_2 \in SState.\ s_1 \equiv_A s_2 \Rightarrow s_1 \equiv_{A \setminus B} s_2$$

Lemma 44.

$$\forall A, B \in \mathcal{P}(RType).\ \forall \varsigma_1, \varsigma_2 \in AMod.\ \varsigma_1 \leq_A \varsigma_2 \Rightarrow \varsigma_1 \leq_{A \setminus B} \varsigma_2$$

Lemma 45.

$$\forall A \in \mathcal{P}(RType).\ \forall \varsigma \in AMod.\ \varsigma \equiv_A \varsigma|_A$$

Lemma 46.

$$\forall A \in \mathcal{P}(RType).\ \forall \varsigma_1, \varsigma_2 \in AMod.\ \forall t \in dom(\varsigma_1).\ \forall \alpha \in AArg.\ \varsigma_1 \leq_A \varsigma_2 \Rightarrow \varsigma_1(t)(\alpha) \subseteq \varsigma_2|_A(t)(\alpha)$$

Lemma 47.

$$\forall A \in \mathcal{P}(RType).\ \forall \varsigma \in AMod.\ \forall t \in dom(\varsigma).\ \forall \alpha \in AArg.\ \varsigma(t)(\alpha) \subseteq \varsigma|_A(t)(\alpha)$$

Lemma 48.

$$\begin{aligned}&\forall i \in \mathbb{N}.\ \forall l, l' \in \mathit{LState}.\ \forall s_1, s_2, s_1', s_2' \in \mathit{SState}.\ \forall \varsigma_1, \varsigma_2 \in \mathit{AMod}.\\ &\quad (s_1 \equiv_A s_1' \wedge s_2 \equiv_A s_2' \wedge \ulcorner (l, s_1) \urcorner = \ulcorner (l', s_1') \urcorner \wedge\\ &\quad (\forall r \in \mathit{dom}(s_1).\ s_1(r) = s_2(r) \Rightarrow s_1'(r) = s_2'(r))) \ \Rightarrow\\ &\qquad (l, s_1, \varsigma_1)\ \hat{R}_i^{\mathit{RType}}\ (l, s_2, \varsigma_2) \Rightarrow (l', s_1', \varsigma_1|_A)\ \hat{R}_i^{\mathit{RType}}\ (l', s_2', \varsigma_2|_A)\end{aligned}$$

Proof.

- by definition of $\hat{R}_i^{\mathit{RType}}$

  $$s_1 \leq s_2 \qquad\qquad \varsigma_1 \leq \varsigma_2$$

  and there exists a $c = \pi_c(\ulcorner (l, s_1) \urcorner)$ such that for all $r \in \mathit{dom}(s_1)$:

  $$s_1(r) = s_2(r) \vee (\exists \alpha \in \mathit{AId}.\ c(r, \alpha) < 1 \wedge (i, s_1, s_2) \in \varsigma_1(s_1(r).t)(s_1(r).a, r, \alpha))$$

- from the definition of $\equiv_A$ and $(-)|_A$ it thus follows that,

  $$s_1' \leq s_2' \qquad\qquad \varsigma_1|_A \leq \varsigma_2|_A$$

- let $r \in \mathit{dom}(s_1') = \mathit{dom}(s_1)$
  - case $s_1(r) = s_2(r)$:
    - by assumption $s_1'(r) = s_2'(r)$
  - case $c(r, \alpha) < 1$, $(i, s_1, s_2) \in \varsigma_1(s_1(r).t)(s_1(r).a, r, \alpha)$:
    - by definition of $(-)|_A$ it thus follows that

      $$(i, s_1', s_2') \in (\varsigma_1|_A)(s_1(r).t)(s_1(r).a, r, \alpha)$$

□

4.4.1 Support

Support assertion $\boxed{\mathit{supp}_{(-)}(=) : \Delta(\mathcal{P}(\mathit{RType})) \times \mathit{Prop} \to \mathit{Spec} \in \text{ASets}}$

$$\begin{aligned}\mathit{supp}_A(p) \stackrel{\text{def}}{=} \{i \in \mathbb{N} \mid\ &\forall j \leq i.\ \forall (j, (l, s, \varsigma)) \in p.\ \forall s' \in \mathit{SState}.\ \forall \varsigma' \in \mathit{AMod}.\\ &s|_A = s'|_A \wedge \varsigma \equiv_A \varsigma' \Rightarrow (j, (l, s', \varsigma')) \in p\}\end{aligned}$$

We use $\mathit{supp}_A(p_1, ..., p_n)$ as shorthand for $\mathit{supp}_A(p_1) \cap \cdots \cap \mathit{supp}_A(p_n)$.

Lemma 49.

$$\forall A \in \mathcal{P}(\mathit{RType}).\ \forall s_1, s_2 \in \mathit{SState}.\ s_1|_A \equiv_A s_2 \Rightarrow s_1|_A = s_2$$

Proof.

- by definition of $\equiv_A$, $dom(s_1|_A) = dom(s_2)$
- and for every $r \in dom(s_1|_A)$, $((s_1|_A)(r)).t \in A$
- and thus, $((s_1|_A)(r)).l = (s_2(r)).l$

□

Lemma 50.

$$\forall A \in \mathcal{P}(RType).\ \forall R \in Action.\ \forall i \in \mathbb{N}.\ \forall s_1, s_2 \in SState.$$
$$s_1 \leq s_2 \wedge (i, s_1|_A, s_2|_A) \in R \Rightarrow (i, s_1, s_2) \in R$$

Proof. Follows from the assumption that $good(R)$. □

Lemma 51.

$$\forall A, B \in \mathcal{P}(RType).\ \forall p \in \mathcal{V}.\ supp_A(p) \subseteq supp_{A \cup B}(p)$$

Lemma 52.

$$\forall A \in \mathcal{P}(RType).\ \forall r \in RId.\ \forall t \in RType.\ \forall a \in Val.\ \forall p \in \mathcal{V}.$$
$$supp_{A \setminus \{t\}}(p) \subseteq supp_{A \cup \{t\}}(\boxed{p}^{r,t,a})$$

Proof.

- let $l \in LState$, $s, s' \in SState$, and $\varsigma, \varsigma' \in AMod$ such that

$$(j, (l, s, \varsigma)) \in \boxed{p}^{r,t,a} \qquad s|_{A \cup \{t\}} = s'|_{A \cup \{t\}} \qquad \varsigma \equiv_{A \cup \{t\}} \varsigma'$$

- then $s(r).t = t$ and $(j, (s(r).l, s, \varsigma)) \in p$
- and since $s|_{A \setminus \{t\}} = s'|_{A \setminus \{t\}}$ and $\varsigma \equiv_{A \setminus \{t\}} \varsigma'$ it follows that

$$(j, (s(r).l, s', \varsigma')) \in p$$

- and since region r has region type t it follows that $s(r) = s'(r)$ and thus,

$$(j, (l, s', \varsigma')) \in \boxed{p}^{r,t,a}$$

□

Lemma 53.

$$\forall A \in \mathcal{P}(RType).\ \forall i \in \mathbb{N}.\ \forall t \in RType.\ \forall I_p, I_q \in Val \times AId \times Val \to \mathcal{V}.$$
$$(\forall x \in Val \times AId \times Val.\ i \in supp_{A \setminus \{t\}}(I_p(x), I_q(x))) \Rightarrow$$
$$i \in supp_{A \cup \{t\}}(protocol(t, I_p, I_q))$$

Proof.

- let $j \leq i$, $l \in \mathit{LState}$, $s, s' \in \mathit{SState}$, and $\varsigma, \varsigma' \in \mathit{AMod}$ such that

$$(j, (l, s, \varsigma)) \in \mathit{protocol}(t, I_p, I_q) \qquad s|_{A\cup\{t\}} = s'|_{A\cup\{t\}} \qquad \varsigma \equiv_{A\cup\{t\}} \varsigma'$$

- then

$$\forall x \in \mathit{AArg}.\ \varsigma(t)(x)|_j = \mathit{act}(t, I_p, I_q)(x)|_j$$

- it thus suffices to show that $\forall x \in \mathit{AArg}.\ \varsigma(t)(x)|_j = \varsigma'(t)(x)|_j$
- let $(a, r, \alpha) \in \mathit{AArg}$
- assume $(k, s_1, s_2) \in \varsigma'(t)(a, r, \alpha)|_j$:
  - then, by Lemma 46, $(k, s_1, s_2) \in \varsigma|_{A\cup\{t\}}(t)(a, r, \alpha)$
  - hence, there exists $s'_1, s'_2 \in \mathit{SState}$ such that

$$s_1 \equiv_{A\cup\{t\}} s'_1 \qquad s_2 \equiv_{A\cup\{t\}} s'_2 \qquad (k, s'_1, s'_2) \in \varsigma(t)(a, r, \alpha)$$

  - hence, there exists $l_1, l_2 \in \mathit{LState}$, $\varsigma'', \varsigma''' \in \mathit{AMod}$, and $v \in \mathit{Val}$ such that

$$(k, (l_1, s'_1, \varsigma'')) \in I_p(a, \alpha, v) \qquad (k, (l_2, s'_2, \varsigma''')) \in I_q(a, \alpha, v)$$

    and

$$s'_i(r) = (l_i, t, a) \qquad s'_1 \leq s'_2 \qquad \varsigma'' \leq \varsigma'''$$

  - since $s_i \equiv_{A\cup\{t\}} s'_i$ it follows that

$$s_i|_{A\setminus\{t\}} = s'_i|_{A\setminus\{t\}} \qquad s_i(r) = (l_i, t, a)$$

  - since $i \in \mathit{supp}_{A\setminus\{t\}}(I_p(a, \alpha, v), I_q(a, \alpha, v))$ it thus follows that

$$(k, (l_1, s_1, \varsigma'')) \in I_p(a, \alpha, v) \qquad (k, (l_2, s_2, \varsigma''')) \in I_q(a, \alpha, v)$$

  - and thus, $(k, s_1, s_2) \in \varsigma(t)(a, r, \alpha)|_j$
- assume $(k, s_1, s_2) \in \varsigma(t)(a, r, \alpha)|_j$:
  - then there exists $l_1, l_2 \in \mathit{LState}$, $\varsigma'', \varsigma''' \in \mathit{AMod}$, and $v \in \mathit{Val}$ such that

$$(k, (l_1, s_1, \varsigma'')) \in I_p(a, \alpha, v) \qquad (k, (l_2, s_2, \varsigma''')) \in I_q(a, \alpha, v)$$

    and

$$s_i(r) = (l_i, t, a) \qquad s_1 \leq s_2 \qquad \varsigma'' \leq \varsigma'''$$

– since $i \in supp_{A\setminus\{t\}}(I_p(a,\alpha,v), I_q(a,\alpha,v))$ we thus have that,

$$(k,(l_1,s_1|_{A\setminus\{t\}},\varsigma'')) \in I_p(a,\alpha,v) \qquad (k,(l_2,s_2|_{A\setminus\{t\}},\varsigma''')) \in I_q(a,\alpha,v)$$

– and thus $(k, s_1|_{A\setminus\{t\}}, s_2|_{A\setminus\{t\}}) \in \varsigma(t)(a,r,\alpha)$

– hence, by $\varsigma \equiv_{A\cup\{t\}} \varsigma'$, there exists $s_1', s_2' \in SState$ such that

$$s_1|_{A\setminus\{t\}} \equiv_{A\cup\{t\}} s_1' \qquad s_2|_{A\setminus\{t\}} \equiv_{A\cup\{t\}} s_2' \qquad (k,s_1',s_2') \in \varsigma'(t)(\alpha)$$

– hence, by Lemmas 43 and 49,

$$s_1' = s_1|_{A\setminus\{t\}} \qquad\qquad s_2' = s_2|_{A\setminus\{t\}}$$

– and thus, by Lemma 50,

$$(k,s_1,s_2) \in \varsigma'(t)(a,r,\alpha)$$

□

Lemma 54.

$$\forall A \in \mathcal{P}(RType).\ \forall p_1,p_2 \in \mathcal{V}.\ supp_A(p_1) \cap supp_A(p_2) \subseteq supp_A(p_1 \cap p_2) \cap supp_A(p_1 \cup p_2)$$

Lemma 55.

$$\forall A \in \mathcal{P}(RType).\ \forall p_1,p_2 \in \mathcal{V}.\ supp_A(p_1) \cap supp_A(p_2) \subseteq supp_A(p_1 \Rightarrow p_2)$$

Lemma 56.

$$\forall A \in \mathcal{P}(RType).\ \forall p_1,p_2 \in \mathcal{V}.\ supp_A(p_1) \cap supp_A(p_2) \subseteq supp_A(p_1 * p_2)$$

Proof.

- let $l \in LState$, $s, s' \in SState$ and $\varsigma, \varsigma' \in AMod$ such that

$$(j,(l,s,\varsigma)) \in p_1 * p_2 \qquad\qquad s|_A = s'|_A \qquad\qquad \varsigma \equiv_A \varsigma'$$

- hence, there exists $l_1, l_2 \in LState$ such that

$$l = l_1 \bullet l_2 \qquad\qquad (j,(l_1,s,\varsigma)) \in p_1 \qquad\qquad (j,(l_2,s,\varsigma)) \in p_2$$

- hence, $(j,(l_1,s',\varsigma')) \in p_1$ and $(j,(l_2,s',\varsigma')) \in p_2$ and thus $(j,(l,s',\varsigma')) \in p_1 * p_2$

□

4.4.2 Stability

Interference decomposition $\boxed{\hat{R}^{(=)}_{(-)} \in \mathbb{N} \times (RId \times \mathcal{P}(AId)) \to \mathcal{P}(\mathcal{M} \times \mathcal{M})}$

$$
\begin{array}{l}
(l_1, s_1, \varsigma_1)\ \hat{R}_i^{r,A}\ (l_2, s_2, \varsigma_2) \quad \text{iff} \\
\quad l_1 = l_2 \wedge s_1 \leq s_2 \wedge \varsigma_1 \leq \varsigma_2 \wedge \\
\quad\quad \exists c \in Cap.\ c = \pi_c(\ulcorner (l_1, s_1) \urcorner). \\
\quad\quad\quad (\forall r' \in dom(s_1).\ s_1(r') = s_2(r') \vee \\
\quad\quad\quad\quad (\exists \alpha \in AId.\ (r' = r \Rightarrow \alpha \in A) \wedge c(r', \alpha) < 1 \\
\quad\quad\quad\quad\quad \wedge\ (i, s_1, s_2) \in \varsigma_1(s_1(r).t)(s_1(r).a, r, \alpha)))
\end{array}
$$

Stability decomposition $\boxed{stable^{(-)} : \Delta(\mathcal{P}(AId) \times RId) \to Prop \to Spec \in \text{ASets}}$

$$stable^{r,A}(p) \stackrel{\text{def}}{=} \{i \in \mathbb{N} \mid \forall j \leq i.\ \forall m, m' \in \mathcal{M}.\ (j, m) \in p \wedge m\ \hat{R}_i^{r,A}\ m' \Rightarrow (j, m') \in p\}$$

Lemma 57.

$$\forall i \in \mathbb{N}.\ \forall r \in RId.\ \hat{R}_i^{r,AId} = \hat{R}_i^{RType}$$

Lemma 58.

$$\forall i \in \mathbb{N}.\ \forall A_1, A_2 \in \mathcal{P}(AId).\ \forall r \in RId.\ A_1 \subseteq A_2 \Rightarrow \hat{R}_i^{r,A_1} \subseteq \hat{R}_i^{r,A_2}$$

Lemma 59.

$$
\begin{array}{l}
\forall A_1, A_2 \in \mathcal{P}(AId).\ \forall r \in RId.\ \forall p \in \mathcal{V}. \\
\quad stable^{r,A_1}(p) \cap stable^{r,A_2}(p) \subseteq stable^{r,A_1 \cup A_2}(p)
\end{array}
$$

Lemma 60.

$$
\begin{array}{l}
\forall i \in \mathbb{N}.\ \forall p, q \in \mathcal{V}.\ \forall I_p, I_q : Val \times AId \times Val \to \mathcal{V}. \\
\forall r \in RId.\ \forall t \in RType.\ \forall a \in Val.\ \forall \alpha \in AId. \\
\quad i \in stable(p * q) \wedge i \in supp_{A \setminus \{t\}}(p, q)\ \wedge \\
\quad i \in pure_{protocol}(p) \wedge pure_{state}(q)\ \wedge \\
\quad (\forall v : Val.\ ((I_p(a, \alpha, v) \cap p) \subseteq \bot) \vee (I_q(a, \alpha, v) \subseteq p)) \\
\quad \Rightarrow i \in stable^{r,\{\alpha\}}(\boxed{p * q}^{r,t,a}_{I_p, I_q})
\end{array}
$$

Proof.

- assume

$$j \leq i \qquad (j, (l, s, \varsigma)) \in \boxed{p * q}^{r,t,a}_{I_p, I_q} \qquad (l, s, \varsigma)\ \hat{R}_j^{r,\{\alpha\}}\ (l, s', \varsigma')$$

- then $(j, (s(r).l, s, \varsigma)) \in p$ and $(j, (l, s, \varsigma)) \in q$
- case $s(r) = s'(r)$:
  - by Lemma 45, $\varsigma \equiv_{A\setminus\{t\}} \varsigma|_{A\setminus\{t\}}$, and hence, since $i \in supp_{A\setminus\{t\}}(p, q)$,
    $$(j, (s(r).t, s[r \mapsto (l, t, a)], \varsigma|_{A\setminus\{t\}})) \in p * q$$
  - by Lemma 58 it follows that $(l, s, \varsigma)\ \hat{R}_j^{r,AId}\ (l, s', \varsigma')$ and thus, by Lemma 57,
    $$(l, s, \varsigma)\ \hat{R}_j^{RType}\ (l, s', \varsigma')$$
  - and since $\ulcorner (l, s) \urcorner = \ulcorner (s(r).l, s[r \mapsto (l, t, a)]) \urcorner$,
    $$s \equiv_{A\setminus\{t\}} s[r \mapsto (l, t, a)] \qquad\qquad s' \equiv_{A\setminus\{t\}} s'[r \mapsto (l, t, a)]$$
    and
    $$\forall r' \in dom(s[r \mapsto (l, t, a)]).\ s(r') = s'(r') \Rightarrow (s[r \mapsto (l, t, a)])(r') = (s'[r \mapsto (l, t, a)])(r')$$
    it follows from Lemma 48 that,
    $$(s(r).l, s[r \mapsto (l, t, a)], \varsigma|_{A\setminus\{t\}})\ \hat{R}_j^{RType}\ (s(r).l, s'[r \mapsto (l, t, a)], \varsigma'|_{A\setminus\{t\}})$$
  - hence, by stability of $p * q$,
    $$(j, (s(r).l, s'[r \mapsto (l, t, a)], \varsigma'|_{A\setminus\{t\}})) \in p * q$$
  - and, since $i \in supp_{A\setminus\{t\}}(p, q)$,
    $$(j, (s'(r).l, s', \varsigma')) \in p * q$$
  - hence,
    $$(j, (l, s', \varsigma')) \in \boxed{p * q}^{r,t,a}_{I_p, I_q}$$
- case $s(r) \neq s'(r)$:
  - then $(j, s, s') \in \varsigma(t)(a, r, \alpha)$
  - hence, there exists $l_1, l_2 \in LState$, $\varsigma'', \varsigma''' \in AMod$, and $v \in Val$ such that
    $$(j, (l_1, s, \varsigma'')) \in I_p(a, \alpha, v) \qquad s(r) = (l_1, t, a)$$
    $$(j, (l_2, s', \varsigma''')) \in I_q(a, \alpha, v) \qquad s'(r) = (l_2, t, a)$$
  - case $I_p(a, \alpha, v) \cap p \subseteq \bot$:
    - since $i \in pure_{protocol}(p)$, $(j, (s(r).l, s, \varsigma'')) \in p$

  - hence $(j, (s(r).l, s, \varsigma)) \in \bot$, which is a contradiction
- case $I_q(a, \alpha, v) \subseteq p$:
  - then $(j, (s'(r).l, s', \varsigma''')) \in p$
  - since $i \in pure_{protocol}(p)$ we thus have that $(j, (s'(r).l, s', \varsigma')) \in p$
  - furthermore, since $(j, (l, s, \varsigma)) \in q$, $\varsigma \leq \varsigma'$ and $i \in pure_{state}(q)$,

$$(j, (\varepsilon, s', \varsigma')) \in q$$

  and thus $(j, (l, s', \varsigma')) \in \boxed{p * q}^{r,t,a}_{I_p, I_q}$

□

4.4.3 View shifts

Lemma 61.

$$\forall i \in \mathbb{N}.\ \forall A \in \mathcal{P}(RType).\ \forall m_1, m_2, m_3, m_4 \in \mathcal{M}.$$
$$m_1\ \hat{R}^{\emptyset}_i\ m_2 \wedge m_2\ \hat{R}^{A}_i\ m_3 \wedge m_3\ \hat{R}^{\emptyset}_i\ m_4 \Rightarrow m_1\ \hat{R}^{A}_i\ m_4$$

Lemma 62.

$$\forall A \in \mathcal{P}(RType).\ \forall p, p', q, q' \in Prop.$$
$$(p \sqsubseteq^{A} p') \cap (a\ sat^{\emptyset}\ \{p'\}\{q'\}) \cap (q' \sqsubseteq^{\emptyset} q) \subseteq (a\ sat^{A}\ \{p\}\{q\}) \wedge$$
$$(p \sqsubseteq^{\emptyset} p') \cap (a\ sat^{A}\ \{p'\}\{q'\}) \cap (q' \sqsubseteq^{\emptyset} q) \subseteq (a\ sat^{A}\ \{p\}\{q\}) \wedge$$
$$(p \sqsubseteq^{\emptyset} p') \cap (a\ sat^{\emptyset}\ \{p'\}\{q'\}) \cap (q' \sqsubseteq^{A} q) \subseteq (a\ sat^{A}\ \{p\}\{q\})$$

Proof.

- let $1 \leq j \leq i$, $m_1, m_2 \in \mathcal{M}$ and $h, h' \in Heap$ such that

$$(j, m_1) \in p \qquad h \in \lfloor m_1 \bullet m_2 \rfloor \qquad h' \in [\![a]\!](h)$$

- then there exists $m_1', m_2' \in \mathcal{M}$ such that

$$(j, m_1') \in p' \qquad h \in \lfloor m_1' \bullet m_2' \rfloor \qquad m_2\ \hat{R}^{A}_j\ m_2'$$

- hence, there exists $m_1'', m_2'' \in \mathcal{M}$ such that

$$(j-1, m_1'') \in q' \qquad h' \in \lfloor m_1'' \bullet m_2'' \rfloor \qquad m_2'\ \hat{R}^{\emptyset}_{j-1}\ m_2''$$

- hence, there exists $m_1''', m_2''' \in \mathcal{M}$ such that

$$(j-1, m_1''') \in q \qquad h' \in \lfloor m_1''' \bullet m_2''' \rfloor \qquad m_2''\ \hat{R}^{\emptyset}_{j-1}\ m_2'''$$

- hence, by Lemma 61, $m_2\ \hat{R}^A_{j-1}\ m_2'''$ and thus

$$(p \sqsubseteq^A p') \cap (a\ \mathrm{sat}^{\emptyset}\ \{p'\}\{q'\}) \cap (q' \sqsubseteq^{\emptyset} q) \subseteq (a\ \mathrm{sat}^A\ \{p\}\{q\})$$

- the other cases follow the same structure

□

Lemma 63.

$$\forall i \in \mathbb{N}.\ \forall I_p, I_q \in \mathit{Val} \times \mathit{AId} \times \mathit{Val} \to \mathit{Prop}.\ \forall t \in \mathit{RType}.$$
$$i \in (\mathit{emp} \sqsubseteq^{\emptyset} \exists s \in \mathit{RType}.\ t \leq s * \mathit{protocol}(s, I_p, I_q))$$

Proof.

- let $j \leq i$, $l_1, l_2 \in \mathit{LState}$, $s \in \mathit{SState}$, $\varsigma \in \mathit{AMod}$ and $h \in \mathit{Heap}$ such that

$$(j, (l_1, s, \varsigma)) \in \mathit{emp} \qquad h \in \lfloor (l_1, s, \varsigma) \bullet (l_2, s, \varsigma) \rfloor$$

- by assumption $\mathit{dom}(\varsigma)$ is finite and thus $t^* \setminus \mathit{dom}(\varsigma)$ is infinite
- pick $s \in \mathit{RType}$ such that $s \in t^* \setminus \mathit{dom}(\varsigma)$
- then

$$(l_2, s, \varsigma)\ \hat{R}^{\emptyset}_j\ (l_2, s, \varsigma[s \mapsto \mathit{act}(s, I_p, I_q)])$$

- and

$$(j, (l_1, s, \varsigma[s \mapsto \mathit{act}(s, I_p, I_q)])) \in \mathit{protocol}(s, I_p, I_q)$$

- and lastly,

$$h \in \lfloor (l_1, s, a[s \mapsto \mathit{act}(s, I_p, I_q)]) \bullet (l_2, s, a[s \mapsto \mathit{act}(s, I_p, I_q)]) \rfloor$$

□

Lemma 64.

$$\forall i \in \mathbb{N}.\ \forall A \in \mathcal{P}_{fin}(\mathit{AId}).\ \forall t \in \mathit{RType}.\ \forall a \in \mathit{Val}.\ \forall p \in \mathit{Prop}.$$
$$i \in (p \sqsubseteq^{\emptyset} \exists r : \mathit{RId}.\ \boxed{p}^{r,t,a} * \circledast_{\alpha \in A} [\alpha]^r_1)$$

Proof.

- let $j \leq i$, $l_1, l_2 \in \mathit{LState}$, $s \in \mathit{SState}$, $\varsigma \in \mathit{AMod}$ and $h \in \mathit{Heap}$ such that

$$(j, (l_1, s, \varsigma)) \in p \qquad h \in \lfloor (l_1, s, \varsigma) \bullet (l_2, s, \varsigma) \rfloor$$

- by assumption $dom(s) \cup dom_r(\pi_c(l_1 \bullet l_2))$ is finite
- pick $r \in RId$ such that $r \notin dom(s) \cup dom_r(\pi_c(l_1 \bullet l_2))$
- then
$$(l_2, s, \varsigma)\ \hat{R}^{\emptyset}_j\ (l_2, s[r \mapsto (l_1, t, a)], \varsigma)$$
- and
$$(j, ((\varepsilon, [(r, A) \mapsto 1]), s[r \mapsto (l_1, t, a)], \varsigma)) \in \boxed{p}^{r,t,a} * \circledast_{\alpha \in A} [\alpha]^r_1$$
- and lastly,
$$h \in \lfloor ((\varepsilon, [(r, A) \mapsto 1]), s[r \mapsto (l_1, t, a)], \varsigma) \bullet (l_2, s[r \mapsto (l_1, t, a)], \varsigma) \rfloor$$

□

Phantom points-to $\boxed{(-)_{(=)} \mapsto (\equiv) : \Delta(OId) \times \Delta(FName) \times \Delta(Val) \to Prop \in \text{ASets}}$

$$x_f \mapsto y \stackrel{\text{def}}{=} \{(i, m) \in \mathbb{N} \times \mathcal{M} \mid (m.l.p)(x, f) = y\}$$

Lemma 65.

$$\forall t \in RType.\ \forall x \in OId.\ \forall f \in FName.\ \forall v_1, v_2 \in Val.$$
$$\forall i \in \mathbb{N}.\ i \in (x_f \mapsto v_1 \sqsubseteq^{\emptyset} x_f \mapsto v_2)$$

Proof.

- let $j \leq i$, $r \in Prop$, $m_1, m_2 \in \mathcal{M}$ and $h \in Heap$ such that
$$(j, m_1) \in x_f \mapsto v_1 \qquad\qquad h \in \lfloor m_1 \bullet m_2 \rfloor$$
- hence, $(x, f) \in dom(m_1.l.p)$ and $(x, f) \notin dom(m_2.l.p)$
- thus,
$$(j, m_1[(x, f) \mapsto_p v_2]) \in x_f \mapsto v_2 \qquad \lfloor m_1 \bullet m_2 \rfloor = \lfloor m_1[(x, f) \mapsto_p v_2] \bullet m_2 \rfloor$$
where $m[(x, f) \mapsto_p v]$ is notation for $((m.l.h, m.l.p[(x, f) \mapsto v], m.l.c), m.s, m.a)$

□

Atomic update allowed

$$\boxed{\rightsquigarrow^{(-)}_{(=)} : \Delta(RId \times \mathcal{P}(RType)) \times \Delta(Action) \to Prop \times Prop \to Spec \in \mathrm{ASets}}$$

$$\begin{aligned} p \rightsquigarrow^{r,A}_a q \stackrel{\text{def}}{=} \{i \in \mathbb{N} \mid\ & \forall j \leq i.\ \forall m_1, m_2 \in \mathcal{M}.\ \forall h_1 \in \lfloor m_1 \rfloor.\ \forall h_2 \in \lfloor m_2 \rfloor. \\ & 1 \leq j \wedge (j, m_1) \in p \wedge (j-1, m_2) \in q \wedge h_2 \in [\![a]\!](h_1)\ \Rightarrow \\ & \exists \alpha \in AId. \\ & (m_1.l.c)(r, \alpha) > 0 \wedge \\ & (j-1, (m_1.s)|_A, (m_2.s)|_A) \in (m_1.a)((m_1.s)(r).t)((m_1.s)(r).a, r, \alpha)\} \end{aligned}$$

View shift allowed

$$\boxed{\rightsquigarrow^{(-)} : \Delta(RId \times \mathcal{P}(RType)) \to Prop \times Prop \to Spec \in \mathrm{ASets}}$$

$$\begin{aligned} p \rightsquigarrow^{r,A} q \stackrel{\text{def}}{=} \{i \in \mathbb{N} \mid\ & \forall j \leq i.\ \forall m_1, m_2 \in \mathcal{M}.\ \forall h \in \lfloor m_1 \rfloor.\ \forall h \in \lfloor m_2 \rfloor. \\ & (j, m_1) \in p \wedge (j, m_2) \in q\ \Rightarrow \\ & \exists \alpha \in AId. \\ & (m_1.l.c)(r, \alpha) > 0 \wedge \\ & (j, (m_1.s)|_A, (m_2.s)|_A) \in (m_1.a)((m_1.s)(r).t)((m_1.s)(r).a, r, \alpha)\} \end{aligned}$$

Lemma 66.

$$\begin{aligned} & \forall i \in \mathbb{N}.\ \forall A \in \mathcal{P}(RType).\ \forall r \in RId.\ \forall t \in RType.\ \forall b \in Val.\ \forall p_1, p_2, q_1, q_2 \in Prop. \\ & \quad valid(p_1 * p_2 \Rightarrow q_1 * q_2) \cap supp_{A \setminus \{t\}}(p_1, p_2, q_1, q_2) \cap (\boxed{p_1}^{r,t,b} * p_2 \rightsquigarrow^{r, A \cup \{t\}} \boxed{q_1}^{r,t,b} * q_2) \\ & \quad \subseteq\ (\boxed{p_1}^{r,t,b} * p_2 \sqsubseteq^{A \cup \{t\}} \boxed{q_1}^{r,t,b} * q_2) \end{aligned}$$

Proof.

- let $j \leq i$, $(l_1, s, \varsigma), (l_2, s, \varsigma), (l_3, s, \varsigma) \in \mathcal{M}$ and $h \in Heap$ such that

  $$(j, (l_1, s, \varsigma)) \in \boxed{p_1}^{r,t,b} \qquad\qquad (j, (l_2, s, \varsigma)) \in p_2$$

  and

  $$h \in \lfloor (l_1, s, \varsigma) \bullet (l_2, s, \varsigma) \bullet (l_3, s, \varsigma) \rfloor$$

- thus

  $$(l_r, s, \varsigma) \in p_1 \qquad\qquad s(r) = (l_1, t, b)$$

  where $l_r = s(r).l$

- since $i \in supp_{A\setminus\{t\}}(p_1, p_2)$, we thus have that

$$(j, (l_r, s_r, \varsigma)) \in p_1 \qquad\qquad (j, (l_2, s_r, \varsigma)) \in p_2$$

  where $s_r = s[r \mapsto (\varepsilon, t, b)]$
- and by upwards-closure, $(j, (l_r \bullet l_1, s_r, \varsigma)) \in p_1$
- hence, there exists $(l_1', s', \varsigma'), (l_2', s', \varsigma') \in \mathcal{M}$ such that

$$(j, (l_1', s', \varsigma')) \in q_1 \qquad\qquad (j, (l_2', s', \varsigma')) \in q_2$$

  and

$$(l_r \bullet l_1, s_r, \varsigma) \bullet (l_2, s_r, \varsigma) = (l_1', s', \varsigma') \bullet (l_2', s', \varsigma')$$

- hence, $s' = s_r$, $\varsigma' = \varsigma$ and $l_r \bullet l_1 \bullet l_2 = l_1' \bullet l_2'$
- since $i \in supp_{A\setminus\{t\}}(q_1, q_2)$ it follows that

$$(j, (\varepsilon, s_r', \varsigma)) \in \boxed{q_1}^{r,t,b} \qquad\qquad (j, (l_2', s_r', \varsigma)) \in q_2$$

  where $s_r' = s[r \mapsto (l_1', t, b)]$
- furthermore, as

$$\begin{aligned} \lfloor (l_1, s, \varsigma) \bullet (l_2, s, \varsigma) \bullet (l_3, s, \varsigma) \rfloor &= \lfloor (l_r \bullet l_1, s_r, \varsigma) \bullet (l_2, s_r, \varsigma) \bullet (l_3, s_r, \varsigma) \rfloor \\ &= \lfloor (l_1', s_r, \varsigma) \bullet (l_2', s_r, \varsigma) \bullet (l_3, s_r, \varsigma) \rfloor \\ &= \lfloor (\varepsilon, s_r', \varsigma) \bullet (l_2', s_r', \varsigma) \bullet (l_3, s_r', \varsigma) \rfloor \end{aligned}$$

  it suffices to prove that $(l_3, s, \varsigma)\ \hat{R}_j^{A\cup\{t\}}\ (l_3, s_r', \varsigma)$
- since $i \in (\boxed{p_1}^{r,t,b} * p_2 \rightsquigarrow^{r,A\cup\{t\}} \boxed{q_1}^{r,t,b} * q_2)$ there exists an $\alpha \in AId$ such that

$$\pi_c(l_1 \bullet l_2)(r, \alpha) > 0 \qquad\qquad (j, s|_{A\cup\{t\}}, s_r'|_{A\cup\{t\}}) \in \varsigma(t)(b, r, \alpha)$$

- and for all $r' \in dom(s)$ such that $r \neq r'$, $s(r') = s_r'(r')$

□

Lemma 67.

$\forall i \in \mathbb{N}.\ \forall A, B \in \mathcal{P}(RType).\ \forall l \in LState.\ \forall s_1, s_2, s_1', s_2' \in SState.\ \forall \varsigma_1, \varsigma_2 \in AMod.$
$s_1 \equiv_A s_1' \wedge s_2 \equiv_A s_2' \wedge (\forall r \in dom(s_1'|_{RType\setminus(A\cup B)}).\ s_1'(r) = s_2'(r)) \wedge \ulcorner (l, s_1) \urcorner = \ulcorner (l, s_1') \urcorner \wedge$
$(\forall r \in dom(s_1'|_B).\ \exists \alpha \in AId.\ \pi_c(\ulcorner (l_1, s_1) \urcorner)(r, \alpha) < 1 \wedge$
$(i, s_1'|_{A\cup B}, s_2'|_{A\cup B}) \in \varsigma_1(s_1'(r).t)(s_1'(r).a, r, \alpha))\ \Rightarrow$
$(l, s_1, \varsigma_1)\ \hat{R}_i^A\ (l, s_2, \varsigma_2) \Rightarrow (l, s_1', \varsigma_1)\ \hat{R}_i^{A\cup B}\ (l, s_2', \varsigma_2)$

Proof.

- by assumption, $s_1 \leq s_2$ and $\varsigma_1 \leq \varsigma_2$
- since, $s_1 \equiv_A s_1'$ and $s_2 \equiv_A s_2'$, $s_1' \leq s_2'$
- let $r \in dom(s_1') = dom(s_1)$
- case $s_1'(r) = s_2'(r)$: trivial
- case $s_1'(r) \neq s_2'(r)$:
  - case $r \in dom(s_1'|_B)$:
    - by assumption, there exists an $\alpha \in AId$ such that

$$\pi_c(\ulcorner (l, s_1') \urcorner)(r, \alpha) < 1 \quad (i, s_1'|_{A \cup B}, s_2'|_{A \cup B}) \in \varsigma_1(s_1'(r).t)(s_1'(r).a, r, \alpha)$$

  - case $r \in dom(s_1'|_A)$:
    - then $s_1(r) = s_1'(r) \neq s_2'(r) = s_2(r)$
    - there thus exists an $\alpha \in AId$ such that $\pi_c(\ulcorner (l, s_1) \urcorner)(r, \alpha) < 1$ and

$$(i, s_1|_A, s_2|_A) \in \varsigma_1(s_1(r).g)(s_1(r).a, r, \alpha)$$

    - hence, $\pi_c(\ulcorner (l, s_1') \urcorner)(r, \alpha) < 1$ and

$$(i, s_1'|_{A \cup B}, s_2'|_{A \cup B}) \in \varsigma_1(s_1'(r).t)(s_1'(r).a, r, \alpha)$$

  - case $r \in dom(s_1'|_{RType \setminus (A \cup B)})$: trivial

□

Lemma 68.

$\forall A \in \mathcal{P}(RType).\ \forall a \in Act.\ \forall r \in RId.\ \forall t \in RType.\ \forall b \in Val.\ \forall p_1, p_2, q_1, q_2 \in X \to Prop.$

$$supp_{A \setminus \{t\}}(p_1, p_2, q_1, q_2) \cap pure_{perm}(p_1) \cap$$
$$(\exists x : X.\ \boxed{p_1(x)}^{r,t,b} * p_2(x) \rightsquigarrow_a^{r, A \cup \{t\}} \exists x : X.\ \boxed{q_1(x)}^{r,t,b} * q_2(x)) \cap$$
$$(a\ sat^{A \setminus \{t\}}\ \{\exists x : X.\ p_1(x) * p_2(x)\}\{\exists x : X.\ q_1(x) * q_2(x)\})$$
$$\subseteq\ (a\ sat^{A \cup \{t\}}\ \{\exists x : X.\ \boxed{p_1(x)}^{r,t,b} * p_2(x)\}\{\exists x : X.\ \boxed{q_1(x)}^{r,t,b} * q_2(x)\})$$

Proof.

- let $1 \leq j \leq i$, $(l_1, s, \varsigma), (l_2, s, \varsigma), (l_3, s, \varsigma) \in \mathcal{M}$, $x \in X$ and $h, h' \in Heap$ such that

$$(j, (l_1, s, \varsigma)) \in \boxed{p_1(x)}^{r,t,b} \qquad (j, (l_2, s, \varsigma)) \in p_2(x)$$

  and

$$h \in \lfloor (l_1, s, \varsigma) \bullet (l_2, s, \varsigma) \bullet (l_3, s, \varsigma) \rfloor \qquad h' \in [\![a]\!](h)$$

- thus

$$(l_r, s, \varsigma) \in p_1(x) \qquad\qquad s(r) = (l_1, t, b)$$

  where $l_r = (s(r).l$

- since $i \in supp_{A\setminus\{t\}}(p_1, p_2) \cap pure_{perm}(p_1)$, we thus have that

$$(j, ((l_r.h, l_r.p, \varepsilon), s_r, \varsigma)) \in p_1(x) \qquad\qquad (j, (l_2, s_r, \varsigma)) \in p_2(x)$$

  where $s_r = s[r \mapsto ((\varepsilon, \varepsilon, l_r.c), t, b)]$

- and by upwards-closure, $(j, ((l_r.h, l_r.p, \varepsilon) \bullet l_1, s_r, \varsigma)) \in p_1(x)$

- furthermore,

$$\lfloor (l_1, s, \varsigma) \bullet (l_2, s, \varsigma) \bullet (l_3, s, \varsigma) \rfloor = \lfloor ((l_r.h, l_r.p, \varepsilon) \bullet l_1, s_r, \varsigma) \bullet (l_2, s_r, \varsigma) \bullet (l_3, s_r, \varsigma) \rfloor$$

- hence, there exists $(l_1', s', \varsigma'), (l_2', s', \varsigma'), (l_3', s', \varsigma') \in \mathcal{M}$, and $x' \in X$ such that

$$(j-1, (l_1', s', \varsigma')) \in q_1(x') \quad (j-1, (l_2', s', \varsigma')) \in q_2(x') \quad (l_3, s_r, \varsigma)\ \hat{R}_{j-1}^{A\setminus\{t\}}\ (l_3', s', \varsigma')$$

  and

$$h' \in \lfloor (l_1', s', \varsigma') \bullet (l_2', s', \varsigma') \bullet (l_3', s', \varsigma') \rfloor$$

- since $i \in supp_{A\setminus\{t\}}(q_1, q_2)$, we thus have that,

$$(j-1, (\varepsilon, s_r', \varsigma')) \in \boxed{q_1(x')}^{r,t,b} \qquad\qquad (j-1, (l_2', s_r', \varsigma')) \in q_2(x')$$

  where $s_r' = s[r \mapsto (l_1', t, b)]$

- furthermore, as $s'(r) = s_r(r) = ((\varepsilon, \varepsilon, \pi_c(l_r)), t, b)$ it follows that

$$h' \in \lfloor (\varepsilon, s_r', \varsigma') \bullet (l_2', s_r', \varsigma') \bullet (l_3', s_r', \varsigma') \rfloor$$

- since

$$i \in (\exists x : X.\ \boxed{p_1(x)}^{r,t,b} * p_2(x) \rightsquigarrow_a^{r, A\cup\{t\}} \exists x : X.\ \boxed{q_1(x)}^{r,t,b} * q_2(x))$$

  there exists an $\alpha \in AId$ such that,

$$\pi_c(l_1 \bullet l_2)(r, \alpha) > 0 \qquad\qquad (j-1, s|_{A\cup\{n\}}, s_r'|_{A\cup\{t\}}) \in \varsigma(t)(b, r, \alpha)$$

- lastly, since $s_r \equiv_{A\setminus\{t\}} s$ and $s' \equiv_{A\setminus\{t\}} s_r'$ it thus follows from Lemma 67 that,

$$(l_3, s, \varsigma)\ \hat{R}_{j-1}^{A\cup\{t\}}\ (l_3', s_r', \varsigma')$$

□

4.4.4 Atomic update allowed

Lemma 69.

$$\forall A \in \mathcal{P}(RType).\ \forall r \in RId.\ \forall a \in Act.\ \forall p_1, p_2, q_1, q_2 \in Prop.$$
$$(p_1 \rightsquigarrow_a^{r,A} q_1) \subseteq (p_1 * p_2 \rightsquigarrow_a^{r,A} q_1 * q_2)$$

Proof. Follows from the fact that $p * q \subseteq p$. □

Lemma 70.

$$\forall I_p, I_q \in Val \times AId \times Val \to Prop.\ \forall t \in RType.\ \forall r \in RId.\ \forall a \in Act.\ \forall b, v \in Val.\ \forall \pi \in \mathsf{Perm}.$$
$$supp_{A\cup\{t\}}(I_p(b,\alpha,v), I_q(b,\alpha,v)) \subseteq (\boxed{I_p(b,\alpha,v)}^{r,t,b}_{I_p,I_q} * [\alpha]^r_\pi \rightsquigarrow_a^{r,A\cup\{t\}} \boxed{I_q(b,\alpha,v)}^{r,t,b}_{I_p,I_q} * [\alpha]_\pi)$$

Proof.

- let $1 \le j \le i$, $(l_1, s_1, \varsigma_1), (l_2, s_2, \varsigma_2) \in \mathcal{M}$, $h, h' \in Heap$ such that

$$(j, (l_1, s_1, \varsigma_1)) \in \boxed{I_p(b,\alpha,v)}^{r,t,b}_{I_p,I_q} * [\alpha]^r_\pi$$
$$(j-1, (l_2, s_2, \varsigma_2)) \in \boxed{I_q(b,\alpha,v)}^{r,t,b}_{I_p,I_q} * [\alpha]^r_\pi$$

  and

$$h \in \lfloor (l_1, s_1, \varsigma_1) \rfloor \qquad h' \in \lfloor (l_2, s_2, \varsigma_2) \rfloor \qquad h' \in [\![a]\!](h)$$

- then $\varsigma_1(t)(b, r, \alpha)|_j = act(t, I_p, I_q)(b, r, \alpha)|_j$
- furthermore,

$$(j, (s_1(r).l, s_1, \varsigma_1)) \in I_p(b,\alpha,v) \qquad (j-1, (s_2(r).l, s_2, \varsigma_2)) \in I_q(b,\alpha,v)$$

- and, since $i \in supp_A(I_p(b,\alpha,v), I_q(b,\alpha,v))$,

$$(j, (s_1(r).l, s_1|_{A\cup\{t\}}, \varsigma_1)) \in I_p(b,\alpha,v) \quad (j-1, (s_2(r).l, s_2|_{A\cup\{t\}}, \varsigma_2)) \in I_q(b,\alpha,v)$$

- and thus, $(j-1, s_1|_{A\cup\{t\}}, s_2|_{A\cup\{t\}}) \in act(t, I_p, I_q)(b, r, \alpha)|_j = \varsigma_1(t)(b, r, \alpha)|_j$

□

Points-to $\boxed{(-).(=) \mapsto (\equiv) : \Delta(OId) \times \Delta(FName) \times \Delta(CVal) \to Prop \in \text{ASets}}$

$$x.f \mapsto y \stackrel{\text{def}}{=} \{(i, m) \in \mathbb{N} \times \mathcal{M} \mid (m.l.h.o)(x, f) = y\}$$

Lemma 71.

$$\begin{array}{l}
\forall A, B \in \mathcal{P}(RType).\ \forall p_1, p_2, q_1, q_2, q_3 \in Prop.\ \forall t \in RType.\ \forall r \in RId. \\
\forall a \in Act.\ \forall x \in OId.\ \forall b \in Val.\ \forall v_1, v_2 \in CVal.\ \forall f \in FName.\ v_1 \neq v_2 \ \Rightarrow \\
\quad supp_{A \setminus \{t\}}(p_1, p_2, q_1, q_2) \cap (a\ sat\ \{p_1 * p_2\}\{q_3\}) \cap \\
\quad (valid((q_1 * q_2) \Rightarrow x.f \mapsto v_1)) \cap (valid(q_3 \Rightarrow x.f \mapsto v_2)) \\
\qquad \subseteq\ (\boxed{p_1}^{r,t,b} * p_2 \rightsquigarrow_a^{r,B} \boxed{q_1}^{r,t,b} * q_2)
\end{array}$$

Proof.

- let $j \in \mathbb{N}$, $m_1, m_2 \in \mathcal{M}$ and $h, h' \in Heap$ such that

  $$(j, m_1) \in \boxed{p_1}^{r,t,b} * p_2 \qquad (j, m_2) \in \boxed{q_1}^{r,t,b} * q_2$$

  and

  $$h \in \lfloor m_1 \rfloor \qquad h' \in \lfloor m_2 \rfloor \qquad h' \in [\![a]\!](h)$$

- since $j \in supp_{A \setminus \{t\}}(p_1, p_2, q_1, q_2)$ it thus follows that

  $$\begin{array}{l}
  (j, (m_1.l \bullet (m_1.s)(r).l, (m_1.s)[r \mapsto \bot], m_1.a)) \in p_1 * p_2 \\
  (j, (m_2.l \bullet (m_2.s)(r).l, (m_2.s)[r \mapsto \bot], m_2.a)) \in q_1 * q_2
  \end{array}$$

  where $s[r \mapsto \bot]$ is notation for $s|_{dom(s) \setminus \{r\}}$

- and

  $$\begin{array}{l}
  \lfloor m_1 \rfloor = \lfloor (m_1.l \bullet (m_1.s)(r).l, (m_1.s)[r \mapsto \bot], m_1.a) \rfloor \\
  \lfloor m_2 \rfloor = \lfloor (m_2.l \bullet (m_2.s)(r).l, (m_2.s)[r \mapsto \bot], m_2.a) \rfloor
  \end{array}$$

- hence, there exists an $m_3 \in \mathcal{M}$ such that

  $$(j - 1, m_3) \in q_3 \qquad h' \in \lfloor m_3 \rfloor$$

- hence,

  $$\begin{array}{l}
  (j - 1, (m_2.l \bullet (m_2.s)(r).l, (m_2.s)[r \mapsto \bot], m_2.a)) \in x.f \mapsto v_1 \\
  (j - 1, m_3) \in x.f \mapsto v_2
  \end{array}$$

- and thus $v_1 = h'(x, f) = v_2$, contradicting $v_1 \neq v_2$.

□

5 Interpretation

In this section we define the interpretation of the logic in the model, and sketch the soundness of a few representative proof rules.

Types $\boxed{[\![\vdash \tau : \mathsf{Type}]\!] \in \mathrm{ASets}}$

Types are interpreted as step-indexed equivalence relations. All base types except propositions and specifications have a plain identity as the step-indexed equivalence. Specifications and assertion propositions are considered i-equal if the agree up to level i. Products and function spaces are interpreted using the cartesian closed structure of ASets.

$$
\begin{aligned}
[\![\vdash 1 : \mathsf{Type}]\!] &= 1 \\
[\![\vdash \tau \to \sigma : \mathsf{Type}]\!] &= [\![\vdash \tau : \mathsf{Type}]\!] \to [\![\vdash \sigma : \mathsf{Type}]\!] \\
[\![\vdash \tau \times \sigma : \mathsf{Type}]\!] &= [\![\vdash \tau : \mathsf{Type}]\!] \times [\![\vdash \sigma : \mathsf{Type}]\!] \\
[\![\vdash \mathsf{Prop} : \mathsf{Type}]\!] &= \mathit{Prop} \\
[\![\vdash \mathsf{Spec} : \mathsf{Type}]\!] &= \mathit{Spec} \\
[\![\vdash \mathsf{Val} : \mathsf{Type}]\!] &= \Delta(\mathit{Val}) \\
[\![\vdash \mathsf{Class} : \mathsf{Type}]\!] &= \Delta(\mathit{CName}) \\
[\![\vdash \mathsf{Method} : \mathsf{Type}]\!] &= \Delta(\mathit{MName}) \\
[\![\vdash \mathsf{Field} : \mathsf{Type}]\!] &= \Delta(\mathit{FName}) \\
[\![\vdash \mathsf{Region} : \mathsf{Type}]\!] &= \Delta(\mathit{RId}) \\
[\![\vdash \mathsf{Action} : \mathsf{Type}]\!] &= \Delta(\mathit{AId}) \\
[\![\vdash \mathsf{RType} : \mathsf{Type}]\!] &= \Delta(\mathit{RType}) \\
[\![\vdash \mathsf{Perm} : \mathsf{Type}]\!] &= \Delta((0,1])
\end{aligned}
$$

Context $\boxed{[\![\Gamma]\!] \in \mathrm{ASets}, [\![\Delta]\!] \in \mathrm{ASets}}$

$$
\begin{aligned}
[\![\Gamma, \mathsf{x} : \tau]\!] &\stackrel{\text{def}}{=} [\![\Gamma]\!] \times [\![\vdash \tau : \mathsf{Type}]\!] & [\![\varepsilon]\!] &\stackrel{\text{def}}{=} 1 \\
[\![\Delta, \mathsf{x} : \mathsf{Val}]\!] &\stackrel{\text{def}}{=} [\![\Delta]\!] \times [\![\vdash \mathsf{Val} : \mathsf{Type}]\!] & [\![\varepsilon]\!] &\stackrel{\text{def}}{=} 1
\end{aligned}
$$

Terms $\boxed{[\![\Gamma; \Delta \vdash \mathsf{M} : \tau]\!] : [\![\Gamma]\!] \times [\![\Delta]\!] \to [\![\tau]\!] \in \mathrm{ASets}}$

Terms are interpreted as morphisms in ASets and thus as non-expansive functions.

Lambda calculus

$$
\begin{aligned}
[\![\Gamma; \Delta \vdash \mathsf{x} : \tau]\!](\vartheta, \delta) &= \pi_{\mathsf{x}}(\vartheta) \\
[\![\Gamma; \Delta \vdash \mathsf{x} : \mathsf{Val}]\!](\vartheta, \delta) &= \pi_{\mathsf{x}}(\delta) \\
[\![\Gamma; \Delta \vdash \lambda \mathsf{x} : \tau.\ \mathsf{M} : \tau \to \sigma]\!](\vartheta, \delta) &= \lambda v \in [\![\vdash \tau : \mathsf{Type}]\!].\ [\![\Gamma, \mathsf{x} : \tau; \Delta \vdash \mathsf{M} : \sigma]\!]((\vartheta, v), \theta) \\
[\![\Gamma; \Delta \vdash \mathsf{M}\ \mathsf{N} : \sigma]\!](\vartheta, \delta) &= ([\![\Gamma; \Delta \vdash \mathsf{M} : \tau \to \sigma]\!](\vartheta, \delta))([\![\Gamma; \Delta \vdash \mathsf{N} : \tau]\!](\vartheta, \delta))
\end{aligned}
$$

Assertion logic

$$
\begin{aligned}
[\![\Gamma; \Delta \vdash \bot : \mathsf{Prop}]\!](\vartheta, \delta) &= \emptyset \\
[\![\Gamma; \Delta \vdash \top : \mathsf{Prop}]\!](\vartheta, \delta) &= \mathbb{N} \times \mathcal{M} \\
[\![\Gamma; \Delta \vdash \mathsf{P} \wedge \mathsf{Q} : \mathsf{Prop}]\!](\vartheta, \delta) &= [\![\Gamma; \Delta \vdash \mathsf{P} : \mathsf{Prop}]\!](\vartheta, \delta) \cap [\![\Gamma; \Delta \vdash \mathsf{Q} : \mathsf{Prop}]\!](\vartheta, \delta) \\
[\![\Gamma; \Delta \vdash \mathsf{P} \vee \mathsf{Q} : \mathsf{Prop}]\!](\vartheta, \delta) &= [\![\Gamma; \Delta \vdash \mathsf{P} : \mathsf{Prop}]\!](\vartheta, \delta) \cup [\![\Gamma; \Delta \vdash \mathsf{Q} : \mathsf{Prop}]\!](\vartheta, \delta) \\
[\![\Gamma; \Delta \vdash \mathsf{P} \Rightarrow \mathsf{Q} : \mathsf{Prop}]\!](\vartheta, \delta) &= \{(i, m) \in \mathbb{N} \times \mathcal{M} \mid \forall j \leq i.\ \forall n \geq m. \\
&\qquad (j, n) \in [\![\Gamma; \Delta \vdash \mathsf{P} : \mathsf{Prop}]\!](\vartheta, \delta)\ \Rightarrow \\
&\qquad (j, n) \in [\![\Gamma; \Delta \vdash \mathsf{Q} : \mathsf{Prop}]\!](\vartheta, \delta)\} \\
[\![\Gamma; \Delta \vdash \forall \mathsf{x} : \tau.\ \mathsf{P} : \mathsf{Prop}]\!](\vartheta, \delta) &= \bigcap_{v \in [\![\vdash \tau : \mathsf{Type}]\!]} [\![\Gamma, \mathsf{x} : \tau; \Delta \vdash \mathsf{P} : \mathsf{Prop}]\!]((\vartheta, v), \delta) \\
[\![\Gamma; \Delta \vdash \exists \mathsf{x} : \tau.\ \mathsf{P} : \mathsf{Prop}]\!](\vartheta, \delta) &= \bigcup_{v \in [\![\vdash \tau : \mathsf{Type}]\!]} [\![\Gamma, \mathsf{x} : \tau; \Delta \vdash \mathsf{P} : \mathsf{Prop}]\!]((\vartheta, v), \delta)
\end{aligned}
$$

Specification logic

$$
\begin{aligned}
[\![\Gamma \vdash \bot : \mathsf{Spec}]\!](\vartheta) &= \emptyset \\
[\![\Gamma \vdash \top : \mathsf{Spec}]\!](\vartheta) &= \mathbb{N} \\
[\![\Gamma \vdash \mathsf{S} \wedge \mathsf{T} : \mathsf{Spec}]\!](\vartheta) &= [\![\Gamma \vdash \mathsf{S} : \mathsf{Spec}]\!](\vartheta) \cap [\![\Gamma \vdash \mathsf{T} : \mathsf{Spec}]\!](\vartheta) \\
[\![\Gamma \vdash \mathsf{S} \vee \mathsf{T} : \mathsf{Spec}]\!](\vartheta) &= [\![\Gamma \vdash \mathsf{S} : \mathsf{Spec}]\!](\vartheta) \cup [\![\Gamma \vdash \mathsf{T} : \mathsf{Spec}]\!](\vartheta) \\
[\![\Gamma \vdash \mathsf{S} \Rightarrow \mathsf{T} : \mathsf{Spec}]\!](\vartheta) &= \{i \in \mathbb{N} \mid \forall j \leq i. \\
&\qquad j \in [\![\Gamma \vdash \mathsf{S} : \mathsf{Spec}]\!](\vartheta)\ \Rightarrow \\
&\qquad j \in [\![\Gamma \vdash \mathsf{T} : \mathsf{Spec}]\!](\vartheta)\}
\end{aligned}
$$

$$
\begin{aligned}
[\![\Gamma \vdash \forall \mathsf{x} : \tau.\ \mathsf{S} : \mathsf{Spec}]\!](\vartheta) &= \bigcap_{v \in [\![\vdash \tau : \mathsf{Type}]\!]} [\![\Gamma, \mathsf{x} : \tau \vdash \mathsf{S} : \mathsf{Spec}]\!]((\vartheta, v)) \\
[\![\Gamma \vdash \exists \mathsf{x} : \tau.\ \mathsf{S} : \mathsf{Spec}]\!](\vartheta) &= \bigcup_{v \in [\![\vdash \tau : \mathsf{Type}]\!]} [\![\Gamma, \mathsf{x} : \tau \vdash \mathsf{S} : \mathsf{Spec}]\!]((\vartheta, v)) \\
[\![\Gamma \vdash \mathsf{M} =_\tau \mathsf{N} : \mathsf{Spec}]\!](\vartheta) &= \{i \in \mathbb{N} \mid [\![\Gamma \vdash \mathsf{M} : \tau]\!](\vartheta) =_i [\![\Gamma \vdash \mathsf{N} : \tau]\!](\vartheta)\}
\end{aligned}
$$

Separation logic

$$
\begin{aligned}
[\![\Gamma; \Delta \vdash \mathsf{emp} : \mathsf{Prop}]\!](\vartheta, \delta) &= emp \\
[\![\Gamma; \Delta \vdash \mathsf{P} * \mathsf{Q} : \mathsf{Prop}]\!](\vartheta, \delta) &= [\![\Gamma; \Delta \vdash \mathsf{P} : \mathsf{Prop}]\!](\vartheta, \delta) * [\![\Gamma; \Delta \vdash \mathsf{Q} : \mathsf{Prop}]\!](\vartheta, \delta) \\
[\![\Gamma; \Delta \vdash \mathsf{P} \mathrel{-\!\!*} \mathsf{Q} : \mathsf{Prop}]\!](\vartheta, \delta) &= \{(i, m) \in \mathbb{N} \times \mathcal{M} \mid \forall j \leq i.\ \forall m' \geq m.\ \forall m'' \in \mathcal{M}. \\
&\qquad (m' \bullet m'' \text{ defined} \wedge (j, m'') \in [\![\Gamma; \Delta \vdash \mathsf{P} : \mathsf{Prop}]\!](\vartheta, \delta)) \\
&\qquad \Rightarrow (j, m' \bullet m'') \in [\![\Gamma; \Delta \vdash \mathsf{Q} : \mathsf{Prop}]\!](\vartheta, \delta)\}
\end{aligned}
$$

$\mathsf{C}^\sharp$

$$
\begin{aligned}
[\![\Gamma; \Delta \vdash \mathsf{x} : \mathsf{Val}]\!](\vartheta, \delta) &= \delta(\mathsf{x}) \\
[\![\Gamma; \Delta \vdash \mathsf{null} : \mathsf{Val}]\!](\vartheta, \delta) &= null \\
[\![\Gamma; \Delta \vdash \mathsf{C} : \mathsf{Class}]\!](\vartheta, \delta) &= \mathsf{C} \\
[\![\Gamma; \Delta \vdash \mathsf{m} : \mathsf{Method}]\!](\vartheta, \delta) &= \mathsf{m} \\
[\![\Gamma; \Delta \vdash \mathsf{f} : \mathsf{Field}]\!](\vartheta, \delta) &= \mathsf{f}
\end{aligned}
$$

$$
\begin{aligned}
[\![\Gamma; \Delta \vdash \mathsf{M.F} \mapsto \mathsf{N} : \mathsf{Prop}]\!](\vartheta, \delta) = \{&(i, m) \in \mathbb{N} \times \mathcal{M} \mid \exists o \in OId.\ \exists v \in CVal.\ \exists f \in FName. \\
&o = [\![\Gamma \vdash \mathsf{M} : \mathsf{Val}]\!](\vartheta, \delta) \wedge \\
&f = [\![\Gamma \vdash \mathsf{F} : \mathsf{Field}]\!](\vartheta, \delta) \wedge \\
&v = [\![\Gamma \vdash \mathsf{N} : \mathsf{Val}]\!](\vartheta, \delta) \wedge \\
&(m.l.h.o)(o, f) = v\} \\
[\![\Gamma; \Delta \vdash \mathsf{M}_\mathsf{F} \mapsto \mathsf{N} : \mathsf{Prop}]\!](\vartheta, \delta) = \{&(i, m) \in \mathbb{N} \times \mathcal{M} \mid \exists o \in OId.\ \exists v \in CVal.\ \exists f \in FName. \\
&o = [\![\Gamma \vdash \mathsf{M} : \mathsf{Val}]\!](\vartheta, \delta) \wedge \\
&f = [\![\Gamma \vdash \mathsf{F} : \mathsf{Field}]\!](\vartheta, \delta) \wedge \\
&v = [\![\Gamma \vdash \mathsf{N} : \mathsf{Val}]\!](\vartheta, \delta) \wedge \\
&(m.l.p)(o, f) = v\} \\
[\![\Gamma; \Delta \vdash \mathsf{N}_1 \mapsto \mathsf{N}_2.\mathsf{M} : \mathsf{Prop}]\!](\vartheta, \delta) = \{&(i, m) \in \mathbb{N} \times \mathcal{M} \mid \exists c \in CId.\ \exists o \in OId.\ \exists m \in MName. \\
&c = [\![\Gamma \vdash \mathsf{N}_1 : \mathsf{Val}]\!](\vartheta, \delta) \wedge \\
&o = [\![\Gamma \vdash \mathsf{N}_2 : \mathsf{Val}]\!](\vartheta, \delta) \wedge \\
&m = [\![\Gamma \vdash \mathsf{M} : \mathsf{Method}]\!](\vartheta, \delta) \wedge \\
&\pi_c(\pi_h(\pi_l(m)))(c) = (o, m)\} \\
[\![\Gamma; \Delta \vdash \mathsf{M} : \mathsf{C} : \mathsf{Prop}]\!](\vartheta, \delta) = \{&(i, m) \in \mathbb{N} \times \mathcal{M} \mid \\
&(m.l.h.t)([\![\Gamma \vdash \mathsf{M} : \mathsf{Val}]\!](\vartheta, \delta)) = [\![\Gamma \vdash \mathsf{C} : \mathsf{Class}]\!](\vartheta, \delta)\}
\end{aligned}
$$

Embeddings and guarded recursion

$$\begin{aligned}
[\![\Gamma \vdash \mathsf{valid}(\mathsf{P}) : \mathsf{Spec}]\!](\vartheta) &= \mathit{valid}([\![\Gamma; - \vdash \mathsf{P} : \mathsf{Prop}]\!](\vartheta)) \\
[\![\Gamma \vdash \mathsf{guarded}_\tau(\mathsf{M}) : \mathsf{Spec}]\!](\vartheta) &= \mathit{guarded}([\![\Gamma \vdash \mathsf{M} : (\tau \to \mathsf{Prop}) \to (\tau \to \mathsf{Prop})]\!](\vartheta)) \\
[\![\Gamma \vdash \triangleright \mathsf{S} : \mathsf{Spec}]\!](\vartheta) &= \triangleright([\![\Gamma \vdash \mathsf{S} : \mathsf{Spec}]\!](\vartheta))
\end{aligned}$$

$$\begin{aligned}
[\![\Gamma; \Delta \vdash \mathsf{asn}(\mathsf{S}) : \mathsf{Prop}]\!](\vartheta, \delta) &= \mathit{asn}([\![\Gamma \vdash \mathsf{S} : \mathsf{Spec}]\!](\vartheta)) \\
[\![\Gamma; \Delta \vdash \mathsf{fix}_\tau(\mathsf{M}) : \tau \to \mathsf{Prop}]\!](\vartheta, \delta) &= \mathit{fix}([\![\Gamma; \Delta \vdash \mathsf{M} : (\tau \to \mathsf{Prop}) \to (\tau \to \mathsf{Prop})]\!](\vartheta, \delta)) \\
[\![\Gamma; \Delta \vdash \triangleright \mathsf{P} : \mathsf{Prop}]\!](\vartheta, \delta) &= \triangleright([\![\Gamma; \Delta \vdash \mathsf{P} : \mathsf{Prop}]\!](\vartheta, \delta))
\end{aligned}$$

Region Types

$$\begin{aligned}
[\![\Gamma; \Delta \vdash \bot : \mathsf{RType}]\!](\vartheta, \delta) &= \text{the empty string} \\
[\![\Gamma; \Delta \vdash \mathsf{M} \leq \mathsf{N} : \mathsf{Prop}]\!](\vartheta, \delta) &= \{(i, m) \in \mathbb{N} \times \mathcal{M} \mid \\
&\qquad [\![\Gamma; \Delta \vdash \mathsf{N} : \mathsf{RType}]\!](\vartheta, \delta) \in ([\![\Gamma; \Delta \vdash \mathsf{M} : \mathsf{RType}]\!](\vartheta, \delta))^*\} \\
[\![\Gamma; \Delta \vdash \mathsf{M} \sqcap \mathsf{N} : \mathsf{RType}]\!](\vartheta, \delta) &= \text{longest common prefix of} \\
&\qquad [\![\Gamma; \Delta \vdash \mathsf{M} : \mathsf{RType}]\!](\vartheta, \delta) \text{ and } [\![\Gamma; \Delta \vdash \mathsf{N} : \mathsf{RType}]\!](\vartheta, \delta)
\end{aligned}$$

Concurrent abstract predicates

$$\begin{aligned}
[\![\Gamma; \Delta \vdash \boxed{\mathsf{P}}^{\mathsf{R},\mathsf{T},\mathsf{A}} : \mathsf{Prop}]\!](\vartheta, \delta) = \mathit{region}([\![&\Gamma; \Delta \vdash \mathsf{R} : \mathsf{Region}]\!](\vartheta, \delta), \\
&[\![\Gamma; \Delta \vdash \mathsf{T} : \mathsf{RType}]\!](\vartheta, \delta), \\
&[\![\Gamma; \Delta \vdash \mathsf{A} : \mathsf{Val}]\!](\vartheta, \delta), \\
&[\![\Gamma; \Delta \vdash \mathsf{P} : \mathsf{Prop}]\!](\vartheta, \delta)) \\
[\![\Gamma; \Delta \vdash \mathsf{protocol}(\mathsf{T}, \mathsf{I}_p, \mathsf{I}_q) : \mathsf{Prop}]\!](\vartheta, \delta) = \mathit{protocol}([\![&\Gamma; \Delta \vdash \mathsf{T} : \mathsf{RType}]\!](\vartheta, \delta), \\
&[\![\Gamma; \Delta \vdash \mathsf{I}_p : \mathsf{Val} \times \mathsf{Action} \times \mathsf{Val} \to \mathsf{Prop}]\!](\vartheta, \delta), \\
&[\![\Gamma; \Delta \vdash \mathsf{I}_q : \mathsf{Val} \times \mathsf{Action} \times \mathsf{Val} \to \mathsf{Prop}]\!](\vartheta, \delta)) \\
[\![\Gamma; \Delta \vdash [\mathsf{A}]^{\mathsf{R}}_{\mathsf{P}} : \mathsf{Prop}]\!](\vartheta, \delta) = \mathit{action}([\![&\Gamma; \Delta \vdash \mathsf{A} : \mathsf{Action}]\!](\vartheta, \delta), \\
&[\![\Gamma; \Delta \vdash \mathsf{R} : \mathsf{Region}]\!](\vartheta, \delta), \\
&[\![\Gamma; \Delta \vdash \mathsf{P} : \mathsf{Perm}]\!](\vartheta, \delta))
\end{aligned}$$

$$
\begin{aligned}
[\![\Gamma \vdash \mathsf{stable(P)} : \mathsf{Spec}]\!](\vartheta) &= \mathit{stable}([\![\Gamma \vdash \mathsf{P} : \mathsf{Prop}]\!](\vartheta)) \\
[\![\Gamma \vdash \mathsf{stable}^{\mathsf{R}}_{\mathsf{A}}\mathsf{(P)} : \mathsf{Spec}]\!](\vartheta) &= \mathit{stable}^{[\![\Gamma \vdash \mathsf{R:Region}]\!](\vartheta)}_{\{[\![\Gamma \vdash \mathsf{A:Action}]\!](\vartheta)\}}([\![\Gamma \vdash \mathsf{P} : \mathsf{Prop}]\!](\vartheta)) \\
[\![\Gamma \vdash \mathsf{dep}_{\mathsf{T}}\mathsf{(P)} : \mathsf{Spec}]\!](\vartheta) &= \mathit{supp}_{([\![\Gamma \vdash \mathsf{T:RType}]\!](\vartheta))^*}([\![\Gamma \vdash \mathsf{P} : \mathsf{Prop}]\!](\vartheta)) \\
[\![\Gamma \vdash \mathsf{indep}_{\mathsf{T}}\mathsf{(P)} : \mathsf{Spec}]\!](\vartheta) &= \mathit{supp}_{\mathit{RType}\setminus([\![\Gamma \vdash \mathsf{T:RType}]\!](\vartheta))^*}([\![\Gamma \vdash \mathsf{P} : \mathsf{Prop}]\!](\vartheta)) \\
[\![\Gamma \vdash \mathsf{pure}_{\mathsf{protocol}}\mathsf{(P)} : \mathsf{Spec}]\!](\vartheta) &= \mathit{pure}_{\mathit{protocol}}([\![\Gamma \vdash \mathsf{P} : \mathsf{Prop}]\!](\vartheta)) \\
[\![\Gamma \vdash \mathsf{pure}_{\mathsf{state}}\mathsf{(P)} : \mathsf{Spec}]\!](\vartheta) &= \mathit{pure}_{\mathit{state}}([\![\Gamma \vdash \mathsf{P} : \mathsf{Prop}]\!](\vartheta)) \\
[\![\Gamma \vdash \mathsf{pure}_{\mathsf{perm}}\mathsf{(P)} : \mathsf{Spec}]\!](\vartheta) &= \mathit{pure}_{\mathit{perm}}([\![\Gamma \vdash \mathsf{P} : \mathsf{Prop}]\!](\vartheta)) \\
[\![\Gamma \vdash \mathsf{P} \sqsubseteq \mathsf{Q} : \mathsf{Spec}]\!](\vartheta) &= [\![\Gamma \vdash \mathsf{P} : \mathsf{Prop}]\!](\vartheta) \sqsubseteq [\![\Gamma \vdash \mathsf{Q} : \mathsf{Prop}]\!](\vartheta) \\
[\![\Gamma \vdash \mathsf{P} \sqsubseteq^{\mathsf{T}} \mathsf{Q} : \mathsf{Spec}]\!](\vartheta) &= [\![\Gamma \vdash \mathsf{P} : \mathsf{Prop}]\!](\vartheta) \sqsubseteq^{\mathit{RType}\setminus([\![\Gamma \vdash \mathsf{T:RType}]\!](\vartheta))^*} [\![\Gamma \vdash \mathsf{Q} : \mathsf{Prop}]\!](\vartheta) \\
[\![\Gamma \vdash \mathsf{P} \rightsquigarrow^{\mathsf{R},\mathsf{T}} \mathsf{Q} : \mathsf{Spec}]\!](\vartheta) &= \{i \in \mathbb{N} \mid \exists r \in \mathit{RId}.\ \exists t \in \mathit{RType}. \\
&\qquad r = [\![\Gamma \vdash \mathsf{R} : \mathsf{Region}]\!](\vartheta) \wedge \\
&\qquad t = [\![\Gamma \vdash \mathsf{T} : \mathsf{RType}]\!](\vartheta) \wedge \\
&\qquad i \in ([\![\Gamma \vdash \mathsf{P} : \mathsf{Prop}]\!](\vartheta) \rightsquigarrow^{r,\mathit{RType}\setminus t^*} [\![\Gamma \vdash \mathsf{Q} : \mathsf{Prop}]\!](\vartheta))\} \\
[\![\Gamma \vdash \mathsf{P} \rightsquigarrow^{\mathsf{R},\mathsf{T}}_{(\Delta).\langle \mathsf{s} \rangle} \mathsf{Q} : \mathsf{Spec}]\!](\vartheta) &= \{i \in \mathbb{N} \mid \exists p, q \in [\![\Delta]\!] \to \mathit{Prop}.\ \exists r \in \mathit{RId}.\ \exists t \in \mathit{RType}. \\
&\qquad \forall l, l' \in \mathit{Stack}.\ \forall a \in \mathit{Act}. \\
&\qquad r = [\![\Gamma \vdash \mathsf{R} : \mathsf{Region}]\!](\vartheta) \wedge \\
&\qquad t = [\![\Gamma \vdash \mathsf{T} : \mathsf{RType}]\!](\vartheta) \wedge \\
&\qquad p = \lambda\delta \in [\![\Delta]\!].\ [\![\Gamma; \Delta \vdash \mathsf{P} : \mathsf{Prop}]\!](\vartheta, \delta) \wedge \\
&\qquad q = \lambda\delta \in [\![\Delta]\!].\ [\![\Gamma; \Delta \vdash \mathsf{Q} : \mathsf{Prop}]\!](\vartheta, \delta) \wedge \\
&\qquad (l, \mathsf{s}) \xrightarrow{a} (l', \varepsilon) \wedge \\
&\qquad i \in (||p||_\Delta(l) \rightsquigarrow^{r,\mathit{RType}\setminus t^*}_a ||q||_\Delta(l'))\}
\end{aligned}
$$

Hoare assertions

$$
\begin{aligned}
&[\![\Gamma \vdash (\Delta).\{\mathsf{P}\}\bar{\mathsf{s}}\{\mathsf{Q}\}]\!](\vartheta) = \\
&\quad \{i \in \mathbb{N} \mid \exists p, q \in [\![\Delta]\!] \to \mathit{Prop}.\ \forall j \leq i.\ \forall t \in \mathit{TId}.\ \forall l \in \mathit{Stack}. \\
&\qquad\qquad p = \lambda\delta \in [\![\Delta]\!].\ [\![\Gamma; \Delta \vdash \mathsf{P} : \mathsf{Prop}]\!](\vartheta, \delta) \wedge \\
&\qquad\qquad q = \lambda\delta \in [\![\Delta]\!].\ [\![\Gamma; \Delta \vdash \mathsf{Q} : \mathsf{Prop}]\!](\vartheta, \delta) \wedge \\
&\qquad\qquad \mathit{safe}_j((t, l, \mathit{stm}(\bar{\mathsf{s}})), ||p||_\Delta(l), ||q||_\Delta)\} \\
&[\![\Gamma \vdash \mathsf{C.M} : (\Delta).\{\mathsf{P}\}\{\mathsf{r.Q}\}]\!](\vartheta) = \\
&\quad \{i \in \mathbb{N} \mid \exists p \in [\![\Delta, \mathtt{this}]\!] \to \mathit{Prop}.\ \exists q \in [\![\Delta, \mathtt{this}, \mathsf{r}]\!]. \\
&\qquad\qquad \exists c \in \mathit{CName}.\ \exists m \in \mathit{MName}. \\
&\qquad\qquad \forall j \leq i.\ \forall t \in \mathit{TId}.\ \forall l \in \mathit{Stack}.\ \forall o_t \in \mathit{OId}.\ \forall v_{\bar{\mathsf{y}}} \in \mathit{CVal}. \\
&\qquad\qquad\quad p = \lambda\delta \in [\![\Delta, \mathtt{this}]\!].\ [\![\Gamma; \Delta \vdash \mathsf{P} : \mathsf{Prop}]\!](\vartheta, \delta) \wedge \\
&\qquad\qquad\quad q = \lambda\delta \in [\![\Delta, \mathtt{this}, \mathsf{z}]\!].\ [\![\Gamma; \Delta \vdash \mathsf{Q}[\mathsf{z}/\mathsf{r}] : \mathsf{Prop}]\!](\vartheta, \delta) \wedge \\
&\qquad\qquad\quad c = [\![\Gamma; \Delta \vdash \mathsf{C} : \mathsf{Class}]\!](\vartheta) \wedge \\
&\qquad\qquad\quad m = [\![\Gamma; \Delta \vdash \mathsf{M} : \mathsf{Method}]\!](\vartheta) \wedge \\
&\qquad\qquad\quad \mathsf{body}(c, m) = \{\overline{\mathsf{C}\mathsf{y}}; \bar{\mathsf{s}}; \mathtt{return}\ \mathsf{z}\} \wedge \\
&\qquad\qquad\quad \mathit{safe}_j((t, (l[\mathtt{this} \mapsto o_t, \bar{\mathsf{y}} \mapsto v_{\bar{\mathsf{y}}}]), \mathit{stm}(\bar{\mathsf{s}})), ||p||_{\Delta,\mathtt{this}}(l[\mathtt{this} \mapsto o_t]), ||q||_{\Delta,\mathtt{this},\mathsf{z}})\} \\
&[\![\Gamma \vdash (\Delta).\{\mathsf{P}\}\langle s\rangle\{\mathsf{Q}\} : \mathsf{Spec}]\!](\vartheta) = \\
&\quad \{i \in \mathbb{N} \mid \exists p, q \in [\![\Delta]\!] \to \mathit{Prop}.\ \forall j \leq i.\ \forall t \in \mathit{TId}.\ \forall l, l' \in \mathit{Stack}.\ \forall a \in \mathit{Act}. \\
&\qquad\qquad p = \lambda\delta \in [\![\Delta]\!].\ [\![\Gamma; \Delta \vdash \mathsf{P} : \mathsf{Prop}]\!](\vartheta, \delta) \wedge \\
&\qquad\qquad q = \lambda\delta \in [\![\Delta]\!].\ [\![\Gamma; \Delta \vdash \mathsf{Q} : \mathsf{Prop}]\!](\vartheta, \delta) \wedge \\
&\qquad\qquad (l, s) \xrightarrow{a} (l', \varepsilon) \wedge \\
&\qquad\qquad j \in (a\ \mathrm{sat}\ \{||p||_\Delta(l)\}\{||q||_\Delta(l')\})\} \\
&[\![\Gamma \vdash (\Delta).\{\mathsf{P}\}\langle s\rangle^{\mathsf{T}}\{\mathsf{Q}\} : \mathsf{Spec}]\!](\vartheta) = \\
&\quad \{i \in \mathbb{N} \mid \exists p, q \in [\![\Delta]\!] \to \mathit{Prop}.\ \exists rt \in \mathit{RType}.\ \forall j \leq i.\ \forall t \in \mathit{TId}.\ \forall l, l' \in \mathit{Stack}.\ \forall a \in \mathit{Act}. \\
&\qquad\qquad p = \lambda\delta \in [\![\Delta]\!].\ [\![\Gamma; \Delta \vdash \mathsf{P} : \mathsf{Prop}]\!](\vartheta, \delta) \wedge \\
&\qquad\qquad q = \lambda\delta \in [\![\Delta]\!].\ [\![\Gamma; \Delta \vdash \mathsf{Q} : \mathsf{Prop}]\!](\vartheta, \delta) \wedge \\
&\qquad\qquad rt = [\![\Gamma \vdash \mathsf{T} : \mathit{RType}]\!](\vartheta) \wedge \\
&\qquad\qquad (l, s) \xrightarrow{a} (l', \varepsilon) \wedge \\
&\qquad\qquad j \in (a\ \mathrm{sat}^{\mathit{RType}\setminus rt^*}\ \{||p||_\Delta(l)\}\{||q||_\Delta(l')\})\}
\end{aligned}
$$

where $||-||_\Delta : ([\![\Delta]\!] \to \mathcal{V}) \to (\mathit{Stack} \to \mathcal{V})$ is defined as follows:

$$
||p||_\Delta(l) \stackrel{\text{def}}{=} \begin{cases} p(l(\mathsf{x}_1), ..., l(\mathsf{x}_n)) & \text{if } \Delta = \mathsf{x}_1, ..., \mathsf{x}_n \text{ and } \mathsf{x}_1, ..., \mathsf{x}_n \in \mathit{dom}(l) \\ \emptyset & \text{otherwise} \end{cases}
$$

Assertion logic entailment $\boxed{[\![\Gamma; \Delta \mid \Phi \mid \mathsf{P} \vdash \mathsf{Q}]\!] : 2}$

$$
\begin{array}{l}
[\![\Gamma; \Delta \mid \Phi \mid \mathsf{P} \vdash \mathsf{Q}]\!] = \\
\quad \forall \vartheta \in [\![\Gamma]\!].\ \forall \delta \in [\![\Delta]\!].\ \forall i \in [\![\Phi]\!](\vartheta).\ \forall m \in \mathcal{M}. \\
\quad\quad (i, m) \in [\![\Gamma; \Delta \vdash \mathsf{P} : \mathsf{Prop}]\!](\vartheta, \delta) \Rightarrow (i, m) \in [\![\Gamma; \Delta \vdash \mathsf{Q} : \mathsf{Prop}]\!](\vartheta, \delta)
\end{array}
$$

Specification logic entailment $\boxed{[\![\Gamma \mid \mathsf{S}_1, ..., \mathsf{S}_n \vdash \mathsf{T}]\!] : 2}$

$$
[\![\Gamma \mid \mathsf{S}_1, ..., \mathsf{S}_n \vdash \mathsf{T}]\!] = \forall \vartheta \in [\![\Gamma]\!].\ \left(\bigcap_{i \in \{1,...,n\}} [\![\Gamma \vdash \mathsf{S}_i : \mathsf{Spec}]\!](\vartheta) \right) \subseteq [\![\Gamma \vdash \mathsf{T} : \mathsf{Spec}]\!](\vartheta)
$$

5.1 Soundness

We have already proven the soundness of the most important proof rules for CAP, guarded recursion, the embeddings and phantom state in Section 4. In this section we consider the soundness of the Hoare logic. In particular, we prove the soundness of two representative proof rules, namely the frame rule (Lemma 75) and the proof rule for forking a new thread (Lemma 77).

Since framing of assertions is explicitly build into the interpretation of the *safe* predicate, the soundness of the frame rule reduces to proving that if $\mathsf{mod}(\bar{\mathsf{s}}) \cap \mathsf{FV}(\mathsf{R}) = \emptyset$ then the interpretation of R is the same in initial and terminal stack of $\bar{\mathsf{s}}$. Lemma 74 expresses that mod is an over-approximation of the set of modified stack variables. Note that mod is only defined on sequences of statements (i.e., mod is *not* defined on thread call stacks) and this lemma is explicitly stated in terms of a sequence of statements. Since method and delegate calls introduce a new stack frame, which is restored upon their return, we can ignore the bodies of method and delegate calls. This is achieved using Lemma 73, which allows us strengthen the post-condition of $s_1; s_2$, by strengthening the post-condition of s_2 (under an arbitrary pre-condition).

Lemma 72 (FV)**.** *If* $\Gamma; \Delta \vdash \mathsf{M} : \tau$ *then*

$$
\begin{array}{l}
\forall \vartheta \in [\![\Gamma]\!].\ \forall \delta_1, \delta_2 \in [\![\Delta]\!]. \\
\quad (\forall x \in \mathsf{FV}(\mathsf{M}).\ \delta_1(x) = \delta_2(x)) \Rightarrow [\![\Gamma; \Delta \vdash \mathsf{M} : \tau]\!](\vartheta, \delta_1) = [\![\Gamma; \Delta \vdash \mathsf{M} : \tau]\!](\vartheta, \delta_2)
\end{array}
$$

Lemma 73.

$$
\begin{array}{l}
\forall i \in \mathbb{N}.\ \forall t \in \mathit{TId}.\ \forall l \in \mathit{Stack}.\ \forall s_1, s_2 \in \mathit{TCStack}.\ \forall p \in \mathit{Prop}.\ \forall q, q' \in \mathit{Stack} \to \mathit{Prop}. \\
\quad (\forall j \leq i.\ \forall r \in \mathit{Prop}.\ \forall l' \in \mathit{LState}.\ \mathit{safe}_j((t, l', s_2), r, q) \Rightarrow \mathit{safe}_j((t, l', s_2), r, q')) \ \Rightarrow \\
\quad\quad \mathit{safe}_i((t, l, s_1; s_2), p, q) \Rightarrow \mathit{safe}_i((t, l, s_1; s_2), p, q')
\end{array}
$$

Proof. By induction on i. As safety is trivial for $i = 0$, assume $i = j + 1$.

- Case $irr(s_1; s_2)$ or $s_1 = \varepsilon$:
  - then $s_1 = \varepsilon$
  - hence $safe_i((t, l, s_2), p, q)$ and thus

$$safe_i((t, l, s_2), p, q')$$

- Case $(t, l, s_1; s_2) \xrightarrow{a} \{(t, l', s')\} \uplus T$ and $s_1 \neq \varepsilon$:
  - by Lemma 16 there thus exists an s_1' such that $s' = s_1'; s_2$
  - furthermore, by safety there exists a

$$p' \in \{(t, l', s_1'; s_2)\} \uplus T \to Prop$$

    such that

$$\begin{array}{l} \forall z \in \{(t, l', s_1'; s_2)\} \uplus T.\ i \in stable(p'(z)) \\ a\ sat_i\ \{p\}\ \{p'(t, l', s_1'; s_2) * \circledast_{z \in T} p'(z)\} \\ safe_j((t, l', s_1'; s_2), p'(t, l', s_1'; s_2), q) \\ \forall z \in T.\ safe_j(z, p'(z), \lambda\_.\ \top) \end{array}$$

  - hence, by the induction hypothesis,

$$safe_j((t, l', s_1'; s_2), p'(t, l', s_1'; s_2), q')$$

□

Lemma 74 (Modifies clause)**.**

$$\begin{array}{l} \forall i \in \mathbb{N}.\ \forall t \in TId.\ \forall l \in Stack.\ \forall s \in seq\ Stm.\ \forall p \in Prop.\ \forall q \in Stack \to Prop.\ \forall x \in Var. \\ \quad x \in dom(l) \wedge x \notin \mathsf{mod}(s) \wedge safe_i((t, l, stm(s)), p, q)\ \Rightarrow \\ \qquad safe_i((t, l, stm(s)), p, \lambda l' \in LState.\ q(l') \wedge l(x) = l'(x)) \end{array}$$

Proof. By induction on i. As safety is trivial for $i = 0$, assume $i = j + 1$.

- Case $irr(t, l, s)$:
  - by assumption

$$p \sqsubseteq_i q(l)$$

  - and clearly,

$$q(l) = (q(l) \wedge l(x) = l(x))$$

- Case $(t, l, s) \xrightarrow{a} \{(t, l', s')\} \uplus T$:
  - Case SEQ followed by MCALL:

* there exists $y, z, \bar{u}, r \in \mathit{Var}$ and $s_b, s'' \in \mathit{seq}\ \mathit{Stm}$ such that

$$s = (y = z.\mathsf{m}(\bar{u}); s'') \qquad s' = s_b; \mathit{return}\ (l, y, r); s''$$

and $T = \emptyset$

* since $x \notin \mathsf{mod}(s)$, $x \neq y$ and $x \notin \mathsf{mod}(s'')$
* by safety of s there exists a $p' \in \mathit{Prop}$ such that

$$i \in \mathit{stable}(p') \qquad i \in (a\ \mathit{sat}\ \{i\}\{p\}p')$$

$$\mathit{safe}_j((t, l', s_b; \mathit{return}\ (l, y, r); s''), p', q)$$

* by Lemma 73 it thus suffices to prove

$\forall k \leq j.\ \forall p'' \in \mathit{Prop}.\ \forall l' \in \mathit{LState}.$
$\mathit{safe}_k((t, l', \mathit{return}\ (l, y, r); s''), p'', q)\ \Rightarrow$
$\mathit{safe}_k((t, l', \mathit{return}\ (l, y, r); s''), p'', \lambda l' \in \mathit{LState}.\ q(l') \wedge l(x) = l'(x))$

Case $(t, l', \mathit{return}\ (l, y, r); s'') \xrightarrow{id} (t, l[y \mapsto l'(r)], s'')$:

· by assumption there thus exists a $p''' \in \mathit{Prop}$ such that

$$k \in \mathit{stable}(p''') \qquad k \in (\mathit{id}\ \mathit{sat}\ \{p''\}\{p'''\})$$

$$\mathit{safe}_{k-1}((t, l[y \mapsto l'(r)], s''), p''', q)$$

· hence, by the induction hypothesis (as $x \notin \mathsf{mod}(s'')$):

$$\mathit{safe}_{k-1}((t, l[y \mapsto l'(r)], s''), p''', \lambda l' \in \mathit{LState}.\ q(l') \wedge (l[y \mapsto l'(r)])(x) = l'(x))$$

· hence, as $x \neq y$:

$$\mathit{safe}_{k-1}((t, l[y \mapsto l'(r)], s''), p''', \lambda l' \in \mathit{LState}.\ q(l') \wedge l(x) = l'(x))$$

– Case Seq followed by DCall: same as above
– Case Seq followed by Assign/FRead/FWrite/IfT/IfF/CAlloc/DAlloc:

* then there exists $s_1 \in \mathit{Stm}$ and $s_1', s_2 \in \mathit{seq}\ \mathit{Stm}$ such that

$$s = s_1; s_2 \qquad s' = s_1'; s_2 \qquad x \notin \mathsf{mod}(s') \qquad T = \emptyset$$

* by safety there thus exists a $p' \in \mathit{Prop}$ such that

$$i \in \mathit{stable}(p) \qquad i \in (a\ \mathit{sat}\ \{p\}\{p'\})$$

$$\mathit{safe}_j((t, l', s'), p', q)$$

∗ and, by the induction hypothesis,

$$safe_j((t, l', s'), p', \lambda l'' \in LState.\ q(l') \wedge l'(x) = l''(x))$$

∗ since $x \notin \mathsf{mod}(s)$, $l(x) = l'(x)$ and thus,

$$safe_j((t, l', s'), p', \lambda l'' \in LState.\ q(l') \wedge l(x) = l''(x))$$

– Case Fork: similar to the above cases

□

Lemma 75 (Frame rule)**.** *The following rule is sound.*

$$\frac{\Gamma \mid \Phi \vdash (\Delta).\{\mathsf{P}\}\bar{\mathsf{s}}\{\mathsf{Q}\} \qquad \mathsf{mod}(\bar{\mathsf{s}}) \cap \mathsf{FV}(\mathsf{R}) = \emptyset}{\Gamma \mid \Phi \vdash (\Delta).\{\mathsf{P} * \mathsf{R}\}\bar{\mathsf{s}}\{\mathsf{Q} * \mathsf{R}\}}$$

Proof.

- suppose $\vartheta \in [\![\Gamma]\!]$ and $i \in [\![\Phi]\!]$
- as safety is trivial for $i = 0$, assume $i = j + 1$
- let

$$p = \lambda\delta \in [\![\Delta]\!].\ [\![\Gamma; \Delta \vdash \mathsf{P} : \mathsf{Prop}]\!](\vartheta, \delta)$$
$$q = \lambda\delta \in [\![\Delta]\!].\ [\![\Gamma; \Delta \vdash \mathsf{Q} : \mathsf{Prop}]\!](\vartheta, \delta)$$
$$r = \lambda\delta \in [\![\Delta]\!].\ [\![\Gamma; \Delta \vdash \mathsf{R} : \mathsf{Prop}]\!](\vartheta, \delta)$$

- suppose $t \in TId$ and $l \in Stack$
- by assumption,

$$safe_i((t, l, \bar{\mathsf{s}}), ||p||_\Delta(l), ||q||_\Delta)$$

- hence, by Lemma 15,

$$safe_i((t, l, \bar{\mathsf{s}}), ||p||_\Delta(l) * ||r||_\Delta(l), \lambda l' \in Stack.\ ||q||_\Delta(l') * ||r||_\Delta(l))$$

- since $\mathsf{mod}(\mathsf{s}) \cap \mathsf{FV}(\mathsf{R}) = \emptyset$ it thus follows by Lemma 74 that,

$$safe_i((t, l, \bar{\mathsf{s}}), ||p||_\Delta(l) * ||r||_\Delta(l), \lambda l' \in Stack.\ ||q||_\Delta(l') * ||r||_\Delta(l) \wedge \bigwedge_{x \in \mathsf{FV}(\mathsf{R})} l'(x) = l(x))$$

- hence, by Lemma 72,

$$safe_i((t, l, \bar{\mathsf{s}}), ||p||_\Delta(l) * ||r||_\Delta(l), \lambda l' \in Stack.\ ||q||_\Delta(l') * ||r||\Delta(l'))$$

□

Lemma 76.

$$\forall i \in \mathbb{N}.\ \forall t \in \mathit{TId}.\ \forall l \in \mathit{Stack}.\ \forall s \in \mathit{seq}\ \mathit{Stm}.\ \forall p \in \mathit{Prop}.\ \forall q \in \mathit{Stack} \to \mathit{Prop}.$$
$$\mathit{safe}_i((t, l, \mathit{stm}(s)), p, q) \Rightarrow \mathit{safe}_i((t, l, \mathit{stm}(s)), p, \lambda\_.\ \top)$$

Lemma 77 (Forking)**.** *The following proof rule is sound.*

$$\frac{\Gamma \mid \Phi \vdash \triangleright(\mathsf{C.m} : (-).\{\mathsf{P}\}\{\mathsf{ret.Q}\}) \qquad \Gamma \vdash \mathsf{C} : \mathsf{Class} \qquad \Gamma \vdash \mathsf{m} : \mathsf{Method} \qquad \Gamma, \Delta \mid \Phi \vdash \mathit{stable}(\mathsf{P}) \wedge \mathit{stable}(\mathsf{Q})}{\Gamma \mid \Phi \vdash (\Delta).\{\mathsf{P}[\mathsf{y}/\mathtt{this}] * \mathsf{x} \mapsto \mathsf{y.m} * \mathsf{y} : \mathsf{C}\}\mathit{fork}(\mathsf{x})\{\mathsf{emp}\}}$$

Proof.

- suppose $\vartheta \in [\![\Gamma]\!]$, $i \in [\![\Phi]\!]$, $t \in \mathit{TId}$, and $l \in \mathit{Stack}$
- as safety is trivial if $i = 0$ assume $i = j + 1$
- suppose $(t, l, \mathtt{fork}(\mathsf{x})) \xrightarrow{a} \{(t, l', s')\} \uplus T$
- then there exists $\mathsf{C} \in \mathit{CName}$, $o \in \mathit{OId}$, $\mathsf{m}_b \in \mathit{MName}$, $\mathsf{r}, \bar{\mathsf{y}} \in \mathit{Var}$, $l'' \in \mathit{Stack}$, $t' \in \mathit{TId}$, $\bar{\mathsf{s}} \in \mathit{Stm}$ such that,

  $$a = \mathit{ctype}(l(\mathsf{x}), \mathsf{C}, o, \mathsf{m}_b) \qquad T = \{(t', l'', \bar{\mathsf{s}})\} \qquad l'' = [\mathtt{this} \mapsto o, \bar{\mathsf{y}} \mapsto \mathit{null}]$$

  and

  $$\mathsf{body}(\mathsf{C}, \mathsf{m}_b) = \mathtt{void}\ \mathsf{m}_b()\{\overline{\mathsf{Cy}}; \bar{\mathsf{s}}; \mathtt{return}\ \mathsf{r}\} \qquad l' = l \qquad s' = \varepsilon$$

- let

  $$\begin{aligned}
  C &= [\![\Gamma \vdash \mathsf{C} : \mathsf{Class}]\!](\vartheta) \\
  m &= [\![\Gamma \vdash \mathsf{m} : \mathsf{Method}]\!](\vartheta) \\
  p_1 &= \lambda\delta \in [\![\mathtt{this}]\!].\ [\![\Gamma; \mathtt{this} \vdash \mathsf{P} : \mathsf{Prop}]\!](\vartheta, \delta) \\
  q_1 &= \lambda\delta \in [\![\mathtt{this}, \mathsf{r}]\!].\ [\![\Gamma; \mathtt{this}, \mathsf{r} \vdash \mathsf{Q}[\mathsf{r}/\mathsf{ret}] : \mathsf{Prop}]\!](\vartheta, \delta) \\
  p_2 &= \lambda\delta \in [\![\Delta]\!].\ [\![\Gamma; \Delta \vdash \mathsf{P}[\mathsf{y}/\mathtt{this}] : \mathsf{Prop}]\!](\vartheta, \delta) \\
  p_3 &= \lambda\delta \in [\![\Delta]\!].\ [\![\Gamma; \Delta \vdash \mathsf{x} \mapsto \mathsf{y.m} * \mathsf{y} : \mathsf{C} : \mathsf{Prop}]\!](\vartheta, \delta) \\
  p_4 &= \lambda\delta \in [\![\Delta]\!].\ [\![\Gamma; \Delta \vdash \mathsf{P}[\mathsf{y}/\mathtt{this}] * \mathsf{x} \mapsto \mathsf{y.m} * \mathsf{y} : \mathsf{C} : \mathsf{Prop}]\!](\vartheta, \delta) \\
  &= \lambda\delta \in [\![\Delta]\!].\ p_2(\delta) * p_3(\delta) \\
  q_2 &= \lambda\delta \in [\![\Delta]\!].\ [\![\Gamma; \Delta \vdash \mathsf{emp} : \mathsf{Prop}]\!](\vartheta, \delta) \\
  &= \lambda\delta \in [\![\Delta]\!].\ \mathit{emp}
  \end{aligned}$$

- then

  $$i \in (\mathit{ctype}(l(\mathsf{x}), \mathsf{C}, o, \mathsf{m}_b)\ \mathit{sat}\ \{||p_2||_\Delta(l)\}\{||p_2||_\Delta(l) * l(\mathsf{y}) = o * \mathsf{m}_b = m * \mathsf{C} = C\})$$

- suppose $l(\mathsf{y}) = o$, $\mathsf{m}_b = m$, and $\mathsf{C} = C$ (safety follows trivially otherwise)
- hence, as $l(\mathsf{y}) = o$,
$$||p_2||_\Delta(l) = ||p_1||_{\mathtt{this}}(l'')$$
- furthermore, by assumption,
$$i \in \mathit{stable}(||p_2||_\Delta(l)) \qquad \mathit{safe}_j((t', l'', s_b), ||p_1||_{\mathtt{this}}(l''), ||q||_{\mathtt{this},\mathsf{r}})$$
- and thus
$$\mathit{safe}_j((t', l'', s_b), ||p_2||_\Delta(l), ||q||_{\mathtt{this},\mathsf{r}})$$
- hence, by Lemma 76,
$$\mathit{safe}_j((t', l'', s_b), ||p_2||_\Delta(l), \lambda\_.\ \top)$$
- and
$$\mathit{safe}_j((t, l, \varepsilon), l(\mathsf{y}) = o * m_b = m * C = T, ||q_2||_\Delta)$$
as $p \sqsubseteq \mathit{emp}$ for all $p \in \mathit{Prop}$

□

Theorem 2 (Soundness)**.** *The specification logic and the assertion logic is sound:*

If $\Gamma; \Delta \mid \Phi \mid \mathsf{P} \vdash \mathsf{Q}$ *then* $[\![\Gamma; \Delta \mid \Phi \mid \mathsf{P} \vdash \mathsf{Q}]\!]$.
If $\Gamma \vdash \Phi \vdash \mathsf{S}$ *then* $[\![\Gamma \vdash \Phi \vdash \mathsf{S}]\!]$.

Theorem 3. *If* $\Gamma \mid \Phi \vdash (\Delta).\{\mathsf{P}\}\bar{\mathsf{s}}\{\mathsf{Q}\}$ *then for all*

$$\vartheta \in [\![\Gamma]\!] \qquad i \in [\![\Phi]\!](\vartheta) \qquad l \in [\![\Delta]\!]$$

and

$$t \in \mathit{TId} \qquad j \leq i \qquad h \in \lfloor ||\lambda\delta.\ [\![\Gamma; \Delta \vdash \mathsf{P} : \mathit{Prop}]\!](\vartheta, \delta)||_\Delta(l) \rfloor_i$$

if

$$(h, \{(t, l, \mathit{stm}(\bar{\mathsf{s}}))\}) \to^j (h', \{(t, l', \varepsilon)\} \uplus T')$$

and T' *is irreducible then* $h' \in \lfloor ||\lambda\delta.\ [\![\Gamma; \Delta \vdash \mathsf{Q} : \mathit{Prop}]\!](\vartheta, \delta)||_\Delta(l') \rfloor_{i-j}$.

References

[1] L. Birkedal, R. Møgelberg, J. Schwinghammer, and K. Støvring. First steps in synthetic guarded domain theory: step-indexing in the topos of trees. In *Proceedings of LICS*, 2011.

[2] L. Birkedal, B. Reus, J. Schwinghammer, K. Støvring, J. Thamsborg, and H. Yang. Step-indexed Kripke models over recursive worlds. In *Proceedings of POPL*, 2011.

[3] T. Dinsdale-Young, L. Birkedal, P. Gardner, M. Parkinson, and H. Yang. Views: Compositional Reasoning for Concurrent Programs. In *Proceedings of POPL*, 2013.

[4] J. Schwinghammer, L. Birkedal, B. Reus, and H. Yang. Nested Hoare triples and frame rules for higher-order store. In *Proceedings of CSL*, Apr. 2009.

[5] J. Schwinghammer, L. Birkedal, B. Reus, and H. Yang. Nested Hoare Triples and Frame Rules for Higher-Order Store. *LMCS*, 7(3:21), 2011.

[6] K. Svendsen, L. Birkedal, and M. Parkinson. A Specification of the Joins Library in Higher-order Separation Logic, 2012.

[7] K. Svendsen, L. Birkedal, and M. Parkinson. Verification of the joins library in higher-order separation logic. Technical report, IT University of Copenhagen, 2012. Available at `www.itu.dk/people/kasv/joins-tr.pdf`.

Chapter 3

The Joins library

Joins: A Case Study in Modular Specification of a Concurrent Reentrant Higher-order Library

Kasper Svendsen[1], Lars Birkedal[1], and Matthew Parkinson[2]

[1] IT University of Copenhagen, {kasv,birkedal}@itu.dk
[2] Microsoft Research Cambridge, mattpark@microsoft.com

Abstract. We present a case study of formal specification for the C♯ joins library, an advanced concurrent library implemented using both shared mutable state and higher-order methods. The library is specified and verified in HOCAP, a higher-order separation logic extended with a higher-order variant of concurrent abstract predicates.

1 Introduction

It is well-known that modular specification and verification of concurrent higher-order imperative programs is very challenging. In the last decade good progress has been made on reasoning about subsets of these programming language features. For example, higher-order separation logic with nested triples has proved useful for modular specification and verification of higher-order imperative programs that use state with little sharing, e.g., [23, 16, 15]. Nested triples support specification of higher-order methods and higher-order quantification allows library specifications to abstract over the internal state maintained by the library and the state effects of function arguments.

Likewise, concurrent abstract predicates [7] has proved useful for reasoning about shared mutable data structures in a concurrent setting. Concurrent abstract predicates (CAP) extends separation logic with protocols governing access to shared mutable state. Thus CAP supports modular specification of shared mutable data structures that abstract over the *internal sharing*, e.g., [5]. However, CAP does not support modular reasoning about *external sharing* – the sharing of *other* mutable data structures through a shared mutable data structure. For instance, CAP does not support modular reasoning about locks[3] – the canonical example of a shared mutable data structure used to facilitate external sharing.

We have recently proposed HOCAP [26], a new program logic which combines higher-order separation logic with concurrent abstract predicates and extends concurrent abstract predicates with *state-independent higher-order protocols*. To reason about external sharing through a data structure, we parameterise the specification of the data structure with assertions that clients can instantiate to describe the resources they wish to share through the data structure. Higher-order protocols allow us to impose protocols on these external resources when

[3] See Section 6 for a discussion of this issue.

reasoning about the implementation of the data structure. State-independent higher-order protocols allow us to reason about non-circular external sharing patterns.

HOCAP is thus intended as a general purpose program logic for modular specification and verification of concurrent higher-order imperative programs with support for modular reasoning about *both* internal and external sharing. We have previously verified simple examples in HOCAP. In this paper we report on an extensive case study of a sophisticated and realistic library that combines all these challenges in one, to test whether HOCAP can in fact be used to give an abstract formal specification.

In particular, we explore how to give a modular specification to a concurrent library that features internal sharing and is used to facilitate external sharing. Clients interact with the library using reentrant callbacks. The specification should thus abstract over the internal state while allowing abstract reasoning about external sharing through the library and reentrant calls back into the library. Furthermore, the specification should of course be strong enough to reason about clients, and weak enough to allow the implementation of the library to be verified against the specification.

Our case study of choice is the C♯ joins library [20]. The joins library, which is based on the join calculus [8, 9], provides a declarative concurrency model based on message passing. Declarative message patterns are used to specify synchronisation conditions and function arguments are used to specify synchronisation actions. Synchronisation actions might themselves cause new messages to be sent, leading to reentrant callbacks. The joins concurrency model is useful for defining new synchronisation primitives – i.e., to facilitate external sharing. Finally, the library itself is implemented using internal state.

In this paper we present a formal specification of a subset of the C♯ joins library in HOCAP. The specification is expressed in terms of the high-level join primitives exposed by the library and hides all internal state from clients. Moreover, we test the specification of the joins library by verifying a number of synchronisation primitives for which there are already accepted specifications in the literature. For example, we verify that a reader-writer lock implemented using joins can be proved to satisfy the standard separation logic specification for a reader-write lock. We have chosen to focus on synchronisation primitives because synchronisation primitives are specifically designed to facilitate external sharing.

In addition to its role as a case study of a higher-order reentrant concurrent library, the specification of the joins library is itself interesting. The main idea behind the specification is to allow clients of the joins library to impose ownership transfer protocols at the level of the join primitives exposed by the library. As illustrated with several examples, this leads to natural and short proofs of synchronisation primitives implemented using the joins library.

We have also verified a simple lock-based implementation of the joins library. However, in this paper we focus on the joins specification and the use thereof, since the main point is to investigate how HOCAP can be used to give abstract specifications for concurrent higher-order imperative libraries. We refer

the interested reader to the accompanying technical report for details about the verification of the joins implementation [25]. This paper does not require the reader to understand all the details of HOCAP.

Outline. The remainder of the paper is organised as follows. In Section 2 we give an extensive introduction to the joins library using a series of examples to explain each feature of the library. Along the way, we sketch how one can reason informally, in separation-logic style, about the correctness of the applications. In Section 3 we summarise the necessary bits of HOCAP. This leads us to Section 4, where we introduce the formal specification of the joins library. In Section 5 we revisit a couple of the example applications and show how the informal proof sketches from Section 2 can be turned into formal proofs using the formal specification from Section 4. Finally, we evaluate and discuss the case study in Section 6.

2 Introducing joins

The joins concurrency model is based on the concept of messages, which are used both for synchronisation and communication between threads. Conceptually, a join instance consists of a single message pool and a number of *channels* for adding messages to this pool. Channels come in two varieties, *synchronous* and *asynchronous*. Sending a message via a synchronous channel adds the message to the message pool and blocks the sender until the message has been received. Asynchronous channels simply add messages to the message pool, without blocking the sender.

The power of the joins calculus stems from how messages are received. One declares a set of *chords*, each consisting of a *pattern* (a condition on the message pool) and a continuation. When a pattern matches a set of messages in the message pool, the chord may *fire*, causing the continuation to execute. Crucially, once a chord fires, the messages that matched the pattern are removed from the message pool *atomically*, making them unavailable for future matches. Upon termination of the continuation, the clients that added the removed messages via synchronous channels are woken up and allowed to continue. We say that a message has been *received* when it has been matched by a chord and the chord continuation has terminated.

In the rest of this section we introduce the C# joins library, one feature at a time. Each new feature is introduced with a joins example of a synchronisation primitive implemented using this feature. For each example, we sketch an informal proof of the synchronisation primitive in separation logic. The examples thus serve both to introduce the joins library and motivate the main ideas behind our formal specification of the joins library.

We take as a starting point Russo's joins library for C$^\sharp$ [20], with a slightly simplified API. In particular, we have omitted value-carrying channels, as value-carrying channels do not add any conceptual difficulties.

2.1 Synchronous channels

Sending a message via a synchronous channel causes the sender to block until the message has been received. To illustrate, we consider the example of a 2-barrier – an asymmetric barrier restricted to two clients.

Implementation. One can implement a 2-barrier as a joins instance with two synchronous channels – one for each client to signal its arrival. Clients should block at the barrier until both clients have signalled their arrival. This can be achieved with a single chord with a pattern that allows it to fire when there is a pending message on both channels (i.e., when both clients have arrived). The C♯ code for a 2-barrier is given in Figure 1.

The TwoBarrier constructor creates a join instance, j, and two synchronous channels, ch1 and ch2, attached to the underlying message pool of this join instance. Next, the constructor creates a pattern p that matches any pair of messages in the message pool consisting of a ch1 message (i.e., a message added via the ch1 channel) and a ch2 message. Lastly, it registers this pattern as a chord without a continuation. Hence, this chord may fire when there is a pending message on both channels and when it fires, it atomically removes and receives these two messages from the message pool. Each Arrive method signals the client's arrival by sending a message on the corresponding channel using the Call method.

```
class TwoBarrier {
  private SyncChannel ch1;
  private SyncChannel ch2;

  public TwoBarrier() {
    Join j = new Join();
    ch1 = new SyncChannel(j);
    ch2 = new SyncChannel(j);
    Pattern p = j.When(ch1).And(ch2);
    p.Do();
  }

  public void Arrive1() { ch1.Call(); }
  public void Arrive2() { ch2.Call(); }
}
```

Fig. 1. Joins 2-barrier implementation.

All the examples we consider in this article follow the same structure as the above example: the constructor creates a join instance with accompanying channels and registers a number of chords. After this initialisation phase, the chords and channels stay fixed and interaction with the joins instance is limited to the sending of messages.

We now sketch a proof of this 2-barrier implementation using separation logic. Recall that separation logic assertions, say P and Q, describe and assert ownership of resources and that $P * Q$ holds if P and Q describe (conceptually) disjoint resources. The logic will be introduced in greater detail in Section 4 when we get to the formal specification and formal reasoning.

Desired specification. From the point of view of resources, a barrier allows clients to exchange resources. We call these resources external as they are typically external to the barrier data structure itself. On arrival at the barrier each client may transfer ownership of some resource to the barrier, which is then redistributed atomically once both clients have arrived. For the purpose of this

introduction we will make the simplifying assumption that each client transfers the same resource to the barrier on each arrival and that these resources are redistributed in the same way at each round of synchronisation. In Section 5.2 we consider a general specification without these simplifying assumptions.

Under these assumptions we can specify the barrier in terms of two predicates, B_i^{in} and B_i^{out}, where B_i^{in} describes the resources client i transfers to the barrier upon arrival, and B_i^{out} describes the resources client i expects to receive back from the barrier upon leaving. These predicates thus describe the external resources clients intend to share through the barrier. Since a barrier can only redistribute resources (i.e., it cannot create resources out of thin air), the combined resources transferred to the barrier must imply the combined resources transferred back from the barrier: $B_1^{\mathsf{in}} * B_2^{\mathsf{in}} \Rightarrow B_1^{\mathsf{out}} * B_2^{\mathsf{out}}$.

The *client* of the barrier is thus free to pick any B_i^{in} and B_i^{out} predicates satisfying the above redistribution property. We can now express the expected specification of a 2-barrier b in terms of these abstract predicates:

$$\{B_1^{\mathsf{in}}\}\ \mathsf{b.Arrive1()}\ \{B_1^{\mathsf{out}}\} \qquad\qquad \{B_2^{\mathsf{in}}\}\ \mathsf{b.Arrive2()}\ \{B_2^{\mathsf{out}}\}$$

That is, for client 1 to arrive at the barrier (i.e., to call Arrive1), it has to provide the resource described by B_1^{in}, and if the call to Arrive1 terminates (i.e., client 1 has left the barrier), it will have received the resource described by B_1^{out}.

Proof sketch. The main idea behind our specification of the joins library is to allow clients to impose an ownership transfer protocol on messages. An ownership transfer protocol consists of a *channel precondition* and a *channel postcondition* for each channel. The channel precondition describes the resources the sender is required to transfer to the recipient when sending a message on the channel. The channel postcondition describes the resources the recipient is required to transfer to the sender upon receiving the message.

In the 2-barrier example, sending a message on a channel corresponds to signalling one's arrival at the barrier. The channel preconditions of the barrier thus describe the resources clients are required to transfer to the barrier upon their arrival. Hence, we take each channel precondition to be the corresponding B^{in} predicate: $P_{\mathsf{ch1}} = B_1^{\mathsf{in}}$ and $P_{\mathsf{ch2}} = B_2^{\mathsf{in}}$. Throughout this section we use the notation P_{ch} to refer to the channel precondition of channel ch and Q_{ch} to refer to the channel postcondition.

The barrier implementation features a single chord that matches and receives both arrival messages, once both clients have arrived. The channel postconditions of the barrier thus describes the resources the barrier is required to transfer back to the clients, once both clients have arrived. We thus take each channel postcondition to be the corresponding B^{out} predicate: $Q_{\mathsf{ch1}} = B_1^{\mathsf{out}}$ and $Q_{\mathsf{ch2}} = B_2^{\mathsf{out}}$.

One can thus think of the channel pre- and postconditions as specifications for channels. Since the channel postcondition describes the resources transferred back to the sender *once* its message has been received, one should think of it as a partial correctness specification. In particular, without any chords to receive messages on a given channel we can pick any channel postcondition, as no message sent on that channel will ever be received. Conversely, whenever

we add a new chord we have to prove that it satisfies the chosen ownership transfer protocol. For chords without continuations, this reduces to proving that the preconditions of the channels that match the chord's pattern imply the postconditions of these channels.

The 2-barrier consists of a single chord that matches any pair of messages consisting of a ch1 message and a ch2 message. Correctness thus reduces to proving $P_{\mathsf{ch1}} * P_{\mathsf{ch2}} \Rightarrow Q_{\mathsf{ch1}} * Q_{\mathsf{ch2}}$, which follows from the assumed redistribution property.

2.2 Asynchronous channels

The previous example illustrated the use of synchronous channels that block the sender until its message has been received. The joins library also supports *asynchronous* channels, allowing messages to be sent without blocking the sender. A lock is a simple example that illustrates the use of both asynchronous and synchronous channels. Acquiring a lock must wait for the previous thread using the lock to finish: it is synchronous. However, releasing a lock need not wait for the next thread to attempt to acquire it: it is asynchronous.

Implementation. We can implement a lock using the joins library as follows:

We use two channels acq and rel to represent the two actions one can perform on a lock. The join instance has a single chord with a pattern that matches any pair of messages consisting of an acq message and a rel message. Thus, to acquire the lock, a thread sends a message on the acq channel; the call will block until the chord fires, which can only happen if there is a rel message in the message pool. The lock is thus unlocked if and only if there is a pending rel message in the message pool. The release can happen asynchronously; it does not have to wait for the next thread to attempt to acquire the lock.

The lock is initially unlocked by calling rel.

```
class Lock {
  private SyncChannel acq;
  private AsyncChannel rel;

  public Lock() {
    Join j = new Join();
    acq = new SyncChannel(j);
    rel = new AsyncChannel(j);
    j.When(acq).And(rel).Do();
    rel.Call();
  }

  public void Acquire() { acq.Call(); }
  public void Release() { rel.Call(); }
}
```

Fig. 2. A Joins implementation of a lock.

Desired specification. Locks are used to ensure exclusive access to some shared resource. We can specify a lock in separation logic in terms of an abstract resource predicate R (picked by the client of the lock) as follows:

$$\{R\}\ \textsf{new Lock()}\ \{\text{emp}\} \qquad \{\text{emp}\}\ \textsf{l.Acquire()}\ \{R\} \qquad \{R\}\ \textsf{l.Release()}\ \{\text{emp}\}$$

When the lock is unlocked the resource described by R is owned by the lock. Upon acquiring the lock, the client takes ownership of R, until it releases the lock again. Since the lock is initially unlocked, creating a new lock requires ownership of R to be transferred to the lock. This is the standard separation logic

specification for a lock [17, 10, 11]. Here R thus describes the external resources shared through the lock.

Proof sketch. Informally, we can understand the rel message as moving the resource protected by the lock from the thread to the join instance, and the acq message as doing the converse. This can be stated more formally using channel pre- and postconditions as follows: $P_{\mathsf{acq}} = \text{emp}$, $Q_{\mathsf{acq}} = R$, $P_{\mathsf{rel}} = R$, and $Q_{\mathsf{rel}} = \text{emp}$.

Recall that channel postconditions describe the resources the recipient is required to transfer to the sender upon receiving the message. Since the sender of a message on an asynchronous channel has no way of knowing if its message has been received, channel postconditions do not make sense for asynchronous channels. We thus require channel postconditions for asynchronous channels to be empty, emp.

As before, to prove that the acq and rel chord satisfies the channel postconditions, we have to show that the combined channel preconditions imply the combined channel postconditions: $P_{\mathsf{acq}} * P_{\mathsf{rel}} \Rightarrow Q_{\mathsf{acq}} * Q_{\mathsf{rel}}$. This follows directly from the fact that $*$ is commutative.

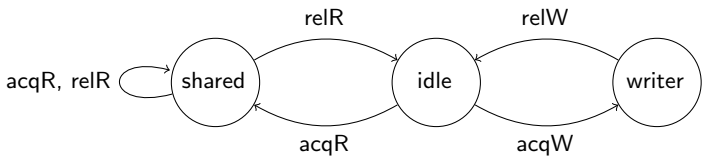
2.3 Continuations

So far, every chord we have considered has simply matched and removed messages from the message pool. In general, a chord can have a continuation that is executed when the chord fires, before any blocked synchronous senders are allowed to continue.

Continuations can, for instance, be used to automatically send a message on a certain channel when a chord fires. Thus they can be used to encode a state machine. Moreover, one can also ensure that a state of the state machine is correlated with the history or state of the synchronisation primitive that one is implementing. To illustrate, we extend the lock from the previous example into a biased reader/writer lock.[4]

A reader/writer lock [4] generalises a lock by introducing read-only permissions. This allows multiple readers to access a shared resource concurrently. To determine whether a read or write access request should be granted, three states suffice: (idle) no readers or writers, (writer) exactly one writer, or (shared) one or more readers. In the idle state there are no readers or writers, so it is safe to grant both read and write access. In the shared state, as one client has already been granted read access, it is only safe to grant read access. We can express this as a state machine as follows:

[4] Biased here means that this reader/writer implementation may starve the writer thread. It is possible to extend this implementation into an unbiased reader/writer lock by introducing an additional asynchronous channel to distinguish between whether or not there are any pending writers when a reader request has been granted.



Here acqR and acqW refers to the acquire read and acquire write operation, and relR and relW refers to the release read and release write operation.

Implementation. The idea is to encode this state machine using three asynchronous channels, idle, shared, and writer, with the invariant that there is at most one pending asynchronous message in the message pool at any given time. This gives a direct encoding of the three states in the above state machine, and adds a fourth intermediate state (when there is no pending message on any of the three asynchronous channels). The intermediate state is necessary for the implementation, as it does not transition atomically between the states of the above state machine. The joins implementation is given below.

```
class RWLock {
  private SyncChannel acqR, acqW, relR, relW;
  private AsyncChannel idle, shared, writer;
  private int readers = 0;

  public RWLock() {
    Join j = new Join();
    // ... initialise channels ...

    j.When(acqR).And(idle).Do(AcqR);
    j.When(acqR).And(shared).Do(AcqR);
    j.When(relR).And(shared).Do(RelR);
    j.When(acqW).And(idle).Do(writer.Call);
    j.When(relW).And(writer).Do(idle.Call);

    idle.Call();
  }

  private void AcqR() {
    readers++;
    shared.Call();
  }

  private void RelR() {
    if (--readers == 0)
      idle.Call();
    else
      shared.Call();
  }

  public void AcquireR() { acqR.Call(); }
  public void AcquireW() { acqW.Call(); }
  public void ReleaseR() { relR.Call(); }
  public void ReleaseW() { relW.Call(); }
}
```

We use three asynchronous channels to encode the current state in the above state machine and thus to determine whether a read or write access can be granted. In addition, we use the readers field to count the actual number of readers, to determine which state to transition to when releasing a reader. Note that the continuation given to Do is a named C♯ delegate, and that in all five cases, the given continuation sends a message on an asynchronous channel. These calls are *reentrant* calls back into the joins library, making these continuations *reentrant callbacks*.

Note further that all five chords consume exactly one asynchronous message and sends exactly one asynchronous message. Between consuming and sending the asynchronous message, there are no pending asynchronous messages and the reader/writer is in the previously mentioned fourth state. Hence, between

consuming and sending an asynchronous message, no other chord can fire and the currently executing continuation has exclusive access to the internal state of the reader/writer lock (i.e., the readers field).

Desired specification. The standard separation logic specification for a reader/writer lock is expressed using counting permissions [2]. Counting permissions allow a full write permission to be split into any number of read permissions, counting the total number of read permissions, to allow them to be joined up to a full write permission later. The standard specification is given below in terms of an abstract resource predicate for writing to the resource R_{write} and an abstract resource predicate for reading the resource R_{read}.

$$\begin{array}{rcl} \{R_{write}\}\ \textsf{new RWLock()} & & \{\text{emp}\} \\ \{\text{emp}\} & \textsf{l.AcquireR()} & \{R_{read}\} \\ \{\text{emp}\} & \textsf{l.AcquireW()} & \{R_{write}\} \\ \{R_{read}\} & \textsf{l.ReleaseR()} & \{\text{emp}\} \\ \{R_{write}\} & \textsf{l.ReleaseW()} & \{\text{emp}\} \end{array} \tag{1}$$

To avoid introducing counting permissions directly, we specify the reader/write lock in terms of an additional family of abstract resource predicates $R(n)$, indexed by $n \in \mathbb{N}$, satisfying that $R(0)$ is the full write permission R_{write}, and $R(n)$ is the permission left after splitting off n read permissions. Thus R should satisfy, $\forall n \in \mathbb{N}.\ R(n) \Leftrightarrow R_{read} * R(n+1)$ and $R(0) \Leftrightarrow R_{write}$. Note that a client of the reader/writer lock is free to pick any R_{write}, R_{read} and R that satisfies these two properties.

Proof sketch. The three asynchronous channels encode the current state of the reader/writer lock. The channel preconditions of the three asynchronous channels thus describe the resources owned by the reader/writer lock in the idle, shared and writer state, respectively. In the idle state (no readers or writers), the reader/writer lock owns the readers field and the full write permission, and the readers field contains 0. In the shared state (one or more readers), the reader/writer lock owns the readers field and the remaining permission after splitting off n read permissions and the readers field contains n. Lastly, in the writer state (exactly one writer), the writer owns the full resource and the reader/writer lock only owns the readers field.

$$P_{\textsf{idle}} = \textsf{readers} \mapsto 0 * R(0) \qquad P_{\textsf{writer}} = \textsf{readers} \mapsto 0$$
$$P_{\textsf{shared}} = \exists n \in \mathbb{N}.\ n > 0 * \textsf{readers} \mapsto n * R(n)$$

Since idle, shared, and writer are asynchronous, their channel postconditions must be empty (as explained earlier).

For the synchronous channels we can read off their channel pre- and postconditions directly from the desired specification (1):

$$P_{\textsf{acqR}} = \text{emp} \qquad Q_{\textsf{acqR}} = R_{read} \qquad P_{\textsf{acqW}} = \text{emp} \qquad Q_{\textsf{acqW}} = R(0)$$
$$P_{\textsf{relR}} = R_{read} \qquad Q_{\textsf{relR}} = \text{emp} \qquad P_{\textsf{relW}} = R(0) \qquad Q_{\textsf{relW}} = \text{emp}$$

To register a chord without a continuation we had to show that the combined channel preconditions *implied* the combined channel postconditions. What about

the present case with a proper continuation? Since the continuation runs before the release of any blocked synchronous callers, we have to show that the continuation *transforms* the combined channel preconditions to the combined channel postconditions. For the reader/writer lock we thus have to show the proof obligations on the left in Figure 3. These proof obligations are all completely standard and mostly trivial separation logic proofs. For instance, the proof of the first obligation is given on the right in Figure 3. Note that in this proof we use the

$$
\begin{array}{ccc}
\{P_{\mathsf{acqR}} * P_{\mathsf{idle}}\} & \mathsf{AcqR()} & \{Q_{\mathsf{acqR}} * Q_{\mathsf{idle}}\} \\
\{P_{\mathsf{acqR}} * P_{\mathsf{shared}}\} & \mathsf{AcqR()} & \{Q_{\mathsf{acqR}} * Q_{\mathsf{shared}}\} \\
\{P_{\mathsf{relR}} * P_{\mathsf{shared}}\} & \mathsf{RelR()} & \{Q_{\mathsf{relR}} * Q_{\mathsf{shared}}\} \\
\{P_{\mathsf{acqW}} * P_{\mathsf{idle}}\} & \mathsf{writer.Call()} & \{Q_{\mathsf{acqW}} * Q_{\mathsf{idle}}\} \\
\{P_{\mathsf{relW}} * P_{\mathsf{writer}}\} & \mathsf{idle.Call()} & \{Q_{\mathsf{relW}} * Q_{\mathsf{writer}}\}
\end{array}
$$

$$
\begin{array}{l}
\{P_{acqR} * P_{idle}\} \\
\{readers \mapsto 0 * R(0)\} \\
\quad \mathsf{readers{+}{+};} \\
\{readers \mapsto 1 * R(0)\} \\
\{readers \mapsto 1 * R_{\mathsf{read}} * R(1)\} \\
\quad \mathsf{shared.Call();} \\
\{R_{\mathsf{read}}\} \\
\{Q_{acqR} * Q_{idle}\}
\end{array}
$$

Fig. 3. Left: Proof obligations for the reader/writer lock chords. Right: A proof sketch for the first proof obligation of the reader/writer lock.

channel pre- and postcondition of the shared channel. These proofs thus have a similar character to partial correctness proofs of a recursive method, where one is allowed to assume the specification of a method while proving that its body satisfies the assumed specification. Here we assume the shared channel obeys the chosen ownership transfer protocol while proving that the first chord obeys the chosen protocol.

2.4 Nonlinear patterns

The public interface of the 2-barrier in Section 2.1 is slightly non-standard, as it has two distinct arrival methods. A more standard barrier interface would provide a common Arrive method, for both clients. The joins library also supports an implementation of a barrier with such an interface, through the use of *nonlinear patterns*. Nonlinear patterns match multiple messages from the same channel.

Implementation. We can thus implement a more standard 2-barrier as a joins instance with a single synchronous arrival channel and a single chord with a nonlinear pattern that matches two messages on the arrival channel. Clearly this generalises to an n-barrier, which can be implemented as follows.

```
class Barrier {
  private SyncChannel arrive;

  public Barrier(int n) {
    Join j = new Join(); arrive = new SyncChannel(j); Pattern p = j.When(arrive);
    for(int i = 1; i < n; i++) { p = p.And(arrive); };
    p.Do();
  }

  public void Arrive() { arrive.Call(); }
```

}

This code registers a single chord with a pattern that matches n messages on the synchronous arrive channel.

Desired specification. As before, assume predicates B_i^{in} and B_i^{out} (picked by the client), where B_i^{in} describes the resources client i transfers to barrier upon arrival and B_i^{out} describes the resources client i expects to receive back from the barrier upon leaving. These predicates should satisfy the following redistribution property, $\circledast_{i\in\{1,\ldots,n\}} B_i^{\mathsf{in}} \Rightarrow \circledast_{i\in\{1,\ldots,n\}} B_i^{\mathsf{out}}$, to allow the barrier to redistribute the combined resources, once every client has arrived.

From the informal description of B_i^{in} and B_i^{out} one might thus expect an n-barrier b to satisfy the following specification:

$$\{B_i^{\mathsf{in}}\}\ \mathsf{b.Arrive()}\ \{B_i^{\mathsf{out}}\}$$

That is, if a client transfers B_i^{in} to the barrier upon arrival, it should receive back B_i^{out} from the barrier upon leaving. However, this specification is not quite right. In particular, what prevents client i from impersonating client j when it arrives at the barrier? To apply the redistribution property to the combined resources transferred to the barrier we need to ensure that when client i arrives at the barrier, it actually transfers B_i^{in} to the barrier, even if it also happens to own B_j^{in}. Hence, while the barrier *implementation* no longer distinguishes between clients, we still need a way to distinguish clients *logically*. We can achieve this by introducing a client identity predicate, $\mathsf{id}(i)$ to assert that the owner is client i. By making this predicate non-duplicable, we can enforce that clients cannot impersonate each other.

We can now express a correct barrier specification in terms of this id predicate as follows:

$$\{\mathrm{emp}\}\ \textbf{new}\ \mathsf{Barrier(n)}\ \{\circledast_{i\in\{1,\ldots,n\}}\mathsf{id}(i)\} \quad \{B_i^{\mathsf{in}} * \mathsf{id}(i)\}\ \mathsf{b.Arrive()}\ \{B_i^{\mathsf{out}} * \mathsf{id}(i)\}$$

Upon creation of a new n-barrier we get back n id assertions. These are then distributed to each client to witness their identity when they arrive at the barrier.

Proof sketch. Our proof sketch of the 2-barrier in Section 2.1 exploited that the implementation used distinct channels to signal the arrival of each client, which allowed us to pick different channel pre- and postconditions for each client. Since the above implementation uses a single arrival channel we have to pick a common channel pre- and postcondition that works for every client. We can achieve this using a *logical argument* to relate the channel precondition and the channel postcondition. In this case we index the channel pre- and postcondition with the client identifier i: $P_{\mathsf{arrive}}(i) = B_i^{\mathsf{in}} * \mathsf{id}(i)$ and $Q_{\mathsf{arrive}}(i) = B_i^{\mathsf{out}} * \mathsf{id}(i)$.

For the id predicate to witness the identity of clients, it must be non-duplicable. That is, it must satisfy, $\mathsf{id}(i) * \mathsf{id}(j) \Rightarrow i \neq j$. To define the id predicate such that it satisfies the above property, we need to introduce a bit more of our logic. We return to this example in Section 5.2.

3 Logic

The program logic is a higher-order separation logic [1] with support for reasoning about concurrency, shared mutable data structures [7, 6], and recursive delegates [23]. We use this one program logic to reason about both *clients* of the joins library, and an *implementation* of the joins library.

Our program logic is a general purpose logic for reasoning about higher-order concurrent C♯ programs. We have presented the logic in a separate paper [26]. The full logic and its soundness proof is included in the accompanying technical report [24] of that paper. For the present paper we limit our attention to those features necessary to verify our client examples. To this end, it suffices to consider a minor extension of higher-order separation logic with fractional permissions, phantom/auxiliary state and nested triples [21].

Higher-order separation logic. Every specification in Section 2 was expressed in terms of abstract resource predicates, such as the lock invariant R. This is easily and directly expressible in a higher-order logic, by quantification over predicates [1, 19].

Our assertion logic is an intuitionistic higher-order separation logic over a simply typed term language. The set of types is closed under function space and products, and includes the type of propositions, Prop, the type of specifications, Spec, and the type of mathematical values, Val. The Val type includes all C♯ values and strings, and is closed under formation of pairs, such that mathematical sequences and other mathematical objects can be conveniently represented.[5]

Fractional permissions. The notion of ownership in standard separation logic is very limited, supporting only two extremes: exclusive ownership and no ownership. To formalise the examples from the previous section we need a middle ground of read-only permissions, which can be freely duplicated. Fractional permissions [3] provide a popular solution to this problem, by annotating the points-to predicate with a fraction $p \in (0, 1]$, written $x.f \overset{p}{\mapsto} v$. Full permission corresponds to $p = 1$ and grants exclusive access to the field f. Permissions can be split and combined arbitrarily ($x.f \overset{p}{\mapsto} v * x.f \overset{q}{\mapsto} v \Leftrightarrow x.f \overset{p+q}{\mapsto} v$). Any fraction less than 1 grants partial read-only access to the field f. We write $x.f \mapsto v$ as shorthand for $x.f \overset{1}{\mapsto} v$ and $x.f \overset{-}{\mapsto} v$ as shorthand for $\exists p \in (0, 1].\ x.f \overset{p}{\mapsto} v$.

Phantom state. Auxiliary variables [18] are commonly used as an abstraction of the history of execution and state in Hoare logics. Normally, one declares a subset of program variables as auxiliary variables that can be updated using standard variable assignments, but are not allowed to affect the flow of execution. To support this style of reasoning, we extend separation logic with phantom state. Like standard auxiliary variables, phantom state allows us to record an abstraction of the history of execution, but unlike standard auxiliary variables, phantom state is purely a logical construct (i.e., the operational semantics of the programming language is not altered to accommodate phantom state and

[5] We use a single universe Val for the universe of mathematical values to avoid also having to quantify over types in the logic. We omit explicit encodings of pairs and write $(v_1, ..., v_n)$ for tuples coded as elements of Val.

phantom state is not updated through programming level assignments). When combined with logical arguments, phantom state allows us to logically distinguish and relate multiple messages on the same channel, as needed for the n-barrier example.

Phantom state extends objects with a logical notion of phantom fields and an accompanying phantom points-to predicate, written $x_f \overset{p}{\mapsto} v$, to make assertions about the value and ownership of these phantom fields. To support read-only phantom fields, we further enrich the notion of ownership with fractional permissions. Thus $x_f \overset{p}{\mapsto} v$ asserts the ownership of phantom field f of object x, with fractional permission p, and that this phantom field currently contains the value v. Like the normal points-to predicate, phantom points-to satisfies $x_f \overset{p_1}{\mapsto} v_1 * x_f \overset{p_2}{\mapsto} v_2 \Rightarrow v_1 = v_2$ as phantom fields contain a single fixed value at any given point in time.

Phantom fields are updated using a *view shift*. The notion of a view shift comes from the Views framework for compositional reasoning about concurrency [6], and generalises assertion implication. A view shift from assertion p to assertion q is written $p \sqsubseteq q$. Views shifts can be applied to pre- and postconditions using the following generalised rule of consequence:

$$\frac{p \sqsubseteq p' \qquad \{p'\}c\{q'\} \qquad q' \sqsubseteq q}{\{p\}c\{q\}}$$

Given full ownership (fractional permission 1) of a phantom field f, one can perform a logical update of the field ($x_f \overset{1}{\mapsto} v_1 \sqsubseteq x_f \overset{1}{\mapsto} v_2$). To create a phantom field f we require that the field does not already exist, so that we can take full ownership of the field. We thus require all phantom fields of an object o to be created simultaneously when o is first constructed.

Figure 4 contains a selection of inference rules from our program logic, related to view shifts and phantom state.

Nested triples. To reason about delegates we use nested triples [21]. We write $x \mapsto \{P\}\{Q\}$ to assert that x refers to a delegate satisfying the given specification.

$$\frac{p \Rightarrow q}{p \sqsubseteq q} \qquad \frac{p \sqsubseteq q}{p * r \sqsubseteq q * r} \qquad \frac{p \sqsubseteq p' \qquad \{p'\}c\{q'\} \qquad q' \sqsubseteq q}{\{p\}c\{q\}}$$

$$\frac{}{x_f \overset{1}{\mapsto} v_1 \sqsubseteq x_f \overset{1}{\mapsto} v_2} \qquad \frac{}{x_f \overset{p_1}{\mapsto} v_1 * x_f \overset{p_2}{\mapsto} v_2 \Rightarrow v_1 = v_2} \qquad \frac{}{x_f \overset{p}{\mapsto} v * x_f \overset{q}{\mapsto} v \Leftrightarrow x_f \overset{p+q}{\mapsto} v}$$

Fig. 4. Selected program logic inference rules

Reasoning about the implementation. Fractional permissions introduce a more lenient ownership discipline that allows for read-only sharing. To verify the *implementation* of the joins library, we need even more general forms of sharing. To reason about general sharing patterns we base our logic on concurrent abstract predicates [7].

Conceptually, concurrent abstract predicates (CAP) partitions the heap into a set of regions that each come with a protocol governing how the state in that region may evolve. This allows *stable* assertions – assertions that are closed under changes permitted by the protocol – to be freely duplicated and shared. To ensure soundness, the logic requires that all pre- and postconditions in the specification logic are stable. We thus introduce a new type, SProp, of stable assertions.

Concurrent abstract predicates with first-order protocols (i.e., protocols that only refer to the state of their own region) suffice for reasoning about sharing of *primitive resources* such as individual heap cells. To reason about sharing of *shared resources* requires *higher-order protocols* that can relate the state of multiple regions. In general, to reason about sharing of shared resources requires reasoning about circular sharing patterns. HOCAP extends concurrent abstract predicates with a limited form of higher-order protocols – called state-independent higher-order protocols – and introduce partial orders to explicitly rule out these circular sharing patterns.

Since we are using the same program logic to reason about join clients and the underlying join implementation, join clients could themselves use CAP to describe shared resources when picking the channel pre- and postconditions. This could potentially introduce circular sharing patterns. To simplify the presentation and focus on the main ideas behind our specification of the joins library we have chosen to present a specification that *does not* allow clients to use CAP in their channel pre- and postconditions. This allows us to give a simple specification without any proof obligations about the absence of circular sharing patterns. In the accompanying technical report, we define a stronger joins specification that *does* allow clients to use CAP, but requires clients to prove the absence of circular sharing patterns. In the technical report we verify the joins implementation against this stronger specification. See Section 6 for further discussion.

To prevent joins clients from using CAP, we introduce a new type, LProp, of local propositions. Every predicate expressible in the language of higher-order separation logic extended with phantom state and nested triples is of type LProp, provided all higher-order quantifications quantify over LProp rather than Prop. However, LProp is not closed under region and action assertions for reasoning about shared mutable data structures using CAP. All assertions of type LProp are trivially stable and LProp is thus a subtype of SProp. We thus require all channel pre- and postconditions to be of type LProp. This ensures that clients do not introduce circular sharing patterns.[6]

For details about the logic see our HOCAP paper and accompanying technical report [26, 24].

[6] This circular sharing pattern has been allowed by the first two authors recent work [22].

4 Joins specification

In this section we present our formal specification for the joins library.

The full specification of the joins library is presented in Figure 5. To simplify the specification and exposition of the joins library, we require all channels and chords be registered before clients start sending messages.[7] Formally, we introduce three phases:

ch: This phase allows new channels to be registered.
pat: This phase allows new chords to be registered.
call: This phase allows messages to be sent.

A newly created join instance starts in the ch phase. Once all channels have been registered, it transitions to the pat phase. Once all chords have been registered, it transitions to the call phase. In the call phase, the only way to interact with the join instance is by sending messages on its channels.

The specification is expressed in terms of a number of abstract representation predicates. We use three join representation predicates, $\mathsf{join}_{\mathsf{ch}}, \mathsf{join}_{\mathsf{pat}}$ and $\mathsf{join}_{\mathsf{call}}$ – one for each phase – which will be explained below. In addition, we use two representation predicates for channels and patterns:

$\mathsf{ch}(c, j)$: This predicate asserts that c refers to a channel registered with join instance j.
$\mathsf{pat}(p, j, X)$: This predicate asserts that p refers to a pattern on join instance j that matches the multi-set of channels X.

These representation predicates are all existentially quantified in the specification; clients thus reasons abstractly in terms of these predicates.

Channel initialisation phase. In the first phase we use the join representation predicate: $\mathsf{join}_{\mathsf{ch}}(A, S, j)$. This predicate asserts that j refers to a join instance with asynchronous channels A and synchronous channels S.

The join constructor (Join) returns a new join instance in the ch phase with no registered channels.

The two rules for creating and registering new channels (Sync and Async) take as argument a join instance j in the ch phase and return a reference to a new channel. In both cases, we get back a ch assertion, ch(r,j), that asserts that this newly created channel is registered with join instance j. In addition, both postconditions explicitly assert that this newly created channel is distinct from all previously registered channels, $\mathsf{r} \notin A \cup S$. As the channel predicate is duplicable ($\mathsf{ch}(c, j) \Leftrightarrow \mathsf{ch}(c, j) * \mathsf{ch}(c, j)$), to allow multiple clients to use the same channel, we have to state this explicitly.

Chord initialisation phase. In the second phase we use the join representation predicate: $\mathsf{join}_{\mathsf{pat}}(P, Q, j)$. This predicate asserts that j refers to a join instance with channel preconditions P and channel postconditions Q. Here P and Q are functions that assign channel pre- and postconditions to each channel.

[7] This restriction rules out reasoning about self-modifying synchronisation primitives. We are not aware of any examples of self-modifying join clients.

To relate the pre- and postcondition of a channel (as needed, e.g., in the n-barrier example to distinguish clients), we index each channel pre- and postcondition with a logical argument of type Val.[8] Formally P and Q are thus functions of type $P, Q : \mathrm{Val} \times \tau_{\mathrm{chan}} \to \mathrm{LProp}$ where τ_{chan} is the type of channel references.[9]

Once sufficient channels have been registered, the join instance can transition into the chord initialisation phase using a view shift:

$$\frac{\forall c, a.\ c \in A \Rightarrow Q(a, c) = \mathrm{emp}}{\mathsf{join}_{\mathsf{ch}}(A, S, j) \sqsubseteq \mathsf{join}_{\mathsf{pat}}(P, Q, j)}$$

This forces all channel pre- and postconditions to be fixed before any chords can be registered. This rule explicitly requires that the channel postconditions of asynchronous channel are empty, emp, as explained in Section 2.2.

Rules When and And create a new singleton pattern, and add new channels to an existing pattern, respectively. Note that a pattern matches a multi-set of channels and the set-union in And is thus multi-set union.

The rules for Do are more interesting. Rule Do1 deals with patterns without a continuation. Recall from our informal proof sketches that to add a new chord without a continuation we showed that the combined channel preconditions of the chord pattern *implied* the combined channel postconditions. Our specification generalises this to require that the combined channel preconditions can be *view shifted* to the combined channel postconditions. This generalisation allows us to perform logical updates of phantom state when the chord fires. We will see why this is useful in Section 5.2.

Furthermore, since our channel pre- and postconditions are now indexed by a logical argument, we have to prove that we can perform this view shift for any logical arguments (we have a logical argument for each channel). Formally,

$$\forall Y \in \mathcal{P}_m(\mathrm{Val} \times \tau_{\mathrm{chan}}).\ \pi_{\mathsf{ch}}(Y) = X \Rightarrow \circledast_{y \in Y} P(y) \sqsubseteq \circledast_{y \in Y} Q(y)$$

where $\mathcal{P}_m(-)$ denotes the finite power multi-set operator and π_{ch} is the power set lifting of π_2. Y thus associates a logical argument with each channel. To register a chord that matches channels x and y, this thus reduces to two universally quantified logical arguments, say a and b:

$$\forall a, b \in \mathrm{Val}.\ P(a, x) * P(b, y) \sqsubseteq Q(a, x) * Q(b, y)$$

The rule for Do with a continuation (Do2) is very similar, but instead of requiring a view-shift, it takes a delegate b that transforms the combined preconditions into the combined postconditions. Crucially, the delegate is given access to the join instance in the message phase. This enables it to send messages, as used in the reader/writer lock example (Section 2.3).

Message phase. The final phase allows messages to be sent. We use a third abstract predicate, $\mathsf{join}_{\mathsf{call}}(P, Q, j)$, with the same parameters as the previous abstract predicate $\mathsf{join}_{\mathsf{pat}}(P, Q, j)$. Once all chords have been registered, the join

[8] As Val is closed under pairs this allows us to encode an arbitrary number of logical arguments of type Val.

[9] Formally, τ_{chan} is simply a synonym for Val, introduced to improve the exposition.

instance can transition into the third phase using a view shift: $\mathsf{join}_{\mathsf{pat}}(P, Q, j) \sqsubseteq \mathsf{join}_{\mathsf{call}}(P, Q, j)$.

The only operation in the third phase is to send messages using Call. The rule for sending a message is very similar to the standard method call rule: we provide the precondition $P(a, c)$ and get back the postcondition $Q(a, c)$. Here a is the logical argument, which the client is free to pick.

Both the $\mathsf{join}_{\mathsf{call}}$ and $\mathsf{ch}(-)$ predicate is freely duplicable, to allow multiple clients to send messages on the same channel:

$$\mathsf{ch}(c, j) \Leftrightarrow \mathsf{ch}(c, j) * \mathsf{ch}(c, j)$$
$$\mathsf{join}_{\mathsf{call}}(P, Q, j) \Leftrightarrow \mathsf{join}_{\mathsf{call}}(P, Q, j) * \mathsf{join}_{\mathsf{call}}(P, Q, j)$$

Reasoning about joins. We have verified a simple lock-based implementation of the joins library (see the accompanying technical report for details). We have thus given concrete definitions of the abstract predicates pat, ch, $\mathsf{join}_{\mathsf{ch}}$, $\mathsf{join}_{\mathsf{pat}}$, $\mathsf{join}_{\mathsf{call}}$ and proved that the implementation satisfies a generalisation of the joins specification in Figure 5.

5 Reasoning with joins

In this section we revisit the lock and the n-barrier example, and sketch their formal correctness proofs in terms of our formal specification of the joins library. The lock example is intended to illustrate the joins specification in general, and has thus been written out in full. The n-barrier example is intended to illustrate the use of logical arguments and phantom state.

5.1 Lock

We begin by formalising the previous informal lock specification. As mentioned in Section 3, to avoid reasoning about sharing of shared mutable data structures through themselves, we require all channel pre- and postconditions to be local assertions – i.e., assertions of type LProp. Since the channel pre- and postconditions are defined in terms of the lock resource invariant, the lock resource invariant must be a local assertion. The formal specification of the lock is thus:

$$\begin{array}{l}\forall R : \mathrm{LProp}.\ \exists\, \mathsf{lock} : \mathrm{Val} \to \mathrm{SProp}.\\ \qquad \begin{array}{rl} \{R\} & \textbf{new Lock()}\ \{\mathsf{r}.\ \mathsf{lock}(\mathsf{r})\} \\ \{\mathsf{lock}(\mathsf{l})\} & \mathsf{l.Acquire()}\ \ \{\mathsf{lock}(\mathsf{l}) * R\} \\ \{\mathsf{lock}(\mathsf{l}) * R\} & \mathsf{l.Release()}\ \ \{\mathsf{lock}(\mathsf{l})\} \end{array}\\ \qquad \wedge\ \ \forall x : \mathrm{Val}.\ \mathsf{lock}(x) \Leftrightarrow \mathsf{lock}(x) * \mathsf{lock}(x)\end{array}$$

This specification introduces an explicit lock representation predicate, lock, which is freely duplicable.

We now formalise the proof sketch of the joins-based lock implementation from Section 2. Hence, for any predicate R, we have to define a concrete lock predicate and show that the above specifications for the lock operations hold for the concrete lock predicate.

Channel initialisation phase

$$\frac{}{\{\mathrm{emp}\}\,\textbf{new}\ \mathsf{Join}()\,\{\mathsf{r}.\ \mathsf{join}_{\mathsf{ch}}(\emptyset,\emptyset,\mathsf{r})\}}\ \textsc{Join}$$

$$\frac{}{\{\mathsf{join}_{\mathsf{ch}}(A,S,\mathsf{j})\}\,\textbf{new}\ \mathsf{SyncChannel}(\mathsf{j})\,\{\mathsf{r}.\ \mathsf{join}_{\mathsf{ch}}(A,S\cup\{\mathsf{r}\},\mathsf{j})*\mathsf{ch}(\mathsf{r},\mathsf{j})*\mathsf{r}\notin A\cup S\}}\ \textsc{Sync}$$

$$\frac{}{\{\mathsf{join}_{\mathsf{ch}}(A,S,\mathsf{j})\}\,\textbf{new}\ \mathsf{AsyncChannel}(\mathsf{j})\,\{\mathsf{r}.\ \mathsf{join}_{\mathsf{ch}}(A\cup\{\mathsf{r}\},S,\mathsf{j})*\mathsf{ch}(\mathsf{r},\mathsf{j})*\mathsf{r}\notin A\cup S\}}\ \textsc{Async}$$

Chord initialisation phase

$$\frac{}{\{\mathsf{join}_{\mathsf{pat}}(P,Q,\mathsf{j})*\mathsf{ch}(\mathsf{c},\mathsf{j})\}\,\mathsf{j}.\mathsf{When}(\mathsf{c})\,\{\mathsf{r}.\ \mathsf{join}_{\mathsf{pat}}(P,Q,\mathsf{j})*\mathsf{pat}(\mathsf{r},\mathsf{j},\{\mathsf{c}\})\}}\ \textsc{When}$$

$$\frac{}{\left\{\begin{array}{c}\mathsf{join}_{\mathsf{pat}}(P,Q,j)*\\ \mathsf{pat}(\mathsf{p},j,X)*\mathsf{ch}(\mathsf{c},j)\end{array}\right\}\mathsf{p}.\mathsf{And}(\mathsf{c})\left\{\begin{array}{c}\mathsf{join}_{\mathsf{pat}}(P,Q,j)*\\ \mathsf{pat}(\mathsf{p},j,X\cup\{\mathsf{c}\})\end{array}\right\}}\ \textsc{And}$$

$$\frac{\forall Y\in\mathcal{P}_m(\mathcal{E}).\ \pi_{\mathsf{ch}}(Y)=X\Rightarrow \circledast_{y\in Y}P(y)\sqsubseteq\circledast_{y\in Y}Q(y)}{\{\mathsf{join}_{\mathsf{pat}}(P,Q,j)*\mathsf{pat}(\mathsf{p},j,X)\}\,\mathsf{p}.\mathsf{Do}()\,\{\mathsf{join}_{\mathsf{pat}}(P,Q,j)\}}\ \textsc{Do1}$$

$$\frac{}{\left\{\begin{array}{l}\mathsf{join}_{\mathsf{pat}}(P,Q,j)*\mathsf{pat}(\mathsf{p},j,X)*\circledast_{z\in Z}\mathsf{ch}(z,j)*\\ \quad\forall Y\in\mathcal{P}_m(\mathcal{E}).\ \pi_{\mathsf{ch}}(Y)=X\Rightarrow\\ \quad b\mapsto\left\{\begin{array}{l}\mathsf{join}_{\mathsf{call}}(P,Q,j)*\\ \circledast_{z\in Z}\mathsf{ch}(z,j)*\circledast_{y\in Y}P(y)\end{array}\right\}\left\{\begin{array}{l}\circledast_{y\in Y}Q(y)\\ \end{array}\right\}\end{array}\right\}\mathsf{p}.\mathsf{Do}(\mathsf{b})\,\{\mathsf{join}_{\mathsf{pat}}(P,Q,j)\}}\ \textsc{Do2}$$

Message phase

$$\frac{}{\{\mathsf{join}_{\mathsf{call}}(P,Q,j)*\mathsf{ch}(\mathsf{c},j)*P(a,\mathsf{c})\}\,\mathsf{c}.\mathsf{Call}()\,\{\mathsf{join}_{\mathsf{call}}(P,Q,j)*Q(a,\mathsf{c})\}}\ \textsc{Call}$$

Phase transitions

$$\frac{\forall c,a.\ c\in A\Rightarrow Q(a,c)=\mathrm{emp}}{\mathsf{join}_{\mathsf{ch}}(A,S,j)\sqsubseteq\mathsf{join}_{\mathsf{pat}}(P,Q,j)}\qquad\frac{}{\mathsf{join}_{\mathsf{pat}}(P,Q,j)\sqsubseteq\mathsf{join}_{\mathsf{call}}(P,Q,j)}$$

$$\mathsf{ch}(c,j)\Leftrightarrow\mathsf{ch}(c,j)*\mathsf{ch}(c,j)\qquad\mathsf{join}_{\mathsf{call}}(P,Q,j)\Leftrightarrow\mathsf{join}_{\mathsf{call}}(P,Q,j)*\mathsf{join}_{\mathsf{call}}(P,Q,j)$$

Abstract predicates

$$\begin{array}{lll}\mathsf{pat} & :\tau_{\mathrm{pat}}\times\tau_{\mathrm{join}}\times\mathcal{P}_m(\tau_{\mathrm{chan}})\to\mathrm{SProp} & \mathsf{ch}:\tau_{\mathrm{chan}}\times\tau_{\mathrm{join}}\to\mathrm{SProp}\\ \mathsf{join}_{\mathsf{ch}} & :\mathcal{P}_m(\tau_{\mathrm{chan}})\times\mathcal{P}_m(\tau_{\mathrm{chan}})\to\mathrm{SProp} & \\ \mathsf{join}_{\mathsf{pat}},\mathsf{join}_{\mathsf{call}} & :(\mathcal{E}\to\mathrm{LProp})\times(\mathcal{E}\to\mathrm{LProp})\times\mathrm{Val}\to\mathrm{SProp} & \end{array}$$

Here $\mathcal{P}_m(X)$ denotes the set of finite multi-subsets of X and

$$\begin{array}{l}\tau_{\mathrm{join}}=\tau_{\mathrm{chan}}=\tau_{\mathrm{pat}}\stackrel{\mathrm{def}}{=}\mathrm{Val}\qquad\mathcal{E}\stackrel{\mathrm{def}}{=}\mathrm{Val}\times\tau_{\mathrm{chan}}\\ \pi_{\mathsf{ch}}(X)\stackrel{\mathrm{def}}{=}\{\pi_2(x)\mid x\in X\}:\mathcal{P}_m(\mathcal{E})\to\mathcal{P}_m(\tau_{\mathrm{chan}})\end{array}$$

Fig. 5. Specification of the joins library.

The channel pre- and postconditions do not change relative to the informal proof. For any pair of channels c_a and c_r we define the channel pre- and postcondition, $P(c_a, c_r), Q(c_a, c_r) : \mathcal{E} \to \text{LProp}$, as follows:

$$P(c_a, c_r)(a, c) = \begin{cases} \text{emp} & \text{if } c = c_a \\ R & \text{if } c = c_r \\ \bot & \text{otherwise} \end{cases} \qquad Q(c_a, c_r)(a, c) = \begin{cases} R & \text{if } c = c_a \\ \text{emp} & \text{if } c = c_r \\ \bot & \text{otherwise} \end{cases}$$

In the proof, c_a will be instantiated with the acquire channel and c_r with the release channel. Note that the logical argument a is simply ignored.

The lock predicate then asserts that there exists some join instance and that fields acq and rel refer to channels with the above channel pre- and postcondition.

$$\begin{aligned} \mathsf{lock}(x) = \exists a, r, j : \text{Val}.\ a \neq r \wedge x.\mathsf{acq} \mapsto\!\!\!\!\sim\ a * x.\mathsf{rel} \mapsto\!\!\!\!\sim\ r \\ * \ \mathsf{ch}(a, j) * \mathsf{ch}(r, j) * \mathsf{join}_{\mathsf{call}}(P(a, r), Q(a, r), j) \end{aligned}$$

We explicitly require that a and r are distinct to ensure that the above definition of P and Q by case analysis on the second argument is well-defined. The lock predicate only asserts partial ownership of fields acq and rel, to allow the lock predicate to be freely duplicated.

Below is a full proof outline for the lock constructor.

public Lock() {
  Join j; Pattern p;
{this.acq $\mapsto$ *null* $*$ this.rel $\mapsto$ *null* $* R$}
  j = **new** Join();
{this.acq $\mapsto$ *null* $*$ this.rel $\mapsto$ *null* $* R *$ $\mathsf{join}_{\mathsf{ch}}(\emptyset, \emptyset, \mathsf{j})$}
  acq = **new** SyncChannel(j);
  rel = **new** AsyncChannel(j);
{$R *$ this.acq $\mapsto a *$ this.rel $\mapsto r *$ $\mathsf{join}_{\mathsf{ch}}(\{r\}, \{a\}, \mathsf{j}) * a \neq r *$ $\mathsf{ch}(a, \mathsf{j}) *$ $\mathsf{ch}(r, \mathsf{j})$}
{$R *$ this.acq $\mapsto a *$ this.rel $\mapsto r *$ $\mathsf{join}_{\mathsf{pat}}(P(a, r), Q(a, r), \mathsf{j}) * a \neq r *$ $\mathsf{ch}(a, \mathsf{j}) *$ $\mathsf{ch}(r, \mathsf{j})$}
  p = j.When(acq).And(rel);
{$R *$ this.acq $\mapsto a *$ this.rel $\mapsto r * a \neq r *$ $\mathsf{join}_{\mathsf{pat}}(P(a, r), Q(a, r), \mathsf{j})$
  $* \ \mathsf{ch}(a, \mathsf{j}) * \mathsf{ch}(r, \mathsf{j}) * \mathsf{pat}(\mathsf{p}, \mathsf{j}, \{a, r\})$}
  p.Do();
{$R *$ this.acq $\mapsto a *$ this.rel $\mapsto r *$ $\mathsf{join}_{\mathsf{pat}}(P(a, r), Q(a, r), \mathsf{j}) * a \neq r *$ $\mathsf{ch}(a, \mathsf{j}) *$ $\mathsf{ch}(r, \mathsf{j})$}
{$R *$ this.acq $\mapsto a *$ this.rel $\mapsto r *$ $\mathsf{join}_{\mathsf{call}}(P(a, r), Q(a, r), \mathsf{j}) * a \neq r *$ $\mathsf{ch}(a, \mathsf{j}) *$ $\mathsf{ch}(r, \mathsf{j})$}
  rel.Call();
{this.acq $\mapsto a *$ this.rel $\mapsto r *$ $\mathsf{join}_{\mathsf{call}}(P(a, r), Q(a, r), \mathsf{j}) * a \neq r *$ $\mathsf{ch}(a, \mathsf{j}) *$ $\mathsf{ch}(r, \mathsf{j})$}
{lock(this)}
}

The call to Do further requires that we prove:

$$\forall Y \in \mathcal{P}_m(\mathcal{E}).\ \pi_{\mathsf{ch}}(Y) = \{a, r\} \Rightarrow \circledast_{y \in Y} P(a, r)(y) \sqsubseteq \circledast_{y \in Y} Q(a, r)(y)$$

which follows easily from the commutativity of $*$.

The full proof outline for Acquire is given below. The proof for Release is similar.

public void Acquire() {
  SyncChannel c;

$\{\mathsf{lock(this)}\}$
$\{\mathsf{this.acq} \mapsto a * \mathsf{this.rel} \mapsto r * \mathsf{join}_{\mathsf{call}}(P(a,r), Q(a,r), \mathsf{j}) * a \neq r * \mathsf{ch}(a,\mathsf{j}) * \mathsf{ch}(r,\mathsf{j})\}$
  $\mathsf{c} = \mathbf{this}.\mathsf{acq};$
$\{\mathsf{this.acq} \mapsto \mathsf{c} * \mathsf{this.rel} \mapsto r * \mathsf{join}_{\mathsf{call}}(P(\mathsf{c},r), Q(\mathsf{c},r), \mathsf{j}) * \mathsf{c} \neq r * \mathsf{ch}(\mathsf{c},\mathsf{j}) * \mathsf{ch}(r,\mathsf{j})\}$
  $\mathsf{c.Call}();$
$\{\mathsf{this.acq} \mapsto \mathsf{c} * \mathsf{this.rel} \mapsto r * \mathsf{join}_{\mathsf{call}}(P(\mathsf{c},r), Q(\mathsf{c},r), \mathsf{j}) * \mathsf{c} \neq r$
    $* \mathsf{ch}(\mathsf{c},\mathsf{j}) * \mathsf{ch}(r,\mathsf{j}) * Q(\mathsf{c},r)(0,\mathsf{c})\}$
$\{\mathsf{lock(this)} * R\}$
}

When we call the acq channel we have to pick a logical argument a. Since the channel pre- and postcondition ignores the a, we can pick anything. In the above proof we arbitrarily picked 0, hence the $Q(\mathsf{c},r)(0,\mathsf{c})$ in the postcondition.

5.2 n-barrier

In this section we formalise a proof of the n-barrier from Section 2.4. This example illustrates how logical arguments combined with phantom state allows us to logically distinguish messages on a single channel. The example also illustrates the use of a non-trivial view-shift to update a phantom field upon firing of a chord.

Desired specification. In Section 2.4 we gave an informal specification of an n-barrier, under the assumption that clients transferred the same resources to the barrier at every round of synchronisation, and that the barrier redistributed these resources in the same way at every round of synchronisation. As these assumptions are unrealistic, we start by generalising the specification.

The simplified n-barrier specification was expressed in terms of two assertions B_i^{in} and B_i^{out} that described the resources client i transferred to and from the barrier at every round of synchronisation. Here, instead, we take B^{in} and B^{out} to be predicates indexed by a client identifier i and the current round of synchronisation m. The general n-barrier specification is given in Figure 6.

$$
\begin{array}{l}
\forall n \in \mathbb{N}.\ \forall B^{\mathsf{in}}, B^{\mathsf{out}} : \{1,...,n\} \times \mathbb{N} \to \mathrm{LProp}. \\
\quad (\forall m \in \mathbb{N}.\ \circledast_{i\in\{1,...,n\}} B_i^{\mathsf{in}}(m) \sqsubseteq \circledast_{i\in\{1,...,n\}} B_i^{\mathsf{out}}(m))) \ \Rightarrow \\
\qquad \exists \mathsf{barrier} : \mathrm{Val} \to \mathrm{SProp}.\ \exists \mathsf{client} : \mathrm{Val} \times \{1,...,n\} \times \mathbb{N} \to \mathrm{SProp}. \\
\qquad\quad \{\mathsf{n} = n\}\,\mathbf{new}\ \mathsf{Barrier(n)}\,\{\mathsf{ret}.\ \mathsf{barrier(ret)} * \circledast_{i\in\{1,...,n\}} \mathsf{client(ret},i,0)\} \\
\qquad \wedge\ \forall i \in \{1,...,n\}.\ \forall m \in \mathbb{N}. \\
\qquad\quad \{\mathsf{barrier(b)} * \mathsf{client(b},i,m) * B_i^{\mathsf{in}}(m)\} \\
\qquad\quad\ \ \mathsf{b.Arrive}() \\
\qquad\quad \{\mathsf{barrier(b)} * \mathsf{client(b},i,m+1) * B_i^{\mathsf{out}}(m)\} \\
\qquad \wedge\ \forall x : \mathrm{Val}.\ \mathsf{barrier}(x) \Leftrightarrow \mathsf{barrier}(x) * \mathsf{barrier}(x)
\end{array}
$$

Fig. 6. General n-barrier specification. This specification requires that the number of clients, n, is known statically. This simplifies the exposition. We can also specify and verify a specification without this assumption.

Here barrier is the barrier representation predicate, which can be freely duplicated. The client predicate plays two roles: namely, (1) to witness the identity of each barrier client (like the id predicate from Section 2.4), and (2) to ensure that every client of the barrier agrees on the round of synchronisation, m, whenever they arrive at the barrier. These two properties are necessary to ensure that we can redistribute the combined resources when every client has arrived at the barrier. When one creates a new n-barrier, one thus receives n client predicates – one for each client – each with 0 as the current round of synchronisation. The current round of synchronisation is incremented by one at each arrival at the barrier.

Predicate definitions. We start by giving concrete definitions for the abstract barrier and client predicate. Hence, assume $n \in \mathbb{N}$ clients and abstract predicates $B^{\mathsf{in}}, B^{\mathsf{out}} : \{1, ..., n\} \times \mathbb{N} \to \text{LProp}$ satisfying,

$$\forall m \in \mathbb{N}.\ \circledast_{i \in \{1,...,n\}} B_i^{\mathsf{in}}(m) \sqsubseteq \circledast_{i \in \{1,...,n\}} B_i^{\mathsf{out}}(m) \tag{2}$$

Since the n-barrier only has a single channel, we need to pick a single channel pre- and postcondition that works for every client, for every round of synchronisation. We thus take the logical argument for the arrival channel to be a pair consisting of a client identifier i and the current synchronisation round m. From the specification above, when client i arrives for synchronisation round m it transfers $B_i^{\mathsf{in}}(m)$ to the barrier and expects to receive back $B_i^{\mathsf{out}}(m)$. In addition, the client gives up its client predicate and gets back a new one, with the same logical client identifier i and an incremented synchronisation round, $m + 1$. For any barrier b and channel c_1 we thus define the channel pre- and postcondition $P(b, c_1), Q(b, c_1) : \mathcal{E} \to \text{LProp}$ as follows:

$$P(b, c_1)((i, m), c) = \begin{cases} \mathsf{client}(b, i, m) * B_i^{\mathsf{in}}(m) & \text{if } c = c_1 \\ \bot & \text{otherwise} \end{cases}$$

$$Q(b, c_1)((i, m), c) = \begin{cases} \mathsf{client}(b, i, m + 1) * B_i^{\mathsf{out}}(m) & \text{if } c = c_1 \\ \bot & \text{otherwise} \end{cases}$$

Here the (i, m) is the logical argument consisting of the logical client identifier i and synchronisation round m. In the proof, c_1 will be instantiated with the arrival channel.

Above, we defined the channel pre- and postcondition in terms of an abstract client predicate, which we have not defined yet. We thus need to define client. This is the main technical challenge of the proof. So, to motivate its definition, we start by considering what properties the client predicate should satisfy. Recall that we use the client predicate to (1) witness the identity of clients, and to (2) ensure that clients agree on the current round of synchronization when they arrive at the barrier.

To witness the identity of clients, disjoint client predicates must refer to distinct clients, as expressed by property (3) below. To ensure that clients agree on the current round of synchronisation, the client predicate should also satisfy (4). Lastly, to update the current round of synchronisation when every client

has arrived at the barrier, the client predicate should satisfy (5).

$$\forall b, i, j, m.\ \mathsf{client}(b, i, m) * \mathsf{client}(b, j, m) \Rightarrow i \neq j \tag{3}$$

$$\forall b, i, j, m, k.\ \mathsf{client}(b, i, m) * \mathsf{client}(b, j, k) \Rightarrow m = k \tag{4}$$

$$\forall b, m.\ \circledast_{i \in \{1,\ldots,n\}} \mathsf{client}(b, i, m) \sqsubseteq \circledast_{i \in \{1,\ldots,n\}} \mathsf{client}(b, i, m+1) \tag{5}$$

Note that (5) is consistent with (4), since we update all n client predicates simultaneously.

We can satisfy (4) and (5) by introducing a phantom field to keep track of the current round of synchronisation. By giving each client $1/n$-th permission of this phantom field, we ensure that every client agrees on the current round of synchronisation, (4). Furthermore, given all n client predicates, these fractions combine to the full permission, allowing the phantom field to be updated arbitrarily, and thus in particular, to be incremented; thus satisfying (5). We can satisfy (3) by associating each client identifier i with a non-duplicable resource $\bullet_i$ in the logic, and requiring ownership of $\bullet_i$ in the client predicate. We thus define client as follows, $\mathsf{client}(b, i, m) = b_{\mathsf{round}} \overset{1/n}{\longmapsto} m * \bullet_i^b$, where $\bullet_i^b$ is defined as follows: $\bullet_i^b = \exists v : \mathrm{Val}.\ b_i \mapsto v$.

The barrier predicate is now trivial to define:

$$\mathsf{barrier}(b) = \exists j, c : \mathrm{Val}.\ b.\mathsf{arrive} \mapsto c * \mathsf{join}_{\mathsf{call}}(P(b,c), Q(b,c), j) * \mathsf{ch}(c, j)$$

It simply asserts that arrive refers to a channel on a join instance with the channel pre- and postcondition we defined above.

Proof. Now that we have defined a client predicate satisfying (3), (4), and (5), we can proceed with the verification of the n-barrier. The main proof obligation is proving that the barrier chord satisfies the postconditions of the channels it matches. Since the barrier chord matches n arrival messages, by rule Do1 we thus have to prove that:

$$\forall Y \in \mathcal{P}_m(\mathcal{E}).\ \pi_{\mathsf{ch}}(Y) = \{c_1^n\} \Rightarrow \circledast_{y \in Y} P(b, c_1)(y) \sqsubseteq \circledast_{y \in Y} Q(b, c_1)(y)$$

To simplify the exposition, we consider the case for $n = 2$. The proof for $n > 2$ follows the same structure. For $n = 2$ the above proof obligation reduces to:

$$\begin{aligned} &\forall i_1, i_2, m_1, m_2. \\ &\quad \mathsf{client}(b, i_1, m_1) * B_{i_1}^{\mathsf{in}}(m_1) * \mathsf{client}(b, i_2, m_2) * B_{i_2}^{\mathsf{in}}(m_2) \ \sqsubseteq \\ &\qquad \mathsf{client}(b, i_1, m_1 + 1) * B_{i_1}^{\mathsf{out}}(m_1) * \mathsf{client}(b, i_2, m_2 + 1) * B_{i_2}^{\mathsf{out}}(m_2) \end{aligned} \tag{6}$$

At this point we cannot directly apply the user-supplied redistribution property, (2), as it requires that $m_1 = m_2$ and $i_1 \neq i_2$. First, we need to use properties (3) and (4) to constrain what logical arguments clients could have chosen when they send their arrival messages. By property (4) it follows that $m_1 = m_2$. Furthermore, from property (3) it follows that i_1 and i_2 are distinct. Since $i_1, i_2 \in \{1, 2\}$, (6) thus reduces to:

$$\begin{aligned} &\forall m.\ \mathsf{client}(b, 1, m) * B_1^{\mathsf{in}}(m) * \mathsf{client}(b, 2, m) * B_2^{\mathsf{in}}(m) \ \sqsubseteq \\ &\qquad \mathsf{client}(b, 1, m + 1) * B_1^{\mathsf{out}}(m) * \mathsf{client}(b, 2, m + 1) * B_2^{\mathsf{out}}(m) \end{aligned} \tag{7}$$

Using the redistribution property, (2), and (5) it follows that,

$$\forall m.\ B_1^{\mathsf{in}}(m) * B_2^{\mathsf{in}}(m) \sqsubseteq B_1^{\mathsf{out}}(m) * B_2^{\mathsf{out}}(m)$$
$$\forall m.\ \mathsf{client}(b,1,m) * \mathsf{client}(b,2,m) \sqsubseteq \mathsf{client}(b,1,m+1) * \mathsf{client}(b,2,m+1)$$

Combining these two we thus get (7). We have thus proven (6). Note that here we implicitly used the ability to perform a view shift when a chord fires, to increment the value of the phantom field round.

The verification of the constructor and Arrive method is now straightforward.

In summary, using logical arguments and phantom state we can thus show that the generalised n-barrier from Section 2.4 satisfies the generalised barrier specification. While the proof is more technically challenging than any of the previous examples, it is still a high-level proof about *barrier* concepts. Informally, we proved that clients agree on the current synchronisation round and that clients identify themselves correctly; both natural proof obligations for a barrier.

6 Discussion

We first relate our specification of joins and the clients thereof to earlier work and then evaluate what we have learned about HOCAP from this case study.

In terms of reasoning about external sharing, O'Hearn's original concurrent separation logic supports reasoning about shared variable concurrency using critical regions [17]. This was subsequently extended to a language with locking primitives by Hobor et al. [11] and Gotsman et al. [10], and to a language with barrier primitives by Hobor et al. [12]. In all four cases, the underlying synchronisation primitives were taken as language primitives and their soundness was proven meta-theoretically.

Concurrent abstract predicates by Dinsdale-Young et al. [7] extends standard separation logic with support for reasoning about shared mutable state by imposing protocols on shared resources. Dinsdale-Young et al. used this logic to verify a spin-lock implemented using compare-and-swap. The spin-lock was verified against a non-standard lock specification *without* built-in support for reasoning about external sharing. Hence, to reason about external sharing, *clients* would have to define a protocol of their own, relating ownership of the shared resources with the state of the lock. This type of reasoning is *not* modular, as it requires the specification of concurrent libraries to expose *internal* implementation details of synchronisation primitives, to allow clients to define a protocol governing the external sharing.

Jacobs and Piessens recently extended their VeriFast tool with support for fine-grained concurrency [14] and verified a lock-based barrier implementation [13] inspired by [11]. They verify the implementation against a specification without built-in support for reasoning about external sharing. Compared to our barrier specification, their specification is thus fairly low-level, requiring *clients* of the barrier to use auxiliary variables to encode who has arrived and what resources they have transferred to the barrier.

The goal of this case study was to test whether HOCAP supports modular reasoning about concurrent higher-order imperative libraries. To this end, we have proposed an abstract specification of the C♯ joins library, expressed in terms of high-level join primitives. We have demonstrated that this abstract specification suffices for formal reasoning about a series of classic synchronisation primitives, which allow for external sharing. Compared to previous work on verifying synchronisation primitives using separation logic, our specifications are stronger and our proofs are considerably simpler. Thus, from this perspective, our case study supports the thesis that HOCAP is useful for modular reasoning about concurrent higher-order imperative libraries. However, as explained in Section 3, the joins specification presented in this paper is restricted to local pre- and postconditions for channels, which means that synchronization primitives implemented using joins can only have local assertions as resource invariants. Recall, e.g., the lock specification in Section 5.1, where the resource invariant ranges over LProp, which means that clients of the lock cannot use CAP when picking a resource invariant for the lock. In the technical report [25] we have presented a stronger specification of joins, which does allow clients to use CAP for such resource invariants, but that is at the expense of complicating the specification, to avoid circular sharing patterns. Thus future work includes finding stronger models of HOCAP that support simple specifications and circular sharing patterns.

Acknowledgements

We would like to thank Mike Dodds, Bart Jacobs, Jonas Braband Jensen, Hannes Mehnert, Claudio Russo, and Aaron Turon for helpful discussions and feedback.

References

1. B. Biering, L. Birkedal, and N. Torp-Smith. BI-Hyperdoctrines, Higher-order Separation Logic, and Abstraction. *ACM TOPLAS*, 2007.
2. R. Bornat, C. Calcagno, P. W. O'Hearn, and M. J. Parkinson. Permission accounting in separation logic. In *Proceedings of POPL*, pages 259–270, 2005.
3. J. Boyland. Checking interference with fractional permissions. In *Static Analysis: 10th International Symposium*, pages 55–72, 2003.
4. P.-J. Courtois, F. Heymans, and D. L. Parnas. Concurrent Control with "Readers" and "Writers". *Commun. ACM*, 14(10):667–668, 1971.
5. P. da Rocha Pinto, T. Dinsdale-Young, M. Dodds, P. Gardner, and M. Wheelhouse. A simple abstraction for complex concurrent indexes. *SIGPLAN Not.*, 46(10):845–864, Oct. 2011.
6. T. Dinsdale-Young, L. Birkedal, P. Gardner, M. Parkinson, and H. Yang. Views: Compositional Reasoning for Concurrent Programs. In *Proceedings of POPL*, 2013.
7. T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis. Concurrent Abstract Predicates. In *Proceedings of ECOOP*, 2010.
8. C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of POPL*, 1996.

9. C. Fournet and G. Gonthier. The Join Calculus: A Language for Distributed Mobile Programming. In *Proceedings of APPSEM*, pages 268–332, 2000.
10. A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, and M. Sagiv. Local Reasoning for Storable Locks and Threads. In *Proceedings of APLAS*, pages 19–37, 2007.
11. A. Hobor, A. W. Appel, and F. Z. Nardelli. Oracle Semantics for Concurrent Separation Logic. In *Proceedings of ESOP*, pages 353–367, 2008.
12. A. Hobor and C. Gherghina. Barriers in concurrent separation logic. In *Proceedings of ESOP*, pages 276–296, 2011.
13. B. Jacobs. Verified general barriers implementation. http://people.cs.kuleuven.be/~bart.jacobs/verifast/examples/barrier.c.html.
14. B. Jacobs and F. Piessens. Expressive modular fine-grained concurrency specification. In *Proceedings of POPL*, pages 271–282, 2011.
15. N. Krishnaswami. *Verifying Higher-Order Imperative Programs with Higher-Order Separation Logic*. PhD thesis, Carnegie Mellon University, 2012.
16. N. Krishnaswami, L. Birkedal, and J. Aldrich. Verifying Event-Driven Programs using Ramified Frame Properties. In *Proceedings of TLDI*, 2010.
17. P. W. O'Hearn. Resources, Concurrency and Local Reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.
18. S. S. Owicki. *Axiomatic Proof Techniques for Parallel Programs*. PhD thesis, Cornell, 1975.
19. M. J. Parkinson and G. M. Bierman. Separation logic and abstraction. In *Proceedings of POPL*, pages 247–258, 2005.
20. C. V. Russo. The Joins Concurrency Library. In *Proceedings of PADL*, pages 260–274, 2007.
21. J. Schwinghammer, L. Birkedal, B. Reus, and H. Yang. Nested Hoare Triples and Frame Rules for Higher-Order Store. *LMCS*, 7(3:21), 2011.
22. K. Svendsen and L. Birkedal. Impredicative Concurrent Abstract Predicates. Under submission, 2013.
23. K. Svendsen, L. Birkedal, and M. Parkinson. Verifying Generics and Delegates. In *Proceedings of ECOOP*, pages 175–199, 2010.
24. K. Svendsen, L. Birkedal, and M. Parkinson. Higher-order Concurrent Abstract Predicates. Technical report, IT University of Copenhagen, 2012. Available at `www.itu.dk/people/kasv/hocap-tr.pdf`.
25. K. Svendsen, L. Birkedal, and M. Parkinson. Verification of the Joins Library in Higher-order Separation Logic. Technical report, IT University of Copenhagen, 2012. Available at `www.itu.dk/people/kasv/joins-tr.pdf`.
26. K. Svendsen, L. Birkedal, and M. Parkinson. Modular Reasoning about Separation for Concurrent Data Structures. In *Proceedings of ESOP*, 2013.

Verification of the Joins Library in Higher-Order Separation Logic

Kasper Svendsen
IT University of Copenhagen
kasv@itu.dk

Lars Birkedal
IT University of Copenhagen
birkedal@itu.dk

Matthew Parkinson
Microsoft Research, Cambridge
mattpark@microsoft.com

Contents

1 Introduction

In this technical report we define a formal specification for the joins library and verify a naive lock-based implementation against this specification. We assume the reader is familiar with the joins library and has read the accompanying article [3]. We take as our program logic the higher-order version of concurrent abstract predicates described in [4] and formally defined in [2]. We assume the reader is familiar with this logic.

The joins library implementation makes for an interesting verification challenge, as it combines state, sharing, concurrency and reentrancy, in a realistic and reasonably small library. We use our higher-order variant of Concurrent Abstract Predicates to reason about sharing and concurrency and we use representation predicates defined by guarded recursion and nested Hoare triples to reason about reentrancy.

2 Implementation

We start by sketching our naive lock-based implementation. Figure 1 defines the static structure of the implementation. The implementation consists of seven main classes:

- Join: Join objects represent join instances. Each join object consists a list of registered chords and a list of registered channels. Conceptually, we think of a join instance as having a single common message pool. However, in the implementation, each channel maintains its own pool of messages. Each join object contains a lock to synchronize access to its internal state. Since we have restricted attention to non-self-modifying join clients, once all chords and channels have been registered, all internal state is fixed, except the message pools. The lock is thus only necessary to synchronize access to message pools.

- Pattern: Pattern objects represent conditions on the message pool. The type of conditions supported by this version of the joins library can be represented as a list of channels – representing the condition that matches a distinct message from each channel in the list.

- Chord: Chord objects represent chords, which are simply implemented as a pair consisting of a Pattern and an Action (a standard C$^\sharp$ delegate type for delegates without a return value).

- MessagePool, Message: The abstract MessagePool class implements a message pool. Each message pool consists of a list of pending messages. Each message consists of an integer status field indicating whether the message has been received. This status field is thus 0 (not received) for all pending messages. When a message is matched by a pattern it is removed from its message pool and once the chord continuation has terminated its status field is set to 1 (received).

- AsyncChannel, SyncChannel: AsyncChannel and SyncChannel objects represent channels. They both inherit from the abstract MessagePool class.

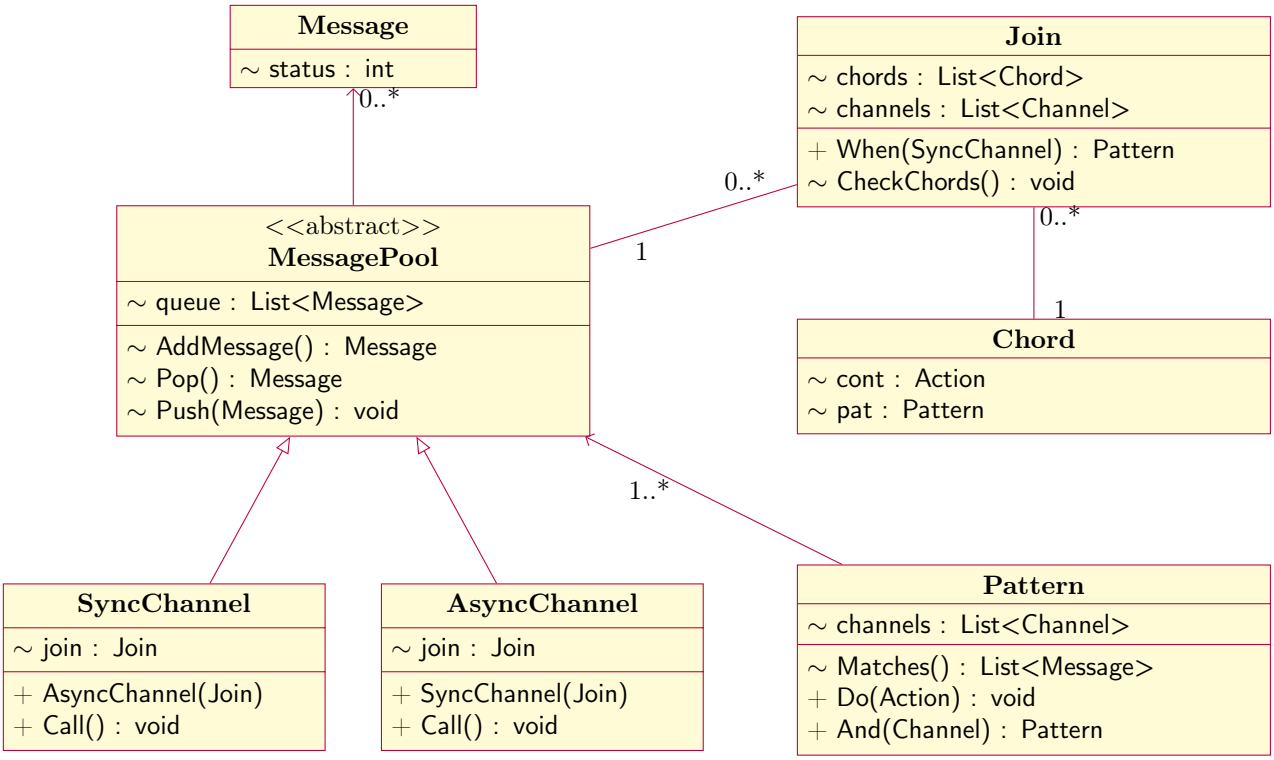


Figure 1: Class diagram

The meat of the implementation is the Call methods for sending a message on a channel. Both Call methods start by acquiring the join lock. Once the lock has been acquired, in each case they add a new pending message to the message pool of the given channel (using the AddMessage method). In the asynchronous case, this is all that happens; Call simply releases the lock and returns. However, in the synchronous case, the method enters a busy-loop, constantly checking whether its message has been received and firing any chords ready to fire (using the CheckChords method). To support reentrant callbacks, the CheckChords method releases and reacquires the join lock before executing chord continuations.

The CheckChords method iterates through the list of chords, checking if each chord is ready to fire (using the Matches method). Whenever it encounters a chord ready to fire, it (1) removes the messages that matched the chord pattern from their message pools, (2) releases the join lock (to allow continuations to send messages on the same join instance without dead-locking), (3) calls the continuation, (4) reacquires the join lock and (5) updates the status of all the matched messages to 1 (received).

The Matches method optimistically tries to match and remove a pending message from each channel of the given pattern (using the Pop method). If it succeeds, it simply returns a list of the messages it matched. If it fails, it adds back the messages already removed to their respective message pools (using the Push method), and returns `null`. This Matches implementation, which optimistically matches messages is unnecessary in our lock-based implementation, as the thread already owns the global message pool lock at this point. However, this implementation is easier to verify and adapt to a proof of the non-locking implementation, for which it is required.

The implementation makes use of three auxiliary classes: Lock, List and Pair. The Lock class implements a spin-lock. The List class implements a singly-linked list. It supports methods Push, Pop and Count, to modify and query lists. In addition, it supports higher-order methods ForEach and Map, to iterate over lists. Lastly, the Pair class implements a pair.

3 Specification

The abstract specification of the joins library is given below. This is a generalized version of the specification that appears in the accompanying article [3]. In particular, this specification supports channel pre- and post-conditions expressed using CAP. Since the overall idea behind the specification remains unchanged, the reader is referred to the accompanying article for an explanation of the specification. To support channel pre- and post-conditions expressed using CAP we require that channel pre- and post-conditions are uniformly expressible using state independent protocols, that they are independent of the chosen region type (`t`) and stable.

$\exists \mathsf{join}_{\mathsf{init\text{-}ch}} : \mathrm{RType} \times \mathcal{P}_m(\mathrm{Val}) \times \mathcal{P}_m(\mathrm{Val}) \times \mathrm{Val} \to \mathrm{Prop}.$

$\exists \mathsf{join}_{\mathsf{init\text{-}path}}, \mathsf{join}_{\mathsf{call}} : \mathrm{RType} \times (\mathrm{Val} \times \mathrm{Val} \to \mathrm{Prop}) \times (\mathrm{Val} \times \mathrm{Val} \to \mathrm{Prop}) \times \mathrm{Val} \to \mathrm{Prop}.$

$\exists \mathsf{chan} : \mathrm{Val} \times \mathrm{Val} \to \mathrm{Prop}.\ \exists \mathsf{pattern} : \mathrm{Val} \times \mathrm{Val} \times \mathcal{P}_m(\mathrm{Val}) \to \mathrm{Prop}.$

$\forall \mathsf{t} : \mathrm{RType}.\ \forall \mathsf{P}, \mathsf{Q} : \mathrm{Val} \times \mathrm{Val} \to \mathrm{Prop}.\ \forall \mathsf{C}_A, \mathsf{C}_S, \mathsf{X}, \mathsf{Z} : \mathcal{P}_m(\mathrm{Val}).\ \forall \mathsf{a}, \mathsf{c}, \mathsf{j} : \mathrm{Val}.$

$\mathsf{usip}(\mathsf{P}, \mathsf{Q}) \land \mathsf{indep}_{\mathsf{t}}(\mathsf{P}) \land \mathsf{indep}_{\mathsf{t}}(\mathsf{Q}) \land \mathsf{stable}(\mathsf{P}) \land \mathsf{stable}(\mathsf{Q}) \Rightarrow$

$\mathsf{Join} : (\text{-}).\ \{\mathrm{emp}\}\ \{\mathsf{ret}.\ \mathsf{join}_{\mathsf{init\text{-}ch}}(\mathsf{t}, \emptyset, \emptyset, \mathsf{ret})\}$

$\mathsf{SyncChannel} : (\mathsf{j}).\ \{\mathsf{join}_{\mathsf{init\text{-}ch}}(\mathsf{t}, \mathsf{C}_A, \mathsf{C}_S,\ \mathsf{j})\}$
$\{\mathsf{ret}.\ \mathsf{join}_{\mathsf{init\text{-}ch}}(\mathsf{t}, \mathsf{C}_A, \mathsf{C}_S \cup \{\mathsf{ret}\},\ \mathsf{j}) * \mathsf{chan}(\mathsf{ret},\ \mathsf{j}) * \mathsf{ret} \notin \mathsf{C}_A \cup \mathsf{C}_S\}$

$\mathsf{AsyncChannel} : (\mathsf{j}).\ \{\mathsf{join}_{\mathsf{init\text{-}ch}}(\mathsf{t}, \mathsf{C}_A, \mathsf{C}_S,\ \mathsf{j})\}$
$\{\mathsf{ret}.\ \mathsf{join}_{\mathsf{init\text{-}ch}}(\mathsf{t}, \mathsf{C}_A \cup \{\mathsf{ret}\}, \mathsf{C}_S,\ \mathsf{j}) * \mathsf{chan}(\mathsf{ret},\ \mathsf{j}) * \mathsf{ret} \notin \mathsf{C}_A \cup \mathsf{C}_S\}$

$\mathsf{Join.When} : (\mathsf{c}).\ \{\mathsf{join}_{\mathsf{init\text{-}pat}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathtt{this}) * \mathsf{chan}(\mathsf{c}, \mathsf{j})\}$
$\{\mathsf{ret}.\ \mathsf{join}_{\mathsf{init\text{-}pat}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathtt{this}) * \mathsf{pattern}(\mathsf{ret}, \mathsf{j}, \{\ \mathsf{c}\})\}$

$\mathsf{Pattern.And} : (\mathsf{c}).\ \{\mathsf{join}_{\mathsf{init\text{-}pat}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathsf{j}) * \mathsf{pattern}(\mathtt{this}, \mathsf{j}, \mathsf{X}) * \mathsf{chan}(\mathsf{c}, \mathsf{j})\}$
$\{\mathsf{ret}.\ \mathsf{join}_{\mathsf{init\text{-}pat}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathsf{j}) * \mathsf{pattern}(\mathsf{ret}, \mathsf{j}, \mathsf{X} \cup \{\mathsf{c}\})\}$

$$\mathsf{Pattern.Do} : (\mathsf{act}).\ \left\{ \begin{array}{l} \mathsf{join}_{\mathsf{init\text{-}pat}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathsf{j}) * \mathsf{pattern}(\mathtt{this}, \mathsf{j}, \mathsf{X}) * \circledast_{\mathsf{z} \in \mathsf{Z}} \mathsf{chan}(\mathsf{z}, \mathsf{j}) * \\ \quad \forall \mathsf{Y} : \mathcal{P}_m(\mathrm{Val} \times \mathrm{Val}).\ \pi_{\mathsf{chan}}(\mathsf{Y}) = \mathsf{X} \Rightarrow \\ \qquad \mathsf{act} \mapsto \{\mathsf{join}_{\mathsf{call}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathsf{j}) * \circledast_{\mathsf{z} \in \mathsf{Z}} \mathsf{chan}(\mathsf{z}, \mathsf{j}) * \circledast_{\mathsf{y} \in \mathsf{Y}} \mathsf{P}(\mathsf{y})\} \\ \qquad\quad \{\circledast_{\mathsf{y} \in \mathsf{Y}} \mathsf{Q}(\mathsf{y})\} \end{array} \right\}$$
$\{\mathsf{join}_{\mathsf{init\text{-}pat}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathsf{j})\}$

$\mathsf{AsyncChannel.Call}, \mathsf{SyncChannel.Call} : (\text{-}).\ \{\mathsf{join}_{\mathsf{call}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathsf{j}) * \mathsf{chan}(\mathtt{this}, \mathsf{j}) * \mathsf{P}(\mathsf{a}, \mathtt{this})\}$
$\{\mathsf{join}_{\mathsf{call}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathsf{j}) * \mathsf{chan}(\mathtt{this}, \mathsf{j}) * \mathsf{Q}(\mathsf{a}, \mathtt{this})\}$

$\boldsymbol{valid}\ (\mathsf{join}_{\mathsf{call}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathsf{j}) \Leftrightarrow \mathsf{join}_{\mathsf{call}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathsf{j}) * \mathsf{join}_{\mathsf{call}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathsf{j}))$

$\boldsymbol{valid}\ (\mathsf{chan}(\mathsf{c}, \mathsf{j}) \Leftrightarrow \mathsf{chan}(\mathsf{c}, \mathsf{j}) * \mathsf{chan}(\mathsf{c}, \mathsf{j}))$

$\mathsf{stable}(\mathsf{join}_{\mathsf{init\text{-}ch}}) \land \mathsf{stable}(\mathsf{join}_{\mathsf{init\text{-}pat}}) \land \mathsf{stable}(\mathsf{join}_{\mathsf{call}}) \land \mathsf{stable}(\mathsf{chan}) \land \mathsf{stable}(\mathsf{pattern})$

$(\forall \mathsf{a} : \mathrm{Val}.\ \forall \mathsf{c} \in \mathsf{C}_A.\ \mathsf{Q}(\mathsf{a}, \mathsf{c}) = \mathrm{emp}) \Rightarrow \mathsf{join}_{\mathsf{init\text{-}ch}}(\mathsf{t}, \mathsf{C}_A, \mathsf{C}_S, \mathsf{j}) \sqsubseteq \mathsf{join}_{\mathsf{init\text{-}pat}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathsf{j})$

$\mathsf{join}_{\mathsf{init\text{-}pat}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathsf{j}) \sqsubseteq \mathsf{join}_{\mathsf{call}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathsf{j})$

4 Proof

In the previous sections we introduced our naive lock-based implementation of the joins library and formalized our high-level joins specification as a proposition in our specification logic. In this section we prove that our naive lock-based implementation satisfies this specification. As the full implementation consists of approximately 150 lines of mini C# code, to make the proof manageable, the proof is written in the form of proof outlines (that is, code with inline assertions about the current state at that point of execution).

We thus have to instantiate each of the existentially quantified predicates from the specification and prove that every method satisfies its specification with these instantiations. In addition, the implementation contains several internal methods that also needs to be specified and verified. For the Message, MessagePool, and Pattern classes it is possible to define their representation predicates independently of the rest of the joins library. However, for the remaining classes this is not the case due to the possibility of reentrant continuations. We will thus start by verifying the Message, MessagePool and Pattern classes.

The proofs of the List class and Pair class are completely standard, and have thus been omitted. Their specifications are given in Section 4.6. The Lock specification and proof follows by a slightly generalization of the spin-lock example in [2]. The Lock specification is also given in Section 4.6.

4.1 The Message class

Message objects are very simple; they consist of a single integer field, status, which indicates whether the message has been received (status = 0) or not (status = 1). However, conceptually, message objects are significantly more complicated: When a client sends a synchronous message the client (1) creates a new message, (2) transfers ownership of the channel pre-condition to the message, (3) enters a busy-loop waiting for someone to receive the message, before (4) transferring back ownership of the channel post-condition to the sender. During the busy-loop, (3), other threads are allowed to match the message and receive the message, transferring ownership of the channel pre-condition from the message to the continuation and ownership of the channel post-condition from the continuation back to the message.

Conceptually, messages can thus be in one of four states, `pending`, `removed`, `received`, and `released`. Each message starts out in the `pending` state. When a message is matched by a pattern it transitions to the `removed` state. When the continuation of the chord that matched the message terminates, it transitions to the `received` state. Lastly, when the client that inserted the message in the first place notices that the message has been received, and exits its busy-loop, the message transitions to the `released` state. In the pending state the message owns its channel pre-condition and the `status` field is 0. In the `removed` state the `status` field is 0, but it no longer owns its channel precondition. In the `received` state the `status` field is 1 and it owns its channel postcondition. Lastly, in the `released` state the `status` field is 1, but it no longer owns its channel post-condition. Crucially, while every thread is allowed to transition messages from the `pending` state to

the `removed` and `received` state, only the client that inserted the message is allowed to transition the message to the `released` state.

We can thus specify the Message constructor in terms of three message representation predicates, $\mathsf{msgprotocol}(-)$, $\mathsf{pending}(-)$ and $\mathsf{msg}(-)$, as follows:

$$\begin{aligned}\mathsf{new\ Message} : (\text{-}).\ &\{\mathsf{P}(\mathsf{a},\mathsf{c}) * \mathsf{msgprotocol}(\mathsf{t},\mathsf{P},\mathsf{Q})\}\\ &\{\mathsf{ret}.\ \mathsf{pending}(\mathsf{t},\mathsf{P},\mathsf{Q},\mathsf{ret},\mathsf{c}) * \mathsf{msg}(\mathsf{t},\mathsf{P},\mathsf{Q},\mathsf{a},\mathsf{ret},\mathsf{c})\}\end{aligned}$$

Here $\mathsf{msgprotocol}(\mathsf{t},\mathsf{P},\mathsf{Q})$ asserts that the protocol on region type t is a message protocol. The $\mathsf{pending}(\mathsf{t},\mathsf{P},\mathsf{Q},\mathsf{ret},\mathsf{c})$ predicate asserts that ret refers to a message on channel c that is currently in the `pending` state *and* asserts permission to transition the message to the `removed` and `received` state *and* asserts ownership of the channel pre-condition. Likewise, $\mathsf{msg}(\mathsf{t},\mathsf{P},\mathsf{Q},\mathsf{a},\mathsf{ret},\mathsf{c})$ asserts that ret refers to a message on channel c that is currently in the `pending`, `removed` or `received` state *and* asserts permission to transition the message from the `received` state to the `released` state.

These predicates should thus satisfy,

$$\begin{aligned}\mathsf{pending}(\mathsf{t},\mathsf{P},\mathsf{Q},\mathsf{x},\mathsf{c}) &\sqsubseteq \exists \mathsf{a} : \mathrm{Val}.\ \mathsf{P}(\mathsf{a},\mathsf{c}) * \mathsf{removed}(\mathsf{t},\mathsf{P},\mathsf{Q},\mathsf{a},\mathsf{x},\mathsf{c})\\ \mathsf{Q}(\mathsf{a},\mathsf{c}) * \mathsf{removed}(\mathsf{t},\mathsf{P},\mathsf{Q},\mathsf{a},\mathsf{x},\mathsf{c}) &\sqsubseteq \mathsf{received}(\mathsf{t},\mathsf{P},\mathsf{Q},\mathsf{x},\mathsf{c})\\ \mathsf{msg}(\mathsf{t},\mathsf{P},\mathsf{Q},\mathsf{a},\mathsf{x},\mathsf{c}) * \mathsf{received}(\mathsf{t},\mathsf{P},\mathsf{Q},\mathsf{x},\mathsf{c}) &\sqsubseteq \mathsf{Q}(\mathsf{a},\mathsf{c})\end{aligned}$$

Here $\mathsf{removed}(\mathsf{t},\mathsf{P},\mathsf{Q},\mathsf{a},\mathsf{x},\mathsf{c})$ asserts that x refers to a message on channel c in the `removed` state and asserts permission to transition it to the `received` state. Likewise, $\mathsf{received}(\mathsf{t},\mathsf{P},\mathsf{Q},\mathsf{x},\mathsf{c})$ simply asserts that x refers to a message on channel c in the `received` state.

When adding a new message to the message pool, the client that created the message thus keeps the $\mathsf{msg}(-)$ assertion, and transfer ownership of the $\mathsf{pending}(-)$ assertion to the message pool. All the message pools are protected by a global lock, allowing the $\mathsf{pending}(-)$ assertions (and thus channel pre-conditions) to be shared through the lock.

Predicate definitions

We can express this sharing and ownership pattern in CAP, using a shared region to transfer the channel post-condition from the message recipient to the sender of the message. We need two actions, Insert, for transferring ownership of the channel post-condition to the shared region, and Remove, for transferring ownership of the channel post-condition out of the shared region. Given channel pre-conditions P and channel post-conditions Q the protocol on message x on channel c with logical argument a is:

$$\mathsf{I}(\mathsf{Q})(\mathsf{x},\mathsf{c},\mathsf{a}) \stackrel{\text{def}}{=} \left(\begin{array}{r} \textsc{Insert} : \mathsf{x.status} \mapsto 0 \rightsquigarrow \mathsf{x.status} \mapsto 1 * \mathsf{Q}(\mathsf{a},\mathsf{c}) \\ \textsc{Remove} : \mathsf{x.status} \mapsto 1 * \mathsf{Q}(\mathsf{a},\mathsf{c}) \rightsquigarrow \mathsf{x.status} \mapsto 1 \\ \tau_1 : \mathsf{x.status} \mapsto 0 \rightsquigarrow \mathsf{x.status} \mapsto 0 \\ \tau_2 : \mathsf{x.status} \mapsto 1 * \mathsf{Q}(\mathsf{a},\mathsf{c}) \rightsquigarrow \mathsf{x.status} \mapsto 1 * \mathsf{Q}(\mathsf{a},\mathsf{c}) \end{array}\right)$$

Here $\mathsf{I}(\mathsf{Q})$ is thus a parametric protocol with parameters $(\mathsf{x}, \mathsf{c}, \mathsf{a})$. We can now define the message representation predicates as follows:

$$\begin{aligned}
\mathsf{msgprotocol}(\mathsf{t}, \mathsf{P}, \mathsf{Q}) &\stackrel{\text{def}}{=} \mathsf{protocol}(\mathsf{t}, \mathsf{I}(\mathsf{Q})) \\
\mathsf{pending}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathsf{x}, \mathsf{c}) &\stackrel{\text{def}}{=} \exists \mathsf{r} : \mathrm{RId}.\ \exists \pi \in \mathrm{Perm}.\ \exists \mathsf{a} : \mathrm{Val}. \\
&\qquad \mathsf{x}_{larg} \mapsto \mathsf{a} * \mathsf{P}(\mathsf{a}, \mathsf{c}) * [\textsc{Insert}]^{\mathsf{r}}_1 * [\tau_1]^{\mathsf{r}}_{\pi} * [\tau_2]^{\mathsf{r}}_{\pi} \\
&\qquad * \boxed{\mathsf{x.status} \mapsto 0}^{\mathsf{r},\mathsf{t},(\mathsf{x},\mathsf{a},\mathsf{c})}_{\mathsf{I}(\mathsf{Q})} \\
\mathsf{removed}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathsf{a}, \mathsf{x}, \mathsf{c}) &\stackrel{\text{def}}{=} \exists \mathsf{r} : \mathrm{RId}.\ \exists \pi \in \mathrm{Perm}. \\
&\qquad \mathsf{x}_{larg} \mapsto \mathsf{a} * [\textsc{Insert}]^{\mathsf{r}}_1 * [\tau_1]^{\mathsf{r}}_{\pi} * [\tau_2]^{\mathsf{r}}_{\pi} \\
&\qquad * \boxed{\mathsf{x.status} \mapsto 0}^{\mathsf{r},\mathsf{t},(\mathsf{x},\mathsf{a},\mathsf{c})}_{\mathsf{I}(\mathsf{Q})} \\
\mathsf{received}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathsf{x}, \mathsf{c}) &\stackrel{\text{def}}{=} \exists \mathsf{r} : \mathrm{RId}.\ \exists \pi \in \mathrm{Perm}.\ \exists \mathsf{a} : \mathrm{Val}. \\
&\qquad \mathsf{x}_{larg} \mapsto \mathsf{a} * [\textsc{Insert}]^{\mathsf{r}}_1 * [\tau_1]^{\mathsf{r}}_{\pi} * [\tau_2]^{\mathsf{r}}_{\pi} \\
&\qquad * \boxed{\mathsf{x.status} \mapsto 1 * (\mathsf{Q}(\mathsf{a}, \mathsf{c}) \vee \mathrm{emp})}^{\mathsf{r},\mathsf{t},(\mathsf{x},\mathsf{a},\mathsf{c})}_{\mathsf{I}(\mathsf{Q})} \\
\mathsf{msg}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathsf{a}, \mathsf{x}, \mathsf{c}) &\stackrel{\text{def}}{=} \exists \mathsf{r} : \mathrm{RId}.\ \exists \pi \in \mathrm{Perm}. \\
&\qquad \mathsf{x}_{larg} \mapsto \mathsf{a} * [\textsc{Remove}]^{\mathsf{r}}_1 * [\tau_1]^{\mathsf{r}}_{\pi} * [\tau_2]^{\mathsf{r}}_{\pi} \\
&\qquad * \boxed{\mathsf{x.status} \mapsto 0 \vee (\mathsf{x.status} \mapsto 1 * \mathsf{Q}(\mathsf{a}, \mathsf{c}))}^{\mathsf{r},\mathsf{t},(\mathsf{x},\mathsf{a},\mathsf{c})}_{\mathsf{I}(\mathsf{Q})}
\end{aligned}$$

Stability

By assumption, all channel pre- and post-conditions are uniformly expressible using state-independent protocols:

$$\begin{aligned}
&\exists \mathsf{R} : \mathrm{Prop}.\ \exists \mathsf{S}_P, \mathsf{S}_Q : \mathrm{Val} \times \mathrm{Val} \to \mathrm{Prop}.\ \mathsf{pure}_{\mathsf{state}}(R) \wedge \\
&\quad (\forall \mathsf{x} : \mathrm{Val} \times \mathrm{Val}.\ (\mathsf{P}(\mathsf{x}) \Leftrightarrow \mathsf{S}_P(\mathsf{x}) * \mathsf{R}) \wedge \mathsf{pure}_{\mathsf{protocol}}(\mathsf{S}_P(\mathsf{x}))) \wedge \\
&\quad (\forall \mathsf{x} : \mathrm{Val} \times \mathrm{Val}.\ (\mathsf{Q}(\mathsf{x}) \Leftrightarrow \mathsf{S}_Q(\mathsf{x}) * \mathsf{R}) \wedge \mathsf{pure}_{\mathsf{protocol}}(\mathsf{S}_Q(\mathsf{x})))
\end{aligned}$$

Assuming that channel pre- and post-conditions are independent of the message region type t, by rule StableA we can thus prove stability of the message representation predicates by proving that the boxed assertions are closed under actions potentially owned by the environment. For instance, $\mathsf{pending}(-)$ asserts the status field is 0 which is stable as it also asserts full ownership of the only action that allows the status field to change, namely, the Insert action. Formally, we prove that,

$$\begin{aligned}
&\forall \mathsf{x}, \mathsf{c}, \mathsf{a} : \mathrm{Val}.\ \forall \mathsf{t} : \mathrm{RType}.\ (\forall \mathsf{a}, \mathsf{c} : \mathrm{Val}.\ \mathsf{indep}_{\mathsf{t}}(\mathsf{P}(\mathsf{a}, \mathsf{c})) \wedge \mathsf{indep}_{\mathsf{t}}(\mathsf{Q}(\mathsf{a}, \mathsf{c}))) \Rightarrow \\
&\quad \mathsf{stable}(\mathsf{pending}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathsf{x}, \mathsf{c})) \wedge \mathsf{stable}(\mathsf{removed}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathsf{a}, \mathsf{x}, \mathsf{c})) \wedge \\
&\quad \mathsf{stable}(\mathsf{received}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathsf{x}, \mathsf{c})) \wedge \mathsf{stable}(\mathsf{msg}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathsf{a}, \mathsf{x}, \mathsf{c}))
\end{aligned}$$

Proof outline

The proof of the constructor is now fairly straightforward: we allocate a new phantom field, larg, set the status field to 0, and setup a new region with region type t and region arguments (x, a, c).

```
public class Message {
  public int status;

  public Message() {
      {this_larg ↦ a * this.status ↦ _ * P(a, c) * protocol(t, I(Q))}
    this.status = 0;
      {this_larg ↦ a * this.status ↦ 0 * P(a, c) * protocol(t, I(Q))}
      {∃r ∈ RId. this_larg ↦ a * [this.status ↦ 0]^{r,t,(x,a,c)}_{I(Q)}
         * [INSERT]^r_1 * [REMOVE]^r_1 * [τ1]^r_1 * [τ2]^r_1 * P(a, c)}
      {pending(t, P, Q, this, c) * msg(t, P, Q, a, this, c)}
  }
}
```

4.2 The MessagePool class

The abstract MessagePool class implements a message pool. Each message pool maintains a list of pending messages. Message pools supports three methods:

- AddMessage: The AddMessage method creates a new message, adds it to the message pool and returns a reference to the new message.
- Pop: The Pop method optimistically tries to pop a pending message from the message pool, returning `null` if there are no pending messages.
- Push: The Push method adds a given message to the message pool.

Formally, the MessagePool class satisfies the following specification.

$$\begin{aligned}&\mathsf{MessagePool.AddMessage} : (\text{-}).\ \{\mathsf{pool}(t, P, Q, \texttt{this}) * P(a, \texttt{this})\}\\ &\qquad\qquad\qquad\qquad\qquad\qquad \{\mathsf{ret}.\ \mathsf{pool}(t, P, Q, \texttt{this}) * \mathsf{msg}(t, P, Q, a, \mathsf{ret}, \texttt{this})\}\end{aligned}$$

$$\begin{aligned}&\mathsf{MessagePool.Pop} : (\text{-}).\ \{\mathsf{pool}(t, P, Q, \texttt{this})\}\\ &\qquad\qquad\qquad\qquad \{\mathsf{ret}.\ \mathsf{pool}(t, P, Q, \texttt{this}) * (\mathsf{ret} = \texttt{null} \vee \mathsf{pending}(t, P, Q, \mathsf{ret}, \texttt{this})\}\end{aligned}$$

$$\begin{aligned}&\mathsf{MessagePool.Push} : (m).\ \{\mathsf{pool}(t, P, Q, \texttt{this}) * \mathsf{pending}(t, P, Q,\ m, \texttt{this})\}\\ &\qquad\qquad\qquad\qquad\qquad\qquad \{\mathsf{pool}(t, P, Q, \texttt{this})\}\end{aligned}$$

Here the pool(t, P, Q, x) predicate asserts ownership of the message pool x. The AddMessage method thus returns the msg(−) assertion, representing the message sender handle, but keeps the pending(−) assertion, representing the message receiver handle.

Predicate definitions

The pool(−) predicate simply asserts that the queue field refers a list and each element of that list is a message in the pending state.

$$\mathsf{pool}(\mathsf{t},\mathsf{P},\mathsf{Q},\mathsf{c}) \stackrel{\text{def}}{=} \exists \mathsf{x} : \mathrm{Val}.\ \exists \mathsf{M} : \mathcal{P}_m(\mathrm{Val}).\ \mathsf{c.queue} \mapsto \mathsf{x} * \mathsf{lst}(\mathsf{x},\mathsf{M}) \\ * \ \mathsf{protocol}(\mathsf{t},\mathsf{I}(\mathsf{Q})) * \circledast_{\mathsf{m}\in\mathsf{M}} \mathsf{pending}(\mathsf{t},\mathsf{P},\mathsf{Q},\mathsf{m},\mathsf{c})$$

Proof outline

```
public abstract class MessagePool {
  public List<Message> queue;

  public MessagePool() {
      {this.queue ↦ null * protocol(t, I(Q))}
    this.queue = new List<Message>();
      {∃x. this.queue ↦ x * lst(x, ∅)}
      {pool(t, P, Q, this)}
  }

  public Message AddMessage() {
    List<Message> q; Message m;
      {pool(t, P, Q, this) * P(a, this)}
    m = new Message();
      {pool(t, P, Q, this) * pending(t, P, Q, m, this) * msg(t, P, Q, a, m, this)}
    q = this.queue;
      {this.queue ↦ q * lst(q, M) * protocol(t, I(Q)) * ⊛_{m∈M} pending(t, P, Q, m, this)
          * pending(t, P, Q, m, this) * msg(t, P, Q, a, m, this)}
    q.Push(m);
      {this.queue ↦ q * lst(q, {m} ∪ M) * protocol(t, I(Q)) * ⊛_{m∈M} pending(t, P, Q, m, this)
          * pending(t, P, Q, m, this) * msg(t, P, Q, a, m, this)}
      {this.queue ↦ q * lst(q, {m} ∪ M) * protocol(t, I(Q)) * ⊛_{m∈{m}∪M} pending(t, P, Q, m, this)
          * msg(t, P, Q, a, m, this)}
      {pool(t, P, Q, this) * msg(t, P, Q, a, m, this)}
    return m;
      {ret. pool(t, P, Q, this) * msg(t, P, Q, a, ret, this)}
  }

  public Message Pop() {
    List<Message> q; Message m;
      {pool(t, P, Q, this)}
    q = this.queue;
      {this.queue ↦ q * lst(q, M) * protocol(t, I(Q)) * ⊛_{m∈M} pending(t, P, Q, m, this)}
        {lst(q, M) * ⊛_{m∈M} pending(t, P, Q, m, this)}
```

```
    if (q.Count > 0) {
        {lst(q, M) ∗ ⊛m∈M pending(t, P, Q, m, this) ∗ M ≠ ∅}
      m = q.Pop();
        {lst(q, M′) ∗ ⊛m∈M pending(t, P, Q, m, this) ∗ M = M′ ∪ {m}}
        {∃M. lst(q, M) ∗ (⊛m∈M pending(t, P, Q, m, this)) ∗ pending(t, P, Q, m, this)}
    } else {
        {lst(q, M) ∗ ⊛m∈M pending(t, P, Q, m, this)}
      m = null;
        {lst(q, M) ∗ ⊛m∈M pending(t, P, Q, m, this) ∗ m = null}
    }
        {∃M. lst(q, M) ∗ (⊛m∈M pending(t, P, Q, m, this)) ∗ (m = null ∨ pending(t, P, Q, m, this)}
      {∃M. this.queue ↦ q ∗ lst(q, M) ∗ protocol(t, I(Q)) ∗ ⊛m∈M pending(t, P, Q, m, this)
          ∗ (m = null ∨ pending(t, P, Q, m, this)}
      {pool(t, P, Q, this) ∗ (m = null ∨ pending(t, P, Q, m, this)}
    return m;
      {ret. pool(t, P, Q, this) ∗ (ret = null ∨ pending(t, P, Q, ret, this)}
  }

  public void Push(Message m) {
    List<Message> q;
      {pool(t, P, Q, this) ∗ pending(t, P, Q, m, this)}
    q = this.queue;
      {this.queue ↦ q ∗ lst(q, M) ∗ protocol(t, I(Q))
          ∗ pending(t, P, Q, m, this) ∗ ⊛m∈M pending(t, P, Q, m, this)}
      {this.queue ↦ q ∗ lst(q, M) ∗ protocol(t, I(Q))
          ∗ ⊛m∈{m}∪M pending(t, P, Q, m, this)}
        q.Push(m);
      {this.queue ↦ q ∗ lst(q, {m} ∪ M) ∗ protocol(t, I(Q))
          ∗ ⊛m∈{m}∪M pending(t, P, Q, m, this)}
      {pool(t, P, Q, this)}
  }
}
```

4.3 The Pattern class

Pattern objects represent conditions on the message pool. This version of the joins library supports conditions of the form:

> match a distinct message from each channel x from the multiset of channels X

Conditions are thus implemented as a list of channels representing the multiset X.

In addition to its public methods, the Pattern class has an internal method, Matches to determine whether the pattern matches the current message pool. If it does, Matches returns a list of messages that matches the pattern; otherwise, it returns null. Its speci-

fication is as follows:

$$\begin{array}{l}\mathsf{Pattern.Matches}: (\text{-})\\ \quad \{\mathsf{pattern}_{\mathsf{internal}}(\mathtt{this}, \mathsf{j}, \mathsf{X}) * \circledast_{\mathsf{x}\in\mathsf{X}}\mathsf{pool}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathsf{x})\}\\ \quad \left\{\begin{array}{l}\mathsf{ret}.\ \exists \mathsf{Y} : \mathcal{P}_m(\mathrm{Val} \times \mathrm{Val}).\ \mathsf{pattern}_{\mathsf{internal}}(\mathtt{this}, \mathsf{j}, \mathsf{X}) * \circledast_{\mathsf{x}\in\mathsf{X}}\mathsf{pool}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathsf{x})\\ \qquad (\mathsf{ret} = \mathtt{null} \vee (\mathsf{lst}(\mathsf{ret}, \pi_2(\mathsf{Y})) * \circledast_{(\mathsf{c},\mathsf{m})\in\mathsf{Y}}\mathsf{pending}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathsf{m}, \mathsf{c}) * \pi_1(\mathsf{Y}) = \mathsf{X}))\end{array}\right\}\end{array}$$

Here $\mathsf{pattern}_{\mathsf{internal}}(\mathsf{p}, \mathsf{j}, \mathsf{X})$ asserts that p refers to a condition on join instance j that matches a distinct message from each channel of the multiset of channels X.

Predicate definitions

The $\mathsf{pattern}(\mathsf{p}, \mathsf{j}, \mathsf{X})$ predicate asserts full ownership of the underlying the underlying list of channels, whereas $\mathsf{pattern}_{\mathsf{internal}}(\mathsf{p}, \mathsf{j}, \mathsf{X})$ only asserts read-only ownership. Both predicates assert that all the channels of the pattern belong to the correct join instance j using the $\mathsf{chan}(\text{-})$ predicate:

$$\mathsf{pattern}(\mathsf{p}, \mathsf{j}, \mathsf{X}) \stackrel{\text{def}}{=} \exists \mathsf{x} : \mathrm{Val}.\ \mathsf{p.join} \mapsto \mathsf{j} * \mathsf{p.channels} \mapsto \mathsf{x} * \mathsf{lst}(\mathsf{x}, \mathsf{X}) * \circledast_{\mathsf{c}\in\mathsf{X}}\mathsf{chan}(\mathsf{c}, \mathsf{j})$$

$$\mathsf{pattern}_{\mathsf{internal}}(\mathsf{p}, \mathsf{j}, \mathsf{X}) \stackrel{\text{def}}{=} \exists \mathsf{x} : \mathrm{Val}.\ \mathsf{p.join} \mathrel{\overset{\mathsf{r}}{\mapsto}} \mathsf{j} * \mathsf{p.channels} \mathrel{\overset{\mathsf{r}}{\mapsto}} \mathsf{x} * \mathsf{lst}_r(\mathsf{x}, \mathsf{X}) * \circledast_{\mathsf{c}\in\mathsf{X}}\mathsf{chan}(\mathsf{c}, \mathsf{j})$$

The $\mathsf{chan}(\mathsf{c},\mathsf{j})$ predicate asserts that c refers to a channel registered with join instance j. For a channel to be registered with a join instance j, its join field should contain a reference to j *and* the channel should be in the channel list of join instance j. Since we require that the $\mathsf{chan}(\text{-})$ predicate be duplicable, it cannot assert full ownership of the channel list of join instance j. Furthermore, since we use the $\mathsf{chan}(\text{-})$ predicate in the join initialization phases (while the channel list is still growing), $\mathsf{chan}(\text{-})$ cannot assert read-only ownership of the underlying channel list either. Instead, we extend the join instance j with a phantom field chans containing a finite multiset of channels, with a protocol that restricts modifications to addition of new channels:

$$\mathsf{chan}(\mathsf{c}, \mathsf{j}) \stackrel{\text{def}}{=} \exists \mathsf{t} \in \mathrm{RType}.\ \mathsf{j}_{reg_c} \mathrel{\overset{\mathsf{r}}{\mapsto}} \mathsf{t} * \mathsf{c.join} \mathrel{\overset{\mathsf{r}}{\mapsto}} \mathsf{j} * \mathsf{c} \in^{\mathsf{t}} \mathsf{j}_{\mathsf{chans}}$$

Here $\mathsf{c} \in^{\mathsf{t}} \mathsf{j}_{\mathsf{chans}}$ asserts that the channel c is a member of the finite multiset that *chans* contains:

$$\mathsf{c} \in^{\mathsf{t}} \mathsf{j}_{\mathsf{chans}} \stackrel{\text{def}}{=} \exists \mathsf{r} : \mathrm{RId}.\ \exists \mathsf{X} : \mathcal{P}_m(\mathrm{Val}).\ \boxed{\mathsf{j}_{\mathsf{chans}} \mapsto \mathsf{X}}^{\mathsf{r},\mathsf{t},\mathsf{j}}_{\mathsf{I}} * \mathsf{c} \in \mathsf{X}$$

where I is the parametric protocol:

$$\mathsf{I}(\mathsf{j}) = (\textsc{Add} : (\mathsf{Y} : \mathcal{P}_m(\mathrm{Val}), \mathsf{x} : \mathrm{Val}) : \mathsf{j}_{\mathsf{chans}} \mapsto \mathsf{Y} \rightsquigarrow \mathsf{j}_{\mathsf{chans}} \mapsto \mathsf{Y} \cup \{\mathsf{x}\})$$

Clearly $\mathsf{c} \in^{\mathsf{t}} \mathsf{j}_{\mathsf{chans}}$ is stable as the protocol only allows new channels to be added. Furthermore, the $\mathsf{chan}(-)$ predicate is freely duplicable:

$$\mathsf{chan}(\mathsf{c}, \mathsf{j}) \Leftrightarrow \mathsf{chan}(\mathsf{c}, \mathsf{j}) * \mathsf{chan}(\mathsf{c}, \mathsf{j})$$

Proof outline

Morally, Matches is implemented as follows,

```
internal List<Message> Matches() {
  List<Pair<Message, Channel>> consumed = new List<Pair<Message, Channel>>();
  bool rollback;

  channels.ForEach(ch => {
    if(!rollback) {
      Message msg = ch.Pop();
      if (msg != null)
        consumed.Push(new Pair<Message, Channel>(msg, ch));
      else
        rollback = true;
    }
  });

  if(rollback) {
    consumed.ForEach(p => {
      p.snd.Push(p.fst);
    });
    return null;
  } else {
    return consumed.Map(List.pi2<Message, Channel>);
  }
}
```

using the higher-order List.ForEach method to optimistically try to Pop a message from each channel of the pattern. However, as mini C# lacks anonymous delegates (to avoid the difficulties of reasoning about variable capture [1]) we introduce an explicit inner class, Inner, with fields to replace the captured variables consumed and rollback.

Define $\mathsf{inner}(-)$ as the following representation predicate. $\mathsf{inner}(-)$ describes the loop invariant of the first `ForEach` call. Informally, $\mathsf{inner}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathsf{x}, \mathsf{X}, \mathsf{Y})$ asserts that either the consumed field refers to a list of messages, with one message from each channel in the multiset Y, or rollback is true.

$$
\begin{aligned}
&\mathsf{inner}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathsf{x}, \mathsf{X}, \mathsf{Y}) = \\
&\quad \exists \mathsf{Z} : \mathcal{P}_m(\mathrm{Val} \times \mathrm{Val}).\ \exists \mathsf{r}, \mathsf{y} : \mathrm{Val}. \\
&\qquad \mathsf{x.consumed} \mapsto \mathsf{y} * \mathsf{lst}_{pair}(\mathsf{y}, \mathsf{Z}) * \circledast_{\mathsf{x} \in \mathsf{X}} \mathsf{pool}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathsf{x}) \\
&\qquad \circledast_{(\mathsf{c},\mathsf{m}) \in \mathsf{Z}}\ \mathsf{pending}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathsf{m}, \mathsf{c}) * \mathsf{x.rollback} \mapsto \mathsf{r} * (\mathsf{r} = \mathtt{true} \vee \pi_1(\mathsf{Z}) = \mathsf{Y})
\end{aligned}
$$

```
public class Inner {
  public List<Pair<Channel, Message>> consumed;
  public bool rollback;
```

```
public Inner() {
    {this.consumed ↦ null ∗ this.rollback ↦ _ ∗ ⊛x∈X pool(t, P, Q, x)}
  consumed = new List<Pair<Channel, Message>>();
  rollback = false;
    {this.consumed ↦ x ∗ lst_pair(x, ∅) ∗ this.rollback ↦ false ∗ ⊛x∈X pool(t, P, Q, x)}
    {inner(t, P, Q, this, X, ∅)}
}

public void Pop(Channel ch) {
  List<Pair<Channel, Message>> con;
  Pair<Channel, Message> p;
  Message msg;
    {inner(t, P, Q, this, X, Y) ∗ ch ∈ X}
  con = this.consumed;
    {this.consumed ↦ con ∗ lst_pair(con, Z) ∗ ⊛x∈X pool(t, P, Q, x) ∗ ⊛(c,m)∈Z pending(t, P, Q, m, c)
       ∗ this.rollback ↦ r ∗ (r = true ∨ π1(Z) = Y) ∗ ch ∈ X}
      {lst_pair(con, Z) ∗ pool(t, P, Q, ch) ∗ ⊛(c,m)∈Z pending(t, P, Q, m, c)
         ∗ this.rollback ↦ r ∗ (r = true ∨ π1(Z) = Y)}
  if (!rollback) {
      {lst_pair(con, Z) ∗ pool(t, P, Q, ch) ∗ ⊛(c,m)∈Z pending(t, P, Q, m, c)
         ∗ this.rollback ↦ false ∗ π1(Z) = Y}
    msg = ch.Pop();
      {lst_pair(con, Z) ∗ pool(t, P, Q, ch) ∗ ⊛(c,m)∈Z pending(t, P, Q, m, c)
         ∗ this.rollback ↦ false ∗ π1(Z) = Y
         ∗ (msg = null ∨ pending(t, P, Q, msg, ch))}
    if (msg != null) {
        {lst_pair(con, Z) ∗ pool(t, P, Q, ch) ∗ ⊛(c,m)∈Z pending(t, P, Q, m, c)
           ∗ this.rollback ↦ false ∗ π1(Z) = Y ∗ pending(t, P, Q, msg, ch))}
      con.Push(new Pair<Channel, Message>(ch, msg));
        {lst_pair(con, {(ch, msg)} ∪ Z) ∗ pool(t, P, Q, ch) ∗ ⊛(c,m)∈{(ch,msg)}∪Z pending(t, P, Q, m, c)
           ∗ this.rollback ↦ false ∗ π1(Z) = Y)}
    } else {
        {lst_pair(con, Z) ∗ pool(t, P, Q, ch) ∗ ⊛(c,m)∈Z pending(t, P, Q, m, c)
           ∗ this.rollback ↦ false ∗ π1(Z) = Y ∗ msg = null)}
      rollback = true;
        {lst_pair(con, Z) ∗ pool(t, P, Q, ch) ∗ ⊛(c,m)∈Z pending(t, P, Q, m, c)
           ∗ this.rollback ↦ true ∗ π1(Z) = Y)}
    }
  }
      {∃Z. lst_pair(con, Z) ∗ pool(t, P, Q, ch) ∗ ⊛(c,m)∈Z pending(t, P, Q, m, c)
         ∗ this.rollback ↦ r ∗ (r = true ∨ π1(Z) = {ch} ∪ Y)}
```

$\{\exists Z.\ \texttt{this.consumed} \mapsto \mathsf{con} * \mathsf{lst}_{pair}(\mathsf{con}, Z) * \circledast_{x \in X} \mathsf{pool}(t, P, Q, x) * \circledast_{(c,m) \in Z} \mathsf{pending}(t, P, Q, m, c)$
$* \texttt{this.rollback} \mapsto r * (r = \texttt{true} \vee \pi_1(Z) = \{ch\} \cup Y)\}$
$\{\mathsf{inner}(t, P, Q, \texttt{this}, X, \{ch\} \cup Y)\}$
}
}

public class Pattern {
internal List<Channel> channels = **new** List<Channel>();
internal Join join;

internal Pattern(Channel ch, Join join) {
$\{\texttt{this.channels} \mapsto \texttt{null} * \texttt{this.join} \mapsto \texttt{null}\}$
Channels channels = **new** List<Channel>();
$\{\texttt{this.channels} \mapsto \texttt{null} * \texttt{this.join} \mapsto \texttt{null} * \mathsf{lst}(\mathsf{channels}, \emptyset)\}$
channels.Push(ch);
$\{\texttt{this.channels} \mapsto \texttt{null} * \texttt{this.join} \mapsto \texttt{null} * \mathsf{lst}(\mathsf{channels}, \{ch\})\}$
this.channels = channels;
this.join = join;
$\{\texttt{this.channels} \mapsto \mathsf{channels} * \texttt{this.join} \mapsto \mathsf{join} * \mathsf{lst}(\mathsf{channels}, \{ch\})\}$
}

internal List<Message> Matches() {
Inner inner; Message msg; List<Message> msgs; List<Channel> chs;
$\{\mathsf{pattern}_{\mathsf{internal}}(\texttt{this}, j, X) * \circledast_{x \in X} \mathsf{pool}(t, P, Q, x)\}$
msgs = **null**;
chs = **this**.channel;
$\{\texttt{this.join} \Rightarrow j * \texttt{this.channels} \Rightarrow \mathsf{chs} * \mathsf{lst}_r(\mathsf{chs}, X)$
$* \mathsf{msgs} = \texttt{null} * (\circledast_{x \in X} \mathsf{chan}(x, j) * \mathsf{pool}(t, P, Q, x))\}$
$\{\mathsf{lst}_r(\mathsf{chs}, X) * \mathsf{msgs} = \texttt{null} * \circledast_{x \in X} \mathsf{pool}(t, P, Q, x)\}$
inner = **new** Inner();
$\{\mathsf{lst}_r(\mathsf{chs}, X) * \mathsf{msgs} = \texttt{null} * \mathsf{inner}(t, P, Q, \mathsf{inner}, X, \emptyset)$
$* \forall Y.\ \mathsf{inner.Pop} \mapsto \begin{Bmatrix} (ch).\ \{\mathsf{inner}(t, P, Q, \mathsf{inner}, X, Y) * ch \in X\} \\ \{\mathsf{inner}(t, P, Q, \mathsf{inner}, X, \{ch\} \cup Y)\} \end{Bmatrix}$
chs.ForEach(inner.Pop);
$\{\mathsf{lst}_r(\mathsf{chs}, X) * \mathsf{msgs} = \texttt{null} * \mathsf{inner}(t, P, Q, \mathsf{inner}, X, X)\}$
$\{\mathsf{lst}_r(\mathsf{chs}, X) * \mathsf{msgs} = \texttt{null} * \circledast_{x \in X} \mathsf{pool}(t, P, Q, x)$
$* \mathsf{inner.consumed} \mapsto y * \mathsf{lst}_{pair}(y, Y) * \circledast_{(c,m) \in Y} \mathsf{pending}(t, P, Q, m, c)$
$* \mathsf{inner.rollback} \mapsto r * (r = \texttt{true} \vee \pi_1(Y) = X)\}$
if(inner.rollback) {
$\{\mathsf{lst}_r(\mathsf{chs}, X) * \mathsf{msgs} = \texttt{null} * \circledast_{x \in X} \mathsf{pool}(t, P, Q, x)$
$* \mathsf{inner.consumed} \mapsto y * \mathsf{lst}_{pair}(y, Y) * \circledast_{(c,m) \in Y} \mathsf{pending}(t, P, Q, m, c)$
$* \mathsf{inner.rollback} \mapsto \texttt{true}\}$
inner.consumed.ForEach(Push);

```
      {lst_r(chs, X) * msgs = null * ⊛_{x∈X} pool(t, P, Q, x)
          * inner.consumed ↦ y * lst_pair(y, Y) * inner.rollback ↦ true}
      {lst_r(chs, X) * ⊛_{x∈X} pool(t, P, Q, x) * msgs = null}
  } else {
      {lst_r(chs, X) * msgs = null * ⊛_{x∈X} pool(t, P, Q, x)
          * inner.consumed ↦ y * lst_pair(y, Y) * ⊛_{(c,m)∈Y} pending(t, P, Q, m, c)
          * inner.rollback ↦ false * π_1(Y) = X}
    msgs = inner.consumed.Map(pi2<Channel, Message>);
      {lst_r(chs, X) * lst(msgs, π_2(Y)) * ⊛_{x∈X} pool(t, P, Q, x)
          * inner.consumed ↦ y * lst_pair(y, Y) * ⊛_{(c,m)∈Y} pending(t, P, Q, m, c)
          * inner.rollback ↦ false * π_1(Y) = X}
  }
      {lst_r(chs, X) * ⊛_{x∈X} pool(t, P, Q, x)
          * (msgs = null ∨ (lst(msgs, π_2(Y)) * ⊛_{(c,m)∈Y} pending(t, P, Q, m, c) * π_1(Y) = X))}
    {this.join ↦ j * this.channels ↦ chs * lst_r(chs, X) * (⊛_{x∈X} chan(x, j) * pool(t, P, Q, x))
        * (msgs = null ∨ (lst(msgs, π_2(Y)) * ⊛_{(c,m)∈Y} pending(t, P, Q, m, c) * π_1(Y) = X))}
    {pattern_internal(this, j, X) * ⊛_{x∈X} pool(t, P, Q, x)
        * (msgs = null ∨ (lst(msgs, π_2(Y)) * ⊛_{(c,m)∈Y} pending(t, P, Q, m, c) * π_1(Y) = X))}
  return msgs;
}

internal void Push(Pair<Channel, Message> p) {
  Channel ch; Message msg;
    {∃c, m. pair(p, c, m) * pending(t, P, Q, m, c) * pool(t, P, Q, c)}
  ch = p.Fst();
  msg = p.Snd();
    {pair(p, ch, msg) * pending(t, P, Q, msg, ch) * pool(t, P, Q, ch)}
  ch.Push(msg);
    {pair(p, ch, msg) * pool(t, P, Q, ch)}
    {pool(t, P, Q, ch)}
}

internal B pi2<A,B>(Pair<A,B> p) {
  return p.snd;
}

public Pattern And(Channel ch) {
  List<Channel> chs;
    {pattern(this, j, X) * chan(ch, j)}
  chs = this.channels;
    {this.channels ↦ chs * lst(chs, X) * ⊛_{c∈X} chan(c, j) * chan(ch, j)}
  chs.Push(ch);
    {this.channels ↦ chs * lst(chs, {ch} ∪ X) * ⊛_{c∈{ch}∪X} chan(c, j)}
```

```
      {pattern(this, j, {ch} ∪ X)}
    return this;
  }

  public void Do(Action cont) {
    Chord chord = new Chord(this, cont);
    join.chords.Push(chord);
  }
}
```

To verify the `Do` method we first need to define the last representation predicates.

4.4 The AsyncChannel, SyncChannel, Chord and Join classes

For the Message, MessagePool and Pattern classes it was possible to define each of their representation predicates independently of the rest of the joins library. However, this is not the case for the AsyncChannel, SyncChannel, Chord and Join classes, due to the possibility of the reentrant continuations. In this section we will thus define the remaining representation predicates up front, followed by proof outlines for the rest of the implementation.

Predicate definitions

We start by defining the internal representation predicates for joins instances in the third phase. At this point in time all channels and chords have already been initialized and the only way to interact with the joins instance is by sending messages. Except for the underlying message pools, all the state maintained by the joins instance is thus fixed. It is thus sufficient for most of these representation predicates to only assert read-only permission, allowing them to be freely duplicated.

The chord representation predicate, $\mathsf{chord}(\mathsf{f}, \mathsf{P}, \mathsf{Q}, \mathsf{c}, \mathsf{j})$, asserts that c refers to a chord registered with joins instance j. Chord continuations are specified in terms of the $\mathsf{join}_{\mathsf{call}}(-)$ predicate, which in turn is defined in terms of the $\mathsf{chord}(-)$ predicate. We thus initially define the $\mathsf{chord}(-)$ predicate in terms of an abstract $\mathsf{join}_{\mathsf{call}}(-)$ predicate f, closing the loop using guarded recursion in the definition of the $\mathsf{join}_{\mathsf{call}}(-)$ predicate below.

$$
\begin{array}{l}
\mathsf{chord}(\mathsf{f}, \mathsf{P}, \mathsf{Q}, \mathsf{c}, \mathsf{j}) \stackrel{\text{def}}{=} \\
\quad \exists \mathsf{x}, \mathsf{y} : \mathrm{Val}.\ \exists \mathsf{X}, \mathsf{Z} : \mathcal{P}_m(\mathrm{Val}). \\
\qquad \mathsf{c.pat} \mapsto \mathsf{x} * \mathsf{pattern}_{\mathsf{internal}}(\mathsf{x}, \mathsf{j}, \mathsf{X}) * \mathsf{c.cont} \mapsto \mathsf{y} * \circledast_{\mathsf{z} \in \mathsf{Z}} \mathsf{chan}(\mathsf{z}, \mathsf{j}) \\
\qquad * \ (\forall \mathsf{Y} \in \mathcal{P}_m(\mathrm{Val} \times \mathrm{Val}).\ \pi_{\mathsf{chan}}(\mathsf{Y}) = \mathsf{X} \ \Rightarrow \\
\qquad\qquad \triangleright\, \mathsf{y} \mapsto \{\mathsf{f}(\mathsf{j}) * \circledast_{\mathsf{z} \in \mathsf{Z}} \mathsf{chan}(\mathsf{z}, \mathsf{j}) * \circledast_{\mathsf{y} \in \mathsf{Y}} \mathsf{P}(\mathsf{y})\}\{\circledast_{\mathsf{y} \in \mathsf{Y}} \mathsf{Q}(\mathsf{y})\})
\end{array}
$$

The internal join representation predicate, $\mathsf{join}_{\mathsf{internal}}(\mathsf{t}, \mathsf{f}, \mathsf{P}, \mathsf{Q}, \mathsf{j})$, asserts that j refers to

a join instance.

$$\begin{aligned}\mathsf{join}_{\mathsf{internal}}(\mathsf{t},\mathsf{f},\mathsf{P},\mathsf{Q},\mathsf{j}) \stackrel{\text{def}}{=}\ & \exists \mathsf{x},\mathsf{y} : \text{Val}.\ \exists \mathsf{X},\mathsf{Y} : \mathcal{P}_m(\text{Val}).\\ & \quad \mathsf{j.channels} \mapsto \mathsf{x} * \mathsf{lst}_r(\mathsf{x},\mathsf{X})\\ & \quad * \ \mathsf{j.chords} \mapsto \mathsf{y} * \mathsf{lst}_r(\mathsf{y},\mathsf{Y})\\ & \quad * \circledast_{\mathsf{c}\in\mathsf{X}} \mathsf{chan}(\mathsf{c},\mathsf{j}) * \mathsf{pool}(\mathsf{t},\mathsf{P},\mathsf{Q},\mathsf{c})\\ & \quad * \circledast_{\mathsf{c}\in\mathsf{Y}} \mathsf{chord}(\mathsf{f},\mathsf{P},\mathsf{Q},\mathsf{c},\mathsf{j})\\ & \quad * \ (\forall \mathsf{a} : \text{Val}.\ \forall \mathsf{x} \in \mathsf{X}.\ \mathsf{x} : \mathsf{AsyncChannel} \Rightarrow \mathsf{Q}(\mathsf{a},\mathsf{x}) = \text{emp})\end{aligned}$$

We can now define the $\mathsf{join}_{\mathsf{call}}(-)$ predicate by guarded recursion:

$$\begin{aligned}\mathsf{join}_{\mathsf{call}}(\mathsf{t},\mathsf{P},\mathsf{Q}) \stackrel{\text{def}}{=}\ & \boldsymbol{fix}(\lambda \mathsf{f} : \text{Val} \to \text{Prop}.\ \lambda \mathsf{j} : \text{Val}.\ \exists \mathsf{I} : \text{Val}.\ \exists \mathsf{t}_1,\mathsf{t}_2,\mathsf{t}_3 \in \text{RType}.\\ & \quad \mathsf{t} \le \mathsf{t}_1 * \mathsf{t} \le \mathsf{t}_2 * \mathsf{t} \le \mathsf{t}_3 * \mathsf{j}_{reg_l} \mapsto \mathsf{t}_1 * \mathsf{j}_{reg_m} \mapsto \mathsf{t}_2 * \mathsf{j}_{reg_c} \mapsto \mathsf{t}_3\\ & \quad * \ \mathsf{j.lock} \mapsto \mathsf{I} * \text{isLock}(\mathsf{t}_1,\mathsf{I},\mathsf{join}_{\mathsf{internal}}(\mathsf{t}_2,\mathsf{f},\mathsf{P},\mathsf{Q},\mathsf{j}))))\end{aligned}$$

This is well-defined as the occurrence of f in $\mathsf{chord}(-)$ is guarded by $\triangleright$. Furthermore, the lock specification (See Section 4.6) requires that the resource invariant – in this case $\mathsf{join}_{\mathsf{internal}}(\mathsf{t}_2,\mathsf{f},\mathsf{P},\mathsf{Q},\mathsf{j})$ – is expressible using first-order protocols. By assumption, the channel pre- and post-conditions are uniformly expressible using first-order protocols. Hence, for a fixed $\mathsf{t} : \text{RType}$, $\mathsf{join}_{\mathsf{internal}}(\mathsf{t},\mathsf{f},\mathsf{P},\mathsf{Q},\mathsf{j})$ is expressible using state-independent protocols, as the parameterized message protocols used in the $\mathsf{pool}(-)$ predicate, the $\mathsf{chan}(-)$ predicate and the protocols used by the channel pre- and post-conditions, can all be pulled out under all the existential quantifiers. Formally, we prove that,

$$\begin{aligned}&\forall \mathsf{t} : \text{RType}.\ \forall \mathsf{f} : \text{Val} \to \text{Prop}.\ \forall \mathsf{j} : \text{Val}.\ \exists \mathsf{S},\mathsf{T} : \text{Prop}.\\ &\quad \mathsf{pure}_{\mathsf{protocol}}(\mathsf{S}) \wedge \mathsf{pure}_{\mathsf{state}}(\mathsf{T}) \wedge \boldsymbol{valid}(\mathsf{join}_{\mathsf{internal}}(\mathsf{t},\mathsf{f},\mathsf{P},\mathsf{Q},\mathsf{j}) \Leftrightarrow \mathsf{S} * \mathsf{T})\end{aligned}$$

This is provable without any assumptions about f, as f is only used in the pre-condition of a nested triple. Stability of $\mathsf{join}_{\mathsf{call}}(-)$ thus follows from the stability of $\mathsf{isLock}(-)$. To simplify the notation, in the following proofs we will use $\mathsf{join}_{\mathsf{call}}(\mathsf{t},\mathsf{P},\mathsf{Q},\mathsf{j})$ and $\mathsf{join}_{\mathsf{call}}(\mathsf{t},\mathsf{P},\mathsf{Q})(\mathsf{j})$ interchangeably.

In the running phase no new chords or channels can be registered with the join instance and read-only permissions thus suffices. In the initialization phase this is not the case and $\mathsf{join}_{\mathsf{init\text{-}ch}}(-)$ and $\mathsf{join}_{\mathsf{init\text{-}pat}}(-)$ thus assert full ownership of the channel and chord list:

$$\begin{aligned}\mathsf{join}_{\mathsf{init\text{-}ch}}(\mathsf{t},\mathsf{C}_A,\mathsf{C}_S,\mathsf{j}) \stackrel{\text{def}}{=}\ & \exists \mathsf{x},\mathsf{y},\mathsf{l} : \text{Val}.\ \exists \mathsf{t}_l,\mathsf{t}_c \in \text{RType}.\\ & \quad \mathsf{j.channels} \mapsto \mathsf{x} * \mathsf{lst}(\mathsf{x},\mathsf{C}_A \cup \mathsf{C}_S) * \circledast_{\mathsf{c}\in\mathsf{C}_A\cup\mathsf{C}_S} \mathsf{chan}(\mathsf{c},\mathsf{j})\\ & \quad * \ \mathsf{j.chords} \mapsto \mathsf{y} * \mathsf{lst}(\mathsf{y},\emptyset)\\ & \quad * \ \mathsf{j.lock} \mapsto \mathsf{l} * \mathsf{lock}(\mathsf{t}_l,\mathsf{l})\\ & \quad * \ \mathsf{j}_{reg_l} \mapsto \mathsf{t}_l * \mathsf{j}_{reg_m} \mapsto \_ * \mathsf{j}_{reg_c} \mapsto \mathsf{t}_c * \mathsf{t} \le \mathsf{t}_c * \mathsf{t} \le \mathsf{t}_l\\ & \quad * \ (\forall \mathsf{c} \in \mathsf{C}_A.\ \mathsf{c} : \mathsf{AsyncChannel})\\ & \quad * \ (\forall \mathsf{c} \in \mathsf{C}_S.\ \mathsf{c} : \mathsf{SyncChannel})\\ & \quad * \ \mathsf{j}_{\mathsf{chans}} \mapsto^{\mathsf{t}_c}_{=} \mathsf{C}_A \cup \mathsf{C}_S\end{aligned}$$

Here $\mathsf{j_{chans}} \mapsto^{\mathsf{t}}_{=} \mathsf{C}_A \cup \mathsf{C}_S$ asserts that the phantom field chans contains exactly $\mathsf{C}_A \cup \mathsf{C}_S$ *and* full ownership of the [Add] action:

$$\mathsf{j_{chans}} \mapsto^{\mathsf{t}}_{=} \mathsf{X} \stackrel{\text{def}}{=} \exists \mathsf{r} : \mathrm{RId}.\ \boxed{\mathsf{j_{chans}} \mapsto \mathsf{X}}^{\mathsf{r},\mathsf{t},\mathsf{j}} * [\textsc{Add}]^{\mathsf{r}}_1$$

where I is the previously defined protocol from Section 4.3. The $\mathsf{join}_{\mathsf{init\text{-}ch}}(-)$ further asserts that j.lock refers to an uninitialized lock (See Section 4.6), as the lock invariant depends on the channel pre- and post-conditions provided by the client in the view-shift to $\mathsf{join}_{\mathsf{init\text{-}pat}}(-)$. The lock is initialized in the view shift from $\mathsf{join}_{\mathsf{init\text{-}pat}}(-)$ to $\mathsf{join}_{\mathsf{call}}(-)$.

Chord continuations can send messages on channels on their own join instance. In the definition of $\mathsf{join}_{\mathsf{init\text{-}pat}}(-)$ we thus use the previously defined $\mathsf{join}_{\mathsf{call}}(-)$ predicate to give meaning to chords:

$$\begin{aligned}\mathsf{join}_{\mathsf{init\text{-}pat}}(\mathsf{t},\mathsf{P},\mathsf{Q},\mathsf{j}) \stackrel{\text{def}}{=}\ & \exists \mathsf{x},\mathsf{y} : \mathrm{Val}.\ \exists \mathsf{X},\mathsf{Y} : \mathcal{P}_m(\mathrm{Val}).\ \exists \mathsf{t}_l,\mathsf{t}_c \in \mathrm{RType}. \\ & \mathsf{j.channels} \mapsto \mathsf{x} * \mathsf{lst}(\mathsf{x},\mathsf{X}) * \circledast_{\mathsf{c}\in\mathsf{X}} \mathsf{chan}(\mathsf{c},\mathsf{j}) \\ & * \ \mathsf{j.chords} \mapsto \mathsf{y} * \mathsf{lst}(\mathsf{y},\mathsf{Y}) * \circledast_{\mathsf{c}\in\mathsf{Y}} \mathsf{chord}(\mathsf{join}_{\mathsf{call}}(\mathsf{t},\mathsf{P},\mathsf{Q}),\mathsf{P},\mathsf{Q},\mathsf{c},\mathsf{j}) \\ & * \ \mathsf{j.lock} \mapsto \mathsf{l} * \mathsf{lock}(\mathsf{t}_l,\mathsf{l}) \\ & * \ \mathsf{j}_{reg_l} \mapsto \mathsf{t}_l * \mathsf{j}_{reg_m} \mapsto \_ * \mathsf{j}_{reg_c} \Rightarrow\!\!\!\!\!\!- \ \mathsf{t}_c * \mathsf{t} \le \mathsf{t}_l * \mathsf{t} \le \mathsf{t}_c \\ & * \ (\forall \mathsf{a} : \mathrm{Val}.\ \forall \mathsf{c} \in \mathsf{X}.\ \mathsf{c} : \mathsf{AsyncChannel} \Rightarrow \mathsf{Q}(\mathsf{a},\mathsf{c}) = \mathrm{emp}) \\ & * \ \mathsf{j_{chans}} \mapsto^{\mathsf{t}_c}_{=} \mathsf{X}\end{aligned}$$

Proof outlines

```
internal class Chord {
  internal Pattern pat;
  internal Action cont;

  internal Chord(Pattern pat, Action cont) {
      {this.pat ↦ null * this.cont ↦ null}
    this.cont = cont;
    this.pat = pat;
      {this.pat ↦ pat * this.cont ↦ cont}
  }
}
```

Now we can verify the `Do` method from the `Pattern` class:

```
class Pattern {
  ...

  public void Do(Action cont) {
      {join_init-pat(t, P, Q, j) * pattern(this, j, X) * ⊛_{z∈Z} chan(z, j)
        * ∀Y : P_m(Val × Val). π_chan(Y) = X ⇒
            cont ↦ {join_call(t, P, Q, j) * ⊛_{z∈Z} chan(z, j) * ⊛_{y∈Y} P(y)}{⊛_{y∈Y} Q(y)}}
```

```
    Chord chord = new Chord(this, cont);
      {join_init-pat(t, P, Q, j) * pattern(this, j, X) * ⊛_{z∈Z} chan(z, j)
        * ∀Y : P_m(Val × Val). π_chan(Y) = X ⇒
            cont ↦ {join_call(t, P, Q, j) * ⊛_{z∈Z} chan(z, j) * ⊛_{y∈Y} P(y)}{⊛_{y∈Y} Q(y)}
        * chord.pat ↦ this * chord.cont ↦ cont}
      {join_init-pat(t, P, Q, join) * chord(join_call(t, P, Q), P, Q, chord, join)}
    join.chords.Push(chord);
      {join_init-pat(t, P, Q, join)}
  }
}

public class AsyncChannel : MessagePool {
  Join join;

  public AsyncChannel(Join j) {
    List<Message> chans;
      {join_init-ch(t, C_A, C_S, j) * this : AsyncChannel * this.join ↦ null}
      {join_init-ch(t, C_A, C_S, j) * this : AsyncChannel * this.join ↦ null * this ∉ C_A ∪ C_S}
    this.join = j;
      {join_init-ch(t, C_A, C_S, j) * this : AsyncChannel * this.join ↦ j * this ∉ C_A ∪ C_S}
    chans = join.channels;
      {j.channels ↦ chans * lst(chans, C_A ∪ C_S) * ⊛_{c∈C_A∪C_S} chan(c, j) * this ∉ C_A ∪ C_S
        * j.chords ↦ y * lst(y, ∅) * (∀c ∈ C_A. c : AsyncChannel) * (∀c ∈ C_S. c : SyncChannel)
        * j_{reg_l} ↦ _ * j_{reg_m} ↦ _ * j_{reg_c} ⇨ t_c * t ≤ t_c
        * j_chans ↦^{t_c}_= C_A ∪ C_S * this : AsyncChannel * this.join ↦ j}
    chans.Push(this);
      {j.channels ↦ chans * lst(chans, {this} ∪ C_A ∪ C_S) * ⊛_{c∈C_A∪C_S} chan(c, j) * this ∉ C_A ∪ C_S
        * j.chords ↦ y * lst(y, ∅) * (∀c ∈ C_A. c : AsyncChannel) * (∀c ∈ C_S. c : SyncChannel)
        * j_{reg_l} ↦ _ * j_{reg_m} ↦ _ * j_{reg_c} ⇨ t_c * t ≤ t_c
        * j_chans ↦^{t_c}_= C_A ∪ C_S * this : AsyncChannel * this.join ↦ j}
      {j.channels ↦ chans * lst(chans, {this} ∪ C_A ∪ C_S) * ⊛_{c∈C_A∪C_S} chan(c, j) * this ∉ C_A ∪ C_S
        * j.chords ↦ y * lst(y, ∅) * (∀c ∈ C_A. c : AsyncChannel) * (∀c ∈ C_S. c : SyncChannel)
        * j_{reg_l} ↦ _ * j_{reg_m} ↦ _ * j_{reg_c} ⇨ t_c * t ≤ t_c
        * j_chans ↦^{t_c}_= {this} ∪ C_A ∪ C_S * this : AsyncChannel * chan(this, j)}
      {join_init-ch(t, C_A ∪ {this}, C_S, j) * chan(this, j) * this ∉ C_A ∪ C_S}
  }

  public void call() {
    Join j; Lock l;
      {this : AsyncChannel * join_call(t, P, Q, j) * chan(this, j) * P(a, this)}
      {join_call(t, P, Q, j) * this.join ↦ j * P(a, this) * Q(a, this)}
    j = this.join;
      {join_call(t, P, Q, j) * this.join ↦ j * P(a, this) * Q(a, this)}
```

```
    l = j.lock;
    l.Acquire();
      {j_{reg_l} ↦ t_l * j_{reg_m} ↦ t_m * j.lock ↦ l * locked(t_l, l, join_internal(t_m, join_call(t, P, Q), P, Q, j))
        * join_internal(t_m, join_call(t, P, Q), P, Q, j) * this.join ↦ j * P(a, this) * Q(a, this)}
        {join_internal(t_m, join_call(t, P, Q), P, Q, j) * P(a, this)}
          {pool(t_m, P, Q, this) * P(a, this)}
    AddMessage();
            {∃x : Val. pool(t_m, P, Q, this) * msg(t_m, P, Q, a, x, this)}
        {join_internal(t_m, join_call(t, P, Q), P, Q, j)}
      {j_{reg_l} ↦ t_l * j_{reg_m} ↦ t_m * j.lock ↦ l * locked(t_l, l, join_internal(t_m, join_call(t, P, Q), P, Q, j))
        * join_internal(t_m, join_call(t, P, Q), P, Q, j) * Q(a, this)}
    l.Release();
      {join_call(t, P, Q, j) * Q(a, this)}
  }
}
```

Since the channel pre- and post-condition predicates P and Q are indexed by channel references, when creating a new channel, the client needs to know the newly created channel is distinct from existing channels ($\texttt{this} \notin C_A \cup C_S$). Intuitively, this should obviously hold in the channel constructor; formally, we use the fact that $\texttt{this.join} \mapsto \texttt{null}$ and that all channels registered with a join instance refers back to that join instance (See the definition of the chan(−) predicate).

```
public class SyncChannel : MessagePool {
  Join join;

  public SyncChannel(Join j) {
    List<Message> chans;
      {join_init-ch(t, C_A, C_S, j) * this : SyncChannel * this.join ↦ null}
      {join_init-ch(t, C_A, C_S, j) * this : SyncChannel * this.join ↦ null * this ∉ C_A ∪ C_S}
    this.join = j;
      {join_init-ch(t, C_A, C_S, j) * this : SyncChannel * this.join ↦ j * this ∉ C_A ∪ C_S}
    chans = join.channels;
      {j.channels ↦ chans * lst(chans, C_A ∪ C_S) * ⊛_{c∈C_A∪C_S} chan(c, j) * this ∉ C_A ∪ C_S
        * j.chords ↦ y * lst(y, ∅) * (∀c ∈ C_A. c : AsyncChannel) * (∀c ∈ C_S. c : SyncChannel)
        * j_{reg_l} ↦ _ * j_{reg_m} ↦ _ * j_{reg_c} ⇨ t_c * t ≤ t_c
        * j_chans ↦^{t_c}_{⊒} C_A ∪ C_S * this : SyncChannel * this.join ↦ j}
    chans.Push(this);
      {j.channels ↦ chans * lst(chans, {this} ∪ C_A ∪ C_S) * ⊛_{c∈C_A∪C_S} chan(c, j) * this ∉ C_A ∪ C_S
        * j.chords ↦ y * lst(y, ∅) * (∀c ∈ C_A. c : AsyncChannel) * (∀c ∈ C_S. c : SyncChannel)
        * j_{reg_l} ↦ _ * j_{reg_m} ↦ _ * j_{reg_c} ⇨ t_c * t ≤ t_c
        * j_chans ↦^{t_c}_{⊒} C_A ∪ C_S * this : SyncChannel * this.join ↦ j}
      {j.channels ↦ chans * lst(chans, {this} ∪ C_A ∪ C_S) * ⊛_{c∈C_A∪C_S} chan(c, j) * this ∉ C_A ∪ C_S
        * j.chords ↦ y * lst(y, ∅) * (∀c ∈ C_A. c : AsyncChannel) * (∀c ∈ C_S. c : SyncChannel)
```

$* \; \mathsf{j}_{reg_l} \mapsto \_ * \mathsf{j}_{reg_m} \mapsto \_ * \mathsf{j}_{reg_c} \Rightarrow \mathsf{t}_c * \mathsf{t} \leq \mathsf{t}_c$
$* \; \mathsf{j}_{chans} \mapsto^{\mathsf{t}_c}_{\sqsubseteq} \{\texttt{this}\} \cup \mathsf{C}_A \cup \mathsf{C}_S * \texttt{this} : \mathsf{SyncChannel} * \mathsf{chan}(\texttt{this}, \mathsf{j})\}$
$\{\mathsf{join}_{\mathsf{init\text{-}ch}}(\mathsf{t}, \mathsf{C}_A, \mathsf{C}_S \cup \{\texttt{this}\}, \mathsf{j}) * \mathsf{chan}(\texttt{this}, \mathsf{j}) * \texttt{this} \notin \mathsf{C}_A \cup \mathsf{C}_S\}$
}

public void call() {
Join j; Message msg; bool done; Lock l;
$\{\texttt{this} : \mathsf{SyncChannel} * \mathsf{join}_{\mathsf{call}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathsf{j}) * \mathsf{chan}(\texttt{this}, \mathsf{j}) * \mathsf{P}(\mathsf{a}, \texttt{this})\}$
j = **this**.join;
$\{\mathsf{join}_{\mathsf{call}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathsf{j}) * \mathsf{chan}(\texttt{this}, \mathsf{j}) * \mathsf{P}(\mathsf{a}, \texttt{this})\}$
l = j.lock;
l.Acquire();
$\{\mathsf{j}_{reg_l} \Rightarrow \mathsf{t}_l * \mathsf{j}_{reg_m} \Rightarrow \mathsf{t}_m * \mathsf{j}_{reg_c} \Rightarrow \mathsf{t}_c * \mathsf{j.lock} \Rightarrow \mathsf{l} * \mathsf{t} \leq \mathsf{t}_l * \mathsf{t} \leq \mathsf{t}_m * \mathsf{t} \leq \mathsf{t}_c$
$* \; \mathrm{locked}(\mathsf{t}_l, \mathsf{l}, \mathsf{join}_{\mathsf{internal}}(\mathsf{t}_m, \mathsf{join}_{\mathsf{call}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}), \mathsf{P}, \mathsf{Q}, \mathsf{j}))$
$* \; \mathsf{join}_{\mathsf{internal}}(\mathsf{t}_m, \mathsf{join}_{\mathsf{call}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}), \mathsf{P}, \mathsf{Q}, \mathsf{j}) * \mathsf{chan}(\texttt{this}, \mathsf{j}) * \mathsf{P}(\mathsf{a}, \texttt{this})\}$
$\{\mathsf{pool}(\mathsf{t}_m, \mathsf{P}, \mathsf{Q}, \texttt{this}) * \mathsf{P}(\mathsf{a}, \texttt{this})\}$
Message msg = AddMessage();
$\{\mathsf{pool}(\mathsf{t}_m, \mathsf{P}, \mathsf{Q}, \texttt{this}) * \mathsf{msg}(\mathsf{t}_m, \mathsf{P}, \mathsf{Q}, \mathsf{a}, \mathsf{msg}, \texttt{this})\}$
$\{\mathsf{j}_{reg_l} \Rightarrow \mathsf{t}_l * \mathsf{j}_{reg_m} \Rightarrow \mathsf{t}_m * \mathsf{j}_{reg_c} \Rightarrow \mathsf{t}_c * \mathsf{j.lock} \Rightarrow \mathsf{l} * \mathsf{t} \leq \mathsf{t}_l * \mathsf{t} \leq \mathsf{t}_m * \mathsf{t} \leq \mathsf{t}_c$
$* \; \mathrm{locked}(\mathsf{t}_l, \mathsf{l}, \mathsf{join}_{\mathsf{internal}}(\mathsf{t}_m, \mathsf{join}_{\mathsf{call}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}), \mathsf{P}, \mathsf{Q}, \mathsf{j}))$
$* \; \mathsf{join}_{\mathsf{internal}}(\mathsf{t}_m, \mathsf{join}_{\mathsf{call}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}), \mathsf{P}, \mathsf{Q}, \mathsf{j}) * \mathsf{msg}(\mathsf{t}_m, \mathsf{P}, \mathsf{Q}, \mathsf{a}, \mathsf{msg}, \texttt{this})\}$
done = false;
$\{\mathsf{j}_{reg_l} \Rightarrow \mathsf{t}_l * \mathsf{j}_{reg_m} \Rightarrow \mathsf{t}_m * \mathsf{j}_{reg_c} \Rightarrow \mathsf{t}_c * \mathsf{j.lock} \Rightarrow \mathsf{l} * \mathsf{t} \leq \mathsf{t}_l * \mathsf{t} \leq \mathsf{t}_m * \mathsf{t} \leq \mathsf{t}_c$
$* \; \mathrm{locked}(\mathsf{t}_l, \mathsf{l}, \mathsf{join}_{\mathsf{internal}}(\mathsf{t}_m, \mathsf{join}_{\mathsf{call}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}), \mathsf{P}, \mathsf{Q}, \mathsf{j}))$
$* \; \mathsf{join}_{\mathsf{internal}}(\mathsf{t}_m, \mathsf{join}_{\mathsf{call}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}), \mathsf{P}, \mathsf{Q}, \mathsf{j})$
$* \; (\mathsf{done} = \texttt{true} * \mathsf{Q}(\mathsf{a}, \texttt{this})) \vee (\mathsf{done} = \texttt{false} * \mathsf{msg}(\mathsf{t}_m, \mathsf{P}, \mathsf{Q}, \mathsf{a}, \mathsf{msg}, \texttt{this}))\}$
while (!done) {
$\{\mathsf{j}_{reg_l} \Rightarrow \mathsf{t}_l * \mathsf{j}_{reg_m} \Rightarrow \mathsf{t}_m * \mathsf{j}_{reg_c} \Rightarrow \mathsf{t}_c * \mathsf{j.lock} \Rightarrow \mathsf{l} * \mathsf{t} \leq \mathsf{t}_l * \mathsf{t} \leq \mathsf{t}_m * \mathsf{t} \leq \mathsf{t}_c$
$* \; \mathrm{locked}(\mathsf{t}_l, \mathsf{l}, \mathsf{join}_{\mathsf{internal}}(\mathsf{t}_m, \mathsf{join}_{\mathsf{call}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}), \mathsf{P}, \mathsf{Q}, \mathsf{j}))$
$* \; \mathsf{join}_{\mathsf{internal}}(\mathsf{t}_m, \mathsf{join}_{\mathsf{call}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}), \mathsf{P}, \mathsf{Q}, \mathsf{j}) * \mathsf{msg}(\mathsf{t}_m, \mathsf{P}, \mathsf{Q}, \mathsf{a}, \mathsf{msg}, \texttt{this}))\}$
j.checkChords();
$\{\mathsf{j}_{reg_l} \Rightarrow \mathsf{t}_l * \mathsf{j}_{reg_m} \Rightarrow \mathsf{t}_m * \mathsf{j}_{reg_c} \Rightarrow \mathsf{t}_c * \mathsf{j.lock} \Rightarrow \mathsf{l} * \mathsf{t} \leq \mathsf{t}_l * \mathsf{t} \leq \mathsf{t}_m * \mathsf{t} \leq \mathsf{t}_c$
$* \; \mathrm{locked}(\mathsf{t}_l, \mathsf{l}, \mathsf{join}_{\mathsf{internal}}(\mathsf{t}_m, \mathsf{join}_{\mathsf{call}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}), \mathsf{P}, \mathsf{Q}, \mathsf{j}))$
$* \; \mathsf{join}_{\mathsf{internal}}(\mathsf{t}_m, \mathsf{join}_{\mathsf{call}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}), \mathsf{P}, \mathsf{Q}, \mathsf{j}) * \mathsf{msg}(\mathsf{t}_m, \mathsf{P}, \mathsf{Q}, \mathsf{a}, \mathsf{msg}, \texttt{this}))\}$
if (msg.status == Status.Released) {
$\{\mathsf{j}_{reg_l} \Rightarrow \mathsf{t}_l * \mathsf{j}_{reg_m} \Rightarrow \mathsf{t}_m * \mathsf{j}_{reg_c} \Rightarrow \mathsf{t}_c * \mathsf{j.lock} \Rightarrow \mathsf{l} * \mathsf{t} \leq \mathsf{t}_l * \mathsf{t} \leq \mathsf{t}_m * \mathsf{t} \leq \mathsf{t}_c$
$* \; \mathrm{locked}(\mathsf{t}_l, \mathsf{l}, \mathsf{join}_{\mathsf{internal}}(\mathsf{t}_m, \mathsf{join}_{\mathsf{call}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}), \mathsf{P}, \mathsf{Q}, \mathsf{j}))$
$* \; \mathsf{join}_{\mathsf{internal}}(\mathsf{t}_m, \mathsf{join}_{\mathsf{call}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}), \mathsf{P}, \mathsf{Q}, \mathsf{j}) * \mathsf{Q}(\mathsf{a}, \texttt{this})\}$
done = true;
$\{\mathsf{j}_{reg_l} \Rightarrow \mathsf{t}_l * \mathsf{j}_{reg_m} \Rightarrow \mathsf{t}_m * \mathsf{j}_{reg_c} \Rightarrow \mathsf{t}_c * \mathsf{j.lock} \Rightarrow \mathsf{l} * \mathsf{t} \leq \mathsf{t}_l * \mathsf{t} \leq \mathsf{t}_m * \mathsf{t} \leq \mathsf{t}_c$
$* \; \mathrm{locked}(\mathsf{t}_l, \mathsf{l}, \mathsf{join}_{\mathsf{internal}}(\mathsf{t}_m, \mathsf{join}_{\mathsf{call}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}), \mathsf{P}, \mathsf{Q}, \mathsf{j}))$
$* \; \mathsf{join}_{\mathsf{internal}}(\mathsf{t}_m, \mathsf{join}_{\mathsf{call}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}), \mathsf{P}, \mathsf{Q}, \mathsf{j}) * \mathsf{Q}(\mathsf{a}, \texttt{this}) * \mathsf{done} = \texttt{true}\}$
} **else** {

```
        {j_reg_l ⇨ t_l * j_reg_m ⇨ t_m * j_reg_c ⇨ t_c * j.lock ⇨ l * t ≤ t_l * t ≤ t_m * t ≤ t_c
          * locked(t_l, l, join_internal(t_m, join_call(t, P, Q), P, Q, j))
          * join_internal(t_m, join_call(t, P, Q), P, Q, j) * msg(t_m, P, Q, a, msg, this))}
      l.Release();
        {join_call(t, P, Q, j) * j_reg_l ⇨ t_l * j_reg_m ⇨ t_m * j_reg_c ⇨ t_c * msg(t_m, P, Q, a, msg, this))}
      Thread.Sleep(1);
        {join_call(t, P, Q, j) * j_reg_l ⇨ t_l * j_reg_m ⇨ t_m * j_reg_c ⇨ t_c * msg(t_m, P, Q, a, msg, this))}
      l.Acquire();
        {j_reg_l ⇨ t_l * j_reg_m ⇨ t_m * j_reg_c ⇨ t_c * j.lock ⇨ l * t ≤ t_l * t_m * t ≤ t_c
          * locked(t_l, l, join_internal(t_m, join_call(t, P, Q), P, Q, j))
          * join_internal(t_m, join_call(t, P, Q), P, Q, j) * msg(t_m, P, Q, a, msg, this))}
    }
  }
    {j_reg_l ⇨ t_l * j_reg_m ⇨ t_m * j_reg_c ⇨ t_c * j.lock ⇨ l * t ≤ t_l * t ≤ t_m * t ≤ t_c
      * locked(t_l, l, join_internal(t_m, join_call(t, P, Q), P, Q, j))
      * join_internal(t_m, join_call(t, P, Q), P, Q, j) * Q(a, this)}
  l.Release();
    {join_call(t, P, Q, j) * Q(a, this)}
 }
}
```

In the proof of the SyncChannel.Call method, when the busy loop releases the lock, the client keeps some fractional permission of the read-only phantom fields containing the region types. This ensures that when the client re-acquires the lock and eliminates the existentially quantified region types, they match the region types from the loop invariant.

4.5 The Join class

```
public class Join {
  public LinkedList<Chord> chords;
  public LinkedList<Channel> channels;
  public Lock lock;

  public Join() {
      {this_reg_l ↦ _ * this_reg_c ↦ _ * this_reg_m ↦ _ * this_chans ↦ _
        * this.channels ↦ null * this.chords ↦ null * this.lock ↦ null}
    channels = new LinkedList<Channel>();
    chords = new LinkedList<Chord>();
    lock = new Lock();
      {this_reg_l ↦ _ * this_reg_c ↦ _ * this_reg_m ↦ _ * this_chans ↦ _
        * this.channels ↦ c * this.chords ↦ ch * this.lock ↦ l
        * lst(c, ∅) * lst(ch, ∅) * lock(t_l, l) * t ≤ t_l}
      {this_reg_l ↦ t_l * this_reg_c ↦ t_c * this_reg_m ↦ _ * this_chans ↦ ∅
        * this.channels ↦ c * this.chords ↦ ch * this.lock ↦ l
```

$* \mathit{lst}(\mathsf{c}, \emptyset) * \mathit{lst}(\mathsf{ch}, \emptyset) * \mathsf{lock}(\mathsf{t}_l, \mathsf{l}) * \mathsf{t} \leq \mathsf{t}_l * \mathsf{t} \leq \mathsf{t}_c\}$
$\{\mathsf{join}_{\mathsf{init\text{-}ch}}(\mathsf{t}, \emptyset, \emptyset, \mathtt{this})\}$
}

public Pattern When(SyncChannel ch) {
$\{\mathsf{join}_{\mathsf{init\text{-}pat}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathtt{this}) * \mathsf{chan}(\mathsf{ch}, \mathtt{this})\}$
Pattern pat = **new** Pattern(ch, **this**);
$\{\mathsf{join}_{\mathsf{init\text{-}pat}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathtt{this}) * \mathsf{chan}(\mathsf{ch}, \mathtt{this})$
$* \mathsf{pat.channels} \mapsto \mathsf{x} * \mathsf{pat.join} \mapsto \mathtt{this} * \mathsf{lst}(\mathsf{x}, \{\mathsf{ch}\})\}$
$\{\mathsf{join}_{\mathsf{init\text{-}pat}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathtt{this}) * \mathsf{pattern}(\mathsf{pat}, \mathtt{this}, \{\mathsf{ch}\})\}$
return pat;
$\{\mathsf{ret}.\ \mathsf{join}_{\mathsf{init\text{-}pat}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \mathtt{this}) * \mathsf{pattern}(\mathsf{ret}, \mathtt{this}, \{\mathsf{ch}\})\}$
}

internal **void** checkChord(Chord c) {
List<Message> consumed; Action act; Pattern pat;
$\{\mathtt{this}_{reg_l} \hookrightarrow \mathsf{t}_l * \mathtt{this}_{reg_m} \hookrightarrow \mathsf{t}_m * \mathtt{this}_{reg_c} \hookrightarrow \mathsf{t}_c * \mathtt{this.lock} \hookrightarrow \mathsf{l} * \mathsf{t} \leq \mathsf{t}_l * \mathsf{t} \leq \mathsf{t}_m * \mathsf{t} \leq \mathsf{t}_c$
$* \mathrm{locked}(\mathsf{t}_l, \mathsf{l}, \mathsf{join}_{\mathsf{internal}}(\mathsf{t}_m, \mathsf{join}_{\mathsf{call}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}), \mathsf{P}, \mathsf{Q}, \mathtt{this}))$
$* \mathsf{join}_{\mathsf{internal}}(\mathsf{t}_m, \mathsf{join}_{\mathsf{call}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}), \mathsf{P}, \mathsf{Q}, \mathtt{this})$
$* \mathsf{chord}(\mathsf{join}_{\mathsf{call}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}), \mathsf{P}, \mathsf{Q}, \mathsf{c}, \mathtt{this})\}$
pat = c.pat;
act = c.cont;
$\{\exists \mathsf{Z}.\ \mathtt{this}_{reg_l} \hookrightarrow \mathsf{t}_l * \mathtt{this}_{reg_m} \hookrightarrow \mathsf{t}_m * \mathtt{this}_{reg_c} \hookrightarrow \mathsf{t}_c$
$* \mathtt{this.lock} \hookrightarrow \mathsf{l} * \mathsf{t} \leq \mathsf{t}_l * \mathsf{t} \leq \mathsf{t}_m * \mathsf{t} \leq \mathsf{t}_c$
$* \mathrm{locked}(\mathsf{t}_l, \mathsf{l}, \mathsf{join}_{\mathsf{internal}}(\mathsf{t}_m, \mathsf{join}_{\mathsf{call}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}), \mathsf{P}, \mathsf{Q}, \mathtt{this}))$
$* \mathsf{join}_{\mathsf{internal}}(\mathsf{t}_m, \mathsf{join}_{\mathsf{call}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}), \mathsf{P}, \mathsf{Q}, \mathtt{this})$
$* (\circledast_{\mathsf{z} \in \mathsf{Z}} \mathsf{chan}(\mathsf{z}, \mathtt{this})) * \mathsf{pattern}_{\mathsf{internal}}(\mathsf{pat}, \mathtt{this}, \mathsf{X})$
$* \forall \mathsf{Y} \in \mathcal{P}_m(\mathrm{Val} \times \mathrm{Val}).\ \pi_{\mathsf{chan}}(\mathsf{Y}) = \mathsf{X} \Rightarrow$
$\triangleright\ \mathsf{act} \mapsto \begin{array}{l}(\text{-}).\ \{\mathsf{join}_{\mathsf{call}}(\mathsf{t}, \mathsf{P}, \mathsf{Q})(\mathtt{this}) * \circledast_{\mathsf{z} \in \mathsf{Z}} \mathsf{chan}(\mathsf{z}, \mathtt{this}) * \circledast_{\mathsf{y} \in \mathsf{Y}} \mathsf{P}(\mathsf{y})\} \\ \quad \{\circledast_{\mathsf{y} \in \mathsf{Y}} \mathsf{Q}(\mathsf{y})\}\end{array}\}$
consumed = pat.Matches();
$\{\exists \mathsf{Y}, \mathsf{Z}.\ \mathtt{this}_{reg_l} \hookrightarrow \mathsf{t}_l * \mathtt{this}_{reg_m} \hookrightarrow \mathsf{t}_m * \mathtt{this}_{reg_c} \hookrightarrow \mathsf{t}_c$
$* \mathtt{this.lock} \hookrightarrow \mathsf{l} * \mathsf{t} \leq \mathsf{t}_l * \mathsf{t} \leq \mathsf{t}_m * \mathsf{t} \leq \mathsf{t}_c$
$* \mathrm{locked}(\mathsf{t}_l, \mathsf{l}, \mathsf{join}_{\mathsf{internal}}(\mathsf{t}_m, \mathsf{join}_{\mathsf{call}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}), \mathsf{P}, \mathsf{Q}, \mathtt{this}))$
$* \mathsf{join}_{\mathsf{internal}}(\mathsf{t}_m, \mathsf{join}_{\mathsf{call}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}), \mathsf{P}, \mathsf{Q}, \mathtt{this})$
$* (\circledast_{\mathsf{z} \in \mathsf{Z}} \mathsf{chan}(\mathsf{z}, \mathtt{this})) * \mathsf{pattern}_{\mathsf{internal}}(\mathsf{pat}, \mathtt{this}, \mathsf{X})$
$* \forall \mathsf{Y} \in \mathcal{P}_m(\mathrm{Val} \times \mathrm{Val}).\ \pi_{\mathsf{chan}}(\mathsf{Y}) = \mathsf{X} \Rightarrow$
$\triangleright\ \mathsf{act} \mapsto \begin{array}{l}(\text{-}).\ \{\mathsf{join}_{\mathsf{call}}(\mathsf{t}, \mathsf{P}, \mathsf{Q})(\mathtt{this}) * \circledast_{\mathsf{z} \in \mathsf{Z}} \mathsf{chan}(\mathsf{z}, \mathtt{this}) * \circledast_{\mathsf{y} \in \mathsf{Y}} \mathsf{P}(\mathsf{y})\} \\ \quad \{\circledast_{\mathsf{y} \in \mathsf{Y}} \mathsf{Q}(\mathsf{y})\}\end{array}$
$* (\mathsf{consumed} = \mathtt{null} \vee (\mathsf{lst}(\mathsf{consumed}, \pi_2(\mathsf{Y})) * \circledast_{(\mathsf{c},\mathsf{m}) \in \mathsf{Y}} \mathsf{pending}(\mathsf{t}_m, \mathsf{P}, \mathsf{Q}, \mathsf{m}, \mathsf{c}) * \pi_1(\mathsf{Y}) = \mathsf{X}))\}$
if (consumed != **null**) {
$\{\exists \mathsf{Y}, \mathsf{Z}.\ \mathtt{this}_{reg_l} \hookrightarrow \mathsf{t}_l * \mathtt{this}_{reg_m} \hookrightarrow \mathsf{t}_m * \mathtt{this}_{reg_c} \hookrightarrow \mathsf{t}_c$

$$
\begin{array}{l}
\quad * \ \texttt{this.lock} \mapsto \mathsf{I} * \mathsf{t} \leq \mathsf{t}_l * \mathsf{t} \leq \mathsf{t}_m * \mathsf{t} \leq \mathsf{t}_c \\
\quad * \ \mathrm{locked}(\mathsf{t}_l, \mathsf{I}, \mathsf{join}_{\mathsf{internal}}(\mathsf{t}_m, \mathsf{join}_{\mathsf{call}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}), \mathsf{P}, \mathsf{Q}, \texttt{this})) \\
\quad * \ \mathsf{join}_{\mathsf{internal}}(\mathsf{t}_m, \mathsf{join}_{\mathsf{call}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}), \mathsf{P}, \mathsf{Q}, \texttt{this}) \\
\quad * \ (\circledast_{\mathsf{z} \in \mathsf{Z}} \mathsf{chan}(\mathsf{z}, \texttt{this})) * \mathsf{pattern}_{\mathsf{internal}}(\mathsf{pat}, \texttt{this}, \mathsf{X}) \\
\quad * \ \forall \mathsf{Y} \in \mathcal{P}_m(\mathrm{Val} \times \mathrm{Val}).\ \pi_{\mathsf{chan}}(\mathsf{Y}) = \mathsf{X} \Rightarrow \\
\qquad \triangleright\, \mathsf{act} \mapsto \begin{array}{l}(\text{-}).\ \{\mathsf{join}_{\mathsf{call}}(\mathsf{t}, \mathsf{P}, \mathsf{Q})(\texttt{this}) * \circledast_{\mathsf{z} \in \mathsf{Z}} \mathsf{chan}(\mathsf{z}, \texttt{this}) * \circledast_{\mathsf{y} \in \mathsf{Y}} \mathsf{P}(\mathsf{y})\} \\ \quad \{\circledast_{\mathsf{y} \in \mathsf{Y}} \mathsf{Q}(\mathsf{y})\}\end{array} \\
\quad * \ \mathsf{lst}(\mathsf{consumed}, \pi_2(\mathsf{Y})) * \circledast_{(\mathsf{c},\mathsf{m}) \in \mathsf{Y}} \mathsf{pending}(\mathsf{t}_m, \mathsf{P}, \mathsf{Q}, \mathsf{m}, \mathsf{c}) * \pi_1(\mathsf{Y}) = \mathsf{X}\} \\
\{\exists \mathsf{Z}, \mathsf{W}.\ \texttt{this}_{reg_l} \mapsto \mathsf{t}_l * \texttt{this}_{reg_m} \mapsto \mathsf{t}_m * \texttt{this}_{reg_c} \mapsto \mathsf{t}_c \\
\quad * \ \texttt{this.lock} \mapsto \mathsf{I} * \mathsf{t} \leq \mathsf{t}_l * \mathsf{t} \leq \mathsf{t}_m * \mathsf{t} \leq \mathsf{t}_c \\
\quad * \ \mathrm{locked}(\mathsf{t}_l, \mathsf{I}, \mathsf{join}_{\mathsf{internal}}(\mathsf{t}_m, \mathsf{join}_{\mathsf{call}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}), \mathsf{P}, \mathsf{Q}, \texttt{this})) \\
\quad * \ \mathsf{join}_{\mathsf{internal}}(\mathsf{t}_m, \mathsf{join}_{\mathsf{call}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}), \mathsf{P}, \mathsf{Q}, \texttt{this}) \\
\quad * \ (\circledast_{\mathsf{z} \in \mathsf{Z}} \mathsf{chan}(\mathsf{z}, \texttt{this})) * \mathsf{pattern}_{\mathsf{internal}}(\mathsf{pat}, \texttt{this}, \mathsf{X}) \\
\quad * \ \forall \mathsf{Y} \in \mathcal{P}_m(\mathrm{Val} \times \mathrm{Val}).\ \pi_{\mathsf{chan}}(\mathsf{Y}) = \mathsf{X} \Rightarrow \\
\qquad \triangleright \mathsf{act} \mapsto \begin{array}{l}(\text{-}).\ \{\mathsf{join}_{\mathsf{call}}(\mathsf{t}, \mathsf{P}, \mathsf{Q})(\texttt{this}) * \circledast_{\mathsf{z} \in \mathsf{Z}} \mathsf{chan}(\mathsf{z}, \texttt{this}) * \circledast_{\mathsf{y} \in \mathsf{Y}} \mathsf{P}(\mathsf{y})\} \\ \quad \{\circledast_{\mathsf{y} \in \mathsf{Y}} \mathsf{Q}(\mathsf{y})\}\end{array} \\
\quad * \ \mathsf{lst}(\mathsf{consumed}, \{\mathsf{m} \mid (\mathsf{c}, \mathsf{m}, \mathsf{a}) \in \mathsf{W}\}) * \{\mathsf{c} \mid (\mathsf{c}, \mathsf{m}, \mathsf{a}) \in \mathsf{W}\} = \mathsf{X} \\
\quad * \ (\circledast_{(\mathsf{c},\mathsf{m},\mathsf{a}) \in \mathsf{W}} \mathsf{P}(\mathsf{a}, \mathsf{c}) * \mathsf{removed}(\mathsf{t}_m, \mathsf{P}, \mathsf{Q}, \mathsf{a}, \mathsf{m}, \mathsf{c}))\} \\
\mathsf{lock.Release}(); \\
\{\exists \mathsf{Z}, \mathsf{W}.\ \texttt{this}_{reg_l} \mapsto \mathsf{t}_l * \texttt{this}_{reg_m} \mapsto \mathsf{t}_m * \texttt{this}_{reg_c} \mapsto \mathsf{t}_c \\
\quad * \ \mathsf{join}_{\mathsf{call}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \texttt{this}) * (\circledast_{\mathsf{z} \in \mathsf{Z}} \mathsf{chan}(\mathsf{z}, \texttt{this})) \\
\quad * \ \triangleright\, \mathsf{act} \mapsto \begin{array}{l}(\text{-}).\ \{\mathsf{join}_{\mathsf{call}}(\mathsf{t}, \mathsf{P}, \mathsf{Q})(\texttt{this}) * \circledast_{\mathsf{z} \in \mathsf{Z}} \mathsf{chan}(\mathsf{z}, \texttt{this}) * \circledast_{\mathsf{y} \in \{(\mathsf{a},\mathsf{c}) \mid (\mathsf{c},\mathsf{m},\mathsf{a}) \in \mathsf{W}\}} \mathsf{P}(\mathsf{y})\} \\ \quad \{\circledast_{\mathsf{y} \in \{(\mathsf{a},\mathsf{c}) \mid (\mathsf{c},\mathsf{m},\mathsf{a}) \in \mathsf{W}\}} \mathsf{Q}(\mathsf{y})\}\end{array} \\
\quad * \ \mathsf{lst}(\mathsf{consumed}, \{\mathsf{m} \mid (\mathsf{c}, \mathsf{m}, \mathsf{a}) \in \mathsf{W}\}) * \{\mathsf{c} \mid (\mathsf{c}, \mathsf{m}, \mathsf{a}) \in \mathsf{W}\} = \mathsf{X} \\
\quad * \ (\circledast_{(\mathsf{c},\mathsf{m},\mathsf{a}) \in \mathsf{W}} \mathsf{P}(\mathsf{a}, \mathsf{c}) * \mathsf{removed}(\mathsf{t}_m, \mathsf{P}, \mathsf{Q}, \mathsf{a}, \mathsf{m}, \mathsf{c}))\} \\
\mathsf{act}(); \\
\{\exists \mathsf{W}.\ \texttt{this}_{reg_l} \mapsto \mathsf{t}_l * \texttt{this}_{reg_m} \mapsto \mathsf{t}_m * \texttt{this}_{reg_c} \mapsto \mathsf{t}_c \\
\quad * \ \mathsf{join}_{\mathsf{call}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}, \texttt{this}) \\
\quad * \ \mathsf{lst}(\mathsf{consumed}, \{\mathsf{m} \mid (\mathsf{c}, \mathsf{m}, \mathsf{a}) \in \mathsf{W}\}) \\
\quad * \ (\circledast_{(\mathsf{c},\mathsf{m},\mathsf{a}) \in \mathsf{W}} \mathsf{Q}(\mathsf{a}, \mathsf{c}) * \mathsf{removed}(\mathsf{t}_m, \mathsf{P}, \mathsf{Q}, \mathsf{a}, \mathsf{m}, \mathsf{c}))\} \\
\mathsf{lock.Acquire}(); \\
\{\exists \mathsf{W}.\ \texttt{this}_{reg_l} \mapsto \mathsf{t}_l * \texttt{this}_{reg_m} \mapsto \mathsf{t}_m * \texttt{this}_{reg_c} \mapsto \mathsf{t}_c \\
\quad * \ \texttt{this.lock} \mapsto \mathsf{I} * \mathsf{t} \leq \mathsf{t}_l * \mathsf{t} \leq \mathsf{t}_m * \mathsf{t} \leq \mathsf{t}_c \\
\quad * \ \mathrm{locked}(\mathsf{t}_l, \mathsf{I}, \mathsf{join}_{\mathsf{internal}}(\mathsf{t}_m, \mathsf{join}_{\mathsf{call}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}), \mathsf{P}, \mathsf{Q}, \texttt{this})) \\
\quad * \ \mathsf{join}_{\mathsf{internal}}(\mathsf{t}_m, \mathsf{join}_{\mathsf{call}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}), \mathsf{P}, \mathsf{Q}, \texttt{this}) * \mathsf{lst}(\mathsf{consumed}, \{\mathsf{m} \mid (\mathsf{c}, \mathsf{m}, \mathsf{a}) \in \mathsf{W}\}) \\
\quad * \ (\circledast_{(\mathsf{c},\mathsf{m},\mathsf{a}) \in \mathsf{W}} \mathsf{Q}(\mathsf{a}, \mathsf{c}) * \mathsf{removed}(\mathsf{t}_m, \mathsf{P}, \mathsf{Q}, \mathsf{a}, \mathsf{m}, \mathsf{c}))\} \\
\mathsf{consumed.ForEach}(\mathsf{Release}); \\
\{\exists \mathsf{W}.\ \texttt{this}_{reg_l} \mapsto \mathsf{t}_l * \texttt{this}_{reg_m} \mapsto \mathsf{t}_m * \texttt{this}_{reg_c} \mapsto \mathsf{t}_c \\
\quad * \ \texttt{this.lock} \mapsto \mathsf{I} * \mathsf{t} \leq \mathsf{t}_l * \mathsf{t} \leq \mathsf{t}_m * \mathsf{t} \leq \mathsf{t}_c \\
\quad * \ \mathrm{locked}(\mathsf{t}_l, \mathsf{I}, \mathsf{join}_{\mathsf{internal}}(\mathsf{t}_m, \mathsf{join}_{\mathsf{call}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}), \mathsf{P}, \mathsf{Q}, \texttt{this})) \\
\quad * \ \mathsf{join}_{\mathsf{internal}}(\mathsf{t}_m, \mathsf{join}_{\mathsf{call}}(\mathsf{t}, \mathsf{P}, \mathsf{Q}), \mathsf{P}, \mathsf{Q}, \texttt{this}) * \mathsf{lst}(\mathsf{consumed}, \{\mathsf{m} \mid (\mathsf{c}, \mathsf{m}, \mathsf{a}) \in \mathsf{W}\})\} \\
\}
$$

```
    {this_reg_l ↦ t_l * this_reg_m ↦ t_m * this_reg_c ↦ t_c
      * this.lock ↦ l * t ≤ t_l * t ≤ t_m * t ≤ t_c
      * locked(t_l, l, join_internal(t_m, join_call(t, P, Q), P, Q, this))
      * join_internal(t_m, join_call(t, P, Q), P, Q, this)}
  }

  internal void checkChords() {
    LinkedList<Chord> chords;
      {this_reg_l ↦ t_l * this_reg_m ↦ t_m * this_reg_c ↦ t_c * this.lock ↦ l * t ≤ t_l * t ≤ t_m * t ≤ t_c
        * locked(t_l, l, join_internal(t_m, join_call(t, P, Q), P, Q, this))
        * join_internal(t_m, join_call(t, P, Q), P, Q, this)}
    chords = this.chords;
      {this_reg_l ↦ t_l * this_reg_m ↦ t_m * this_reg_c ↦ t_c * this.lock ↦ l * t ≤ t_l * t ≤ t_m * t ≤ t_c
        * locked(t_l, l, join_internal(t_m, join_call(t, P, Q), P, Q, this))
        * join_internal(t_m, join_call(t, P, Q), P, Q, this)
        * lst_r(chords, Y) * ⊛_{c∈Y} chord(join_call(t, P, Q), P, Q, c, this)}
    chords.ForEach(checkChord);
      {this_reg_l ↦ t_l * this_reg_m ↦ t_m * this_reg_c ↦ t_c * this.lock ↦ l * t ≤ t_l * t ≤ t_m * t ≤ t_c
        * locked(t_l, l, join_internal(t_m, join_call(t, P, Q), P, Q, this))
        * join_internal(t_m, join_call(t, P, Q), P, Q, this)}
  }

  public void Release(Message msg) {
      {Q(a, c) * removed(t_m, P, Q, a, msg, c)}
    msg.status = Status.Released;
      {emp}
  }
}
```

4.6 Auxiliary classes

The above proof outlines assume auxiliary classes Lock, List and Pair satisfying the following specifications.

The Lock class

The Lock class implements a spin-lock, using a compare-and-swap to atomically acquire the lock. The standard separation logic specification of a lock, associates a resource invariant R with each lock, which is transferred to the client upon acquiring the lock, and transferred back upon releasing the lock. Since the resource invariant R might itself assert ownership of resourced shared using CAP, we require that R is stable, that it is independent of the lock region type (picked by the client), and that it is expressible using state-independent protocols. To support the joins' view-shift from $\mathsf{join}_{\mathsf{init\text{-}pat}}(-)$ to $\mathsf{join}_{\mathsf{call}}(-)$ we further introduce a corresponding lock initialization phase, allowing the resource invariant R to be picked using a view-shift. We thus assume the following

specification for the Lock library.

$$
\begin{array}{l}
\exists \mathsf{lock} : \mathrm{RType} \times \mathrm{Val} \to \mathrm{Prop}.\ \exists \mathsf{islock}, \mathsf{locked} : \mathrm{RType} \times \mathrm{Prop} \times \mathrm{Val} \to \mathrm{Prop}. \\
\quad \forall \mathsf{t} \in \mathrm{RType}.\ \forall \mathsf{R} : \mathrm{Prop}.\ \mathsf{indep_t}(\mathsf{R}) \wedge \mathsf{sip}(\mathsf{R}) \wedge \mathsf{stable}(\mathsf{R}) \ \Rightarrow \\
\qquad \mathsf{new\ Lock()} : (\text{-}).\ \{\mathrm{emp}\}\ \{\mathsf{ret}.\ \exists \mathsf{s} : \mathrm{RType}.\ \mathsf{lock}(\mathsf{s}, \mathsf{ret}) * \mathsf{t} \leq \mathsf{s}\} \\
\qquad \mathsf{Lock.Acquire} : (\text{-}).\ \{\mathsf{isLock}(\mathsf{t}, \mathsf{R}, \mathtt{this})\}\ \{\mathsf{locked}(\mathsf{t}, \mathsf{R}, \mathtt{this}) * \mathsf{R}\} \\
\qquad \mathsf{Lock.Release} : (\text{-}).\ \{\mathsf{locked}(\mathsf{t}, \mathsf{R}, \mathtt{this}) * \mathsf{R}\}\ \{\mathsf{isLock}(\mathsf{t}, \mathsf{R}, \mathtt{this})\} \\
\qquad \forall \mathsf{x} : \mathrm{Val}.\ \mathsf{R} * \mathsf{lock}(\mathsf{t}, \mathsf{x}) \sqsubseteq \mathsf{isLock}(\mathsf{t}, \mathsf{R}, \mathsf{x}) \\
\qquad \forall \mathsf{x} : \mathrm{Val}.\ \mathsf{stable}(\mathsf{lock}(\mathsf{t}, \mathsf{x})) \wedge \mathsf{stable}(\mathsf{isLock}(\mathsf{t}, \mathsf{R}, \mathsf{x})) \wedge \mathsf{stable}(\mathsf{locked}(\mathsf{t}, \mathsf{R}, \mathsf{x})) \\
\qquad \boldsymbol{valid}\ (\forall \mathsf{x} : \mathrm{Val}.\ \mathsf{isLock}(\mathsf{t}, \mathsf{R}, \mathsf{x}) \Leftrightarrow \mathsf{isLock}(\mathsf{t}, \mathsf{R}, \mathsf{x}) * \mathsf{isLock}(\mathsf{t}, \mathsf{R}, \mathsf{x}))
\end{array}
$$

Here the $\mathsf{lock}(\mathsf{t}, \mathsf{x})$ predicate asserts that x refers to an uninitialized and unlocked lock. The lock is initialized by picking a resource invariant R using the view shift:

$$\mathsf{R} * \mathsf{lock}(\mathsf{t}, \mathsf{x}) \sqsubseteq \mathsf{isLock}(\mathsf{t}, \mathsf{R}, \mathsf{x})$$

The $\mathsf{isLock}(\mathsf{t}, \mathsf{R}, \mathsf{x})$ predicate asserts that x refers to an initialized lock with resource invariant R. The $\mathsf{isLock}(-)$ predicate is freely duplicable, allowing multiple clients to use the same lock. Lastly, the $\mathsf{locked}(\mathsf{t}, \mathsf{R}, \mathsf{x})$ predicate asserts that x refers to an initialized and locked lock with resource invariant R.

With the exception of the delayed choice of resource invariant R, this specification matches the Lock specification from the Example section in [2]. The proof is thus exactly the same, except that the lock region is created in the view-shift when the resource invariant is picked, instead of in the constructor. We will thus not repeat the proof. We refer the interested reader to the Example section in [2].

List class

In the case of the List class, which implements a linked list library, we assume a slightly weaker specification than usual. In particular, for our purposes, we do not care about the ordering of elements, only how many times each element appears in a given list. We thus represent the abstract state of a list as a multiset of elements. Since a lot of the lists maintained by join instances are read-only once the join instance transitions to the call phase, we index the list representation predicate with a fractional permission, to permit

read-only sharing.

$$\exists \mathsf{lst} : \mathrm{Val} \times \mathrm{Perm} \times \mathcal{P}_m(\mathrm{Val}) \to \mathrm{Prop}.$$
$$\forall \mathsf{I} : \mathcal{P}_m(\mathrm{Val}) \to \mathrm{Prop}.\ \forall \mathsf{J}_p, \mathsf{J}_q : \mathrm{Val} \to \mathrm{Prop}.\ \forall \mathsf{K} : \mathcal{P}_m(\mathrm{Val} \times \mathrm{Val}) \to \mathrm{Prop}.\ \forall \pi : \mathrm{Perm}.\ \forall \mathsf{X} : \mathcal{P}_m(\mathrm{Val}).$$

$$\mathsf{new\ List} : (\text{-}).\ \{\mathrm{emp}\}\ \{\mathsf{ret}.\ \mathsf{lst}(\mathsf{ret}, 1, \mathsf{X})\}$$

$$\mathsf{List.Push} : (\mathsf{y}).\ \{\mathsf{lst}(\mathtt{this}, 1, \mathsf{X})\}\ \{\mathsf{lst}(\mathtt{this}, 1, \{\mathsf{y}\} \cup \mathsf{X})\}$$

$$\mathsf{List.Pop} : (\text{-}).\ \{\mathsf{lst}(\mathtt{this}, 1, \mathsf{X}) * \mathsf{X} \neq \emptyset\}$$
$$\{\mathsf{ret}.\ \exists \mathsf{Y} : \mathcal{P}_m(\mathrm{Val}).\ \mathsf{lst}(\mathtt{this}, 1, \mathsf{Y}) * \mathsf{X} = \{\mathsf{ret}\} \cup \mathsf{Y}\}$$

$$\mathsf{List.Count} : (\text{-}).\ \{\mathsf{lst}(\mathtt{this}, \pi, \mathsf{X})\}\ \{\mathsf{ret}.\ \mathsf{lst}(\mathtt{this}, \pi, \mathsf{X}) * \mathsf{ret} = |\mathsf{X}|\}$$

$$\mathsf{List.ForEach} : (\mathsf{f}).\ \left\{ \begin{array}{l} \mathsf{lst}(\mathtt{this}, \pi, \mathsf{X}) * \mathsf{I}(\emptyset, \mathsf{X}) * \circledast_{\mathsf{x} \in \mathsf{X}} \mathsf{J}_p(\mathsf{x}) * \\ \qquad \forall \mathsf{Y}, \mathsf{Z} \in \mathcal{P}_m(\mathrm{Val}).\ \mathsf{f} \mapsto \begin{array}{l} (\mathsf{x}).\ \{\mathsf{I}(\mathsf{Y}, \{\mathsf{x}\} \cup \mathsf{Z}) * \mathsf{J}_p(\mathsf{x}) * \mathsf{x} \in \mathsf{X}\} \\ \qquad\qquad \{\mathsf{I}(\{\mathsf{x}\} \cup \mathsf{Y}, \mathsf{Z}) * \mathsf{J}_q(\mathsf{x})\} \end{array} \end{array} \right\}$$
$$\{\mathsf{lst}(\mathtt{this}, \pi, \mathsf{X}) * \mathsf{I}(\mathsf{X}, \emptyset) * \circledast_{\mathsf{x} \in \mathsf{X}} \mathsf{J}_q(\mathsf{x})\}$$

$$\mathsf{List.Map} : (\mathsf{f}).\ \left\{ \mathsf{lst}(\mathtt{this}, \pi, \mathsf{X}) * \mathsf{K}(\emptyset) * \forall \mathsf{Y} \in \mathcal{P}_m(\mathrm{Val} \times \mathrm{Val}).\ \mathsf{f} \mapsto \begin{array}{l} (\mathsf{x}).\ \{\mathsf{K}(\mathsf{Y}) * \mathsf{x} \in \mathsf{X}\} \\ \qquad \{\mathsf{ret}.\ \mathsf{K}(\{(\mathsf{x}, \mathsf{ret})\} \cup \mathsf{Y})\} \end{array} \right\}$$
$$\{\mathsf{ret}.\ \mathsf{lst}(\mathtt{this}, \pi, \mathsf{X}) * \exists \mathsf{Y} : \mathcal{P}_m(\mathrm{Val} \times \mathrm{Val}).\ \mathsf{lst}(\mathsf{ret}, 1, \pi_2(\mathsf{Y})) * \mathsf{K}(\mathsf{Y}) * \pi_1(\mathsf{Y}) = \mathsf{X}\}$$

$$\textbf{\textit{valid}}\ (\forall \mathsf{x} : \mathrm{Val}.\ \forall p, q : \mathrm{Perm}.$$
$$(\exists r : \mathrm{Perm}.\ p + q = r * \mathsf{lst}(\mathsf{x}, r, \mathsf{X})) \Leftrightarrow \mathsf{lst}(\mathsf{x}, p, \mathsf{X}) * \mathsf{lst}(\mathsf{x}, q, \mathsf{X}))$$

$$\mathsf{stable}(\mathsf{lst}(\mathsf{x}, \pi, \mathsf{X}))$$

The list representation predicate, $\mathsf{lst}(\mathsf{x}, \pi, \mathsf{X})$, thus asserts that x refers to a linked list containing elements X, with fractional permission π. Modifying a list requires full ownership, whereas querying (Count) and iterating (ForEach, Map) only requires read-only permission. In the proof outlines we use $\mathsf{lst}_r(\mathsf{x}, \mathsf{X})$ as shorthand for $\exists \pi : \mathrm{Perm}.\ \mathsf{lst}(\mathsf{x}, \pi, \mathsf{X})$ and $\mathsf{lst}(\mathsf{x}, \mathsf{X})$ as shorthand for $\mathsf{lst}(\mathsf{x}, 1, \mathsf{X})$. We thus have that

$$\mathsf{lst}_r(\mathsf{x}, \mathsf{X}) \Leftrightarrow \mathsf{lst}_r(\mathsf{x}, \mathsf{X}) * \mathsf{lst}_r(\mathsf{x}, \mathsf{X})$$

We omit the completely standard higher-order separation logic proof that a singly-linked list implementation satisfies the above specification.

Pair class

Lastly, the Pair class implements pairing.

$$\exists \mathsf{pair} : \mathrm{Val} \times \mathrm{Val} \times \mathrm{Val} \to \mathrm{Prop}.$$
$$\forall \mathsf{x}, \mathsf{y}, \mathsf{z} : \mathrm{Val}.$$

$$\mathsf{new\ Pair} : (\mathsf{x},\ \mathsf{y}).\ \{\mathrm{emp}\}\ \{\mathsf{ret}.\ \mathsf{pair}(\mathsf{ret}, \mathsf{x}, \mathsf{y})\}$$

$$\mathsf{Pair.Fst} : (\text{-}).\ \{\mathsf{pair}(\mathtt{this}, \mathsf{x}, \mathsf{y})\}\ \{\mathsf{ret}.\ \mathsf{pair}(\mathtt{this}, \mathsf{x}, \mathsf{y}) * \mathsf{ret} = \mathsf{x}\}$$

$$\mathsf{Pair.Snd} : (\text{-}).\ \{\mathsf{pair}(\mathtt{this}, \mathsf{x}, \mathsf{y})\}\ \{\mathsf{ret}.\ \mathsf{pair}(\mathtt{this}, \mathsf{x}, \mathsf{y}) * \mathsf{ret} = \mathsf{y}\}$$

$$\mathsf{stable}(\mathsf{pair}(\mathsf{z}, \mathsf{x}, \mathsf{y}))$$

The pair representation predicate, $\mathsf{pair}(\mathsf{x},\mathsf{y},\mathsf{z})$, asserts that x refers to a pair consisting of y and z.

We use $\mathsf{lst}_{pair}(\mathsf{x},\mathsf{X})$ as shorthand for,

$$\mathsf{lst}_{pair}(\mathsf{x},\mathsf{X}) \stackrel{\text{def}}{=} \exists \mathsf{Y} : \mathcal{P}_m(\text{Val} \times \text{Val} \times \text{Val}).\ \mathsf{lst}(\mathsf{x}, \pi_1(\mathsf{Y})) * \circledast_{(\mathsf{a},\mathsf{y},\mathsf{z})\in\mathsf{Y}} \mathsf{pair}(\mathsf{a},\mathsf{y},\mathsf{z}) * \{(\mathsf{y},\mathsf{z}) \mid (\mathsf{a},\mathsf{y},\mathsf{z}) \in \mathsf{Y}\} = \mathsf{X}$$

where $\mathsf{X} \in \mathcal{P}_m(\text{Val} \times \text{Val})$, to specify lists of pairs.

References

[1] K. Svendsen, L. Birkedal, and M. Parkinson. Verifying Generics and Delegates. In *Proceedings of ECOOP*, pages 175–199, 2010.

[2] K. Svendsen, L. Birkedal, and M. Parkinson. Higher-order Concurrent Abstract Predicates. Technical report, IT University of Copenhagen, 2012. Available at `www.itu.dk/people/kasv/hocap-tr.pdf`.

[3] K. Svendsen, L. Birkedal, and M. Parkinson. Joins: A Case Study in Modular Specification of a Concurrent Reentrant Higher-order Library. In *Proceedings of ECOOP*, 2013.

[4] K. Svendsen, L. Birkedal, and M. Parkinson. Modular Reasoning about Separation for Concurrent Data Structures. In *Proceedings of ESOP*, 2013.

Chapter 4

State and non-termination in CIC

Partiality, State and Dependent Types

Kasper Svendsen[1], Lars Birkedal[1], and Aleksandar Nanevski[2]

[1] IT University of Copenhagen {kasv,birkedal}@itu.dk
[2] IMDEA Software aleks.nanevski@imdea.org

Abstract. Partial type theories allow reasoning about recursively-defined computations using fixed-point induction. However, fixed-point induction is only sound for admissible types and not all types are admissible in sufficiently expressive dependent type theories.

Previous solutions have either introduced explicit admissibility conditions on the use of fixed points, or limited the underlying type theory. In this paper we propose a third approach, which supports Hoare-style partial correctness reasoning, without admissibility conditions, but at a tradeoff that one cannot reason equationally about effectful computations. The resulting system is still quite expressive and useful in practice, which we confirm by an implementation as an extension of Coq.

1 Introduction

Dependent type theories such as the Calculus of Inductive Constructions [2] provide powerful languages for integrated programming, specification, and verification. However, to maintain soundness, they typically require all computations to be pure and terminating, severely limiting their use as general purpose programming languages.

Constable and Smith [9] proposed adding partiality by introducing a type $\bigcirc\tau$ of potentially non-terminating computations of type τ, along with the following fixed point principle for typing recursively defined computations:

$$\text{if } M : \bigcirc\tau \to \bigcirc\tau \quad \text{then} \quad \textbf{\textit{fix}}(M) : \bigcirc\tau$$

Unfortunately, in sufficiently expressive dependent type theories, there exists types τ for which the above fixed point principle is unsound [10]. For instance, in type theories with subset-types, the fixed point principle allows reasoning by a form of fixed point induction, which is only sound for admissible predicates (a predicate is admissible if it holds for the limit whenever it holds for all finite approximations). Previous type theories based on the idea of partial types which admit fixed points have approached the admissibility issue in roughly two different ways:

1. The notion of admissibility is axiomatized in the type theory and explicit admissibility conditions are required in order to use ***fix***. This approach has, e.g., been investigated by Crary in the context of Nuprl [10]. The resulting type theory is expressive, but admissibility conditions lead to significant proof obligations, in particular, when using Σ types.

2. The underlying dependent type theory is restricted in such a way that one can only form types that are trivially admissible. This approach has, e.g., been explored in recent work on Hoare Type Theory (HTT) [21]. The restrictions exclude usage of subset types and Σ types, which are often used for expressing properties of computations and for modularity. Another problem with this approach is that since it limits the underlying dependent type theory one cannot easily implement it as a simple extension of existing implementations.

In this paper we explore a third approach, which ensures that all types are admissible, not by limiting the underlying standard dependent type theory, but by limiting only the partial types. The limitation on partial types consists of equating all effectful computations at a given type: if M and N are both of type $\bigcirc\tau$, then they are propositionally equal. Thus, with this approach, the only way to reason about effectful computations is through their type, rather than via equality or predicates. With sufficiently expressive types, the type of an effectful computation can serve as a partial correctness specification of the computation. Our hypothesis is that this approach allows us to restrict attention to a subset of admissible types, which is closed under the standard dependent type formers and which suffices for reasoning about partial correctness.

To demonstrate that this approach scales to expressive type theories and to effects beyond partiality, we extend the Calculus of Inductive Constructions (CIC) [2] with stateful and potentially non-terminating computations. Since reasoning about these effectful computations is limited to their type, our partial types are further refined into a Hoare-style partial correctness specifications, and have the form $\boldsymbol{ST}\,\tau\,(P, Q)$, standing for computations with pre-condition P, post-condition Q, that diverge or terminate with a value of type τ.

The resulting type theory is an impredicative variant of Hoare Type Theory [17], which differs from previous work on Hoare Type Theory in the scope of features considered and the semantic approach. In particular, this paper is the first to clarify semantically the issue of admissibility in Hoare Type Theory.

Impredicative Hoare Type Theory (iHTT) features the universes of propositions (***prop***), small types (***set***), and large types (***type***), with ***prop*** included in ***set***, ***set*** included in ***type***, and axioms ***prop***:***type*** and ***set***:***type***. The ***prop*** and ***set*** universes are impredicative, while ***type*** is predicative. There are two main challenges in building a model to justify the soundness of iHTT: (1) achieving that Hoare types are small ($\boldsymbol{ST}\ \tau\ s : \boldsymbol{set}$), which enables *higher-order store*; that is, storing side-effectful computations into the heap, and (2) supporting arbitrary Σ types, and more generally, inductive types. In this respect iHTT differs from the previous work on Hoare Type Theory, which either lacks higher-order store [19], lacks strong Σ types [21], or whose soundness has been justified using specific syntactic methods that do not scale to fully general inductive definitions [17, 18].

The model is based on a standard realizability model of partial equivalence relations (PERs) and assemblies over a combinatory algebra A. These give rise to a model of the Calculus of Constructions [14], with ***set*** modelled using PERs. Restricting PERs to complete PERs (i.e., PERs closed under limits of ω-chains)

over a suitable universal domain, allows one to model recursion in a simply-typed setting [4], or in a dependently-typed setting, but without strong Σ types [21].

Our contribution is in identifying a set of complete *monotone* PERs that are closed under Σ types and Hoare types. Complete PERs do not model Σ types because, given a chain of dependent pairs, in general, due to dependency, the second components of the chain are elements of *distinct* complete PERs. To apply completeness, we need a fixed single complete PER. Monotonicity will equate the first components of the chain and give us the needed single complete PER for the second components. Monotonicity further forces a trivial equality on Hoare types, equating all effectful computations satisfying a given specification. However, it does not influence the equality on the total, purely functional, fragment of iHTT, ensuring that we still model CIC. This is sufficient for very expressive Hoare-style reasoning, and avoids admissibility conditions on the use of ***fix***.

As iHTT is an extension of CIC, we have implemented iHTT as an axiomatic extension of Coq [1]. The implementation is carried out in Ssreflect [12] (a recent extension of Coq), based on the previous implementation of predicative Hoare Type Theory [19]. The implementation is available at:

`http://www.itu.dk/people/kasv/ihtt.tgz`.

2 Hoare types by example

To illustrate Hoare types, we sketch a specification of a library for arrays in iHTT. We assume that array indexes range over a finite type $\iota{:}\boldsymbol{set}_{fin}$, that the elements of ι can be enumerated as $\iota_0, \iota_1, \ldots, \iota_n$, and that equality between these elements can be decided by a function $=\!=\colon \iota \to \iota \to \boldsymbol{bool}$.

Each array is implemented as a contiguous block of locations, each location storing a value from the range type $\tau{:}\boldsymbol{set}$. The space occupied by the array is uniquely determined by ι, τ, and the pointer to the first element, justifying that the ***array*** type be defined as this first pointer.

$$\boldsymbol{array} : \boldsymbol{set}_{fin} \to \boldsymbol{set} \to \boldsymbol{set} = \lambda\iota.\, \lambda\tau.\, \boldsymbol{ptr}.$$

Here, ***ptr*** is the type of pointers, which we assume isomorphic to ***nat***. Each array is essentially a stateful implementation of some finite function $f{:}\iota{\to}\tau$. To capture this, we define a predicate indexed by f, that describes the layout of an array in the heap.

$$\begin{aligned}&\mathsf{shape} : (\boldsymbol{array}\ \iota\ \tau) \to (\iota \to \tau) \to \boldsymbol{heap} \to \boldsymbol{prop} = \\ &\quad \lambda a.\, \lambda f.\, \lambda h.\, h = a \mapsto f\,\iota_o \bullet a{+}1 \mapsto f\,\iota_1 \bullet \cdots \bullet a{+}n \mapsto f\,\iota_n.\end{aligned}$$

In other words, h stores an array a, representing a finite function f, if $\mathsf{shape}\ a\ f\ h$ holds, that is, if h consists of $n{+}1$ consecutive locations a, $a{+}1$, …, $a{+}n$, storing $f\,\iota_0, f\,\iota_1, \ldots, f\,\iota_n$, respectively. The property is stated in terms of singleton heaps $a{+}k \mapsto f\,\iota_k$, connected by the operator $\bullet$ for *disjoint* heap union. Later in the text, we will also require a constant empty denoting the empty heap.

The type of arrays comes equipped with several methods for accessing and manipulating the array elements. For example, the method for reading the value at index $k{:}\iota$ can be given the following type.

$$\begin{array}{l}\mathsf{read} : \Pi a{:}\boldsymbol{array}\ \iota\ \tau.\ \Pi k{:}\iota.\\ \quad \boldsymbol{ST}\ \tau\ (\lambda h.\ \exists f.\ \mathsf{shape}\ a\ f\ h,\\ \quad\quad \lambda r.\ \lambda h.\ \lambda m.\ \forall f.\ \mathsf{shape}\ a\ f\ h \to r = f\,k \wedge m = h)\end{array}$$

Informally, $\mathsf{read}\ a\ k$ is specified as a stateful computation whose precondition permits the execution only in a heap h which stores a valid array at adress a ($\exists f.\ \mathsf{shape}\ a\ f\ h$). The postcondition, on the other hand, specifies the result of executing $\mathsf{read}\ a\ k$ as a relation between output result $r{:}\tau$, input heap h and output heap m. In particular, the result r is indeed the array value at index k ($r = f\,k$), and the input heap is unchanged ($m = h$).

Unlike in ordinary Hoare logic, but similar to VDM [6], our postcondition is parametrized wrt. both input and the output heaps in order to directly express the relationship between the two. In particular, when this relationship depends on some specification-level value, such as f above, the dependency can be expressed by an ordinary propositional quantification.

Hoare types employ *small footprint* specifications, as in separation logic [20], whereby the specifications only describe the parts of the heap that the computation traverses. The untraversed parts are by default invariant. To illustrate, consider the type for the method new that generates a fresh array, indexed by ι, and populated by the value $x{:}\tau$.

$$\mathsf{new} : \Pi x{:}\tau.\ \boldsymbol{ST}\ (\boldsymbol{array}\ \iota\ \tau)\ (\lambda h.\ h = \mathsf{empty}, \lambda a.\ \lambda h.\ \lambda m.\ \mathsf{shape}\ a\ (\lambda z.\ x)\ m)$$

The type states *not* that $\mathsf{new}\ x$ can only run in an empty heap, but that $\mathsf{new}\ x$ *changes* the empty subheap of the current heap into a heap m containing an array rooted at a and storing all x's. In other words, new is *adding* fresh pointers, and the resulting array a itself is fresh. On the other hand, unlike in separation logic, we allow that the specifications can directly use and quantify over variables of type ***heap***. For completeness, we next simply list without discussion the types of the other methods for arrays.

$$\begin{array}{l}\mathsf{new\_from\_fun} : \Pi f{:}\iota{\to}\tau.\ \boldsymbol{ST}\ (\boldsymbol{array}\ \iota\ \tau)\ (\lambda h.\ h = \mathsf{empty}, \lambda a\,h\,m.\ \mathsf{shape}\ a\ f\ m)\\ \mathsf{free} : \Pi a{:}\boldsymbol{array}\ \iota\ \tau.\ \boldsymbol{ST}\ \boldsymbol{unit}\ (\lambda h.\ \exists f.\ \mathsf{shape}\ a\ f\ h, \lambda r\,h\,m.\ m = \mathsf{empty})\\ \mathsf{write} : \Pi a{:}\boldsymbol{array}\ \iota\ \tau.\ \Pi k{:}\iota.\ \Pi x{:}\tau.\\ \quad \boldsymbol{ST}\ \boldsymbol{unit}\ (\lambda h.\ \exists f.\ \mathsf{shape}\ a\ f\ h, \lambda r\,h\,m.\ \forall f.\ \mathsf{shape}\ a\ f\ h \to\\ \quad\quad \mathsf{shape}\ a\ (\lambda z.\ \mathsf{if}\ z == k\ \mathsf{then}\ x\ \mathsf{else}\ f(z))\ m)\end{array}$$

At this point, we emphasize that various type theoretic abstractions are quite essential for practical work with Hoare types. The usefulness of Π types and the propositional quantifiers is apparent from the specification of the array methods. But the ability to structure specification is important too. For example, we can pair pre- and postconditions into a type $\boldsymbol{spec}\ \tau = (\boldsymbol{heap} \to \boldsymbol{prop}) \times (\tau \to \boldsymbol{heap} \to \boldsymbol{heap} \to \boldsymbol{prop})$, which is then used to specify the fixpoint combinator.

$$\begin{array}{l}\mathsf{fix} : \Pi\alpha{:}\boldsymbol{set}.\ \Pi\beta{:}\alpha{\to}\boldsymbol{set}.\ \Pi s{:}\Pi x.\ \boldsymbol{spec}\ (\beta\ x).\\ \quad (\Pi x.\ \boldsymbol{ST}\ (\beta\ x)\ (s\ x) \to \Pi x.\ \boldsymbol{ST}\ (\beta\ x)\ (s\ x)) \to \Pi x.\ \boldsymbol{ST}\ (\beta\ x)\ (s\ x).\end{array}$$

Structuring proofs and specifications with programs is also necessary, and is achieved using dependent records (i.e., Σ types), which we illustrate next.

The first example of a dependent record is the $\boldsymbol{set}_{fin}$ type. This is an algebraic structure containing the carrier type σ, the operation $==$ for deciding equality on σ, and a list enumerating σ's elements. Additionally, $\boldsymbol{set}_{fin}$ needs proofs that $==$ indeed decides equality, and that the enumeration list contains each element exactly once. Using the record notation $[x_1{:}\tau_1, \ldots x_n{:}\tau_n]$ instead of the more cumbersome $\Sigma x_1{:}\tau_1 \ldots \Sigma x_n{:}\tau_n.\, 1$, the $\boldsymbol{set}_{fin}$ type is defined as follows.

$$\begin{aligned}\boldsymbol{set}_{fin} = [\,&\sigma : \boldsymbol{set}, \mathsf{enum} : \boldsymbol{list}\ \sigma, ==: \sigma \to \sigma \to \mathsf{bool},\\ &\mathsf{eqp} : \forall x\, y{:}\sigma.\, x == y = \mathsf{true} \iff x = y,\\ &\mathsf{enump} : \forall x{:}\sigma.\, \mathsf{count}\ x\ \mathsf{enum} = 1]\end{aligned}$$

The above dependent record refines a type. In practice, we will also use records that refine *values*. For example, in programming, arrays are often indexed by the type of bounded integers $I_n = [x : \boldsymbol{nat}, \mathsf{boundp} : x \leq n]$. I_n can be extended with appropriate fields, to satisfy the specification for $\boldsymbol{set}_{fin}$, but the important point here is that the elements of I_n are dependent records containing a number x and a proof that $x \leq n$. Of course, during actual execution, this proof can be ignored (proofs are computationally irrelevant), but it is clearly important statically, during verification.

Finally, the library of arrays itself can be ascribed a signature which will serve as an interface to the client programs. This signature too is a dependent record, providing types for all the array methods. Just as in the case of $\boldsymbol{set}_{fin}$ and I_n, the signature may also include properties, similar to object invariants [13, 15]. For example, we have found it useful in practice to hide from the clients the definitions of the array type and the array shape predicate, but expose that two arrays in stable states; that is, between two method calls, stored in compatible heaps must be equal (i.e., that the shape predicate is "functional"):

$$\begin{aligned}\mathsf{functional} : \Pi\mathsf{shape}.\, \forall a_1\ a_2\ f_1\ f_2\ h_1\ h_2.\, \mathsf{shape}\ a_1\ f_1\ h_1 \to \mathsf{shape}\ a_2\ f_2\ h_2 \to\\ (\exists j_1\ j_2.\, h_1 \bullet j_1 = h_2 \bullet j_2) \to a_1 = a_2 \wedge f_1 = f_2 \wedge h_1 = h_2.\end{aligned}$$

Then the signature for arrays indexed by ι, containing values of type τ, is provided by the following dependent record parametrized by ι and τ.

$$\begin{aligned}\mathit{ArraySig} = \Pi\iota{:}\boldsymbol{set}_{fin}.\, \Pi\tau{:}\boldsymbol{set}.\\ [\,&\boldsymbol{array} : \boldsymbol{set},\ \mathsf{shape} : \boldsymbol{array} \to (\iota \to \tau) \to \boldsymbol{heap} \to \boldsymbol{prop},\\ &\mathsf{funcp} : \mathsf{functional\ shape},\ \mathsf{read} : \Pi a{:}\boldsymbol{array}.\, \Pi k{:}\iota.\ \ldots].\end{aligned}$$

Therefore, Σ types are central for building verified libraries of programs, specifications and proofs.

3 Semantics

In this section we present a model for impredicative Hoare Type Theory. We will introduce the relevant parts of iHTT as the model is defined. The full type

theory is defined in Appendix A. In presenting the model, we first focus on the ***set***-universe, and then scale up to cover all of iHTT.

Since the purely-function fragment of IHTT is terminating, we take our universe of realizers to be a universal *pre-domain* with a suitable sub-domain for modelling stateful and potentially non-terminating computations.

Definition 1. *Let $\mathbb{V}$ denote a pre-domain satisfying the following recursive pre-domain equation:*

$$\mathbb{V} \cong 1 + \mathbb{N} + (\mathbb{V} \times \mathbb{V}) + (\mathbb{V} \to_c \mathbb{V}_\perp) + T(\mathbb{V}) + H(\mathbb{V})$$

where $\to_c$ is the space of continuous functions and

$$T(\mathbb{V}) \stackrel{def}{=} H(\mathbb{V}) \to_c ((\mathbb{V} \times H(\mathbb{V})) + 1)_\perp$$
$$H(\mathbb{V}) \stackrel{def}{=} \{h : \mathbf{ptr} \to_c \mathbb{V}_\perp \mid supp(h) \text{ finite} \wedge h(\mathbf{null}) = \perp\}$$
$$supp(h : \mathbf{ptr} \to \mathbb{V}_\perp) \stackrel{def}{=} \{l \in \mathbf{ptr} \mid h(l) \neq \perp\}$$

The first four summands of $\mathbb{V}$ model the underlying dependent type theory, and $T(\mathbb{V})$ and $H(\mathbb{V})$ model computations and heaps, respectively. The ordering on $T(\mathbb{V})$ is the standard pointwise order and the ordering on $H(\mathbb{V})$ is as follows:

$$h_1 \leq h_2 \quad \textit{iff} \quad supp(h_1) = supp(h_2) \wedge \forall n \in supp(h_1).\ h_1(n) \leq h_2(n)$$

Let $in_1, in_{\mathbb{N}}, in_\times, in_\to, in_T$, and in_H denote injections into $\mathbb{V}$ corresponding to each of the above summands.

$\mathbb{V}$ defines a partial combinatory algebra (PCA) with the following partial application operator:

Definition 2. *Let $\cdot : \mathbb{V} \times \mathbb{V} \rightharpoonup \mathbb{V}$ denote the function,*

$$a \cdot b = \begin{cases} f(b) & \text{if } a = in_\to(f) \wedge f(b) \neq \perp \\ \mathbf{undef} & \text{otherwise} \end{cases}$$

We recall some notation and definitions. If $R \subseteq A \times A$ is a PER then its domain, denoted $|R|$, is $\{x \in A \mid (x, x) \in R\}$. If $R, S \subseteq A \times A$ are PERs, then $R \to S$ is the PER $\{(\alpha, \beta) \in A \times A \mid \forall x, y \in A.\ (x, y) \in R \Rightarrow (\alpha \cdot x, \beta \cdot y) \in S\}$. If $R \subseteq A \times A$ is a PER and $f : A \to B$ then $f(R)$ denotes the PER $\{(f(x), f(y)) \mid (x, y) \in R\} \subseteq B \times B$. For a subset $X \subseteq A$, we use $\Delta(X)$ to denote the PER $\{(x, y) \mid x \in X \wedge y \in X\}$. Lastly, if $R \subseteq A \times A$ is a PER, we use $[R]$ to denote the set of R equivalence classes.

Definition 3 (Per(A)). *The category of PERs, Per(A), over a partial combinatory algebra $(A, \cdot)$, has PERs over A as objects. Morphisms from R to S are set-theoretic functions $f : [R] \to [S]$, such that there exists a realizer $\alpha \in A$ such that,*

$$\forall e \in |R|.\ [\alpha \cdot e]_S = f([e]_R)$$

$\mathrm{Per}(\mathbb{V})$ is cartesian closed and thus models simple type theory. To model recursion, note that a realized set-theoretic function is completely determined by its realizers (i.e., $\mathrm{Per}(\mathbb{V})(R,S) \cong [R \to S]$) and that we have the standard least fixed-point operator on the sub-domain of computations of $\mathbb{V}$. This lifts to a least fixed-point operator on those PERs that are admissible on the sub-domain of computations:

Definition 4.

1. *A PER $R \subseteq A \times A$ on a pre-domain A is* complete *if, for all chains $(c_i)_{i\in\mathbb{N}}$ and $(d_i)_{i\in\mathbb{N}}$ such that $(c_i, d_i) \in R$ for all i, also $(\sqcup_i c_i, \sqcup_i d_i) \in R$.*
2. *A PER $R \subseteq A \times A$ on a domain A is* admissible *if it is complete and $\bot \in |R|$.*

Let $\mathrm{CPer}(A)$ *and* $\mathrm{AdmPer}(A)$ *denote the full sub-categories of* $\mathrm{Per}(A)$ *consisting of complete PERs and admissible PERs, respectively.*

Definition 5. *Define $u : \mathbb{V} \to_c (T(\mathbb{V}) \to_c T(\mathbb{V}))$ as follows,*

$$u(x)(y) \stackrel{def}{=} \begin{cases} z & \text{if } x \cdot in_T(y) = in_T(z) \\ \bot & \text{otherwise} \end{cases}$$

and let lfp denote the realizer $in_\to(\lambda x.\ [in_T(\sqcup_n (u(x))^n)])$.

Lemma 1. *Let $R \in \mathrm{AdmPer}(T(\mathbb{V}))$, then*

$$\mathit{lfp} \in |(in_T(R) \to in_T(R)) \to in_T(R)|$$

and for all $\alpha \in |in_T(R) \to in_T(R)|$,

$$\alpha \cdot (\mathit{lfp} \cdot \alpha)\ in_T(R)\ \mathit{lfp} \cdot \alpha$$

Proof. See Lemma 6 in Appendix C.1, which generalizes the fixed point operator to impredicative Π types into an admissible PER.

The above development is standard and suffices to model fixed points over partial types in a non-stateful, simply-typed setting. However, it does not extend directly to a stateful dependently-typed setting: Assume $\vdash \tau : \textbf{\textit{type}}$ and $x : \tau \vdash \sigma : \textbf{\textit{type}}$. Then τ is interpreted as a PER $R \in \mathrm{Per}(\mathbb{V})$, and σ as an $[R]$-indexed family of PERs $S \in [R] \to \mathrm{Per}(\mathbb{V})$, and $\vdash \Sigma x : \tau.\sigma : \textbf{\textit{type}}$ as the PER $\Sigma_R(S)$:

$$\Sigma_R(S) = \{(in_\times(a_1, b_1), in_\times(a_2, b_2)) \mid a_1\ R\ a_2 \wedge b_1\ S([a_1]_R)\ b_2\}.$$

In general, this PER is not chain-complete even if R and each S_x is: given a chain $(a_i, b_i)_{i\in\mathbb{N}}$, we do not know in general that $[a_i]_R = [a_j]_R$ and hence cannot apply the completeness of S_x to the chain $(b_i)_{i\in\mathbb{N}}$ for any $x \in [R]$.

To rectify this problem we impose the following monotonicity condition on the PERs, which ensures exactly that $a_i\ R\ a_j$, for all $i, j \in \mathbb{N}$, and hence that $\Sigma_R(S)$ is chain-complete.

Definition 6 (CMPer(A))**.** *A PER $R \subseteq A \times A$ on a pre-domain A is* monotone *if, for all $x, y \in |R|$ such that $x \leq y$, we have $(x, y) \in R$. Let* CMPer(A) *denote the full sub-category of* Per(A) *on the complete monotone PERs.*

Restricting to complete monotone PERs forces a trivial equality on any particular Hoare type, as all of the elements of the type have to be equal to the diverging computation. However, it does not trivialize the totality of Hoare types, as we can still interpret each distinct Hoare type, $\boldsymbol{ST}\ \tau\ s$, as a distinct PER R, containing the computations that satisfy the specification s.

Restricting to complete monotone PERs does not collapse the equality on types in the purely functional fragment of iHTT, as these types are all interpreted as PERs without a bottom element in their domain. In particular, Π-types, which are modelled as elements of $\mathbb{V} \to \mathbb{V}_\perp$, picks out only those elements that map to non-bottom on their domain.

We shall see later that the monotonicity condition is also used to interpret partial types with a post-condition, which induces a dependency similar to that of Σ types, in the semantics of partial types.

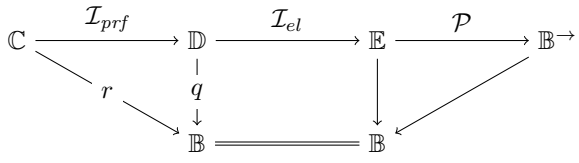
3.1 iHTT

So far, we have informally introduced a PER model of a single dependent type universe extended with partial types. The next step is to scale the ideas to a model of all of iHTT, and to prove that we do indeed get a model of iHTT. We start by showing that CMPERs and assemblies form a model of the underlying dependent type theory. Next, we sketch the interpretation of iHTT specific features such as heaps and computations in the model. Lastly, we show that the model features W-types at both the ***set*** and ***type*** universe.

Underlying DTT. We begin by defining a general class of models for the dependent type theory underlying iHTT, and then present a concrete instance based on complete monotone PERs.

To simplify the presentation and exploit existing categorical descriptions of models of the Calculus of Constructions, the model will be presented using the fibred approach of [14]. To simplify the definition of the interpretation function, we consider a split presentation of the model (i.e., with canonical choice of all fibred structure, preserved on-the-nose).

Definition 7 (Split iHTT structure). *A split iHTT structure is a structure*



such that

– $\mathcal{P}$ *is a split closed comprehension category,*
– $\mathcal{I}_{el}$ *and* $\mathcal{I}_{prf}$ *are split fibred reflections*
– *the coproducts induced by the* $\mathcal{I}_{el}$ *reflection are strong (i.e.,* $\mathcal{P} \circ \mathcal{I}_{el}$ *is a split closed comprehension category), and*
– *there exists objects* $\Omega_{el}, \Omega_{prf} \in \mathbb{E}_1$ *such that* $\{\Omega_{el}\}$ *is a split generic object for the q fibration and* $\{\Omega_{prf}\}$ *is a split generic object for the r fibration, where* $\{-\} = \mathrm{Dom} \circ \mathcal{P} : \mathbb{E} \to \mathbb{B}$.

The idea is to model contexts in $\mathbb{B}$, and the three universes, ***prop***, ***set***, and ***type*** in fibres of $\mathbb{C}$, $\mathbb{D}$ and $\mathbb{E}$, respectively. The split closed comprehension category structure models unit, Π and Σ types in the ***type*** universe. The split fibred reflections models the inclusion of ***prop*** into ***set*** and ***set*** into ***type*** and induces unit, Π and weak Σ types in ***prop*** and ***set***. Lastly, the split generic objects models the axioms ***prop*** : ***type*** and ***set*** : ***type***, respectively.

The concrete model we have in mind is mostly standard: the contexts and the ***type*** universe will be modelled with assemblies, the ***set*** universe with complete monotone PERs, and the ***prop*** universe with regular subobjects of assemblies. We begin by defining a category of uniform families of complete monotone PERs. Uniformity refers to the fact that each morphism is realized by a single $\alpha \in \mathbb{V}$.

Definition 8 ($\mathrm{UFam}(\mathrm{CMPer}(A))$**).**

– *Objects are pairs* $(I, (S_i)_{i \in |I|})$ *where* $I \in \mathrm{Asm}(\mathbb{V})$ *and each* $S_i \in \mathrm{CMPer}(\mathbb{V})$.
– *Morphisms from* (I, S_i) *to* (J, T_j) *are pairs* $(u, (f_i)_{i \in |I|})$ *where*

$$u : I \to J \in \mathrm{Asm}(\mathbb{V}) \qquad \textit{and} \qquad f_i : [S_i] \to [T_{u(i)}]$$

such that there exists an $\alpha \in \mathbb{V}$ *satisfying*

$$\forall i \in |I|.\ \forall e_i \in E_I(i).\ \forall e_v \in |S_i|.\ \alpha \cdot e_i \cdot e_v \in f_i([e_v]_{S_i})$$

Recall [14] that the standard category $\mathrm{UFam}(\mathrm{Per}(A))$ is defined in the same manner, using all PERs instead of only the complete monotone ones.

Let $\mathrm{RegSub}(\mathrm{Asm}(\mathbb{V}))$ denote the standard category of regular subobjects of assemblies and $\mathrm{UFam}(\mathrm{Asm}(\mathbb{V}))$ the standard category of uniform families of assemblies (see [14] for a definition). There is a standard split fibred reflection of $\mathrm{UFam}(\mathrm{Per}(A))$ into $\mathrm{UFam}(\mathrm{Asm}(A))$: the inclusion views a PER R as the assembly $([R], id_{[R]})$ [14]. This extends to a split fibred reflection of $\mathrm{UFam}(\mathrm{CMPer}(A))$ into $\mathrm{UFam}(\mathrm{Asm}(V))$ by composing with the following reflection of $\mathrm{UFam}(\mathrm{CMPer}(A))$ into $\mathrm{UFam}(\mathrm{Per}(A))$.

Lemma 2. *The inclusion* $\mathcal{I} : \mathrm{UFam}(\mathrm{CMPer}(A)) \to \mathrm{UFam}(\mathrm{Per}(A))$ *is a split fibred reflection.*

Proof (Sketch). We show that $\mathrm{CMPer}(A)$ is a reflective sub-category of $\mathrm{Per}(A)$; the same construction applies to uniform families, by a point-wise lifting. The left-ajoint, $\mathcal{R} : \mathrm{Per}(A) \to \mathrm{CMPer}(A)$ is given by monotone completion:

$$\mathcal{R}(S) = \overline{S} \qquad\qquad \mathcal{R}([\alpha]_{R \to S}) = [\alpha]_{\overline{R} \to \overline{S}}$$

where $\overline{R} \stackrel{\text{def}}{=} \bigcap\{S \in \text{CMPer}(\mathbb{V}) \mid R \subseteq S\}$ for a PER $R \in \text{Per}(\mathbb{V})$.

Let $S \in \text{Per}(A)$ and $T \in \text{CMPer}(A)$. Since the underlying realizers are continuous functions, we have that (see Lemma 7 in Appendix C)

$$\overline{S} \to T = S \to T$$

which induces the adjoint-isomorphism:

$$\text{CMPer}(A)(\mathcal{R}(S), T) \cong [\overline{S} \to T] = [S \to T] \cong \text{Per}(A)(S, \mathcal{I}(T))$$

Lemma 3. *The coproducts induced by the*

$$\mathcal{I}_{el} : \text{UFam}(\text{CMPer}(\mathbb{V})) \to \text{UFam}(\text{Asm}(\mathbb{V}))$$

reflection are strong.

Proof. Let $I \in \text{Asm}(\mathbb{V})$, $X \in \text{UFam}(\text{CMPer}(\mathbb{V}))_I$ and $Y \in \text{UFam}(\text{CMPer}(\mathbb{V}))_{\{X\}}$. Then the induced coproduct, $\Sigma_X(Y)$, is given by the family of complete monotone PERs,

$$\Sigma_X(Y) = \left(\overline{\left\{ (\alpha, \beta) \mid \begin{array}{c} \exists x, y \in \coprod\limits_{x \in [X_i]} [Y_{(i,x)}]. \\ \alpha \in E_{\coprod_i}(x) \wedge \beta \in E_{\coprod_i}(y) \wedge x \sim_i y \end{array} \right\}} \right)_{i \in |I|}$$

where

$$\begin{aligned} E_{\coprod_i}(a, b) &= \{in_\times(d, e) \mid d \in a \wedge b \in e\} \\ x \smile_i y \quad &\text{iff} \quad E_{\coprod_i}(x) \cap E_{\coprod_i}(y) = \emptyset \end{aligned}$$

and $\sim_i$ is the transitive closure of $\smile_i$.

Since $\mathcal{I}_{el}(X)$ and $\mathcal{I}_{el}(Y)$ are modest sets, $\smile_i$ is the equality relation on $\coprod_{x \in [X_i]} [Y_{(i,x)}]$, and the above PER thus reduces to the monotone completion of the standard PER interpretation of Σ types.

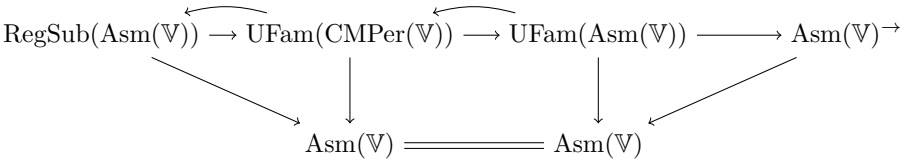
$$\Sigma_X(Y) = (\overline{\{(in_\times(a_1, b_1), in_\times(a_2, b_2)) \mid a_1 \ X_i \ a_2 \wedge b_1 \ Y_{(i,[a_1]_{X_i})} \ b_2\}})_{i \in |I|}$$

Since each X_i and each $Y_{(i,x)}$ is a complete monotone PER, the standard PER interpretation of Σ types is already a complete monotone PER (see Lemma 8 in Appendix C) and thus,

$$\Sigma_X(Y) = (\{(in_\times(a_1, b_1), in_\times(a_2, b_2)) \mid a_1 \ X_i \ a_2 \wedge b_1 \ Y_{(i,[a_1]_{X_i})} \ b_2\})_{i \in |I|}$$

The coproducts thus coincide with the coproducts induced by the $\text{UFam}(\text{Per}(\mathbb{V})) \to \text{UFam}(\text{Asm}(\mathbb{V}))$ reflection, which are strong [14, Section 10.5.8].

Theorem 1. *The diagram below forms a split iHTT structure.*



Interpretation. Except for impredicative Σ types and the axiom $\boldsymbol{set} : \boldsymbol{type}$, the interpretation of the underlying dependent type theory in the above concrete split iHTT structure is exactly the standard PER-assembly interpretation. The type $\boldsymbol{set}$ is interpreted as the set of complete monotone PERs over $\mathbb{V}$, instead of the set of all PERs over $\mathbb{V}$. Impredicative Σ types are interpreted as the monotone completion of the standard PER-assembly interpretation. For completeness, we have written out the concrete interpretation of the underlying dependent type theory in Appendix B.

As terms generally have multiple typing derivations in dependent type theories, the interpretation function is typically defined as a partial function on pre-terms and pre-contexts, and later shown to be defined on well-typed terms and contexts [23]. Formally, the interpretation is given by three mutually recursive partial functions,

$$\begin{aligned}
[\![-]\!]^{\mathrm{Ctx}} &: \mathrm{Ctx} \rightharpoonup obj(\mathrm{Asm}(\mathbb{V})) \\
[\![-]\!]^{\mathrm{Type}} &: \mathrm{Ctx} \times \mathrm{Term} \rightharpoonup obj(\mathrm{UFam}(\mathrm{Asm}(\mathbb{V}))) \\
[\![-]\!]^{\mathrm{Term}} &: \mathrm{Ctx} \times \mathrm{Term} \rightharpoonup hom(\mathrm{UFam}(\mathrm{Asm}(\mathbb{V})))
\end{aligned}$$

where Ctx is the set of pre-contexts and Term the set of pre-terms (defined in Appendix A.1), such that,

$$\begin{aligned}
[\![\Gamma \vdash A]\!]^{\mathrm{Type}} &\in \mathrm{UFam}(\mathrm{Asm}(\mathbb{V}))^{\mathrm{Ctx}}_{[\![\Gamma]\!]} \\
[\![\Gamma \vdash \mathrm{M}]\!]^{\mathrm{Term}} &\in \mathrm{UFam}(\mathrm{Asm}(\mathbb{V}))^{\mathrm{Ctx}}_{[\![\Gamma]\!]}(1[\![\Gamma]\!]^{\mathrm{Ctx}}, [\![\Gamma \vdash A]\!]^{\mathrm{Type}})
\end{aligned}$$

for well-typed types, $\Gamma \vdash A : \boldsymbol{type}$, and well-typed terms, $\Gamma \vdash M : A$. Terms of small types, $\Gamma \vdash \mathrm{M} : \boldsymbol{el}(\tau)$, are thus interpreted as morphisms in $\mathrm{UFam}(\mathrm{CMPer}(\mathbb{V}))$ fibres as follows:

$$\begin{aligned}
[\![\Gamma \vdash \mathrm{M} : \boldsymbol{el}(\tau)]\!]^{\mathrm{Term}} &\in \mathrm{UFam}(\mathrm{Asm}(\mathbb{V}))_I(1I, [\![\Gamma \vdash \boldsymbol{el}(\tau)]\!]^{\mathrm{Type}}) \\
&= \mathrm{UFam}(\mathrm{Asm}(\mathbb{V}))_I(1I, \mathcal{I}_{el}([\![\Gamma \vdash \tau]\!]^{\mathrm{Term}})) \\
&\cong \mathrm{UFam}(\mathrm{CMPer}(\mathbb{V}))_I(1I, [\![\Gamma \vdash \tau]\!]^{\mathrm{Term}})
\end{aligned}$$

where $I = [\![\Gamma]\!]^{\mathrm{Ctx}}$. Furthermore, global sections in $\mathrm{UFam}(\mathrm{CMPer}(\mathbb{V}))$ fibres are uniquely determined by their realizers:

$$\mathrm{UFam}(\mathrm{CMPer}(\mathbb{V}))_I(1I, X) \cong \left[\bigcap_{i \in |I|} \Delta(E_I(i)) \to X_i \right]$$

for $I \in \mathrm{Asm}(\mathbb{V})$ and $X \in \mathrm{UFam}(\mathrm{CMPer}(\mathbb{V}))_I$. We can thus define the interpretation of terms of small types by giving the underlying realizer. This is the view we will employ when defining the interpretation of the basic computations of iHTT.

Heaps. Pre- and post-conditions in iHTT are expressed as predicates over a type ***heap***, of heaps. In addition to a constant denoting the empty heap, this type features the following operations: ***upd***, ***free***, ***max***, ***dom***, and ***peek***. ***upd*** and ***free*** updates and frees the value at a given location, respectively. ***peek*** returns the value (if any) at a given location. ***dom*** decides whether a given location is allocated and ***max*** returns the largest allocated location (and 0 for the empty heap). The ***max*** operation allows one to define a recursion operator on heaps. From these low-level operations we can define the high-level operations such as points-to ($\mapsto$) and disjoint union ($\bullet$) used in Section 2.

heap is a large type (i.e., $\boldsymbol{heap} : \boldsymbol{type}$). The types of the built-in operations are as follows.

$$\begin{aligned}
&\boldsymbol{empty} : \boldsymbol{heap}\\
&\quad \boldsymbol{upd} : \Pi\tau : \boldsymbol{set}.\ \boldsymbol{heap} \to \boldsymbol{el}(\boldsymbol{nat}) \to \boldsymbol{el}(\tau) \to \boldsymbol{heap}\\
&\quad \boldsymbol{free} : \boldsymbol{heap} \to \boldsymbol{el}(\boldsymbol{nat}) \to \boldsymbol{heap}\\
&\quad \boldsymbol{max} : \boldsymbol{heap} \to \boldsymbol{el}(\boldsymbol{nat})\\
&\quad \boldsymbol{dom} : \boldsymbol{heap} \to \boldsymbol{el}(\boldsymbol{nat}) \to \boldsymbol{bool}\\
&\quad \boldsymbol{peek} : \boldsymbol{heap} \to \boldsymbol{el}(\boldsymbol{nat}) \to \boldsymbol{el}(1) + (\Sigma\tau : \boldsymbol{set}.\ \boldsymbol{el}(\tau))
\end{aligned}$$

We use $h[m \mapsto_\tau v]$ as shorthand for $\boldsymbol{upd}\ \tau\ h\ m\ v$ and $h \setminus m$ as shorthand for $\boldsymbol{free}\ h\ m$. These operations satisfy the heap axioms in Appendix A.11, which include several axioms relating the basic operations in addition to an induction principle and an extensionality principle for heaps.

We would like to model the values of ***heap*** as elements of $H(\mathbb{V})$. However, with this interpretation of ***heap*** we cannot interpret the update operation, $h[m \mapsto_\tau v]$, as the definition would not be independent of the choice of realizer for v. Rather, we introduce a notion of a world, which gives a notion of heap equivalence and interpret a heap as a pair consisting of a world and an equivalence class of heaps in the given world. A world is a finite map from locations to small semantic types:

$$\mathbb{W} \stackrel{\text{def}}{=} \boldsymbol{ptr} \xrightarrow{\text{fin}} \mathrm{CMPer}(\mathbb{V})$$

and two heaps $h_1, h_2 \in H(\mathbb{V})$ are considered equivalent in a world $w \in \mathbb{W}$ iff their support equals the domain of the world and their values are point-wise related by the world:

$$h_1 \sim_w h_2 \quad \text{iff} \quad supp(h_1) = supp(h_2) = \mathrm{Dom}(w) \wedge \forall l \in \mathrm{Dom}(w).\ h_1(l)\ w(l)\ h_2(l)$$

A typed heap is then a pair consisting of a world and an equivalence class of domain-theoretic heaps,

$$\mathbb{H}_t \stackrel{\text{def}}{=} \coprod_{w \in \mathbb{W}} [\sim_w]$$

and ***heap*** is interpreted as the set of typed heaps with the underlying domain-theoretic heaps as realizers:

$$[\![\Gamma \vdash \textbf{\textit{heap}}]\!]^{\mathrm{Type}} = (\mathbb{H}_t, (w, U) \mapsto in_H(U))_{i \in |I|}$$

for $I = [\![\Gamma]\!]^{\mathrm{Ctx}}$. That is, for each i, we have the assembly with underlying set $\mathbb{H}_t$ and with realizability map $\mathbb{H}_t \to P(\mathbb{V})$ given by $(w, U) \mapsto in_H(U)$. The realizers themselves do not have to contain any typing information, as we interpret small types with trivial realizability information (i.e., $[\![\Gamma \vdash \textbf{\textit{set}} : \textbf{\textit{type}}]\!] = \nabla(\mathrm{CMPer}(\mathbb{V}))$). Let $I = [\![\Gamma]\!]$, then

$$\begin{aligned}
[\![\Gamma \vdash \textbf{\textit{empty}}]\!]_i^{\mathrm{Term}} &= ([], \{[]\}) \\
[\![\Gamma \vdash \textbf{\textit{upd}}]\!]_i^{\mathrm{Term}}(R)(w, [h]_w)([n])([v]) &= (w[n \mapsto R], [h[n \mapsto v]]_{w[n \mapsto R]}) \\
[\![\Gamma \vdash \textbf{\textit{free}}]\!]_i^{\mathrm{Term}}(w, [h]_w)([n]) &= (w|_{\mathrm{dom}(w) \setminus n}, [h|_{\mathrm{dom}(w) \setminus n}]) \\
[\![\Gamma \vdash \textbf{\textit{max}}]\!]_i^{\mathrm{Term}}(w, [h]_w) &= [in_{\mathbb{N}}(\max\{n \in \mathbb{N} \mid h(n) \neq \bot \vee n = 0\})] \\
[\![\Gamma \vdash \textbf{\textit{dom}}]\!]_i^{\mathrm{Term}}(w, [h]_w)([n]) &= \begin{cases} [in_{\mathbb{N}}(0)] & \text{if } w(n) \text{ defined} \\ [in_{\mathbb{N}}(1)] & \text{otherwise} \end{cases} \\
[\![\Gamma \vdash \textbf{\textit{peek}}]\!]_i^{\mathrm{Term}}(w, [h]_w)([n]) &= \begin{cases} inl(w(n), [h(n)]_{w(n)}) & \text{if } w(n) \text{ defined} \\ inr([*]) & \text{otherwise} \end{cases}
\end{aligned}$$

In the interpretation of update, the world is used to ensure that the interpretation is independent of the choice of realizer v for N_2. Since the underlying domain-theoretic heaps do not contain any typing information, the world is also used in the interpretation of peek, to determine the type of the value stored at the given location.

Note that iHTT has "strong" update and that the world is modified to contain the new type (semantically, the per R) upon update. Thus our notion and use of worlds is different from the use of worlds in models of "weak" ML-like reference types, e.g. [5]; in particular, note that we do not index every type by a world, but only use worlds to interpret the type of heaps and the operations thereon (see also the next subsection for further discussion).

Theorem 2. *The heap axioms in Appendix A.11 hold in the model.*

Hoare Types. We are now ready to sketch the interpretation of Hoare-types. The idea is to interpret Hoare-types as PERs on elements of $T(\mathbb{V})$ that satisfy the given specification, but with a trivial equality. Specifically, given a partial correctness specification, we define an admissible subset $X \subseteq T(\mathbb{V})$ of computations satisfying the specification, and interpret the associated Hoare type as the PER,

$$R = in_T(\Delta(X)) = \{(in_T(f), in_T(g)) \mid f \in X \wedge g \in X\}$$

The trivial equality ensures that R is trivially monotone and admissibility on the sub-domain of computations follows from admissibility of X.

Assume a semantic pre-condition $P \in \mathbb{H}_t \to 2$, a small semantic type $R \in \mathrm{CMPer}(\mathbb{V})$, and a semantic post-condition $Q \in [R] \times \mathbb{H}_t \times \mathbb{H}_t \to 2$. As explained in the previous section, the pre- and post-condition is expressed in terms of a typed heaps, $\mathbb{H}_t$, instead of the underlying domain theoretic heaps. The subset of computations satisfying the specification is thus defined using the usual "forall initial worlds, there exists a terminal world"-formulation, known from models of ML-like references [5]. However, as iHTT supports strong update and deallocation, the terminal world is not required to be an extension of the initial world. Specifically, define $\mathit{hoare}(R, P, Q)$ as the following subset of $T(\mathbb{V})$:

$$\begin{aligned}\mathit{hoare}(R,P,Q) \stackrel{\text{def}}{=} \{f \in T(\mathbb{V}) \mid \\ \forall w \in \mathbb{W}.\, \forall h \in |\sim_w|.\, P(w,[h]_w) = \top \Rightarrow \\ (f(h) = \bot \ \vee \exists v', h'.\, f(h) = (v', h') \wedge v' \in |R| \wedge \\ h' \in \overline{\{h' \in H(\mathbb{V}) \mid \exists w' \in \mathbb{W}.\, Q([v']_R)(w,[h]_w)(w',[h']_{w'}) = \top\}})\}\end{aligned}$$

where $\overline{(-)}$ denotes the chain-completion operator on $T(\mathbb{V})$. The explicit chain-completion of the post-condition is required because of the existential quantification over worlds. Furthermore, since the post-condition is indexed by the return value $[v']_R$, monotonicity is used to collapse a chain of domain-theoretic values $v_1 \leq v_2 \leq \cdots$ into a single type-theoretic value $[v_1]_R$, when proving that $\mathit{hoare}(R, P, Q)$ is an admissible subset of $T(\mathbb{V})$.

A Hoare-type in the is now interpreted as follows:

$$[\![\Gamma \vdash \textbf{\textit{st}}\ \tau\ (\mathrm{P}, \mathrm{Q})]\!]_i^{\mathrm{Type}} = \mathit{in}_T(\Delta(\mathit{hoare}([\![\Gamma \vdash \tau]\!]_i, [\![\Gamma \vdash \mathrm{P}]\!]_i, [\![\Gamma \vdash \mathrm{Q}]\!]_i)))$$

The previous model of iHTT [21] featured a non-trivial computational equality. However, the previous model lacked the worlds introduced in the previous section and with it a useful notion of heap equivalence. As a result, the computational equality in the previous model was very strict, rendering the structural rule for existentials in Hoare logic unsound. The new model validates all the usual structural rules of Hoare-logic.

Computations. iHTT contains five basic computations for returning a value, reading from the heap, writing to the heap, allocating a location and deallocation a location. These basic computations are given by the following terms, where $\{\mathrm{P}\}\tau\{\mathrm{Q}\}$ is shorthand for $\textbf{\textit{st}}\ \tau\ (\mathrm{P}, \mathrm{Q})^T$:

$$\begin{aligned}\textbf{\textit{ret}} &: \Pi\tau : \textbf{\textit{set}}.\ \Pi v : \tau.\ \{\lambda\_.\top\}\tau\{\lambda r, h_i, h_t.\ h_i = h_t \wedge r = v\} \\ \textbf{\textit{read}} &: \Pi\tau : \textbf{\textit{set}}.\ \Pi l : \textbf{\textit{nat}}. \\ &\qquad \{\lambda h.\ \textbf{\textit{dom}}\ h\ l = \textbf{\textit{true}}\}\tau\{\lambda r, h_i, h_t.\ h_i = h_t \wedge \textbf{\textit{peek}}\ h_t\ l = \mathit{inr}(\tau, r)\} \\ \textbf{\textit{write}} &: \Pi\tau : \textbf{\textit{set}}.\ \Pi l : \textbf{\textit{nat}}.\ \Pi v : \tau. \\ &\qquad \{\lambda h.\ \textbf{\textit{dom}}\ h\ n = \textbf{\textit{true}}\}\tau\{\lambda r, h_i, h_t.\ h_t = h_i[l \mapsto_\tau v]\} \\ \textbf{\textit{alloc}} &: \Pi\tau : \textbf{\textit{set}}.\ \Pi v : \tau. \\ &\qquad \{\lambda\_.\top\}\textbf{\textit{nat}}\{\lambda r, h_i, h_t.\ h_t = h_i[r \mapsto_\tau v] \wedge r \neq 0 \wedge \textbf{\textit{dom}}\ h_i\ r = \textbf{\textit{false}}\} \\ \textbf{\textit{dealloc}} &: \Pi l : \textbf{\textit{nat}}.\ \{\lambda h.\ \textbf{\textit{dom}}\ h\ l = \textbf{\textit{true}}\}1\{\lambda\_, h_i, h_t.\ h_t = h_i \setminus l\}\end{aligned}$$

In addition, iHTT contains a term, ***bind***, for combining a two computations in a sequential composition:

$$\begin{aligned}
&\textbf{\textit{bind}} : \Pi\tau, \sigma : \textbf{\textit{set}}.\ \Pi s_1 : \mathit{spec}\ \tau.\ \Pi s_2 : \tau \to \mathit{spec}\ \sigma. \\
&\qquad \textbf{\textit{st}}\ \tau\ s_1 \to (\Pi v : \tau.\ \textbf{\textit{st}}_\sigma\ (s_2\ v)) \to \\
&\qquad\qquad \{\lambda h_i.\ \pi_1(s_1)\ h_i \wedge \forall x h_t.\ \pi_2(s_1)\ x\ h_i\ h_t \Rightarrow \pi_1(s_2\ x)\ h_m\} \\
&\qquad\qquad\quad \sigma \\
&\qquad\qquad \{\lambda r, h_i, h_t.\ \exists x, h.\ \pi_2(s_1)\ x\ h_i\ h_t \wedge \pi_2(s_2)\ r\ h\ h_t\}
\end{aligned}$$

and a term, ***do***, which corresponds to the structural rule of consequence in Hoare-logic:

$$\textbf{\textit{do}} : \Pi\tau : \textbf{\textit{set}}.\ \Pi s_1, s_2 : \mathit{spec}\ \tau.\ \mathit{conseq}\ s_1\ s_2 \to \textbf{\textit{st}}\ \tau\ s_1 \to \textbf{\textit{st}}\ \tau\ s_2$$

where *conseq* is specification implication:

$$\begin{aligned}
&\mathit{conseq} : \Pi\tau : \textbf{\textit{set}}.\ \mathit{spec}\ \tau \to \mathit{spec}\ \tau \to \textbf{\textit{prop}} \\
&\mathit{conseq} = \lambda\tau.\ \lambda s_1.\ \lambda s_2.\ (\forall i.\ \pi_1(s_2)\ i \Rightarrow \pi_1(s_1)\ i) \wedge \\
&\qquad\qquad (\forall y, i, m.\ \pi_1(s_2)\ i \Rightarrow \pi_2(s_1)\ y\ i\ m \Rightarrow \pi_2(s_2)\ y\ i\ m)
\end{aligned}$$

Lastly, for each type A iHTT contains a fixed point operator, $\textbf{\textit{fix}}_A$, for computations with an argument of type A:

$$\begin{aligned}
&\textbf{\textit{fix}}_A : \Pi\tau : \textbf{\textit{set}}.\ \Pi s : \mathit{spec}\ \tau. \\
&\qquad ((\Pi x : A.\ \textbf{\textit{st}}\ \tau\ s) \to (\Pi x : A.\ \textbf{\textit{st}}\ \tau\ s)) \to \Pi x : A.\ \textbf{\textit{st}}\ \tau\ s
\end{aligned}$$

Below we give the interpretation of the basic computations, by giving the underlying realizers. As a notational convenience, we write $\lambda x.\ f$ as shorthand for $\mathit{in}_\to(\lambda x.\ f)$. Let $I = [\![\Gamma]\!]$, then,

$$\begin{aligned}
[\![\Gamma \vdash \textbf{\textit{ret}}]\!] &= \lambda e_i.\ \lambda_.\ \lambda e_v.\ \mathit{in}_T(\lambda h.\ (e_v, h)) \\
[\![\Gamma \vdash \textbf{\textit{read}}]\!] &= \lambda e_i.\ \lambda_.\ \lambda n.\ \mathit{in}_T(\lambda h.\ \textbf{\textit{if}}\ h(n) \neq \bot\ \textbf{\textit{then}}\ (h(n), h)\ \textbf{\textit{else err}}) \\
[\![\Gamma \vdash \textbf{\textit{write}}]\!] &= \lambda e_i.\ \lambda_.\ \lambda n.\ \lambda e_v. \\
&\qquad \mathit{in}_T(\lambda h.\ \textbf{\textit{if}}\ h(n) \neq \bot\ \textbf{\textit{then}}\ (\mathit{in}_1(*), h[n \mapsto e_v])\ \textbf{\textit{else err}}) \\
[\![\Gamma \vdash \textbf{\textit{alloc}}]\!] &= \lambda e_i.\ \lambda_.\ \lambda e_v.\ \mathit{in}_T(\lambda h.\ \textbf{\textit{let}}\ l = \textbf{\textit{leastfree}}(h)\ \textbf{\textit{in}}\ (l, h[l \mapsto e_v])) \\
[\![\Gamma \vdash \textbf{\textit{dealloc}}]\!] &= \lambda e_i.\ \lambda n.\ \lambda h. \\
&\qquad \mathit{in}_T(\lambda h.\ \textbf{\textit{if}}\ h(n) = \bot\ \textbf{\textit{then err else}}\ (*, h[n \mapsto \bot])) \\
[\![\Gamma \vdash \textbf{\textit{bind}}]\!] &= \lambda e_i.\ \lambda_.\ \lambda_.\ \lambda_.\ \lambda_.\ \lambda m.\ \lambda n.\ \mathit{in}_T(\mathit{bind}(m, n)) \\
[\![\Gamma \vdash \textbf{\textit{do}}]\!] &= \lambda e_i.\ \lambda_.\ \lambda_.\ \lambda_.\ \lambda_.\ \lambda m.\ m \\
[\![\Gamma \vdash \textbf{\textit{fix}}_A]\!] &= \lambda e_i.\ \lambda_.\ \lambda_.\ \lambda m.\ \mathit{in}_\to(\lambda v.\ \mathit{in}_T((\sqcup_n (u(m))^n)(v)))
\end{aligned}$$

where

$$bind(m,n) \stackrel{\text{def}}{=} \lambda h.\ \textbf{\textit{case}}\ \pi_T(m)(h)\ \textbf{\textit{of}} \begin{array}{ll} (v,h') & \Rightarrow \pi_T(n\cdot v)(h') \\ \textbf{\textit{err}} & \Rightarrow \textbf{\textit{err}} \\ \bot & \Rightarrow \bot \end{array}$$

$$\textbf{\textit{leastfree}} \stackrel{\text{def}}{=} \lambda h.\ \min\{n \in \mathbb{N}_+ \mid h(n) = \bot\}$$

$$u(x:\mathbb{V})(y:\mathbb{V}\to T(\mathbb{V}))(z:\mathbb{V}) \stackrel{\text{def}}{=} \begin{cases} a & \text{if } x\cdot in_{\to}(\lambda z.\ in_T(y(z)))\cdot z = in_T(a) \\ \bot & \text{otherwise} \end{cases}$$

$$\pi_T(x:\mathbb{V}) \stackrel{\text{def}}{=} \begin{cases} a & \text{if } x\downarrow \text{ and } x = in_T(a) \\ \bot_{T(\mathbb{V})} & \text{otherwise} \end{cases}$$

Theorem 3. *The interpretation of computations is sound, i.e., well-typed computations satisfy their specifications.*

Proof. We prove three representative cases: ***bind***, ***write***, and $\textbf{\textit{fix}}_A$.

In the case of ***bind***, the intermediate state in the execution of a sequential composition satisfies the chain-completion of the post-condition of the first computation. To show that ***bind*** is sound, one uses that the underlying computations are continuous to reason about intermediate states added via the chain-completion. Soundness follows from Lemma 9 in Appendix C.3.

The soundness of $\textbf{\textit{fix}}_A$ follows from Lemma 6 in Appendix C.1, which generalizes the the least fixed point operator on $\mathrm{Per}(\mathbb{V})$ to computations with a single argument of a large type.

Lastly, the soundness of ***write*** follows from Lemma 10 in Appendix C.3.

W-types. The presentation of (co)-inductive types in CIC is based on an intricate syntactic scheme of inductive families. So far, this presentation of (co)-inductive types has eluded a categorical semantics. Martin-Löf type theory features an alternative presentation, based on W-types (a type-theoretic formalization of well-founded trees), which is strong enough to represent a wide range of predicative inductive types in extensional models (such as ours) [11, 3]. Since W-types in addition have a simple categorical semantics, we have chosen to show that iHTT models inductive types by showing that it models W-types. Specifically, we show that it models W-types at both the ***type*** and ***set*** universe, and that the W-types at the ***set*** universe supports elimination over large types.

Semantically, in the setting of locally cartesian closed categories, W-types are modelled as initial algebras of polynomial functors [16]. In the setting of split closed comprehension categories we define:

Definition 9. *A split closed comprehension category $\mathcal{P} : \mathbb{E} \to \mathbb{B}^{\to}$ has split W-types, if for every $I \in \mathbb{B}$, $X \in \mathbb{E}_I$ and $Y \in \mathbb{E}_{\{X\}}$ the endo-functor,*

$$P_{I,X,Y} = \Sigma_X \circ \Pi_Y \circ (\pi_X \circ \pi_Y)^* : \mathbb{E}_I \to \mathbb{E}_I$$

has a chosen initial algebra $\alpha_{I,X,Y} : P_{I,X,Y}(W_{I,X,Y}) \to W_{I,X,Y} \in \mathbb{E}_I$*, which is preserved on-the-nose by re-indexing functors.*

As is well-known, $\mathrm{Asm}(\mathbb{V})$ is locally cartesian closed and all polynomial functors on $\mathrm{Asm}(\mathbb{V})$ have initial algebras. This yields initial algebras for functors $P_{X,Y} : \mathrm{UFam}(\mathrm{Asm}(\mathbb{V}))_1 \to \mathrm{UFam}(\mathrm{Asm}(\mathbb{V}))_1$ for $X \in \mathbb{E}_1$ and $Y \in \mathbb{E}_{\{X\}}$. This lifts to an arbitrary context $I \in \mathrm{Asm}(\mathbb{V})$ by a point-wise construction, which yields split W-types, as re-indexing in $\mathrm{UFam}(\mathrm{Asm}(\mathbb{V}))$ is by composition:

Lemma 4. *The split ccomp* $\mathcal{P} : \mathrm{UFam}(\mathrm{Asm}(\mathbb{V})) \to \mathrm{Asm}(\mathbb{V})^{\to}$ *has split W-types.*

Proof. W-types in $\mathrm{Asm}(\mathbb{V})$ are constructed from the W-types in Sets, by restricting to "heriditarily realized" trees (see Appendix C.4 for the full construction): In the case where $I = 1$, assume $X \in \mathrm{Asm}(\mathbb{V})$ and $Y \in \mathrm{UFam}(\mathrm{Asm}(\mathbb{V}))_X$ and let $T_{X,Y}$ denote a solution to the set-isomorphism,

$$sup_{X,Y} : \coprod_{x \in |X|} |Y_x| \to T_{X,Y} \cong T_{X,Y}$$

and let $E_{X,Y} : T_{X,Y} \to \mathcal{P}(\mathbb{V})$ denote the unique function satisfying,

$$E_{X,Y}(sup(x,f)) = \{in_\times(e_x, e_f) \mid e_x \in E_X(x) \wedge \forall y \in |Y_x|.\ \forall e_y \in E_{Y_x}(y).\ \} \\ e_f \cdot e_y \in E_{X,Y}(f(y))$$

Then $W_{X,Y} \stackrel{\text{def}}{=} (\{w \in T_{X,Y} \mid E_{X,Y}(w) \neq \emptyset\}, E_{X,Y}) \in \mathrm{Asm}(\mathbb{V})$ and $sup : P_{X,Y}(W_{X,Y}) \to W_{X,Y}$ is an initial $P_{X,Y}$ algebra, realized by the identity (Lemma 14 in Appendix C.4).

In the case of an arbitrary context $I \in \mathrm{Asm}(\mathbb{V})$, assume $X \in \mathrm{UFam}(\mathrm{Asm}(\mathbb{V}))_I$ and $Y \in \mathrm{UFam}(\mathrm{Asm}(\mathbb{V}))_{\{X\}}$, then we define $W_{I,X,Y}$ and $\alpha_{I,X,Y}$ by a point-wise lifting:

$$W_{I,X,Y} = \left(W_{X_i, \lambda y.\ Y_{(i,y)}}\right)_{i \in |I|}$$

and

$$\alpha_{I,X,Y} = \left(id_I, \left(sup_{X_i, \lambda y.\ Y_{(i,y)}}\right)_{i \in |I|}\right)$$

$\alpha_{I,X,Y}$ is uniformly realized, since each of the underlying *sup* functions is realized by the identitiy.

To show that these are further preserved on the nose by reindexing, assume $J \in \mathrm{Asm}(\mathbb{V})$ and $u : J \to I \in \mathrm{Asm}(\mathbb{V})$. Then,

$$u^*(\alpha_{I,X,Y}) = \left(id_J, \left(\alpha_{X_{u(j)}, \lambda y.\ Y_{(u(j),y)}}\right)_{j \in |J|}\right) = \alpha_{J, u^*(X), \{\overline{u}(X)\}^*(Y)}$$

This models W-types in the ***type***-universe. In addition, iHTT features small W-types over small types, with elimination over large types. The idea is to model these small W-types by forming the W-type in $\mathrm{UFam}(\mathrm{Asm}(\mathbb{V}))$ and mapping it

back into UFam(CMPer($\mathbb{V}$)) using the reflection. The reflection works by collapsing an assembly into a PER by equating values with overlapping sets of realizers and into a complete monotone PER by taking the monotone completion of said PER. In the case where X and Y are in the image of $\mathcal{I}_{el}$, the construction of $W_{I,X,Y}$ sketched above yields a modest set (i.e., has no overlapping sets of realizers) and furthermore, the induced PER is already monotone and complete. This induces the following isomorphism, which allows us to model small W-types with elimination over large types:

Lemma 5. *For every $I \in \mathbb{B}$, $X \in \mathrm{UFam}(\mathrm{CMPer}(\mathbb{V}))_I$, and $Y \in \mathrm{UFam}(\mathrm{CMPer}(\mathbb{V}))_{\{X\}}$, there is a chosen isomorphism,*

$$W_{I,\mathcal{I}_{el}(X),\mathcal{I}_{el}(Y)} \cong \mathcal{I}_{el}(\mathcal{R}_{el}(W_{I,\mathcal{I}_{el}(X),\mathcal{I}_{el}(Y)})),$$

which is preserved on-the-nose by reindexing functors.

Proof (Sketch). There is a well-known equivalence between PERs and modest sets. This extends to complete monotone PERs as follows: Define CMod($\mathbb{V}$) as the full-subcategory of Mod($\mathbb{V}$), consiting of modest sets (X, E_X), satisfying the following two conditions,

$$\forall x, y \in X.\ (\exists a, b.\ a \in E_X(x) \wedge b \in E_X(y) \wedge a \leq b) \Rightarrow x = y$$
$$\forall x \in X.\ \forall c : \mathbb{N} \to_m \mathbb{V}.\ (\forall n \in \mathbb{N}.\ c(n) \in E_X(x)) \Rightarrow \sqcup_n c(n) \in E_X(x)$$

then $\mathcal{I}_{el}$ and $\mathcal{R}_{el}$ forms an equivalence between CMPer($\mathbb{V}$) and CMod($\mathbb{V}$) and furthermore for every $X \in \mathrm{CMod}(\mathbb{V})$ there is a chosen isomorphism $X \cong \mathcal{I}_{el}(\mathcal{R}_{el}(X))$. This lifts to a fibred equivalence between UFam(CMPer($\mathbb{V}$)) and UFam(CMod($\mathbb{V}$)), with chosen isomorphisms, $X \cong \mathcal{I}_{el}(\mathcal{R}_{el}(X))$ for $X \in \mathrm{UFam}(\mathrm{CMod}(\mathbb{V}))_I$, preserved on the nose by re-indexing functors.

The result thus follows by showing that $W_{X,Y} \in \mathrm{CMPer}(\mathbb{V})$ for $X \in \mathrm{Asm}(\mathbb{V})$ and $Y \in \mathrm{UFam}(\mathrm{Asm}(\mathbb{V}))_X$, which follows by well-founded induction on $T_{X,Y}$ (Lemma 15 in Appendix C.4).

4 Implementation

One of the advantages of weakening Hoare types instead of the underlying dependent type theory – as in the case of [21] – is that it can simplify implementation of the resulting type theory. In our case, presenting iHTT as an extension of CIC allows for easy implementation as an axiomatic extension of the Coq proof assistant.

Our implementation is based on the Coq infrastructure developed by Nanevski et. al. [19], to support efficient reasoning about stateful computations in Coq. This infrastructure defines a new Hoare type on top of the built-in Hoare type with support for more efficient reasoning, based on ideas from separation logic. Specifically, as illustrated in Section 2, this new Hoare type features (1) small

footprint specifications and (2) efficient reasoning about heaps. Efficient reasoning about heaps is achieved by reasoning using the partial commutative monoid $(1+\boldsymbol{heap}, \bullet)$ instead of the total non-commutative monoid $(\boldsymbol{heap}, \bullet)$, where $\boldsymbol{heap}$ is the actual type of heaps and $\bullet$ is heap union [19].

Compared to Nanevski et. al.'s implementation of predicative Hoare Type Theory (pHTT), this new implementation features higher-order store and impredicative quantification, but the predicative hierarchy lacks Hoare-types. The new implementation is almost source compatible with verifications in pHTT that do not exploit the predicative hierarchy.

Compared to the Ynot implementation of pHTT [18], in addition to impredicativity the main difference lies in the treatment of ghost variables. Post conditions in Ynot are unary and thus employ ghost-variables to relate the pre- and post-condition. Ynot expresses ghost variables of a specification as computationally irrelevant arguments to the computation. As Coq lacks support for computationally irrelevant variables, Ynot extends Coq with an injectivity axiom, which gives an embedding of $\boldsymbol{set}$ into the computationally irrelevant $\boldsymbol{prop}$-universe. This axiom is inconsistent with a proof irrelevant $\boldsymbol{prop}$-universe and thus in particular unsound in our model. Additionally, this limits Ynot's ghost variables to small types, whereas iHTT supports ghost variables of large types.

5 Related work

Our approach to partiality is based on the idea of partial types, as introduced by Constable and Smith [9]. We have already discussed its relation to the work on admissibility by Crary [10] in the introduction. Below we first discuss related work on partiality, followed by related work on partial correctness reasoning.

Bove and Capretta [7] proposed representing a partial function $f : A \rightharpoonup B$ as a total function $\overline{f} : \Pi a : A.\ P(a) \to B$, defined by recursion over an inductively defined predicative $P : A \to \boldsymbol{prop}$, expressing the domain of the partial function. This allows the definition of partial computations by general recursion, but does not model non-termination, as $\overline{f}$ can only be applied to arguments on which it terminates. Capretta [8] proposed an alternative co-inductive representation, which does model non-termination, representing a partial function $f : A \rightharpoonup B$ as a total function $\overline{f} : A \to B^v$, where B^v is co-inductive type of partial elements of type B. This representation yields a least fixed point operator on finitrary (continuous) endo-functions on $A \to B^v$. Capretta does not provide a fixed point induction principle, but we believe such a principle would require admissibility proofs.

Another alternative approach to partiality is to give a model of a language featuring general recursion inside a dependent type theory. This allows one to model and reason about partial computations inside the type theory, but does not extend the type theory itself with partial computations. This approach has for instance been studied and implemented by Reus et. al. [22], who formalized Synthetic Domain Theory in the Lego proof checker. The resulting type theory can be seen as a very expressive version of LCF. The synthetic approach allevi-

ates the need for continuity proofs, but still requires admissibility proofs when reasoning by fixed point induction.

Hoare-style specification logics is another line of closely related work. With a collapsed computational equality, reasoning about a partial computation in iHTT is limited to Hoare-style partial correctness reasoning, as in a specification logic. With the modularity features provided by the underlying dependent type theory (i.e., Σ types), iHTT can thus be seen as a modular specification logic for a higher-order programming language.

6 Conclusion

We have presented a new approach for extending dependent type theory with potentially non-terminating computations, without weakening the underlying dependent type theory or adding restrictions on the use of fixed points in defining partial computations. We have shown that it scales to very expressive dependent type theories and effects beyond partiality, by extending the Calculus of Inductive Constructions with stateful and potentially non-terminating computations. We have further demonstrated that this approach is practical, by implementing this extension of CIC as an axiomatic extension of the Coq proof assistant.

To justify the soundness of our extension of CIC, we have presented a realizability model of the theory. For lack of space, we have limited the presentation to a single predicative universe, but the model can be extended to the whole predicative hierarchy $\boldsymbol{type} \subseteq \boldsymbol{type}_1 \subseteq \ldots$ of CIC.

References

1. The Coq Proof Assistant. http://coq.inria.fr/.
2. *Coq Reference Manual, Version 8.3.*
3. M. Abbott, T. Altenkirch, and N. Ghani. Representing nested inductive types using W-types. In *Proc. of ICALP*, 2004.
4. R. M. Amadio. Recursion over Realizability Structures. *Information and Computation*, 91:55–85, 1991.
5. L. Birkedal, K. Støvring, and J. Thamsborg. Realisability semantics of parametric polymorphism, general references and recursive types. *Math. Struct. Comp. Sci.*, 20(4):655–703, 2010.
6. D. Bjørner and C. B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *Lecture Notes in Computer Science*. Springer, 1978.
7. A. Bove. Simple General Recursion in Type Theory. *Nordic Journal of Computing*, 8, 2000.
8. V. Capretta. General Recursion via Coinductive Types. *Logical Methods in Computer Science*, 1(2):1–28, 2005.
9. R. L. Constable and S. F. Smith. Partial Objects in Constructive Type Theory. In *Proceedings of Second IEEE Symposium on Logic in Computer Science*, 1987.
10. K. Crary. Admissibility of Fixpoint Induction over Partial Types. In *Automated Deduction - CADE-15*, 1998.
11. P. Dybjer. Representing inductively defined sets by wellorderings in Martin-Löf's type theory. *Theor. Comput. Sci.*, 176(1-2):329–335, 1997.

12. G. Gonthier and A. Mahboubi. A Small Scale Reflection Extension for the Coq system. Technical report, INRIA, 2007.
13. C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.
14. B. Jacobs. *Categorical Logic and Type Theory*. Elsevier Science, 1999.
15. B. Meyer. *Object-oriented software construction*. Prentice Hall, 1997.
16. I. Moerdijk and E. Palmgren. Wellfounded trees in categories. *Annals of Pure and Applied Logic*, 104(1–3):189–218, 2000.
17. A. Nanevski, G. Morrisett, and L. Birkedal. Hoare Type Theory, Polymorphism and Separation. *Journal of Functional Programming*, 18(5–6):865–911, 2008.
18. A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: Dependent Types for Imperative Programs. In *Proceedings of ICFP 2008*, pages 229–240, 2008.
19. A. Nanevski, V. Vafeiadis, and J. Berdine. Structuring the Verification of Heap-Manipulating Programs. In *Proceedings of POPL 2010*, 2010.
20. P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL'01*, pages 1–19, 2001.
21. R. Petersen, L. Birkedal, A. Nanevski, and G. Morrisett. A Realizability Model of Impredicative Hoare Type Theory. In *Proceedings of ESOP 2008*, 2008.
22. B. Reus. Synthetic Domain Theory in Type Theory: Another Logic of Computable Functions. In *Proceedings of TPHOL 1996*, 1996.
23. T. Streicher. *Semantics of Type Theory: Correctness, Completeness, and Independence Results*. Birkhaeuser Verlag, 1991.

A Impredicative Hoare Type Theory

A.1 Syntax

$$
\begin{array}{rcl}
\Gamma & ::= & \varepsilon \mid \Gamma, x : A \\
\mathrm{M}, \mathrm{N}, A, B, \tau, \sigma, \mathrm{P}, \mathrm{Q} & ::= & x \mid \textbf{\textit{set}} \mid \textbf{\textit{prop}} \mid \textbf{\textit{el}}(\tau) \mid \textbf{\textit{prf}}(\mathrm{P}) \\
& \mid & \Pi^T x : A.B \mid Abs^T_{x:A.B}(\mathrm{M}) \mid App^T_{x:A.B}(\mathrm{M}, \mathrm{N}) \\
& \mid & \Pi^S x : A.\tau \mid Abs^S_{x:A.\tau}(\mathrm{M}) \mid App^S_{x:A.\tau}(\mathrm{M}, \mathrm{N}) \\
& \mid & \forall x : A.\mathrm{P} \mid Abs^P_{x:A.\mathrm{P}}(\mathrm{M}) \mid App^P_{x:A.\mathrm{P}}(\mathrm{M}, \mathrm{N}) \mid \bullet_{\mathrm{P}} \\
& \mid & \Sigma^T x : A.B \mid Pair^T_{x:A.B}(\mathrm{M}, \mathrm{N}) \mid \pi^1_{x:A.B}(\mathrm{M}) \mid \pi^2_{x:A.B}(\mathrm{M}) \\
& \mid & \Sigma^S x : A.\tau \mid Pair^S_{x:A.\tau}(\mathrm{M}, \mathrm{N}) \mid \textbf{\textit{unpack}}\ \mathrm{M}\ \textbf{\textit{as}}\ (x, y)\ \textbf{\textit{in}}\ \mathrm{N} \\
& \mid & A + B \mid \textbf{\textit{inl}}_{A,B}(\mathrm{M}) \mid \textbf{\textit{inr}}_{A,B}(\mathrm{M}) \\
& \mid & \textbf{\textit{case}}_{A,B}\ \mathrm{M}\ \textbf{\textit{in}}\ \textbf{\textit{inl}}(x) \Rightarrow \mathrm{N}_1 \mid \textbf{\textit{inr}}(x) \Rightarrow \mathrm{N}_2 \\
& \mid & \mathcal{W}^T x : A.B \mid sup^T_{x:A.B}(\mathrm{M}) \mid fold^T_{x:A.B}(\mathrm{M}) \\
& \mid & \mathcal{W}^S x : \tau.\sigma \mid sup^S_{x:\tau.\sigma}(\mathrm{M}) \mid fold^S_{x:\tau.\sigma}(\mathrm{M}) \\
& \mid & 1 \mid () \\
& \mid & \textbf{\textit{bool}} \mid \textbf{\textit{true}} \mid \textbf{\textit{false}} \mid \textbf{\textit{if}}\ \mathrm{M}\ \textbf{\textit{then}}\ \mathrm{N}_1\ \textbf{\textit{else}}\ \mathrm{N}_2 \\
& \mid & \textbf{\textit{nat}} \mid \textbf{\textit{zero}} \mid \textbf{\textit{succ}}\ \mathrm{M} \mid \mathrm{M} == \mathrm{N} \mid \mathrm{M} < \mathrm{N} \\
& \mid & \textbf{\textit{heap}} \mid \textbf{\textit{empty}} \mid \textbf{\textit{upd}} \mid \textbf{\textit{free}} \mid \textbf{\textit{max}} \mid \textbf{\textit{dom}} \mid \textbf{\textit{peek}} \\
& \mid & \textbf{\textit{st}} \mid \textbf{\textit{ret}} \mid \textbf{\textit{read}} \mid \textbf{\textit{write}} \mid \textbf{\textit{alloc}} \mid \textbf{\textit{dealloc}} \mid \textbf{\textit{bind}} \mid \textbf{\textit{do}} \mid \textbf{\textit{fix}}_A
\end{array}
$$

A.2 Judgments

$\Gamma\ \mathrm{Ctx}$	Γ is a well-formed context
$\Gamma \vdash A : \textbf{\textit{type}}$	A is a type in context Γ
$\Gamma \vdash \mathrm{M} : A$	M is a term of type A in context Γ
$\Gamma \vdash A = B : \textbf{\textit{type}}$	A and B are convertible in context Γ
$\Gamma \vdash \mathrm{M} = \mathrm{N} : A$	M and N are convertible in context Γ

We reserve the meta-variables A and B for types, the meta-variables τ and σ for small types (terms of type ***set***), and the meta-variables P and Q for propositions (terms of type ***prop***).

A.3 Contexts

$$\frac{}{\varepsilon\ \mathrm{Ctx}} \qquad \frac{x \notin \Gamma \qquad \Gamma \vdash A : \textbf{\textit{type}}}{\Gamma, x : A\ \mathrm{Ctx}}$$

A.4 Structural Rules

$$\frac{\Gamma \vdash A : \textbf{\textit{type}}}{\Gamma, x : A \vdash x : A}\ \pi \qquad \frac{\Gamma \vdash \mathrm{M} : A \qquad \Gamma, x : A, \Delta \vdash \mathcal{J}}{\Gamma, \Delta[\mathrm{M}/x] \vdash \mathcal{J}[\mathrm{M}/x]}\ \textsc{sub}$$

$$\frac{\Gamma, x : A, y : A, \Delta \vdash \mathcal{J}}{\Gamma, x : A, \Delta[x/y] \vdash \mathcal{J}[x/y]}\ \textsc{cont} \qquad \frac{\Gamma \vdash B : \boldsymbol{type} \qquad \Gamma, x : A, y : B, \Delta \vdash \mathcal{J}}{\Gamma, y : B, x : A, \Delta \vdash \mathcal{J}}\ \textsc{ex}$$

$$\frac{\Gamma \vdash A : \boldsymbol{type} \qquad \Gamma \vdash \mathcal{J}}{\Gamma, x : A \vdash \mathcal{J}}\ \textsc{weak} \qquad \frac{\Gamma \vdash \mathrm{M} : A \qquad \Gamma \vdash A = B : \boldsymbol{type}}{\Gamma \vdash \mathrm{M} : B}\ \textsc{conv}$$

A.5 Types

$$\frac{}{\varepsilon \vdash \boldsymbol{set} : \boldsymbol{type}}\ \textsc{type} \qquad \frac{}{\varepsilon \vdash \boldsymbol{prop} : \boldsymbol{type}}\ \textsc{prop}$$

$$\frac{\Gamma \vdash \tau : \boldsymbol{set}}{\Gamma \vdash \boldsymbol{el}(\tau) : \boldsymbol{type}}\ \textsc{ElT} \qquad \frac{\Gamma, x : A \vdash B : \boldsymbol{type}}{\Gamma \vdash \Pi^T x : A.\ B : \boldsymbol{type}}\ \Pi T \qquad \frac{\Gamma, x : A \vdash B : \boldsymbol{type}}{\Gamma \vdash \Sigma^T x : A.\ B : \boldsymbol{type}}\ \Sigma T$$

A.6 Small types

$$\frac{}{\Gamma \vdash 1 : \boldsymbol{set}}\ 1 \qquad \frac{}{\Gamma \vdash \boldsymbol{bool} : \boldsymbol{set}}\ \textsc{bool} \qquad \frac{}{\Gamma \vdash \boldsymbol{nat} : \boldsymbol{set}}\ \textsc{nat} \qquad \frac{\Gamma \vdash \mathrm{P} : \boldsymbol{prop}}{\Gamma \vdash \boldsymbol{prf}(\mathrm{P}) : \boldsymbol{set}}\ \textsc{PrfT}$$

$$\frac{\Gamma, x : A \vdash \tau : \boldsymbol{set}}{\Gamma \vdash \Pi^S x : A.\ \tau : \boldsymbol{set}}\ \Pi S \qquad \frac{\Gamma, x : A \vdash \tau : \boldsymbol{set}}{\Gamma \vdash \Sigma^S x : A.\ \tau : \boldsymbol{set}}\ \Sigma S$$

A.7 Propositions

$$\frac{\Gamma, x : A \vdash \mathrm{P} : \boldsymbol{prop}}{\Gamma \vdash \forall x : A.\mathrm{P} : \boldsymbol{prop}}$$

$$\frac{\Gamma, x : A \vdash \mathrm{M} : \boldsymbol{el}(\boldsymbol{prf}(\mathrm{P}))}{\Gamma \vdash \lambda^P x : A.\mathrm{M} : \boldsymbol{el}(\boldsymbol{prf}(\forall x : A.\mathrm{P}))} \qquad \frac{\Gamma \vdash \mathrm{M} : \boldsymbol{el}(\boldsymbol{prf}(\forall x : A.\mathrm{P})) \qquad \Gamma \vdash \mathrm{N} : A}{\Gamma \vdash App^P_{x:A.\mathrm{P}}(\mathrm{M}, \mathrm{N}) : \boldsymbol{el}(\boldsymbol{prf}(\mathrm{P}[\mathrm{N}/x]))}$$

Using universal quantification, we define the usual logical connectives as follows:

$$\begin{aligned}
\mathrm{P} \Rightarrow \mathrm{Q} &\stackrel{\text{def}}{=} \forall x : \boldsymbol{el}(\boldsymbol{prf}(\mathrm{P})).\ \mathrm{Q} \\
\top &\stackrel{\text{def}}{=} \forall \mathrm{P} : \boldsymbol{prop}.\ \mathrm{P} \Rightarrow \mathrm{Q} \\
\bot &\stackrel{\text{def}}{=} \forall \mathrm{P} : \boldsymbol{prop}.\ \mathrm{P} \\
\mathrm{P} \wedge \mathrm{Q} &\stackrel{\text{def}}{=} \forall \mathrm{R} : \boldsymbol{prop}.\ (\mathrm{P} \Rightarrow \mathrm{Q} \Rightarrow \mathrm{R}) \Rightarrow \mathrm{R} \\
\mathrm{P} \vee \mathrm{Q} &\stackrel{\text{def}}{=} \forall \mathrm{R} : \boldsymbol{prop}.\ (\mathrm{P} \Rightarrow \mathrm{R}) \Rightarrow (\mathrm{Q} \Rightarrow \mathrm{R}) \Rightarrow \mathrm{R} \\
\neg \mathrm{P} &\stackrel{\text{def}}{=} \mathrm{P} \Rightarrow \bot \\
\exists x : A.\mathrm{P} &\stackrel{\text{def}}{=} \forall \mathrm{R} : \boldsymbol{prop}.\ (\Pi^P x : A.\ (\mathrm{P} \Rightarrow \mathrm{R})) \Rightarrow \mathrm{R} \\
a =_A b &\stackrel{\text{def}}{=} \forall \mathrm{P} : A \to \boldsymbol{prop}.\ \mathrm{P}(a) \Rightarrow \mathrm{P}(b)
\end{aligned}$$

A.8 Terms

$$\frac{}{\Gamma \vdash () : \boldsymbol{el}(1)}$$

$$\frac{}{\Gamma \vdash \boldsymbol{true} : \boldsymbol{el}(\boldsymbol{bool})} \qquad \frac{}{\Gamma \vdash \boldsymbol{false} : \boldsymbol{el}(\boldsymbol{bool})}$$

$$\frac{\Gamma \vdash \mathrm{M} : \boldsymbol{bool} \qquad \Gamma \vdash \mathrm{N}_1 : A \qquad \Gamma \vdash \mathrm{N}_2 : A}{\Gamma \vdash \boldsymbol{if}\ \mathrm{M}\ \boldsymbol{then}\ \mathrm{N}_1\ \boldsymbol{else}\ \mathrm{N}_2 : A}$$

$$\frac{}{\Gamma \vdash \boldsymbol{zero} : \boldsymbol{el}(\boldsymbol{nat})} \qquad \frac{\Gamma \vdash \mathrm{M} : \boldsymbol{el}(\boldsymbol{nat})}{\Gamma \vdash \boldsymbol{succ}\ \mathrm{M} : \boldsymbol{el}(\boldsymbol{nat})}$$

$$\frac{\Gamma \vdash \mathrm{M}, \mathrm{N} : \boldsymbol{el}(\boldsymbol{nat})}{\Gamma \vdash \mathrm{M} == \mathrm{N} : \boldsymbol{el}(\boldsymbol{bool})} \qquad \frac{\Gamma \vdash \mathrm{M}, \mathrm{N} : \boldsymbol{el}(\boldsymbol{nat})}{\Gamma \vdash \mathrm{M} < \mathrm{N} : \boldsymbol{el}(\boldsymbol{bool})}$$

$$\frac{\Gamma, x : A \vdash \mathrm{M} : B}{\Gamma \vdash \lambda^T x : A.\ \mathrm{M} : \Pi^T x : A.\ B} \qquad \frac{\Gamma \vdash \mathrm{M} : \Pi^T x : B.\ A \qquad \Gamma \vdash \mathrm{N} : B}{\Gamma \vdash \mathrm{M}\ \mathrm{N} : A[N/x]}$$

$$\frac{\Gamma, x : A \vdash \mathrm{M} : \boldsymbol{el}(\tau)}{\Gamma \vdash \lambda^S x : A.\ \mathrm{M} : \boldsymbol{el}(\Pi^S x : A.\ \tau)} \qquad \frac{\Gamma \vdash \mathrm{M} : \boldsymbol{el}(\Pi^S x : A.\ \tau) \qquad \Gamma \vdash \mathrm{N} : A}{\Gamma \vdash \mathrm{M}\ \mathrm{N} : \boldsymbol{el}(\tau[N/x])}$$

$$\frac{\Gamma \vdash \mathrm{M} : A \qquad \Gamma \vdash \mathrm{N} : B[\mathrm{M}/x]}{\Gamma \vdash (\mathrm{M},\mathrm{N})^T : \Sigma^T x : A.\ B} \qquad \frac{\Gamma \vdash \mathrm{M} : A \qquad \Gamma \vdash \mathrm{N} : \boldsymbol{el}(\tau[\mathrm{M}/x])}{\Gamma \vdash (\mathrm{M},\mathrm{N})^S : \boldsymbol{el}(\Sigma^S x : A.\ \tau)}$$

$$\frac{\Gamma, z : \boldsymbol{el}(\Sigma^S x : A.\tau) \vdash \sigma : \boldsymbol{set} \qquad \Gamma, x : A, y : \tau \vdash \mathrm{M} : \boldsymbol{el}(\sigma[(x,y)^S/z])}{\Gamma, z : \boldsymbol{el}(\Sigma^S x : A.\ \tau) \vdash \boldsymbol{unpack}\ z\ \boldsymbol{as}\ (x,y)\ \boldsymbol{in}\ \mathrm{M} : \boldsymbol{el}(\sigma)}$$

$$\frac{\Gamma \vdash \mathrm{M} : \Sigma^T x : A.B}{\Gamma \vdash \boldsymbol{fst}\ \mathrm{M} : A} \qquad \frac{\Gamma \vdash \mathrm{M} : \Sigma^T x : A.B}{\Gamma \vdash \boldsymbol{snd}\ \mathrm{M} : B[\boldsymbol{fst}\ \mathrm{M}/x]}$$

$$\frac{\Gamma \vdash \mathrm{M} : A}{\Gamma \vdash \boldsymbol{inl}_{A,B}(\mathrm{M}) : A + B} \qquad \frac{\Gamma \vdash \mathrm{M} : B}{\Gamma \vdash \boldsymbol{inr}_{A,B}(\mathrm{M}) : A + B}$$

$$\frac{\Gamma \vdash \mathrm{M} : A + B \qquad \Gamma, x : A \vdash \mathrm{N}_1 : C \qquad \Gamma, x : B \vdash \mathrm{N}_2 : C}{\Gamma \vdash \boldsymbol{case}_{A,B}\ \mathrm{M}\ \boldsymbol{in}\ \boldsymbol{inl}(x) \Rightarrow \mathrm{N}_1 \mid \boldsymbol{inr}(x) \Rightarrow \mathrm{N}_2 : C}$$

A.9 W-types

$$\frac{\Gamma \vdash A : \boldsymbol{type} \qquad \Gamma, x : A \vdash B : \boldsymbol{type}}{\Gamma \vdash \mathcal{W}^T x : A.B : \boldsymbol{type}} \qquad \frac{\Gamma \vdash \tau : \boldsymbol{set} \qquad \Gamma, x : \boldsymbol{el}(\tau) \vdash \sigma : \boldsymbol{set}}{\Gamma \vdash \mathcal{W}^S x : \tau.\sigma : \boldsymbol{set}}$$

$$\frac{\Gamma \vdash \mathrm{M} : \Sigma^T x : A.\ (B \to \mathcal{W} x : A.B)}{\Gamma \vdash sup^T_{x:A.B}(\mathrm{M}) : \mathcal{W}^T x : A.B} \qquad \frac{\Gamma \vdash \mathrm{M} : \Sigma^T x : \boldsymbol{el}(\tau).\ (\boldsymbol{el}(\sigma) \to \boldsymbol{el}(\mathcal{W} x : \tau.\sigma))}{\Gamma \vdash sup^S_{x:\tau.\sigma}(\mathrm{M}) : \boldsymbol{el}(\mathcal{W}^S x : \tau.\sigma)}$$

$$\frac{\begin{array}{c}\Gamma \vdash A : \boldsymbol{type} \qquad \Gamma, x : A \vdash B : \boldsymbol{type} \qquad \Gamma \vdash X : \boldsymbol{type} \\ \Gamma \vdash \mathrm{M} : (\Sigma^T x : A.\ (B \to X)) \to X\end{array}}{\Gamma \vdash fold^T_{x:A.B}(\mathrm{M}) : \mathcal{W}^T x : A.B \to X}$$

$$\frac{\begin{array}{c}\Gamma \vdash \tau : \boldsymbol{set} \qquad \Gamma, x : \boldsymbol{el}(\tau) \vdash \sigma : \boldsymbol{set} \qquad \Gamma \vdash X : \boldsymbol{type} \\ \Gamma \vdash \mathrm{M} : (\Sigma^T x : \boldsymbol{el}(\tau).\ (\boldsymbol{el}(\sigma) \to X)) \to X\end{array}}{\Gamma \vdash fold^S_{x:\tau.\sigma}(\mathrm{M}) : \boldsymbol{el}(\mathcal{W}^S x : \tau.\sigma) \to X}$$

$$\frac{\begin{array}{c}\Gamma \vdash A : \boldsymbol{type} \qquad \Gamma, x : A \vdash B : \boldsymbol{type} \qquad \Gamma \vdash X : \boldsymbol{type} \\ \Gamma \vdash \mathrm{N}_1 : (\Sigma^T x : A.\ (B \to X)) \to X \qquad \Gamma \vdash \mathrm{N}_2 : (\mathcal{W}^T x : A.B) \to X \\ C \equiv (\Sigma^T x : A.\ (B \to \mathcal{W}^T x : A.B))\end{array}}{\Gamma \vdash (\forall y : C.\ \mathrm{N}_2(sup^T_{x:A.B}(y)) = N_1(\pi_1(y), fold^T_{x:A:B}(N_1) \circ \pi_2(y))) \Rightarrow \mathrm{N}_2 = fold^T_{x:A.B}(\mathrm{N}_1)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \tau : \boldsymbol{set} \qquad \Gamma, x : \boldsymbol{el}(\tau) \vdash \sigma : \boldsymbol{set} \qquad \Gamma \vdash X : \boldsymbol{type} \\ \Gamma \vdash \mathrm{N}_1 : (\Sigma^T x : \boldsymbol{el}(\tau).\ (\boldsymbol{el}(\sigma) \to X)) \to X \qquad \Gamma \vdash \mathrm{N}_2 : (\mathcal{W}^S x : \tau.\sigma) \to X \\ C \equiv (\Sigma^T x : \boldsymbol{el}(\tau).\ (\boldsymbol{el}(\sigma) \to \mathcal{W}^S x : \tau.\sigma))\end{array}}{\Gamma \vdash (\forall y : C.\ \mathrm{N}_2(sup^S_{x:\tau.\sigma}(y)) = N_1(\pi_1(y), fold^S_{x:\tau.\sigma}(N_1) \circ \pi_2(y))) \Rightarrow \mathrm{N}_2 = fold^S_{x:\tau.\sigma}(\mathrm{N}_1)}$$

A.10 Heaps

For each term M : A below, iHTT contains a typing rule,

$$\frac{}{\Gamma \vdash \mathrm{M} : A}$$

$$\begin{aligned}
&\boldsymbol{empty} : \boldsymbol{heap} \\
&\quad \boldsymbol{upd} : \Pi^T \tau : \boldsymbol{set}.\ \boldsymbol{heap} \to \boldsymbol{el}(\boldsymbol{nat}) \to \boldsymbol{el}(\tau) \to \boldsymbol{heap} \\
&\quad \boldsymbol{free} : \boldsymbol{heap} \to \boldsymbol{el}(\boldsymbol{nat}) \to \boldsymbol{heap} \\
&\quad \boldsymbol{max} : \boldsymbol{heap} \to \boldsymbol{el}(\boldsymbol{nat}) \\
&\quad \boldsymbol{dom} : \boldsymbol{heap} \to \boldsymbol{el}(\boldsymbol{nat}) \to \boldsymbol{bool} \\
&\quad \boldsymbol{peek} : \boldsymbol{heap} \to \boldsymbol{el}(\boldsymbol{nat}) \to \boldsymbol{el}(1) + (\Sigma^T \tau : \boldsymbol{set}.\ \boldsymbol{el}(\tau))
\end{aligned}$$

A.11 Heap axioms

For each propostion P below, iHTT contains a typing rule,

$$\frac{}{\Gamma \vdash \bullet_{\mathrm{P}} : \boldsymbol{el}(\boldsymbol{prf}(\mathrm{P}))}$$

$$\begin{aligned}
&\boldsymbol{max}\ \boldsymbol{empty} =_{\boldsymbol{el}(\boldsymbol{nat})} \boldsymbol{zero} \\
&\forall \tau : \boldsymbol{set}.\ \forall h : \boldsymbol{heap}.\ \forall n : \boldsymbol{el}(\boldsymbol{nat}).\ \forall v : \boldsymbol{el}(\tau). \\
&\qquad \boldsymbol{max}\ (h[n \mapsto_\tau v]) =_{\boldsymbol{el}(\boldsymbol{nat})} \boldsymbol{if}\ n < \boldsymbol{max}\ h\ \boldsymbol{then}\ \boldsymbol{max}\ h\ \boldsymbol{else}\ \ n
\end{aligned}$$

$$\begin{aligned}
&\forall n : \boldsymbol{el}(\boldsymbol{nat}).\ \boldsymbol{peek}\ \boldsymbol{empty}\ n = \boldsymbol{inl}\ () \\
&\forall \tau : \boldsymbol{set}.\ \forall h : \boldsymbol{heap}.\ \forall n : \boldsymbol{el}(\boldsymbol{nat}).\ \forall v : \boldsymbol{el}(\tau).\ \forall m : \boldsymbol{el}(\boldsymbol{nat}). \\
&\qquad \boldsymbol{peek}\ (h[n \mapsto_\tau v])\ m = \boldsymbol{if}\ n == m\ \boldsymbol{then}\ \boldsymbol{inr}\ Pair_{\tau : \boldsymbol{set}.\boldsymbol{el}(\tau)}(\tau, v)\ \boldsymbol{else}\ \boldsymbol{peek}\ h\ m \\
&\forall h : \boldsymbol{heap}.\ \forall n, m : \boldsymbol{el}(\boldsymbol{nat}). \\
&\qquad \boldsymbol{peek}\ (h \setminus n)\ m = \boldsymbol{if}\ n == m\ \boldsymbol{then}\ \boldsymbol{inl}\ ()\ \boldsymbol{else}\ \boldsymbol{peek}\ h\ m
\end{aligned}$$

$$\begin{aligned}
&\forall n : \boldsymbol{el}(\boldsymbol{nat}).\ \boldsymbol{dom}\ \boldsymbol{empty}\ n = \boldsymbol{false} \\
&\forall \tau : \boldsymbol{set}.\ \forall h : \boldsymbol{heap}.\ \forall n : \boldsymbol{el}(\boldsymbol{nat}).\ \forall v : \boldsymbol{el}(\tau).\ \forall m : \boldsymbol{el}(\boldsymbol{nat}). \\
&\qquad \boldsymbol{dom}\ (h[n \mapsto_\tau v])\ m = \boldsymbol{if}\ n == m\ \boldsymbol{then}\ \boldsymbol{true}\ \boldsymbol{else}\ \boldsymbol{dom}\ h\ m \\
&\forall h : \boldsymbol{heap}.\ \forall nm : \boldsymbol{el}(\boldsymbol{nat}). \\
&\qquad \boldsymbol{dom}\ (h \setminus n)\ m = \boldsymbol{if}\ n == m\ \boldsymbol{then}\ \boldsymbol{false}\ \boldsymbol{else}\ \boldsymbol{dom}\ h\ m
\end{aligned}$$

$\forall h : \textbf{\textit{heap}}.\ \forall \alpha, \beta : \textbf{\textit{set}}.\ \forall n, m : \textbf{\textit{el}}(\textbf{\textit{nat}}).\ \forall a : \textbf{\textit{el}}(\alpha).\ \forall b : \textbf{\textit{el}}(\beta).$
$h[n \mapsto_\alpha a][m \mapsto_\beta b] = \textbf{\textit{if}}\ n == m\ \textbf{\textit{then}}\ h[m \mapsto_\beta b]\ \textbf{\textit{else}}\ h[m \mapsto_\beta b][n \mapsto_\alpha a]$

$\forall h_1, h_2 : \textbf{\textit{heap}}.\ (\forall n : \textbf{\textit{el}}(\textbf{\textit{nat}}).\ \textbf{\textit{peek}}\ h_1\ n = \textbf{\textit{peek}}\ h_2\ n) \Rightarrow h_1 =_{\textbf{\textit{heap}}} h_2$
$\forall \mathrm{P} : \textbf{\textit{heap}} \to \textbf{\textit{prop}}.$
$(\forall h : \textbf{\textit{heap}}.\ \forall \alpha : \textbf{\textit{set}}.\ \forall n : \textbf{\textit{el}}(\textbf{\textit{nat}}).\ \forall v : \textbf{\textit{el}}(\alpha).\ \mathrm{P}\ h \Rightarrow \textbf{\textit{max}}\ h < n \Rightarrow \mathrm{P}\ (h[n \mapsto_\alpha v])) \Rightarrow$
$\mathrm{P}\ \textbf{\textit{empty}} \Rightarrow \forall h : \textbf{\textit{heap}}.\ \mathrm{P}\ h$

where $h[n \mapsto_\tau v]$ is shorthand for $\textbf{\textit{upd}}\ \tau\ h\ n\ v$ and $h \backslash n$ is shorthand for $\textbf{\textit{free}}\ h\ n$.

A.12 Computations

For each term $\mathrm{M} : \tau$ below, iHTT contains a typing rule,

$$\frac{}{\Gamma \vdash \mathrm{M} : \textbf{\textit{el}}(\tau)}$$

$\textbf{\textit{st}} : \Pi^S \tau : \textbf{\textit{set}}.\ \mathit{spec}\ \tau \to \textbf{\textit{set}}$
$\textbf{\textit{ret}} : \Pi^S \tau : \textbf{\textit{set}}.\ \Pi^S v : \textbf{\textit{el}}(\tau).\ \{\lambda\_.\top\}\tau\{\lambda r, h_i, h_t.\ h_i = h_t \wedge r = v\}$
$\textbf{\textit{read}} : \Pi^S \tau : \textbf{\textit{set}}.\ \Pi^S l : \textbf{\textit{el}}(\textbf{\textit{nat}}).$
$\{\lambda h.\ \textbf{\textit{dom}}\ h\ l = \textbf{\textit{true}}\}\tau\{\lambda r, h_i, h_t.\ h_i = h_t \wedge \textbf{\textit{peek}}\ h_t\ l = \mathit{inr}(\tau, r)\}$
$\textbf{\textit{write}} : \Pi^S \tau : \textbf{\textit{set}}.\ \Pi^S l : \textbf{\textit{el}}(\textbf{\textit{nat}}).$
$\Pi^S v : \textbf{\textit{el}}(\tau).\ \{\lambda h.\ \textbf{\textit{dom}}\ h\ n = \textbf{\textit{true}}\}\tau\{\lambda r, h_i, h_t.\ h_t = h_i[l \mapsto_\tau v]\}$
$\textbf{\textit{alloc}} : \Pi^S \tau : \textbf{\textit{set}}.\ \Pi^S v : \textbf{\textit{el}}(\tau).$
$\{\lambda\_.\top\}\textbf{\textit{nat}}\{\lambda r, h_i, h_t.\ h_t = h_i[r \mapsto_\tau v] \wedge r \neq 0 \wedge \textbf{\textit{dom}}\ h_i\ r = \textbf{\textit{false}}\}$
$\textbf{\textit{dealloc}} : \Pi^S l : \textbf{\textit{el}}(\textbf{\textit{nat}}).\ \{\lambda h.\ \textbf{\textit{dom}}\ h\ l = \textbf{\textit{true}}\}1\{\lambda\_, h_i, h_t.\ h_t = h_i \setminus l\}$

$\textbf{\textit{do}} : \Pi^S \tau : \textbf{\textit{set}}.\ \Pi^S s_1, s_2 : \mathit{spec}\ \tau.\ \Pi^S \textbf{\textit{el}}(\textbf{\textit{prf}}(\mathit{conseq}\ s_1, s_2)).\ \textbf{\textit{el}}(\textbf{\textit{st}}\ \tau\ s_1) \to \textbf{\textit{el}}(\textbf{\textit{st}}\ \tau\ s_2)$
$\textbf{\textit{bind}} : \Pi^S \tau, \sigma : \textbf{\textit{set}}.\ \Pi^S s_1 : \mathit{spec}\ \tau.\ \Pi^S s_2 : \textbf{\textit{el}}(\tau) \to \mathit{spec}\ \sigma.$
$\textbf{\textit{el}}(\textbf{\textit{st}}\ \tau\ s_1) \to (\Pi^S v : \tau.\ \textbf{\textit{st}}_\sigma\ (s_2\ v)) \to$
$\{\lambda h_i.\ \pi_1(s_1)\ h_i \wedge \forall x h_t.\ \pi_2(s_1)\ x\ h_i\ h_t \Rightarrow \pi_1(s_2\ x)\ h_m\}$
σ
$\{\lambda r, h_i, h_t.\ \exists x, h.\ \pi_2(s_1)\ x\ h_i\ h_t \wedge \pi_2(s_2)\ r\ h\ h_t\}$
$\textbf{\textit{fix}}_A : \Pi^S \tau : \textbf{\textit{set}}.\ \Pi^S s : \mathit{spec}\ \tau.$
$((\Pi^S x : A.\ \textbf{\textit{st}}\ \tau\ s) \to (\Pi^S x : A.\ \textbf{\textit{st}}\ \tau\ s)) \to \Pi^S x : A.\ \textbf{\textit{st}}\ \tau\ s$

where

$$\mathit{spec} = \lambda \tau : \textbf{\textit{set}}.\ (\textbf{\textit{heap}} \to \textbf{\textit{prop}}) \times (\tau \to \textbf{\textit{heap}} \to \textbf{\textit{heap}} \to \textbf{\textit{prop}})$$

and $\{\mathrm{P}\}\tau\{\mathrm{Q}\}$ is shorthand for $\textbf{\textit{st}}\ \tau\ (\mathrm{P}, \mathrm{Q})^T$.

A.13 Term definitional equality $\boxed{\Gamma \vdash \mathrm{M} = \mathrm{N} : A}$

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash M = M : A} \qquad \frac{\Gamma \vdash M = N : A}{\Gamma \vdash N = M : A} \qquad \frac{\Gamma \vdash M_1 = M_2 : A \qquad \Gamma \vdash M_2 = M_3 : A}{\Gamma \vdash M_1 = M_3 : A}$$

$$\frac{\Gamma, x : A \vdash \mathrm{M} : B \qquad \Gamma \vdash \mathrm{N} : A \qquad U \in \{S, T, P\}}{\Gamma \vdash (\lambda^U x : A.\ \mathrm{M})\ \mathrm{N} = \mathrm{M}[\mathrm{N}/x] : B[\mathrm{M}/x]}$$

$$\frac{\Gamma \vdash \mathrm{M} : A \qquad \Gamma \vdash \mathrm{N} : B[\mathrm{M}/x]}{\Gamma \vdash \textbf{\textit{fst}}\ (\mathrm{M}, \mathrm{N})^T = \mathrm{M} : A} \qquad \frac{\Gamma \vdash \mathrm{M} : A \qquad \Gamma \vdash \mathrm{N} : B[\mathrm{M}/x]}{\Gamma \vdash \textbf{\textit{snd}}\ (\mathrm{M}, \mathrm{N})^T = \mathrm{N} : B[\mathrm{M}/x]}$$

$$\frac{\Gamma \vdash \mathrm{M} : A \qquad \Gamma \vdash \mathrm{N} : \textbf{\textit{el}}(\tau[\mathrm{M}/x]) \qquad \Gamma, x : A, y : \tau \vdash \mathrm{K} : \textbf{\textit{el}}(\sigma)}{\Gamma \vdash \textbf{\textit{unpack}}\ (\mathrm{M}, \mathrm{N})^S\ \textbf{\textit{as}}\ (x, y)\ \textbf{\textit{in}}\ K = K[\mathrm{M}/x, \mathrm{N}/y] : \textbf{\textit{el}}(\sigma)}$$

$$\frac{\Gamma \vdash \mathrm{M} : A \qquad \Gamma, x : A \vdash \mathrm{N}_1 : C \qquad \Gamma, x : B \vdash \mathrm{N}_2 : C}{\Gamma \vdash \textbf{\textit{case}}\ \textbf{\textit{inl}}(\mathrm{M})\ \textbf{\textit{in}}\ \textbf{\textit{inl}}(x) \Rightarrow \mathrm{N}_1 \mid \textbf{\textit{inr}}(x) \Rightarrow \mathrm{N}_2 = \mathrm{N}_1[\mathrm{M}/X] : C}$$

$$\frac{\Gamma \vdash \mathrm{M} : B \qquad \Gamma, x : A \vdash \mathrm{N}_1 : C \qquad \Gamma, x : B \vdash \mathrm{N}_2 : C}{\Gamma \vdash \textbf{\textit{case}}\ \textbf{\textit{inr}}(\mathrm{M})\ \textbf{\textit{in}}\ \textbf{\textit{inl}}(x) \Rightarrow \mathrm{N}_1 \mid \textbf{\textit{inr}}(x) \Rightarrow \mathrm{N}_2 = \mathrm{N}_2[\mathrm{M}/X] : C}$$

$$\frac{\Gamma \vdash \mathrm{M} : \Sigma^T x : A.\ (B \to \mathcal{W}^T x : A.B) \qquad \Gamma \vdash \mathrm{N} : (\Sigma^T x : A.\ (B \to X)) \to X}{\Gamma \vdash fold^T_{x:A.B}(\mathrm{N})(sup^T_{x:A.B}(\mathrm{M})) = \mathrm{N}(\pi_1(\mathrm{M}), fold^T_{x:A.B}(\mathrm{N}) \circ \pi_2(\mathrm{M})) : X}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathrm{M} : \Sigma^T x : \textbf{\textit{el}}(\tau).\ (\textbf{\textit{el}}(\sigma) \to \textbf{\textit{el}}(\mathcal{W}^S a : \tau.\sigma)) \\ \Gamma \vdash \mathrm{N} : (\Sigma^T x : \textbf{\textit{el}}(\tau).\ (\textbf{\textit{el}}(\sigma) \to X)) \to X\end{array}}{\Gamma \vdash fold^S_{x:\tau.\sigma}(\mathrm{N})(sup^S_{x:\tau.\sigma}(\mathrm{M})) = \mathrm{N}(\pi_1(\mathrm{M}), fold^S_{x:\tau.\sigma}(\mathrm{N}) \circ \pi_2(\mathrm{M})) : X}$$

A.14 Type definitional equality $\boxed{\Gamma \vdash A = B : \textbf{\textit{type}}}$

$$\frac{\Gamma \vdash A : \textbf{\textit{type}}}{\Gamma \vdash A = A : \textbf{\textit{type}}} \qquad \frac{\Gamma \vdash A = B : \textbf{\textit{type}}}{\Gamma \vdash B = A : \textbf{\textit{type}}}$$

$$\frac{\Gamma \vdash A = B : \textbf{\textit{type}} \qquad \Gamma \vdash B = C : \textbf{\textit{type}}}{\Gamma \vdash A = C : \textbf{\textit{type}}}$$

$$\frac{\Gamma \vdash A = A' : \boldsymbol{type} \qquad \Gamma, x : A \vdash B = B' : \boldsymbol{type}}{\Gamma \vdash \Pi^T x : A.B = \Pi^T x : A'.B' : \boldsymbol{type}}$$

$$\frac{\Gamma \vdash A = A' : \boldsymbol{type} \qquad \Gamma, x : A \vdash B = B' : \boldsymbol{type}}{\Gamma \vdash \Sigma^T x : A.B = \Sigma^T x : A'.B' : \boldsymbol{type}}$$

$$\frac{\Gamma \vdash A = A' : \boldsymbol{type} \qquad \Gamma, x : A \vdash B = B' : \boldsymbol{type}}{\Gamma \vdash \mathcal{W}^T x : A.B = \mathcal{W}^T x : A'.B'}$$

$$\frac{\Gamma \vdash A = A' : \boldsymbol{type} \qquad \Gamma \vdash B = B' : \boldsymbol{type}}{\Gamma \vdash A + B = A' + B' : \boldsymbol{type}}$$

$$\frac{\Gamma \vdash \tau = \sigma : \boldsymbol{set}}{\Gamma \vdash \boldsymbol{el}(\tau) = \boldsymbol{el}(\sigma) : \boldsymbol{type}}$$

B Interpretation

This appendix contains the concrete interpretation of the underlying dependent type theory. The interpretation of terms of small types is defined in terms of a realizer, which uniquely determines a morphism in the appropriate fibre category.

$$
\begin{aligned}
[\![-]\!]^{\mathrm{Ctx}} &: \mathrm{Ctx} \rightharpoonup \mathit{obj}(\mathrm{Asm}(\mathbb{V})) \\
[\![-]\!]^{\mathrm{Type}} &: \mathrm{Ctx} \times \mathrm{Term} \rightharpoonup \mathit{obj}(\mathrm{UFam}(\mathrm{Asm}(\mathbb{V}))) \\
[\![-]\!]^{\mathrm{Term}} &: \mathrm{Ctx} \times \mathrm{Term} \rightharpoonup \mathit{hom}(\mathrm{UFam}(\mathrm{Asm}(\mathbb{V})))
\end{aligned}
$$

B.1 Contexts

$\boxed{\boldsymbol{\Gamma}}$

$$
\begin{aligned}
[\![\varepsilon]\!]^{\mathrm{Ctx}} &= 1 \\
[\![\Gamma, x : A]\!]^{\mathrm{Ctx}} &= (\Sigma_{i \in I} X_i, (i, x) \mapsto \{(a, b) \mid a \in E(i) \wedge b \in E_{X_i}(x)\})
\end{aligned}
$$

for $(I, E) = [\![\Gamma]\!]^{\mathrm{Ctx}}$ and $(X_i, E_{X_i}) = [\![\Gamma \vdash A]\!]^{\mathrm{Type}}$.

B.2 Types

$\boxed{\boldsymbol{\Gamma \vdash A : \mathit{type}}}$

$$
\begin{aligned}
[\![\Gamma \vdash \boldsymbol{\mathit{set}}]\!]_i^{\mathrm{Type}} &= \nabla(\mathrm{CMPer}(\mathbb{V})) \\
[\![\Gamma \vdash \boldsymbol{\mathit{prop}}]\!]_i^{\mathrm{Type}} &= \nabla(2) \\
[\![\Gamma \vdash \boldsymbol{\mathit{el}}(\tau)]\!]_i^{\mathrm{Type}} &= \mathcal{I}_{el}([\![\Gamma \vdash \tau]\!]_i^{\mathrm{Term}}) \\
[\![\Gamma \vdash \Pi^T x : A.B]\!]_i^{\mathrm{Type}} &= \left(\left\{ f \in \prod_{i \in X_i} Y_{(i,x)} \mid E_{\prod_i}(f) \neq \emptyset \right\}, E_{\prod_i} \right) \\
[\![\Gamma \vdash \Sigma^T x : A.B]\!]_i^{\mathrm{Type}} &= \left(\coprod_{x \in X_i} Y_{(i,x)}, E_{\coprod_i} \right)
\end{aligned}
$$

for $(I, E) = [\![\Gamma]\!]^{\mathrm{Ctx}}$, $(X_i, E_{X_i}) = [\![\Gamma \vdash A]\!]^{\mathrm{Type}}$, and $(Y_{(i,x)}, E_{Y_{(i,x)}}) = [\![\Gamma, x : A \vdash B]\!]^{\mathrm{Type}}$, where

$$
\begin{aligned}
E_{\prod_i}(f) &= \{in_{\rightarrow}(\alpha) \mid \forall x \in X_i.\ \forall e_x \in E_{X_i}(x).\ \alpha \cdot e_x \in E_{Y_{(i,x)}}(f(x))\} \\
E_{\coprod_i}(x, y) &= \{in_{\times}(\alpha, \beta) \mid \alpha \in E_{X_i}(x) \wedge \beta \in E_{Y_{(i,x)}}(y)\}
\end{aligned}
$$

B.3 Small types

$\boxed{\Gamma \vdash \tau : \mathit{set}}$

$$[\![\Gamma \vdash \mathit{prf}(\mathrm{P})]\!]_i^{\mathrm{Term}} = \begin{cases} \mathbb{V} \times \mathbb{V} & \text{if } [\![\Gamma \vdash \mathrm{P}]\!]_i^{\mathrm{Term}} = \top \\ \emptyset & \text{if } [\![\Gamma \vdash \mathrm{P}]\!]_i^{\mathrm{Term}} = \bot \end{cases}$$

$$[\![\Gamma \vdash \Pi^S x : A.\tau]\!]_i^{\mathrm{Term}} = \left\{(a,b) \mid \forall x \in X_i.\ \forall e_1, e_2 \in E_{X_i}(x).\ (a \cdot e_1, b \cdot e_2) \in R_{(i,x)}\right\}$$

$$[\![\Gamma \vdash \Sigma^S x : A.\tau]\!]_i^{\mathrm{Term}} = \overline{\left\{(a,b) \mid \exists x, y \in \coprod_{x \in X_i} [R_{(i,x)}].\ a \in E_{\coprod_i}(x) \wedge b \in E_{\coprod_i}(y) \wedge x \sim_i y\right\}}$$

for $(I, E) = [\![\Gamma]\!]^{\mathrm{Ctx}}$, $(X_i, E_{X_i}) = [\![\Gamma \vdash A]\!]^{\mathrm{Type}}$, and $R_{(i,x)} = [\![\Gamma, x : A \vdash \tau]\!]^{\mathrm{Term}}$, where

$$E_{\coprod_i}(x,y) = \{in_\times(\alpha,\beta) \mid \alpha \in E_{X_i}(x) \wedge \beta \in y\}$$
$$x \smile_i y \quad \text{iff} \quad E_{\coprod_i}(x) \cap E_{\coprod_i}(y) \neq \emptyset$$

$\sim_i$ is the transitive closure of $\smile_i$, and $\overline{(-)}$ denotes monotone completion.

B.4 Terms

$\boxed{\Gamma \vdash M : A}$

$$\begin{aligned}
[\![\Gamma, x : A \vdash x]\!]_{(i,x)}^{\mathrm{Term}} &= x \\
[\![\Gamma \vdash \mathit{Abs}^T_{x:A.B}(\mathrm{M})]\!]_i^{\mathrm{Term}} &= x \in X_i \mapsto [\![\Gamma, x : A \vdash \mathrm{M}]\!]_{(i,x)}^{\mathrm{Term}} \\
[\![\Gamma \vdash \mathit{App}^T_{x:A.B}(\mathrm{M},\mathrm{N})]\!]_i^{\mathrm{Term}} &= [\![\Gamma \vdash \mathrm{M}]\!]_i^{\mathrm{Term}}([\![\Gamma \vdash \mathrm{N}]\!]_i^{\mathrm{Term}}) \\
[\![\Gamma \vdash \mathit{Pair}^T_{x:A.B}(\mathrm{M},\mathrm{N})]\!]_i^{\mathrm{Term}} &= ([\![\Gamma \vdash \mathrm{M}]\!]_i^{\mathrm{Term}}, [\![\Gamma, x : A \vdash \mathrm{N}]\!]^{\mathrm{Term}}_{(i,[\![\Gamma \vdash \mathrm{M}]\!]_i^{\mathrm{Term}})}) \\
[\![\Gamma \vdash \pi^1_{x:A.B}(\mathrm{M})]\!]_i^{\mathrm{Term}} &= \pi_1([\![\Gamma \vdash \mathrm{M}]\!]_i^{\mathrm{Term}}) \\
[\![\Gamma \vdash \pi^2_{x:A.B}(\mathrm{M})]\!]_i^{\mathrm{Term}} &= \pi_2([\![\Gamma \vdash \mathrm{M}]\!]_i^{\mathrm{Term}})
\end{aligned}$$

$$\begin{aligned}
[\![\Gamma \vdash \mathit{Abs}^S_{x:A.\tau}(\mathrm{M})]\!]^{\mathrm{Term}} &= in_\to(\lambda e. in_\to(\lambda x.\ m(in_\times(e,x)))) \\
[\![\Gamma \vdash \mathit{App}^S_{x:A.\tau}(\mathrm{M},\mathrm{N})]\!]^{\mathrm{Term}} &= in_\to\left(\lambda e.\ \begin{array}{c}\textbf{\textit{case}}\ m(e)\ \textbf{\textit{of}} \\ in_\to(f) \Rightarrow f(n(e)) \\ \text{otherwise} \Rightarrow \bot_{\mathbb{V}}\end{array}\right) \\
[\![\Gamma \vdash \mathit{Pair}^S_{x:A.\tau}(\mathrm{M},\mathrm{N})]\!]^{\mathrm{Term}} &= in_\to(\lambda e.\ in_\times(m(e), n(in_\times(e, m(e))))) \\
[\![\Gamma \vdash \textbf{\textit{unpack}}\ \mathrm{M}\ \textbf{\textit{as}}\ (x,y)\ \textbf{\textit{in}}\ \mathrm{N}]\!]^{\mathrm{Term}} &= in_\to\left(\lambda e.\ \begin{array}{c}\textbf{\textit{case}}\ m(e)\ \textbf{\textit{of}} \\ in_\times(a,b) \Rightarrow n(in_\times(in_\times(e,a),b)) \\ \text{otherwise} \Rightarrow \bot_{\mathbb{V}}\end{array}\right)
\end{aligned}$$

where $m = [\![\Gamma \vdash M]\!]^{\mathrm{Term}}$ and $n = [\![\Gamma \vdash N]\!]^{\mathrm{Term}}$.

B.5 Base types and terms

$$
\begin{aligned}
[\![\Gamma \vdash 1]\!]_i^{\mathrm{Term}} &= \{(in_1(*), in_1(*))\} \\
[\![\Gamma \vdash ()]\!]^{\mathrm{Term}} &= in_{\to}(\lambda e.\ in_1(*)) \\
[\![\Gamma \vdash \textbf{\textit{bool}}]\!]_i^{\mathrm{Term}} &= \{(in_{\mathbb{N}}(1), in_{\mathbb{N}}(1)), (in_{\mathbb{N}}(2), in_{\mathbb{N}}(2))\} \\
[\![\Gamma \vdash \textbf{\textit{nat}}]\!]_i^{\mathrm{Term}} &= \{(in_{\mathbb{N}}(n), in_{\mathbb{N}}(n)) \mid n \in \mathbb{N}\} \\
[\![\Gamma \vdash \textbf{\textit{if}}\ \mathrm{M}\ \textbf{\textit{then}}\ \mathrm{N}_1\ \textbf{\textit{else}}\ \mathrm{N}_2]\!]_i^{\mathrm{Term}} &= \begin{cases} [\![\Gamma \vdash \mathrm{N}_1]\!]_i^{\mathrm{Term}} & \text{if } [\![\Gamma \vdash \mathrm{M}]\!]_i^{\mathrm{Term}} = [in_{\mathbb{N}}(1)] \\ [\![\Gamma \vdash \mathrm{N}_2]\!]_i^{\mathrm{Term}} & \text{if } [\![\Gamma \vdash \mathrm{M}]\!]_i^{\mathrm{Term}} = [in_{\mathbb{N}}(2)] \end{cases} \\
[\![\Gamma \vdash \textbf{\textit{true}}]\!]^{\mathrm{Term}} &= in_{\to}(\lambda e.\ in_{\mathbb{N}}(1)) \\
[\![\Gamma \vdash \textbf{\textit{false}}]\!]^{\mathrm{Term}} &= in_{\to}(\lambda e.\ in_{\mathbb{N}}(2)) \\
[\![\Gamma \vdash \textbf{\textit{zero}}]\!]^{\mathrm{Term}} &= in_{\to}(\lambda e.\ in_{\mathbb{N}}(0)) \\
[\![\Gamma \vdash \textbf{\textit{succ}}\ \mathrm{M}]\!]^{\mathrm{Term}} &= in_{\to} \left(\begin{array}{lll} & \textbf{\textit{case}}\ m(e)\ \textbf{\textit{of}} & \\ \lambda e. & in_{\mathbb{N}}(n) & \Rightarrow in_{\mathbb{N}}(n+1) \\ & \text{otherwise} & \Rightarrow \bot_{\mathbb{V}} \end{array} \right)
\end{aligned}
$$

C Proofs

C.1 Recursion

Definition 10. *Given* $X \in \mathrm{Asm}(\mathbb{V})$ *and* $R \in \mathrm{UFam}(\mathrm{CMPer}(\mathbb{V}))_X$*, let* $\Pi_X(R)$ *denote the complete monotone PER,*

$$\Pi_X(R) = \{(\alpha, \beta) \mid \forall x \in |X|.\ \forall e_x \in E_X(x).\ (\alpha \cdot e_x, \beta \cdot e_x) \in R_x\}$$

Definition 11. *Let* $u : \mathbb{V} \to (\mathbb{V} \to T(\mathbb{V})) \to (\mathbb{V} \to T(\mathbb{V}))$ *denote*

$$u(x : \mathbb{V})(y : \mathbb{V} \to T(\mathbb{V}))(z : \mathbb{V}) \stackrel{def}{=} \begin{cases} a & \text{if } x \cdot in_{\to}(\lambda z.\ in_T(y(z))) \cdot z = in_T(a) \\ \bot & \text{otherwise} \end{cases}$$

and let lfp denote $in_{\to}(\lambda e.\ \sqcup_n in_{\to}(\lambda z.\ in_T((u(e))^n)(\bot_{\mathbb{V}\to T(\mathbb{V})})(z)))$.

Lemma 6. *Let* $X \in \mathrm{Asm}(\mathbb{V})$ *and* $R \in \mathrm{UFam}(\mathrm{AdmPer}(T(\mathbb{V})))$*, then*

$$lfp : |(\Pi_X(in_T(R)) \to \Pi_X(in_T(R))) \to \Pi_X(in_T(R))|$$

Proof. Assume $\alpha_1\ (\Pi_X(in_T(R)) \to \Pi_X(in_T(R)))\ \alpha_2$, then we need to show that

$$\begin{aligned}(lfp \cdot \alpha_1, lfp \cdot \alpha_2) = (\ &\sqcup_n in_{\to}(\lambda z.\ in_T(((u(\alpha_1))^n)(\bot_{\mathbb{V}\to T(\mathbb{V})})(z))),\\ &\sqcup_n in_{\to}(\lambda z.\ in_T(((u(\alpha_2))^n)(\bot_{\mathbb{V}\to T(\mathbb{V})})(z)))) \in \Pi_X(in_T(R))\end{aligned}$$

which follows by chain-completeness from,

$$\forall n \in \mathbb{N}.\ in_{\to}(\lambda z.\ in_T((u(\alpha_1))^n(\bot_{\mathbb{V}\to H(\mathbb{V})})(z)))\ \Pi_X(in_T(R))\ in_{\to}(\lambda z.\ in_T((u(\alpha_2))^n(\bot_{\mathbb{V}\to H(\mathbb{V})})(z)))$$

which we prove by induction on n. The base case follows from the admissibility of R. For the inductive case, assume

$$in_{\to}(\lambda z.\ in_T((u(\alpha_1))^n(\bot_{\mathbb{V}\to H(\mathbb{V})})(z)))\ \Pi_X(in_T(R))\ in_{\to}(\lambda z.\ in_T((u(\alpha_2))^n(\bot_{\mathbb{V}\to H(\mathbb{V})})(z)))$$

Assume $x \in |X|$ and $e_1, e_2 \in E_X(x)$, then it follows by the induction hypothesis and definition of $\Pi_X(in_T(R))$ that,

$$\begin{aligned}&(\alpha_1 \cdot in_{\to}(\lambda z.\ in_T((u(\alpha_1))^n(\bot_{\mathbb{V}\to H(\mathbb{V})})(z))) \cdot e_1,\\ &\ \alpha_2 \cdot in_{\to}(\lambda z.\ in_T((u(\alpha_2))^n(\bot_{\mathbb{V}\to H(\mathbb{V})})(z))) \cdot e_2) \in in_T(R)\end{aligned}$$

and thus,

$$(in_{\to}(\lambda z.\ in_T((u(\alpha_1))^n(\bot)(z))) \cdot e_1, in_{\to}(\lambda z.\ in_T((u(\alpha_2))^n(\bot)(z))) \cdot e_2) \in in_T(R)$$

as

$$\begin{aligned}in_{\to}(\lambda z.\ in_t((u(\alpha_i))^{n+1}(\bot(z)))) \cdot e_i &= in_T((u(\alpha_i))((u(\alpha_i))^n(\bot_{\mathbb{V}\to H(\mathbb{V})}))(e_i))\\ &= \alpha_i \cdot in_{\to}(\lambda z.\ in_T((u(\alpha_i))^n(\bot)(z))) \cdot e_i\end{aligned}$$

C.2 Underlying DTT

Lemma 7. *Let* $S, T \in \mathrm{Per}(\mathbb{V})$. *Then*

$$S \to T \subseteq S \to \overline{T} = \overline{S} \to \overline{T}$$

where $\overline{(-)}$ *denotes monotone completion operator:*

$$\overline{R} = \bigcap \{S \in \mathrm{CMPer}(\mathbb{V}) \mid R \subseteq S\}, \quad R \in \mathrm{Per}(\mathbb{V})$$

Proof. The $S \to T \subseteq S \to \overline{T}$ and $\overline{S} \to \overline{T} \subseteq S \to \overline{T}$ inclusions are obvious.

To show $S \to \overline{T} \subseteq \overline{S} \to \overline{T}$, assume $(\alpha_1, \alpha_2) \in S \to \overline{T}$.

Define,

$$U_i \stackrel{\text{def}}{=} \{(x, y) \mid (\alpha_i \cdot x, \alpha_i \cdot y) \in \overline{T}\}$$

U_i is clearly a PER and by assumption $S \subseteq U_i$. Furthermore, U_i is monotone and complete by continuity of α_i and monotone and completeness of $\overline{T}$. Hence, $\overline{S} \subseteq U_i$.

Define

$$U \stackrel{\text{def}}{=} \{(x, y) \in \overline{S} \mid (\alpha_1 \cdot x, \alpha_2 \cdot y) \in \overline{T}\}$$

Since $\overline{S} \subseteq U_i$, U is indeed a PER and by assumption $S \subseteq U$. Again, it follows that U is monotone and complete from the continuity of α_i , monotone and completeness of $\overline{T}$ and that $U_i \subseteq \overline{T}$. Hence, $U = \overline{S}$ and thus $(\alpha_1, \alpha_2) \in \overline{S} \to \overline{T}$.

Lemma 8. *Let* $S \in \mathrm{CMPer}(\mathbb{V})$ *and* $T : \mathbb{V}/S \to \mathrm{CMPer}(\mathbb{V})$, *then*

$$\Sigma_X(Y) = \{(in_\times(a_1, b_1), in_\times(a_2, b_2)) \mid a_1 \ S \ a_2 \wedge b_1 \ T([a_1]_S) \ b_2\} \in \mathrm{CMPer}(\mathbb{V})$$

Proof. $\Sigma_S(T)$ is clearly a PER.

To show that $\Sigma_S(T)$ is monotone, assume $(a_1, b_1), (a_2, b_2) \in |\Sigma_S(T)|$, $a_1 \leq a_2$, and $b_1 \leq b_2$. Then $a_1 \ S \ a_2$ and $b_1 \ T([a_1]_S) \ b_2$ and hence $((a_1, b_1), (a_2, b_2)) \in \Sigma_S(T)$.

To show that $\Sigma_S(T)$ is complete, assume $a_1 \leq a_2 \leq \cdots$, $b_1 \leq b_2 \leq \cdots$, and $(a_i, b_i) \in |\Sigma_S(T)|$. Then by completeness, $\bigsqcup_n a_n \in |S|$. Hence, by monotonicity, $a_i \ S \ \bigsqcup_n a_n$ for every i and thus $b_i \in |T([\bigsqcup_n a_n]_S)|$ for every i, from which it follows that $\bigsqcup_n b_n \in |T([\bigsqcup_n a_n])|$ by completeness.

C.3 Computations

Definition 12. *Let* $R \in \mathrm{CMPer}(\mathbb{V})$, *then*

$$spec(R) \stackrel{def}{=} (\mathbb{H}_t \to 2) \times ([R] \to \mathbb{H}_t \to \mathbb{H}_t \to 2)$$

Lemma 9. *Let* $R, S \in \mathrm{CMPer}(\mathbb{V})$*,* $(p_1, q_1) \in \mathit{spec}(R)$*,* $s_2 \in [R] \to \mathit{spec}(S)$*, and*

$$m \in |\Delta(\mathit{in}_T(\mathit{hoare}(R, p_1, q_1)))|$$
$$n \in |\Pi_{\mathcal{I}(R)}(x \in [R] \mapsto \mathit{in}_T(\Delta(\mathit{hoare}(S, \pi_1(s_2(x)), \pi_2(s_2(x))))))|$$

then $\mathit{bind}(m, n) \in \mathit{hoare}(S, p, q)$*, where*

$$p(h_i \in \mathbb{H}_t) = p_1(h_i) \wedge \forall x \in [R].\ \forall h_t \in \mathbb{H}_t.\ q_1(x)(h_i)(h_t) \Rightarrow \pi_1(s_2(x))(h_t)$$

and

$$q(x \in [S])(h_i \in \mathbb{H}_t)(h_t \in \mathbb{H}_t) = \exists y \in [R].\ \exists h \in \mathbb{H}_t.\ q_1(y)(h_i)(h) \wedge \pi_2(s_2(y))(x)(h)(h_t)$$

Proof. Let $w \in \mathbb{W}$ and $h \in |\sim_w|$ such that $p([h]_w)$.

By assumption, there exists an $\alpha \in \mathbb{V}$, such that $m = \mathit{in}_T(\alpha)$ and $\alpha \in \mathit{hoare}(R, p_1, p_2)$. Since $p_1([h]_w)$ we thus have that that $\alpha(h) \neq \textbf{\textit{err}}$. If $\alpha(h) = \bot$ then $\mathit{bind}(m, n)(h) = \bot \in \mathit{hoare}(S, p, q)$, trivially. Hence, the only case we have to consider is if $\alpha(h) = (v, h')$. In this case $v \in |R|$ and $h' \in \overline{M_Q}$, where

$$M_Q \stackrel{\text{def}}{=} \{h' \in H(\mathbb{V}) \mid \exists w' \in \mathbb{W}.\ q_1([v]_R)([h]_w)([h']_{w'})\}$$

Since $v \in |R|$ there exists a β such $n \cdot v = \mathit{in}_T(\beta)$ and

$$\beta \in \mathit{hoare}(S, \pi_1(s_2([v]_R)), \pi_2(s_2([v]_R))).$$

Define N as the subset of intermediate states for which the terminal state of β satisfies the post-condition:

$$N \stackrel{\text{def}}{=} \{h' \in H(\mathbb{V}) \mid \beta(h') = \bot \vee \exists v', h''.\ \beta(h') = (v', h'') \wedge v' \in |S| \wedge h'' \in \overline{M_B(v')}\}$$
$$M_B(v') \stackrel{\text{def}}{=} \{h'' \in H(\mathbb{V}) \mid \exists w'' \in \mathbb{W}.\ \pi_2(s_2([v]_R))([v']_S)([h]_w)([h'']_{w''})\}\}$$

It is thus sufficient to show that $\overline{M_Q} \subseteq N$. To that end, we show that N is chain-complete and that $M_Q \subseteq N$. Chain-completeness of N follows from the continuity of β and monotonicity of S.

To show that $M_Q \subseteq N$, assume $w' \in \mathbb{W}$ and $h' \in |\sim_{w'}|$ such that $q_1([v]_R)([h]_w)([h']_{w'})$. Since $p([h]_w)$, it follows that $\pi_1(s_2([v]_T))([h']_{w'})$ and hence $\beta(h') = \bot$ or $\beta(h') = (v', h'')$ such that $v' \in |S|$ and $h'' \in \overline{M_S}$, where

$$M_S = \{h'' \in \mathbb{H}_u \mid \exists w'' \in \mathbb{W}.\ \pi_2(s_2([v]_R))([v']_S)([h']_{w'})([h'']_{w''})\}$$

In the first case we trivially have that $h' \in N$. For the second case we have to prove that $\overline{M_S} \subseteq \overline{M_B([v']_S)}$, which follows from $M_S \subseteq M_B([v']_S)$.

Lemma 10. *Let* $R \in \mathrm{CMPer}(\mathbb{V})$*,* $[n] \in [\mathit{in}_{\mathbb{N}}(\{(n, n) \mid n \in \mathbb{N}\})]$*,* $e_n \in [n]$*,* $v \in [R]$*, and* $e_v \in v$*, then*

$$\mathit{write}(e_n, e_v) = (\lambda h.\ \textbf{if}\ h(n) \neq \bot\ \textbf{then}\ (\mathit{in}_1(*), h[e_n \mapsto e_v])\ \textbf{else err}) \in \mathit{hoare}(\{(\mathit{in}_1(*), \mathit{in}_1(*))\}, p, q(v))$$

where

$$p((w, \_) \in \mathbb{H}_t) \stackrel{def}{=} w(n) \downarrow$$
$$q(x \in [R])((w, [h]_w) \in \mathbb{H}_t)(h_t \in \mathbb{H}_t) \stackrel{def}{=} h_t = (w[n \mapsto R], [h[n \mapsto e_v]])$$

Proof. Let $w \in \mathbb{W}$ and $h \in \mid \sim_w \mid$ such that $p(w, [h]_w)$. Then $w(n) \downarrow$ and thus $h(e_n) \neq \bot$. Hence,

$$write(e_n, e_v)(h) = (in_1(*), h[e_n \mapsto e_v])$$

Furthermore, $in_1(*) \in |\{(in_1(*), in_1(*))\}|$ and

$$q([in_1(*)])(w, [h]_w)(w[n \mapsto R], [h[e_n \mapsto e_v]])$$

C.4 W-types

In this appendix we construct W-types in Asm($\mathbb{V}$), from the W-types in Sets. We take it as an axiom that Sets has W-types. From this axiom, we derive an induction principle and a dependent recursion principle for W-types in Sets (Lemmas 11 and 12). W-types in Asm($\mathbb{V}$) are then defined as a "hereditarily realized" subset of the underlying W-type in Sets (Definition 13). From the dependent recursion principle we derive a recursion principle on this "hereditarily realized" subset (Lemma 13), which we use to prove that the "hereditarily realized" subset is indeed a W-type in Asm($\mathbb{V}$) (Lemma 14).

Axiom 1 *Let $A \in$ Sets and $B \in \mathrm{Fam}(\mathrm{Sets})_A$, then there exists a set W with an isomorphism, $sup : \left(\coprod_{a \in A} B_a \to W\right) \cong W$, such that for any set $X \in$ Sets and function $g : (\coprod_{a \in A} B_a \to X) \to X$ there exists a unique function $h : W \to X$, satisfying, $\forall a \in A.\ \forall f \in B_a \to W.\ h(sup(a, f)) = g(a, h \circ f)$.*

Given $A \in$ Sets and $B \in \mathrm{Fam}(\mathrm{Sets})_A$, we use $W_{A,B}$ and $sup_{A,B}$ to denote a set and isomorphism satisfying the conditions of Axiom 1. Given a morphism g as in Axiom 1, we use $fold(g)$ to refer to the meadiating morphism induced by g.

Lemma 11. *Let $A \in$ Sets, $B \in \mathrm{Fam}(\mathrm{Sets})_A$ and $V \subseteq W_{A,B}$, such that*

$$\forall a \in A.\ \forall f \in B_a \to W_{A,B}.\ (\forall b \in B_a.\ f(b) \in V) \Rightarrow sup(a, f) \in V$$

then $V = W$.

Proof. Define $g : (\coprod_{a \in A} B_a \to V) \to V$ as the function, $g(a, f) = sup(a, f)$. Then by the uniqueness of $fold(sup)$ we have that,

$$fold(g) = fold(sup) = id_W$$

and thus $W \subseteq V$.

Lemma 12. *Let* $A \in \mathrm{Sets}$, $B \in \mathrm{Fam}(\mathrm{Sets})_A$, $X \in \mathrm{Fam}(\mathrm{Sets})_W$ *and*

$$g : \Pi_{a\in A}\Pi_{f\in B_a\to W}(\Pi_{b\in B_a} X_{f(b)}) \to X_{sup(a,f)}$$

there exists a unique $h : \Pi_{w\in W} X_w$*, satisfying,*

$$\forall a \in A.\ \forall f \in B_a \to W.\ h(sup(a,f)) = g(a, f, h \circ f)$$

Proof. Define $g' : (\Sigma_{a\in A} B_a \to \Sigma_{w\in W} X_w) \to \Sigma_{w\in W} X_w$ as follows,

$$\begin{aligned} g'(a,f) = (&sup(a, \lambda b \in B_a.\ \pi_1(f(b))),\\ &g(a, \lambda b \in B_a.\ \pi_1(f(b)), \lambda b \in B_a.\ \pi_2(f(b)))) \end{aligned}$$

and let $h' : W \to \Sigma_{w\in W} X_w$ denote the unique function satisfying,

$$\forall a \in A.\ \forall f \in B_a \to W.\ h'(sup(a,f)) = g'(a, h' \circ f) \tag{1}$$

Define

$$V = \{w \in W \mid \pi_1(h'(w)) = w\}$$

then for any $a \in A$, $f \in B_a \to w$, such that $\forall b \in B_a.\ f(b) \in V$, we have that

$$\pi_1(h'(sup(a,f))) = \pi_1(g'(a, h' \circ f)) = sup(a, \lambda b \in B_a.\ \pi_1(h' \circ f)(b)) = sup(a, f)$$

and thus $V = W$. Define $h : \Pi_{w\in W} X_w$ as,

$$h(w) = \pi_2(h'(w))$$

then

$$\begin{aligned} h(sup(a,f)) &= \pi_2(h'(sup(a,f))) = \pi_2(g'(a, h' \circ f))\\ &= g(a, \lambda b \in B_a.\ \pi_1(h'(f(b))), \lambda b \in B_a.\ \pi_2(h'(f(b))))\\ &= g(a, f, h \circ f) \end{aligned}$$

To prove uniqueness, assume $k : \Pi_{w\in W} X_w$ is another meadiating morphism. Then $k' : W \to \Sigma_{w\in W} X_w$ defined as follows,

$$k'(w) \stackrel{\text{def}}{=} (w, k(w))$$

satisfies (1):

$$\begin{aligned} k'(sup(a,f)) &= (sup(a,f), g(a, f, k \circ f))\\ &= (sup(a, \lambda b \in B_a.\ \pi_1(k'(f(b)))),\\ &\quad g(a, \lambda b \in B_a.\ \pi_1(k'(f(b))), \lambda b \in B_a.\ \pi_2(k'(f(b)))))\\ &= g'(a, k' \circ f) \end{aligned}$$

and thus, $k' = h'$ from which it follows that $h = \pi_2 \circ h' = \pi_2 \circ k' = k$.

Definition 13. *Given* $X \in \mathrm{Asm}(\mathbb{V})$ *and* $Y \in \mathrm{UFam}(\mathrm{Asm}(\mathbb{V}))_X$*, let* $\dot{W}_{X,Y}$ *denote the assembly,*

$$\dot{W}_{X,Y} = (\{w \in W_{|X|,|Y|} \mid E_W(w) \neq \emptyset\}, E_W)$$

where $E_W : W_{|X|,|Y|} \to \mathrm{P}(\mathbb{V})$ *denotes the unique function satisfying,*

$$E_W(sup(x,f)) = \{in_\times(e_x, e_f) \mid e_x \in E_X(x) \wedge \forall b \in |Y_x|.\ \forall e_y \in E_{Y_x}(y).\ e_f \cdot e_y \in E_W(f(y))\}$$

Lemma 13. *Let* $X \in \mathrm{Asm}(\mathbb{V})$*,* $Y \in \mathrm{UFam}(\mathrm{Asm}(\mathbb{V}))_X$*,* $Z \in \mathrm{Sets}$ *and* $g : (\coprod_{x \in |X|}(|Y_x| \to Z)) \to Z$*, then there exists a unique* $h : |\dot{W}_{X,Y}| \to Z \in \mathrm{Sets}$ *satisfying,*

$$\forall x \in |X|.\ \forall f \in |Y_x| \to W_{|X|,|Y|}.\ E_W(sup(x,f)) \neq \emptyset \Rightarrow h(sup(x,f)) = g(x, h \circ f) \tag{2}$$

Proof. Define $Z_w = \{* \mid E_W(w) \neq \emptyset\} \to Z$ and

$$g' : \Sigma_{x \in |X|} \Sigma_{f \in |Y_x| \to W_{|X|,|Y|}} (\Pi_{y \in |Y_x|} Z_{f(b)}) \to Z_{sup(x,f)}$$

as follows,

$$g'(x \in |X|, f \in |Y_x| \to W_{|X|,|Y|}, r \in \Pi_{y \in |Y_x|} Z_{f(b)})(*) = g(x, y \in |Y_x| \mapsto r(y)(*))$$

which is well-defined, since if $E_W(sup(x,f)) \neq \emptyset$, then $E_W(f(y)) \neq \emptyset$ for every $y \in |Y_x|$. There thus exists a unique $h' : \Pi_{w \in W} Z_w$ satisfying,

$$\forall x \in |X|.\ \forall f \in |Y_x| \to W_{|X|,|Y|}.\ E_W(sup(x,f)) \neq \emptyset \Rightarrow h'(sup(x,f))(*) = g(x, y \in |Y_x| \mapsto (h' \circ f)(b)(*))$$

Lastly, define $h : |\dot{W}_{X,Y}| \to Z$ as $h(v) = h'(v)(*)$, then

$$\forall x \in |X|.\ \forall f \in |Y_x| \to W_{|X|,|Y|}.\ E_W(sup(x,f)) \neq \emptyset \Rightarrow h(sup(x,f)) = g(x, h \circ f)$$

To show uniqueness, assume $k : |\dot{W}_{X,Y}| \to Z$ satisfying (2). Then,

$$k'(w \in W_{|X|,|Y|}) \stackrel{\text{def}}{=} x \in \{* \mid E_W(w) \neq \emptyset\} \mapsto k(w)$$

satisfies,

$$k'(sup(x,f))(*) = k(sup(x,f)) = g(x, k \circ f) = g(x, b \in B_a \mapsto (k' \circ f)(b)(*))$$

for $x \in |X|$ and $f \in |Y_x| \to W_{|X|,|Y|}$, with $E_W(sup(x,f)) \neq \emptyset$. Hence $k' = h'$ and thus $h(v) = h'(v)(*) = k'(v)(*) = k(v)$.

Definition 14. *Given* $X \in \mathrm{Asm}(\mathbb{V})$ *and* $Y \in \mathrm{UFam}(\mathrm{Asm}(\mathbb{V}))_X$*, let* $P_{X,Y} : \mathrm{Asm}(\mathbb{V}) \to \mathrm{Asm}(\mathbb{V})$ *denote the functor,*

$$P_{X,Y}(A) = (\{(x,f) \in \Sigma_{x \in |X|} |Y_x| \to |A| \mid E_\Pi(f) \neq \emptyset\}, E_P)$$
$$P_{X,Y}(f : A \to B) = (x,g) \mapsto (x, f \circ g)$$

where

$$E_P(x,f) = \{(e_x, e_f) \mid e_x \in E_X(x) \wedge e_f \in E_{\Pi_x}(f)\}$$
$$E_{\Pi_x}(f) = \{\alpha \mid \forall y \in |Y_x|.\ \forall e_y \in E_{Y_x}(y).\ \alpha \cdot e_y \in E_A(f(y))\}$$

Lemma 14. *Let* $X \in \mathrm{Asm}(\mathbb{V})$ *and* $Y \in \mathrm{UFam}(\mathrm{Asm}(\mathbb{V}))_X$ *then*

$$sup_{|X|,|Y|} : P_{X,Y}(\dot{W}_{X,Y}) \to \dot{W}_{X,Y} \in \mathrm{Asm}(\mathbb{V})$$

realized by the identity, is an initial $P_{X,Y}$*-algebra.*

Proof. Given $h : P_{X,Y}(A) \to A \in \mathrm{Asm}(\mathbb{V})$, define $u' : |\dot{W}_{X,Y}| \to |A| + 1$ as the unique map satisfying,

$$u'(sup(x,f)) = \begin{cases} inl(h(x, \pi_l \circ u' \circ f)) & (\forall y \in |Y_x|.\ (\pi_l \circ u' \circ f)(y)\!\downarrow) \wedge E_{\Pi_x}(\pi_l \circ u' \circ f) \neq \emptyset \\ inr(*) & \text{otherwise} \end{cases}$$

where $\pi_l : |A| + 1 \rightharpoonup |A|$ is defined as follows,

$$\pi_l(x) = \begin{cases} y & \text{if } x = inl(y) \\ \textbf{\textit{undef}} & \text{otherwise} \end{cases}$$

Assume e_h is a h-realizer and define α and U as follows,

$$\alpha \stackrel{\text{def}}{=} in_{\to}(fix(\lambda e_u.\ \lambda in_{\times}(e_a, e_f).\ e_h \cdot (e_a, e_u \circ e_f)))$$
$$U \stackrel{\text{def}}{=} \{v \in |\dot{W}_{X,Y}| \mid \exists a \in |A|.\ u'(a) = inl(x) \wedge \forall e_v \in E_W(v).\ \alpha \cdot e_v \in E_A(a)\}$$

where fix is the least fixed point operator on $\mathbb{V} \to \mathbb{V}_\perp$.

To show that $U = |\dot{W}_{X,Y}|$, let $x \in |X|$ and $f : |Y_x| \to W_{|X|,|Y|}$ such that $E_\Pi(x,f) \neq \emptyset$ and assume that

$$\forall y \in |Y_x|.\ f(y) \in U$$

then it follows by the induction hypothesis that

$$\forall y \in |Y_x|.\ (\pi_l \circ u' \circ f)(y)\!\downarrow \wedge E_{\Pi_x}(\pi_l \circ u' \circ f) \neq \emptyset$$

and hence,

$$u'(sup(x,f)) = inl(h(x, \pi_l \circ u' \circ f))$$

Assume $(e_x, e_f) \in E_W(sup(x,f))$. Then $(e_x, \alpha \circ e_f) \in E_P(x, \pi_l \circ u' \circ f)$, since by the induction hypothesis, α realizes $\pi_l \circ u'$ for $w \in W_{|X|,|Y|}$ in the image of f. Hence, as e_h realizes h,

$$\alpha \cdot (e_x, e_f) = e_h \cdot (e_x, \alpha \circ e_f) \in E_A(h(x, \pi_l \circ u' \circ f))$$

and thus $sup(x, f) \in U$.

Define $u : |\dot{W}_{X,Y}| \to |A|$ as

$$u = \pi_l \circ u'$$

then

$$u(sup(a, f)) = h(a, u \circ f)$$

and u is further realized by α.

To prove uniqueness, assume $v : |\dot{W}_{X,Y}| \to |A|$ satisfying $v \circ sup = h \circ P_{X,Y}(v)$. Then $v' = inl \circ v$ satisfies

$$\forall x.\ \forall f.\ v'(sup(x, f)) = inl(h(x, \pi_l \circ v' \circ f))$$

and thus by uniqueness of u', $u' = v'$ and thus $u = v$.

Definition 15. *Let* $(X, E_X) \in \mathrm{Asm}(\mathbb{V})$ *then* (X, E_X) *is complete monotone iff*

$$\forall x, y \in X.\ (\exists a, b \in \mathbb{V}.\ a \in E_X(x) \wedge b \in E_X(y) \wedge a \leq b) \Rightarrow x = y$$

and

$$\forall x \in X.\ \forall c : \mathrm{N} \to_m \mathbb{V}.\ (\forall n \in \mathbb{N}.\ c(n) \in E_X(x)) \Rightarrow \sqcup_n c(n) \in E_X(x)$$

Lemma 15. *Let* $X \in \mathrm{Asm}(\mathbb{V})$ *and* $Y \in \mathrm{UFam}(\mathrm{Asm}(\mathbb{V}))_X$. *If* X *is complete monotone and* Y_x *is complete monotone for each* $x \in |X|$, *then* $\dot{W}_{X,Y}$ *is complete monotone.*

Proof. We prove

$$\begin{aligned}
&\forall w \in W_{|X|,|Y|}.\\
&\quad (\forall w' \in W_{|X|,|Y|}.\ (\exists a, b.\ a \in E_W(w) \wedge b \in E_W(w') \wedge a \leq b) \Rightarrow w = w') \wedge\\
&\quad (\forall c : \mathbb{N} \to_m \mathbb{V}.\ (\forall n \in \mathbb{N}.\ c(n) \in E_W(w)) \Rightarrow \bigsqcup_n c(n) \in E_W(w))
\end{aligned}$$

by induction on $w \in W_{|X|,|Y|}$. Assume $x_1 \in |X|$ and $f_1 \in |Y_x| \to W_{|X|,|Y|}$.

We prove the first conjunct by induction on $w' \in W_{|X|,|Y|}$. Assume $x_2 \in |X|$ and $f_2 \in |Y_x| \to W_{|X|,|Y|}$, such that there exists $(e_{x_1}, e_{f_1}) \in E_W(x_1, f_1)$ and $(e_{x_2}, e_{f_2}) \in E_W(x_2, f_2)$ with $(e_{x_1}, e_{f_1}) \leq (e_{x_2}, e_{f_2})$. By monotonicity of X we thus have that $x_1 = x_2$. Furthermore, given any $y \in |Y_x|$ and $e_{f_1} \cdot e_y \leq e_{f_2} \cdot e_y$, where $e_y \in E_{Y_x}(y)$, and thus $f_1(y) = f_2(y)$, by the induction hypothesis for w.

The second conjunct follows easily from continuity of f_1 realizers and the induction hypothesis.