A separation logic for refining concurrent objects

Aaron Turon   Mitchell Wand
Northeastern University
{turon, wand}@ccs.neu.edu

Abstract

Fine-grained concurrent data structures are crucial for gaining performance from multiprocessing, but their design is a subtle art. Recent literature has made large strides in verifying these data structures, using either atomicity refinement or separation logic with rely-guarantee reasoning. In this paper we show how the ownership discipline of separation logic can be used to enable atomicity refinement, and we develop a new rely-guarantee method that is localized to the definition of a data structure. We present the first semantics of separation logic that is sensitive to atomicity, and show how to control this sensitivity through ownership. The result is a logic that enables compositional reasoning about atomicity and interference, even for programs that use fine-grained synchronization and dynamic memory allocation.

1. Introduction

1.1 The goal

Our story begins with a very simple data structure: the counter. Counters permit a single operation, `inc`, implemented as follows:

```
int inc(int *C) {
  tmp = *C;  *C = tmp+1;  return tmp;
}
```

Of course, this implementation only works in a sequential setting. If multiple threads use it concurrently, an unlucky interleaving can lead to several threads fetching the same value from the counter. The usual reaction to this problem is to use mutual exclusion, wrapping the operation with lock instructions. But as Moir and Shavit put it, "with this arrangement, we prevent the bad interleavings by preventing *all* interleavings" [20]. Fine-grained concurrent objects permit as many good interleavings as possible, without allowing any bad ones. The code

```
int inc(int *C) {
  do { tmp = *C; } until CAS(C, tmp, tmp+1);
  return tmp;
}
```

implements `inc` using an optimistic approach: it takes a snapshot of the counter without acquiring a lock, computes the new value of the counter, and uses compare-and-set (`CAS`) to safely install the new value. The key is that `CAS` compares `*C` with the value of `tmp`, *atomically* updating `*C` with `tmp + 1` and returning `true` if they are the same, and just returning `false` otherwise.

Even for this simple data structure, the fine-grained implementation significantly outperforms the lock-based implementation [17]. Likewise, even for this simple example, we would prefer to think of the counter in a more abstract way when reasoning about its clients, giving it the following specification:

$$\mathsf{inc}(c,\ \mathit{ret})\ \triangleq\ \langle \forall x : c \mapsto x,\ c \mapsto x+1 \wedge \mathit{ret} = x\rangle$$

This specification says that, for any value x, inc atomically transforms a heap in which c points to x into one where c points to $x+1$, moreover ensuring that the value of *ret* (an out-parameter) is x.

We express both implementations and their specifications in a single specification language, and take the perspective of refinement: φ *refines* (implements) ψ if for every context $C[-]$, each behavior of $C[\varphi]$ is a possible behavior of $C[\psi]$.[1] That is, no client can detect that it is interacting with φ rather than ψ, even if the client invokes many operations of φ concurrently.

This paper presents a logic for proving such refinements, based on separation logic. The key idea is to use a notion of *ownership*, expressed through separation logic, to reason about both atomicity and interference. The key contributions are the semantics ("fenced refinement") and proof rules enabling this reasoning. We will be able to easily verify the counter given above (§4.1), as well as more complex, nonblocking data structures (*e.g.* a nonblocking stack §6).

1.2 The approach

Since this paper is focused on verification rather than development, we will take the perspective of *abstraction*, which is converse to refinement: φ refines ψ iff ψ abstracts φ. The appeal of abstraction is that, to prove something about the behaviors of $C[\varphi]$ for a particular client C, it suffices to consider the behaviors of $C[\psi]$ instead—and ψ is usually much simpler than φ. For verifying data structures in a sequential setting, one is primarily interested in *data abstraction*, where ψ abstracts away representation details from φ. In a concurrent setting, we want simplicity in another respect: *atomicity abstraction*, where ψ appears to perform an operation in one atomic step even though φ takes many steps.

Refinement hinges on the power of the context $C[-]$: what can it observe, and with what can it interfere? The more powerful the context, the less freedom we have in implementing a specification. Concurrency changes not *what*, but *when*. In a first-order language, a sequential context can only interact with the state before or after running the program fragment in its hole. A concurrent context might do so at any time.

In his classic 1973 paper, "Protection in Programming Languages", Morris argued that "a programmer should be able to prove that his programs . . . do not malfunction solely on the basis of what

[1] We take the perspective, advanced by Filipović *et al.* [8], that linearizability [11] is a *proof technique* for refinement—and we choose not to use it. See §8 for more discussion.

he can see from his private bailiwick" [22]. That paper and its contemporaries showed how to enable such reasoning by weakening the context, *e.g.* by hiding data through lexical scoping [22], protecting data by dynamic sealing [22] or by giving it an opaque (existential) type [15, 19]. A basic message of this paper is that these and other hiding techniques serve just as well to localize reasoning in a concurrent setting. Of course, hiding mechanisms appear in various guises in concurrency theory (restriction in CCS [18], existentials in Lamport's TLA [13]). So what is new in this paper are the particulars. They are as follows.

We present a new approach for proving refinement of concurrent objects. By *concurrent object* we mean a set of methods, one of which is a constructor, the others operators. Each method takes as an argument a pointer to an object instance. The methods are implemented using purely sequential code, but the client of an object (*i.e.*, its context) may invoke many methods on the same instance, concurrently.

When reasoning about concurrent objects, we make use of an ownership discipline enforced by the logic:

- Each concurrent object instance is owned by the methods associated with the object; clients of an object cannot modify it except by invoking these methods.
- Each method has access to one object instance, but this instance may be concurrently modified by other method invocations.
- Each method may allocate private memory, and may assume this memory will not be interfered with.
- No method may access memory other than its private memory, and its shared object instance. In particular, it cannot access memory that belongs to a client.

Just what constitutes an "object instance" is determined by the prover, who selects a heap predicate as the representation invariant for an object instance. The invariant, in addition to expressing well-formedness constraints, also picks out what portion of the heap belongs to a given instance. Thus, as in concurrent separation logic, "ownership is in the eye of the prover" [23].

A key point about this ownership discipline—typically for a separation logic—is that it is *dynamic*. For example, data private to a method can become reachable as part of an object instance. At that point, the method must relinquish sole ownership of the data, and must contend with concurrent interference on it. On the other hand, a method may withdraw data from an instance, at which point it becomes method-private. It is this dynamic flavor of ownership and hiding that differentiates our logic from prior work applying hiding to concurrency.

We derive two major benefits from our ownership discipline. First, it enables us to perform atomicity abstraction, under the basic principle that "if you don't own it, you can't see it". Thus an atomic step modifying only method-private data can be absorbed into an adjacent atomic step, to form a single, large atomic step.

Claim 1: By making both atomicity and ownership explicit, we can clarify and exploit their relationship: atomicity is relative to ownership.

Atomicity abstraction stems from the notion of private data. The other benefit of ownership is to do with object instances, which are shared among concurrently-running methods.

Claim 2: Because ownership limits possible interference, we can exploit it to give a *modular* and *dynamic* account of rely/guarantee reasoning.

Rely/guarantee reasoning [12] is a well-known technique for compositionally reasoning about interference: you prove that each thread guarantees certain behavior, as long as it can rely on other threads to interfere in a limited way. Traditionally, the rely constraint must be carried through the whole verification, even if it is only relevant to a small portion. Using ownership, we localize rely/guarantee reasoning to the definition of a concurrent object, since we know all interference must stem from the methods making up its definition. In our approach, rely/guarantee reasoning is done with respect to a hypothetical object instance; at runtime, there may be zero, one, or many such instances, each residing in its own region of the heap. A broadly similar form of rely/guarantee reasoning appeared in very recent work by Dinsdale-Young *et al.* [4]; we compare the approaches in §8.

To our claims, we make the following contributions:

- We present a new specification language in the tradition of refinement calculi [1, 21], but tailored to separation logic-style reasoning (§2). We give it an operational semantics, which determines the definition of refinement. Our notion of refinement captures safety properties only, but we expect most of our results to easily transfer to a model incorporating liveness.
- We adapt Brookes's transition trace model [2] to handle dynamic memory allocation, and use it to give our specification language a simple denotational semantics (§3). The semantics is adequate for the operational model, so it can be used to justify refinements. To our knowledge, this is the first model of separation logic that captures atomicity explicitly.
- On the strength of the denotational model, we are able to give many basic refinement laws (§4), and we show how these laws can be used to verify a simple nonblocking counter (§4.1). They are not strong enough, however, to handle more complex data structures—for that, we need ownership-based reasoning.
- In §5.2 we introduce our ownership discipline, formally captured by the notion of *fenced refinement*. The semantics of fenced refinement is a *hybrid* between trace semantics and input/output relations: traces for shared state, I/O for private state. We give laws of fenced refinement that justify our two claims above. Finally, we give a law DATA$_2$ relating fenced and standard refinement, which allows us to employ fenced refinement compositionally in correctness proofs. We sketch the soundness proof for this law in §7.

Domains

x, y, z	$\in$	VAR	variables
		LOC	locations
v	$\in$	VAL	values
ρ	$\in$	$\mathrm{VAR} \rightharpoonup \mathrm{VAL}$	environments
σ	$\in$	$\Sigma \triangleq \mathrm{LOC} \rightharpoonup \mathrm{VAL}$	heaps
o	$\in$	$\Sigma^{\lightning} \triangleq \Sigma \cup \{\lightning\}$	outcomes

Syntax

Specifications	φ, ψ, θ	$::=$	$\varphi;\psi \mid \varphi \parallel \psi \mid \varphi \vee \psi \mid \exists x.\varphi$
		$\mid$	$\mu X.\varphi \mid X \mid \mathbf{let}\ f = F\ \mathbf{in}\ \psi$
		$\mid$	$F\ e \mid \langle \forall \overline{x} : p,\ q \rangle \mid \{p\}$
Procedures	F	$::=$	$f \mid \lambda x.\varphi$
Configurations	κ	$::=$	$\varphi, \sigma \mid o$
Predicates	p, q, r	$::=$	$\mathsf{true} \mid \mathsf{false} \mid \mathsf{emp} \mid e \mapsto e'$
		$\mid$	$e = e' \mid p * q \mid p \wedge q \mid p \vee q$
		$\mid$	$p \Rightarrow q \mid \forall x.p \mid \exists x.p$
Expressions	e	$::=$	$x \mid n \mid e + e' \mid (e, e')$

Figure 1.

- We present a case study—the verification of a nonblocking stack—to demonstrate fenced refinement (§6).

This paper draws on several ideas from recent work, especially that of Vafeiadis *et al.* [27, 28] and Elmas *et al.* [5, 6]. We refer the reader to §8 for a detailed discussion of prior work.

2. Specifications

The standard view of refinement calculus is as follows [21]: we are broadly interested in "specifications", the most concrete of which are "programs". Thus, we have an expressive language of specifications φ (so it is easy to say *what* behavior is wanted), and a sublanguage of programs (that say *how* to produce such behavior). It will be possible, for instance, to write a specification that solves the halting problem, but this will not correspond to any program. The distinction between programs and general specifications is not important for our purposes here; suffice it to say that the implementations we will verify are clearly executable on a machine.

The language of specifications (Figure 1) includes traditional programming constructs like sequential composition $\varphi;\psi$ and parallel composition $\varphi \parallel \psi$, let-binding of first-order procedures, and recursion $\mu X.\varphi$. In addition, there are logical constructs like disjunction $\varphi \vee \psi$ and quantification $\exists x.\varphi$. These latter can be read operationally as nondeterminism: $\varphi \vee \psi$ either behaves like φ or behaves like ψ. The remaining specification constructs (shaded in gray) will be explained in §2.2.

Operational semantics of specifications $\kappa \to \kappa'$

$$\frac{\varphi_1,\sigma \to \varphi_1',\sigma'}{\varphi_1;\varphi_2,\sigma \to \varphi_1';\varphi_2,\sigma'} \qquad \frac{\varphi_1,\sigma \to \sigma'}{\varphi_1;\varphi_2,\sigma \to \varphi_2,\sigma'} \qquad \frac{\varphi_1,\sigma \to \lightning}{\varphi_1;\varphi_2,\sigma \to \lightning}$$

$$\frac{\varphi_1 \parallel \varphi_2,\sigma \to \kappa}{\varphi_2 \parallel \varphi_1,\sigma \to \kappa} \qquad \frac{\varphi_1,\sigma \to \varphi_1',\sigma'}{\varphi_1 \parallel \varphi_2,\sigma \to \varphi_1' \parallel \varphi_2,\sigma'}$$

$$\frac{\varphi_1,\sigma \to \sigma'}{\varphi_1 \parallel \varphi_2,\sigma \to \varphi_2,\sigma'} \qquad \frac{\varphi_1,\sigma \to \lightning}{\varphi_1 \parallel \varphi_2,\sigma \to \lightning}$$

$$\varphi_1 \vee \varphi_2,\sigma \to \varphi_1,\sigma \qquad \varphi_1 \vee \varphi_2,\sigma \to \varphi_2,\sigma$$

$$\frac{v \in \textsc{Val}}{\exists x.\varphi,\sigma \to \varphi[v/x],\sigma} \qquad \mu X.\varphi,\sigma \to \varphi[\mu X.\varphi/X],\sigma$$

$$\textbf{let } f = F \textbf{ in } \varphi,\sigma \to \varphi[F/f],\sigma \qquad (\lambda x.\varphi)e,\sigma \to \varphi[\llbracket e \rrbracket/x],\sigma$$

$$\frac{(\sigma,o) \in \mathsf{act}(p,q)}{\langle \forall \overline{x} : p,\ q\rangle,\sigma \to o} \qquad \frac{\sigma \models p * \mathsf{true}}{\{p\},\sigma \to \sigma} \qquad \frac{\sigma \not\models p * \mathsf{true}}{\{p\},\sigma \to \lightning}$$

In this largely-standard semantics, a specification interacts step-by-step with a heap σ. A *configuration* κ of the abstract machine is typically a pair φ,σ of the remaining specification to execute, and the current heap. In addition, there are two terminal configurations, called *outcomes*: either successful termination with heap σ, or else a *fault* $\lightning$. Faulting will be explained in §2.2; the important point at the moment is that faulty termination propagates up through sequential and parallel composition.

A few other points of note:

- Variables x, y, z are immutable; only the heap is mutable.
- The domain of values $\textsc{Val}$ is unspecified, but at least includes locations, integers, and pairs.
- We assume a denotational semantics $\llbracket e \rrbracket^\rho : \textsc{Val}$ for expressions e, which is straightforward. We write $\llbracket e \rrbracket$ for $\llbracket e \rrbracket^\emptyset$.
- A composition $\varphi;\psi$ steps to ψ exactly when φ terminates.
- The first rule for $\parallel$ makes it symmetric.
- We abbreviate **let** $f = \lambda x.\varphi$ **in** ψ by **let** $f(x) = \varphi$ **in** ψ.

2.1 Heap predicates

To describe the remaining forms of specifications (shaded in gray), we first need to define *heap predicates* p. These are the usual predicates of separation logic: they make assertions about both program variables (*e.g.* $x = 3$) and the heap (*e.g.* $x \mapsto 3$). As such, the entailment relation $\sigma,\rho \models p$ holds when p is true both of the heap σ and the *environment* ρ, which gives the values of program variables:

Predicate semantics $\sigma,\rho \models p$

$$\begin{array}{ll}
\sigma,\rho \models \mathsf{emp} & \text{iff } \sigma = \emptyset \\
\sigma,\rho \models e \mapsto e' & \text{iff } \sigma = [\llbracket e \rrbracket^\rho \mapsto \llbracket e' \rrbracket^\rho] \\
\sigma,\rho \models e = e' & \text{iff } \llbracket e \rrbracket^\rho = \llbracket e' \rrbracket^\rho \\
\sigma,\rho \models p * q & \text{iff } \exists \sigma_1,\sigma_2.\ \sigma = \sigma_1 \uplus \sigma_2,\ \ \sigma_1,\rho \models p,\ \ \sigma_2,\rho \models q \\
\sigma,\rho \models p \vee q & \text{iff } \sigma,\rho \models p \text{ or } \sigma,\rho \models q \\
\sigma,\rho \models \forall x.p & \text{iff } \forall v.\ \sigma,\rho[x \mapsto v] \models p
\end{array}$$

The predicates emp, $e \mapsto e'$ and $p * q$ come from separation logic. The predicate emp asserts that the heap is empty, while $e \mapsto e'$ asserts that the heap contains a *single* location e which contains the value e'. Thus $x \mapsto 3$ is satisfied when ρ takes x to the only location in σ, and σ maps that location to 3. The *separating conjunction* $p{*}q$ is satisfied by any heap separable into one subheap satisfying p and one satisfying q. Thus $x \mapsto 3 {*} y \mapsto 4$ asserts that the heap contains exactly two locations, containing 3 and 4. In particular, it implies that the locations $x \neq y$.

The remaining predicates are familiar from first-order logic; we give two illustrative cases.

Finally, we write $\sigma \models p$ for $\sigma,\emptyset \models p$.

2.2 Actions, assertions, and assumptions

In separation logic, a Hoare triple $\{p\}C\{q\}$ for a command C asserts that, if $\sigma \models p$, then C running on σ

- will not fault (by, say, following a dangling pointer) and,
- if it terminates, will do so with a heap σ' such that $\sigma' \models q$.

Note that p and q may be quite specific about the shape of the heap, for example insisting that it consists of a single cell. A key idea of separation logic is to view these predicates not as describing the entire heap, but as describing the portion relevant to C; the rest of the heap is neither accessed nor modified by C. This idea leads to the *frame rule*, allowing $\{p * r\}C\{q * r\}$ to be inferred from $\{p\}C\{q\}$.

In our specification language, we forgo basic commands in favor of a single construct: the *action* $\langle \forall \overline{x} : p,\ q\rangle$. An action describes an *atomic* step of computation in terms of the strongest partial correctness assertion it satisfies (*cf.* Morgan's specification statements [21]). In other words, it permits every behavior that satisfies the triple $\{p\} - \{q\}$. The variables $\overline{x}$ are in scope for both p and q, and are used to link them together (as in inc above).

To understand the behavior of a simple action $\langle p,\ q\rangle$, it is helpful to think in terms of a specific starting heap σ. First, suppose that $\sigma \not\models p * \mathsf{true}$, meaning that no subheap of σ satisfies p. In this case, the behavior is unconstrained, and in particular the action is permitted to fault. Hence, faults arise when preconditions are not satisfied. On the other hand, if σ can be decomposed into subheaps $\sigma = \sigma_1 \uplus \sigma_2$ such that $\sigma_1 \models p$, then the action must take a step to some heap $\sigma_1' \uplus \sigma_2$ such that $\sigma_1' \models q$; it must establish the postcondition without modifying the frame σ_2, and cannot fault.

The semantics of actions is in essence that of Calcagno *et al.* [3]:

Action semantics $\quad \mathrm{act}(p,q) \subseteq \Sigma \times \Sigma^{\lightning}$

$$\begin{array}{l} (\sigma, o) \in \mathrm{act}(p,q) \text{ iff } \forall \sigma_1, \sigma_2, \rho.\ \exists \sigma_1'. \\ \quad \text{if} \quad \sigma = \sigma_1 \uplus \sigma_2 \text{ and } \sigma_1, \rho \models p \\ \quad \text{then} \quad o = \sigma_1' \uplus \sigma_2 \text{ and } \sigma_1', \rho \models q \end{array}$$

Many familiar commands can be expressed as actions. The following are used in our examples, and provide some intuition for the semantics of actions:

$$\begin{array}{rcl} \mathsf{abort} & \triangleq & \langle \mathsf{false},\ \mathsf{true} \rangle \\ \mathsf{miracle} & \triangleq & \langle \mathsf{true},\ \mathsf{false} \rangle \\ \mathsf{skip} & \triangleq & \langle \mathsf{emp},\ \mathsf{emp} \rangle \\ \mathsf{new}(x, \mathit{ret}) & \triangleq & \langle \mathsf{emp},\ \mathit{ret} \mapsto x \rangle \\ \mathsf{put}(a, x) & \triangleq & \langle a \mapsto -,\ a \mapsto x \rangle \\ \mathsf{get}(a, \mathit{ret}) & \triangleq & \langle \forall x : a \mapsto x,\ a \mapsto x \wedge x = \mathit{ret} \rangle \\ \mathsf{cas}(a, \mathit{old}, \mathit{new}, \mathit{ret}) & \triangleq & \end{array}$$

$$\left\langle \forall x : a \mapsto x, \quad \begin{array}{l} (x \neq \mathit{old} \wedge \mathit{ret} = 0 \wedge a \mapsto x) \\ \vee\ (x = \mathit{old} \wedge \mathit{ret} = 1 \wedge a \mapsto \mathit{new}) \end{array} \right\rangle$$

where the predicate $a \mapsto -$ is shorthand for $\exists z. a \mapsto z$. The abort action can always fault, as its precondition is never satisfied. On the other hand, miracle never faults, but it also never takes any steps, as its postcondition is never satisfied. From the standpoint of the operational semantics, miracle *neither faults nor successfully terminates*; from a partial correctness standpoint, it satisfies every specification [21]. The skip action cannot fault, because any heap has a subheap that satisfies emp. Likewise, new cannot fault, and it furthermore ensures that *ret* is distinct from any address allocated in the frame. Finally, put, get and cas fault if a is unallocated. Notice that get and cas both quantify over a variable x in order to connect an observation made in the precondition to a constraint in the postcondition.

One common theme in these examples is the use of postconditions to filter possible behaviors, miracle being the most extreme case. For procedures like get, the postcondition is used to constrain the out-parameter *ret*. Where one might write **let** $x = \mathsf{get}(a)$ **in** φ as a program, we write $\exists x.\mathsf{get}(a, x); \varphi$ as a specification. Executions where x took on the wrong value simply disappear, neither terminating nor faulting.

One use of postcondition filtering is so common that we introduce shorthand for it: an *assumption* $[p]$ stands for the action $\langle \mathsf{emp},\ \mathsf{emp} \wedge p \rangle$. Because its precondition is emp, an assumption cannot fault, and because its postcondition implies emp as well, it cannot alter the heap; it is a refinement of skip. Since it is conjoined with emp, the predicate p cannot further constrain the heap; it is used only to constrain the environment ρ. Assumptions are used for control flow: the specification **if** p **then** φ **else** ψ is sugar for $([p]; \varphi) \vee ([\neg p]; \psi)$.

Finally, *assertions* $\{p\}$ simply test a predicate: if some subheap satisfies p then the heap is left unchanged, and otherwise the assertion faults. Assertions provide a way to introduce a claim during verification while postponing its justification (*cf.* [5]).

We can write the fine-grained concurrent version of inc in our specification language as follows:

$$\begin{array}{rcl} \mathsf{inc}'(c, \mathit{ret}) & \triangleq & \mu X.\ \exists t.\mathsf{get}(c, t); \\ & & \quad\quad \exists b.\mathsf{cas}(c, t, t+1, b); \\ & & \quad\quad \textbf{if } b = 1 \textbf{ then } [\mathit{ret} = t] \textbf{ else } X \end{array}$$

3. Refinement: model theory

We have seen what specifications are, how they behave, and how they express common programming constructs. But the point of working with specifications is to *compare* them: we want to say when a relatively concrete specification "implements" a more abstract one. Informally, a specification φ implements ψ if no client can observe that it is interacting with φ instead of ψ, *i.e.*, every behavior of φ is a possible behavior of ψ. Formally, this situation is expressed as *refinement*:

Refinement $\quad \varphi \sqsubseteq_{\mathsf{op}} \psi$

$$\varphi \sqsubseteq_{\mathsf{op}} \psi \text{ iff } \forall C, \sigma. \left\{ \begin{array}{ll} \text{if } C[\varphi], \sigma \ \lightning & \text{then } C[\psi], \sigma \ \lightning \\ \text{if } C[\varphi], \sigma \Downarrow & \text{then } C[\psi], \sigma \Downarrow \text{ or } C[\psi], \sigma \ \lightning \end{array} \right.$$

$$\text{where} \quad \kappa \ \lightning \ \text{ iff } \ \kappa \rightarrow^* \lightning \quad \text{and} \quad \kappa \Downarrow \ \text{ iff } \ \exists \sigma. \kappa \rightarrow^* \sigma$$

where C is a specification context (a specification with a hole) closing φ and ψ. The "op" stands for operational semantics. In this definition, we collapse the possible outcomes of a $C[\varphi], \sigma$ to three cases: nontermination, successful termination, and faulting termination. If the specification ψ can fault in a given context, then φ is relieved of any obligations for that context. In this way, faulting behavior is treated as "unspecified" behavior, and as we will soon see, it follows that abort is the most permissive specification ($\varphi \sqsubseteq_{\mathsf{op}} \mathsf{abort}$ for all φ).

Ultimately, our goal is to give useful axioms and inference rules for proving such refinements. However, as usual, the quantification over all contexts in the definition of refinement makes it difficult to work with directly. As a stepping stone, in this section we give a denotational semantics for specifications, which will give us a sound (but not complete) denotational version $\sqsubseteq_{\mathsf{den}}$ of refinement. Readers primarily interested in the proof rules can read §4 first, taking the soundness of the rules on faith.

The denotational model is based on Brookes's transition trace model [2], which gave a fully abstract semantics for a parallel WHILE language. We adjust this model to deal with pointers and the heap, as well as faulting.

The challenge in giving a denotational model for concurrency is the semantics of parallel composition: to define $[\![\varphi \parallel \psi]\!]$ in terms of $[\![\varphi]\!]$ and $[\![\psi]\!]$ we must "leave room" for interference from ψ in the meaning of φ, and *vice-versa*. With transition traces, we give meaning to φ and ψ in terms of discrete timeslices of execution, between which we allow for arbitrary interference. To calculate the meaning of $\varphi \parallel \psi$, we need only interleave these timeslices.

In detail: we model the behavior of each specification as a set of transition traces. A *transition trace* is a finite sequence of *moves* (σ, σ'). Each move represents one timeslice of execution, which may correspond to zero, one, or some finite number of steps $\rightarrow$ in the operational semantics. A trace of φ like $(\sigma_1, \sigma_1')(\sigma_2, \sigma_2')$ arises from an execution where first $\varphi, \sigma_1 \rightarrow^* \varphi', \sigma_1'$, then some as-yet unknown specification in the environment changed the heap from σ_1' to σ_2, then $\varphi', \sigma_2 \rightarrow^* \varphi'', \sigma_2'$. A trace can be terminated by a fault, either on the part of the specification as in $(\sigma, \lightning)$, or on the part of its environment as in $(\lightning, \lightning)$:

Domains

$$\begin{array}{rcl} \textsc{Move} & \triangleq & \Sigma \times \Sigma \\ \textsc{Fault} & \triangleq & \Sigma^{\lightning} \times \{\lightning\} \\ \textsc{Trace} & \triangleq & \textsc{Move}^*; \textsc{Fault}^? \end{array} \qquad \begin{array}{rcl} s, t, u & \in & \textsc{Trace} \\ S, T, U & \subseteq & \textsc{Trace} \end{array}$$

The transition traces of a specification can be read directly from the operational semantics:

Observed traces $\quad t \in \mathcal{O}[\![\varphi]\!]$

$$\frac{}{(\lightning, \lightning) \in \mathcal{O}[\![\varphi]\!]} \qquad \frac{\varphi, \sigma \rightarrow^* o}{(\sigma, o) \in \mathcal{O}[\![\varphi]\!]} \qquad \frac{\varphi, \sigma \rightarrow^* \varphi', \sigma' \quad t \in \mathcal{O}[\![\varphi']\!]}{(\sigma, \sigma')t \in \mathcal{O}[\![\varphi]\!]}$$

The inference rules say, respectively: the environment might fault at any time; a terminating timeslice (whether successful or faulting) results in a singleton trace; a nonterminating timeslice allows the specification to resume later (under a possibly-changed heap).

Our denotational semantics gives an alternative definition of $\mathcal{O}[\![\varphi]\!]$ that is *compositional* in the structure of φ; this compositionality is ultimately how we connect transition traces to refinement (Theorem 1 below).

An important insight in Brookes's model is that the transition traces of a specification are closed under *stuttering* (addition of a step (σ, σ)) and *mumbling* (merging of two steps with a common midpoint). Closure under stuttering means that the context of a specification cannot observe timeslices in which it does not change the heap; this justifies that skip is a unit for sequential composition. Closure under mumbling will imply that, for example, $\langle p,\ r\rangle \sqsubseteq_{\mathsf{op}} \langle p,\ q\rangle\,;\langle q,\ r\rangle$, because the scheduler might give the latter specification a long enough timeslice to execute both actions, so that its behavior looks just like that of the former.

In giving the denotational semantics, we must explicitly apply stuttering and mumbling closure, which we do *via* the closure operator † (in effect a quotient, blurring together observationally indistinguishable trace sets):

Closure $\qquad t \in T^\dagger$

$$\frac{t \in T}{t \in T^\dagger} \qquad \frac{st \in T^\dagger}{s(\sigma,\sigma)t \in T^\dagger} \qquad \frac{s(\sigma,\sigma')(\sigma',o)t \in T^\dagger}{s(\sigma,o)t \in T^\dagger}$$

$$\frac{}{(\lightning\,,\lightning\,) \in T^\dagger} \qquad \frac{t(\sigma,\lightning\,) \in T^\dagger}{t(\sigma,\sigma')u \in T^\dagger}$$

The unshaded rules for † appeared in Brookes's original paper. To them, we add two rules concerning faults. The first captures the fact that the environment of a specification might cause a fault at any time. The second reflects that faulting on the part of a specification is *permissive*, so a specification that faults after some interactions t can be implemented by one that continues without faulting after t.

The reason $(\lightning\,,\lightning\,)$ steps are important is to handle cases like $(\mu X.X) \parallel \mathsf{abort}$. Without the $(\lightning\,,\lightning\,)$ steps, $\mathcal{O}[\![(\mu X.X)]\!]$ would be empty, but the semantics of the composition is nonempty. The effect of including $(\lightning\,,\lightning\,)$ in the closure is that every finite prefix of the behavior of a specification is included in its set of traces, but with the marker $(\lightning\,,\lightning\,)$ at the end signifying that the specification was terminated early by a fault on the part of the environment.

With those preliminaries out of the way, the denotational semantics is straightforward. Sequential composition is concatenation and parallel composition is nondeterministic interleaving. One caveat: when concatenating traces, a fault on the left wipes out everything on the right, since faulting causes early termination: $t(o,\lightning\,); u = t(o,\lightning\,)$. This rule applies to both sequential and parallel composition. Recursion is defined using the Tarskian least-fixed point over closed sets of traces; the order on trace sets is set inclusion. Disjunction and existential quantification are given by least upper bounds according to that ordering. Here, environments ρ map variables x to values v, specification variables X to closed sets of traces T, and procedure variables f to functions $\textsc{Val} \to 2^{\textsc{Trace}}$. Environments are ordered pointwise, leading to the following lemma.

Lemma 1. For each φ, the function $[\![\varphi]\!]$ from environments to trace sets is monotonic.

We connect the denotational semantics to refinement in two steps. First, we show that the denotational semantics gives the sames trace sets as the operational semantics, modulo †-closure:

Lemma 2. If φ is a closed specification then $\mathcal{O}[\![\varphi]\!]^\dagger = [\![\varphi]\!]^\emptyset$.

Denotational semantics of specifications $\qquad [\![\varphi]\!]^\rho \subseteq \textsc{Trace}$

$$\begin{aligned}
[\![\varphi;\psi]\!]^\rho &\triangleq ([\![\varphi]\!]^\rho\,; [\![\psi]\!]^\rho)^\dagger \\
[\![\varphi \parallel \psi]\!]^\rho &\triangleq ([\![\varphi]\!]^\rho \parallel [\![\psi]\!]^\rho)^\dagger \\
[\![\varphi \vee \psi]\!]^\rho &\triangleq [\![\varphi]\!]^\rho \cup [\![\psi]\!]^\rho \\
[\![\exists x.\varphi]\!]^\rho &\triangleq \textstyle\bigcup_v [\![\varphi]\!]^{\rho[x\mapsto v]} \\
[\![\textbf{let } f = F \textbf{ in } \psi]\!]^\rho &\triangleq [\![\psi]\!]^{\rho[f\mapsto [\![F]\!]^\rho]} \\
[\![F(e)]\!]^\rho &\triangleq [\![F]\!]^\rho\,([\![e]\!]^\rho) \\
[\![\mu X.\varphi]\!]^\rho &\triangleq \textstyle\bigcap\{T : T^\dagger = T,\ [\![\varphi]\!]^{\rho[X\mapsto T]} \subseteq T\} \\
[\![X]\!]^\rho &\triangleq \rho(X) \\
[\![\langle\forall\overline{x} : p,\ q\rangle]\!]^\rho &\triangleq \mathsf{act}(\rho(p), \rho(q))^\dagger \\
[\![\{p\}]\!]^\rho &\triangleq \{(\sigma,\sigma)\ :\ \sigma \in [\![p * \mathsf{true}]\!]^\rho\}^\dagger \\
&\quad\ \cup\ \{(\sigma,\lightning\,)\ :\ \sigma \notin [\![p * \mathsf{true}]\!]^\rho\}^\dagger
\end{aligned}$$

Procedures:

$$[\![f]\!]^\rho \triangleq \rho(f) \qquad [\![\lambda x.\varphi]\!]^\rho \triangleq \lambda v.\,[\![\varphi]\!]^{\rho[x\mapsto v]}$$

Figure 2.

This lemma is proved separately in each direction; the $\subseteq$ direction goes by induction on the rules defining $\mathcal{O}[\![-]\!]$, while $\supseteq$ goes by induction on the structure of φ.

We define a denotational version of refinement, and prove that it soundly approximates the operational version ("adequacy"):

Definition 1. $\varphi \sqsubseteq_{\mathsf{den}} \psi$ iff for all closing ρ, $[\![\varphi]\!]^\rho \subseteq [\![\psi]\!]^\rho$.

Theorem 1 (Adequacy)**.** If $\varphi \sqsubseteq_{\mathsf{den}} \psi$ then $\varphi \sqsubseteq_{\mathsf{op}} \psi$.

Proof. Suppose $\varphi \sqsubseteq_{\mathsf{den}} \psi$. Let C be a specification context that closes φ and ψ. By the monotonicity of the semantics, one can show that $[\![C[\varphi]]\!]^\emptyset \subseteq [\![C[\psi]]\!]^\emptyset$. By Lemma 2, $\mathcal{O}[\![C[\varphi]]\!]^\dagger \subseteq \mathcal{O}[\![C[\psi]]\!]^\dagger$. It follows that if $C[\varphi], \sigma \lightning$ then $C[\psi], \sigma \lightning$, and that if $C[\varphi], \sigma \Downarrow$ then either $C[\psi], \sigma \lightning$ or $C[\psi], \sigma \Downarrow$. □

4. Refinement: proof theory

With the denotational semantics in hand, it is easy to prove a number of basic refinement laws. These laws will be powerful enough to verify a basic nonblocking counter (§4.1), but to tackle more advanced examples we will need to employ ownership-based reasoning, the subject of §5.

We write $\sqsubseteq$ for axiomatic refinement, which is justified in terms of $\sqsubseteq_{\mathsf{den}}$, and we write $\equiv$ when the refinement goes in both directions. The benefit of the setup in the previous two sections is that *axiomatic refinement is a congruence*—which is what enables us to verify implementations in a stepwise, compositional manner.

Many of the rules in Figure 3 are familiar. The top group comes from first-order and Hoare logic; we leave those rules unlabeled and use them freely. The two Dst rules, giving the interaction between nondeterminism and sequencing, are standard for a linear-time process calculus [29]. Ind is standard fixpoint induction. Frm is the frame rule from separation logic, capturing the locality of actions. Csq$_1$ is the consequence rule of Hoare logic.

The less familiar rules are still largely straightforward. Ext provides an important case where actions and assertions are equivalent: on *exact* predicates, which are satisfied by exactly one heap, and hence are deterministic as postconditions. The Str rules allow us to manipulate quantifier structure in a way reminiscent of scope extrusion in the π-calculus [25]; like the Dst rules, they express that the semantics is insensitive to *when* a nondeterministic choice

Some laws of refinement $\varphi \sqsubseteq \psi$

$$\begin{array}{lcl} \mathsf{miracle} \sqsubseteq \varphi \sqsubseteq \mathsf{abort} & & \varphi[e/x] \sqsubseteq \exists x.\varphi \sqsubseteq \varphi \\ \varphi \vee \varphi \sqsubseteq \varphi \sqsubseteq \varphi \vee \psi & & \mathsf{skip}; \varphi \equiv \varphi \equiv \varphi; \mathsf{skip} \end{array}$$

$$\begin{array}{llcl} \text{DSTL} & (\varphi_1 \vee \varphi_2); \psi & \equiv & \varphi_1; \psi \vee \varphi_2; \psi \\ \text{DSTR} & \psi; (\varphi_1 \vee \varphi_2) & \equiv & \psi; \varphi_1 \vee \psi; \varphi_2 \\ \text{STR}_1 & \exists x.\varphi; \psi & \equiv & \varphi; (\exists x.\psi) \\ \text{STR}_2 & \exists x.\, \langle \forall \overline{y} : p,\, q \rangle & \sqsubseteq & \langle \forall \overline{y} : p,\, \exists x.q \rangle \\ \text{FRM} & \langle \forall \overline{x} : p,\, q \rangle & \sqsubseteq & \langle \forall \overline{x} : p * r,\, q * r \rangle \\ \text{EXT} & \langle \forall \overline{x} : p,\, p \rangle & \equiv & \{\exists \overline{x}.p\} \quad (p \text{ exact}) \\ \text{IDM}_1 & \{p\}; \{p\} & \equiv & \{p\} \\ \text{IDM}_2 & \{\exists \overline{x}.p\}; \langle \forall \overline{x} : p,\, q \rangle & \equiv & \langle \forall \overline{x} : p,\, q \rangle \\ \text{ASM} & \langle \forall \overline{x} : p,\, q \wedge r \rangle & \equiv & \langle \forall \overline{x} : p,\, q \rangle ; [r] \quad (r \text{ pure}) \end{array}$$

$$\text{IND}\ \frac{\varphi[\psi/X] \sqsubseteq \psi}{\mu X.\varphi \sqsubseteq \psi} \qquad \text{CSQ}_1\ \frac{\forall \overline{x}.\, p \Rightarrow p' \qquad \forall \overline{x}.\, q' \Rightarrow q}{\langle \forall \overline{x} : p',\, q' \rangle \sqsubseteq \langle \forall \overline{x} : p,\, q \rangle} \qquad \text{CSQ}_2\ \frac{q \Rightarrow p}{\{p\} \sqsubseteq \{q\}}$$

NB: syntax appearing both in and outside a binder for x in a refinement (as in $\exists x.\varphi \sqsubseteq \varphi$) cannot mention x.

Figure 3.

is made. The IDM rules express the idempotence of assertions—recall that the precondition of an action acts as a kind of assertion. ASM allows us to move a guard into or out of a postcondition when that guard is *pure* (does not use emp or $\mapsto$). CSQ_2 tells us that assertions are antitonic, which follows from the permissive nature of faulting.

Theorem 2. The laws of refinement are sound: if $\varphi \sqsubseteq \psi$ then $\varphi \sqsubseteq_{\mathsf{op}} \psi$.

Proof. We prove the laws sound using the denotational semantics: we show that $\varphi \sqsubseteq \psi$ implies $\varphi \sqsubseteq_{\mathsf{den}} \psi$, which by Theorem 1 (adequacy) implies $\varphi \sqsubseteq_{\mathsf{op}} \psi$. Using the denotational semantics, the laws are almost trivial to show sound. The only nontrivial case, IND, uses the Knaster-Tarski fixpoint theorem in the standard way. $\square$

As a simple illustration of the proof rules, we have:

Lemma 3.

$$\begin{array}{rcl} & & \mathsf{get}(a, \mathit{ret}) \\ \text{definition} & \equiv & \langle \forall x : a \mapsto x,\ a \mapsto x \wedge x = \mathit{ret} \rangle \\ \text{CSQ}_1 & \sqsubseteq & \langle \forall x : a \mapsto x,\ a \mapsto x \rangle \\ \text{EXT} & \sqsubseteq & \{a \mapsto -\} \end{array}$$

This lemma shows that we can forget (abstract away) the constraint that get places on its out-parameter, allowing it to vary freely. But we cannot forget that $\mathsf{get}(a, \mathit{ret})$ faults when a is unallocated.

4.1 Example: a nonblocking counter

We now return to the example of an atomic counter:

$$\begin{array}{rcl} \mathsf{inc}(c,\ \mathit{ret}) & \triangleq & \langle \forall x : c \mapsto x,\ c \mapsto x + 1 \wedge \mathit{ret} = x \rangle \\ \mathsf{inc}'(c, \mathit{ret}) & \triangleq & \mu X.\ \exists t.\mathsf{get}(c, t); \\ & & \qquad \exists b.\mathsf{cas}(c, t, t+1, b); \\ & & \qquad \textbf{if } b = 1 \textbf{ then } [\mathit{ret} = t] \textbf{ else } X \end{array}$$

To show $\mathsf{inc}' \sqsubseteq \mathsf{inc}$, we first prove some useful lemmas about optimistic concurrency:

Lemma 4.

$$\begin{array}{rl} & \mathsf{cas}(a, t, e, b); [b = 1] \\ \text{def} \equiv & \left\langle \forall x : a \mapsto x,\ \begin{array}{l} (x \neq t \wedge b = 0 \wedge a \mapsto x) \\ \vee\, (x = t \wedge b = 1 \wedge a \mapsto e) \end{array} \right\rangle ; [b = 1] \\ \text{ASM} \equiv & \left\langle \forall x : a \mapsto x,\ \left(\begin{array}{l} (x \neq t \wedge b = 0 \wedge a \mapsto x) \\ \vee\, (x = t \wedge b = 1 \wedge a \mapsto e) \end{array} \right) \wedge b = 1 \right\rangle \\ \text{CSQ}_1 \sqsubseteq & \langle \forall x : a \mapsto x,\ a \mapsto e \wedge x = t \rangle \end{array}$$

Lemma 5. $\mathsf{cas}(a, t, e, b); [b \neq 1] \sqsubseteq \{a \mapsto -\}$.

Proof. Similar. $\square$

Corollary 1.

$$\begin{array}{r} \mu X.\exists t.\mathsf{get}(a, t); \varphi; \\ \exists b.\mathsf{cas}(a, t, e, b); \\ \textbf{if } b = 1 \textbf{ then } \psi \\ \textbf{else } X \end{array} \sqsubseteq \begin{array}{l} \mu X.\exists t.\mathsf{get}(a, t); \varphi; \\ \left(\begin{array}{l} \left\langle \forall x : \begin{array}{l} a \mapsto x, \\ a \mapsto e \wedge x = t \end{array} \right\rangle ; \psi \\ \vee\, \{a \mapsto -\}; X \end{array} \right) \end{array}$$

Proof. Expand **if**, apply DSTR, and use Lemmas 4, 5. $\square$

Lemma 6 (Optimism)**.**

$$\begin{array}{r} \mu X.\exists t.\mathsf{get}(a, t); \varphi; \\ \exists b.\mathsf{cas}(a, t, e, b); \\ \textbf{if } b = 1 \textbf{ then } \psi \\ \textbf{else } X \end{array} \sqsubseteq \begin{array}{l} (\exists t.\mathsf{get}(a, t); \varphi)^*; \\ \exists t.\mathsf{get}(a, t); \varphi; \\ \langle \forall x : a \mapsto x,\ a \mapsto e \wedge x = t \rangle ; \psi \end{array}$$

where θ^* is shorthand for $\mu X.\mathsf{skip} \vee \theta; X$. Notice, too, that the scope of quantifiers continues over line breaks. The specification on the left captures a typical optimistic, nonblocking algorithm: take a snapshot of a cell a, do some work φ, and update a if it has not changed; otherwise, loop. The specification on the right characterizes the (partial correctness) effect of the algorithm: it performs some number of unsuccessful loops, and then updates a.

Proof. Apply Corollary 1. At a high level, the result follows by induction (rule IND), using IDM_2 to remove assertions $\{a \mapsto -\}$. $\square$

Applying Lemma 6 to inc' (letting φ be skip, which we drop), we have:

$$\begin{array}{rl} & \mathsf{inc}'(c, \mathit{ret}) \\ 1 \sqsubseteq & (\exists t.\mathsf{get}(c, t))^*; \\ & \quad \exists t.\mathsf{get}(c, t); \langle \forall x : c \mapsto x,\ c \mapsto x + 1 \wedge x = t \rangle ; [\mathit{ret} = t] \\ 2 \sqsubseteq & (\exists t.\{c \mapsto -\})^*; \\ & \quad \exists t.\{c \mapsto -\}; \langle \forall x : c \mapsto x,\ c \mapsto x + 1 \wedge x = t \rangle ; [\mathit{ret} = t] \\ 3 \sqsubseteq & \{c \mapsto -\}^*; \exists t.\ \langle \forall x : c \mapsto x,\ c \mapsto x + 1 \wedge x = t \rangle ; [\mathit{ret} = t] \\ 4 \sqsubseteq & \{c \mapsto -\}^*; \exists t.\ \langle \forall x : c \mapsto x,\ c \mapsto x + 1 \wedge x = t \wedge \mathit{ret} = t \rangle \\ 5 \sqsubseteq & \{c \mapsto -\}^*; \exists t.\ \langle \forall x : c \mapsto x,\ c \mapsto x + 1 \wedge \mathit{ret} = x \rangle \\ 6 \sqsubseteq & \{c \mapsto -\}^*; \langle \forall x : c \mapsto x,\ c \mapsto x + 1 \wedge \mathit{ret} = x \rangle \\ 7 \sqsubseteq & \langle \forall x : c \mapsto x,\ c \mapsto x + 1 \wedge \mathit{ret} = x \rangle \end{array}$$

Step (1) is the application of Lemma 6. In step (2), we abstract away the gets (the snapshots) using Lemma 3. In (3), we remove the first existential, and apply IDM_2 inductively to coalesce the assertions; (4) applies ASM; (5) applies CSQ_1; (6) removes the remaining existential; (7) applies IDM_2 inductively.

The abstraction of get in step (2) is reminiscent of havoc abstraction [5], and it illustrates that the snapshots are not necessary for the safety of the algorithm (though essential for liveness).

Notice that we do not give and did not use any rules for reasoning about parallel composition. We certainly could give such rules (*e.g.*, an expansion law [18]), but that would be beside the point. Our aim is to reason about concurrent objects, which are defined by *sequential* methods that clients may choose to execute in parallel. Having proved the refinement, we can conclude that even for a concurrent client $C[-]$ we have $C[\mathsf{inc}'] \sqsubseteq C[\mathsf{inc}]$.

5. Ownership

In the previous section, we were able to reason about the non-blocking counter because it possesses a very helpful property: it works correctly *regardless of interference*. No matter what concurrent reads and writes occur to the counter, the inc' method will only modify the counter by atomically incrementing it. Such an implementation is possible because a counter can be represented in a single cell of memory, and so cas can be used to operate on the entire data structure at once. For more complex data structures, such as the stack we will study in §6, this will not be the case; such data structures cannot cope with arbitrary interference.

In this section, we will develop an ownership discipline that will enable reasoning about (lack of) interference. We will leverage the denotational semantics to give a meaning to, and find the consequences of, the ownership discipline. The resulting laws give voice to the key claims of the paper: atomicity is relative to ownership, and interference is modularized by ownership.

5.1 The discipline

The ownership discipline we have in mind is tied to the notion of a *concurrent object*, which is given by a collection of methods, one of which is a constructor. For example, imagine a counter that supported both incrementing and decrementing:

$$\begin{array}{l}\textbf{let } \mathsf{newcnt}(\mathit{ret}) = \mathsf{new}(0, \mathit{ret}) \textbf{ in}\\ \textbf{let } \mathsf{inc}(c,\ \mathit{ret}) = \cdots \textbf{ in}\\ \textbf{let } \mathsf{dec}(c,\ \mathit{ret}) = \cdots \textbf{ in } \varphi\end{array}$$

Intuitively, the representation of a counter can be described by a simple predicate: $\ell \mapsto x$. Notice that this predicate is specific to a counter located at address ℓ and with value x. A client

$$\varphi = \exists a.\mathsf{newcnt}(a); \exists b.\mathsf{newcnt}(b)$$

that called newcnt twice would expect to be left with a subheap satisfying $a \mapsto 0 * b \mapsto 0$.

More complex data structures are described by more complex predicates. Consider the specification of a thread-safe stack:

$$\begin{array}{l}\textbf{let } \mathsf{newStk}(\mathit{ret}) = \mathsf{new}(0, \mathit{ret}) \textbf{ in}\\ \textbf{let } \mathsf{push}(s, x) = \left\langle \forall h : \begin{array}{l} s \mapsto h,\\ \exists h'.\ s \mapsto h' * h' \mapsto (x, h)\end{array} \right\rangle \textbf{ in}\\ \textbf{let } \mathsf{pop}(s, \mathit{ret}) = \left\langle \forall h, h', x : \begin{array}{l} s \mapsto h * h \mapsto (x, h'),\\ s \mapsto h' \wedge \mathit{ret} = x\end{array} \right\rangle \textbf{ in } \varphi\end{array}$$

In this representation, an instance of a stack consists of at least a memory cell s which points to the head of the stack (0 if the stack is empty). The contents of the stack is given by a linked list, which can be described by the following recursive predicate:

$$\begin{array}{rcl}\mathit{list}(\ell, \epsilon) & \triangleq & \ell = 0\\ \mathit{list}(\ell, x \cdot xs) & \triangleq & \exists \ell'.\ell \mapsto (x, \ell') * \mathit{list}(\ell', xs)\end{array}$$

The second parameter to *list* is a sequence of list items; this sequence is "abstract" in the sense that it exists only at the level of the logic, not as a value in a program. Altogether, a stack located at ℓ with abstract contents x is described by the predicate $\exists a.\ \ell \mapsto a * \mathit{list}(a, x)$.

In general, the memory belonging to an instance of a concurrent object is described by a predicate $p[\ell, x]$ with free variables ℓ and x. The variable ℓ gives the location of the object, while x gives its abstract value. The predicate $\exists x.p[\ell, x]$ then describes a heap that contains an object instance at ℓ, with unknown contents. We call this latter predicate the *representation invariant* for a concurrent object, and use the metavariable I to designate such predicates. We introduce a new metavariable not just as a mnemonic, but also because we restrict representation invariants to be *precise*. A predicate is precise if, for every heap σ, there is at most one way to split $\sigma = \sigma_1 \uplus \sigma_2$ such that σ_1 satisfies the predicate. Thus, precise predicates serve to pick out ("fence" [7]) precisely the region of memory relevant to an object instance. For example, the invariant $I[\ell] = \exists a.\exists x.\ell \mapsto a * \mathit{list}(a, x)$ in some sense "discovers" the linked list associated with a stack, whatever its contents may be.

Our ownership discipline works as follows: a concurrent object is given (in the logic) a representation invariant $I[\ell]$ parameterized by location ℓ; an object instance at ℓ consists of the subheap described by $I[\ell]$. Clients may not access these subheaps directly, but must invoke methods of the object instead. Each method is parameterized by a location ℓ of the object to operate on, and is given access only to the subheap described by $I[\ell]$. Since multiple methods may be invoked on the same instance concurrently, this memory is shared between invocations. But because the object's methods are the *only* code with access to this memory, we may assume that any concurrent interference is due to one of these methods.

When a method begins executing, its view of the heap consists solely of some object instance $I[\ell]$. As it executes, the method may in addition allocate private memory, which is initially free from interference. If, however, the private memory is made reachable from the object instance, so that it falls into the region described by $I[\ell]$, it at that point becomes shared, and subject to interference. Conversely, memory unlinked from the object instance becomes private to the method. An implementation of push, for example, will first allocate a private linked list node, and only later actually link it into the list.

5.2 Fenced refinement

Now for the punchline: our ownership discipline enables partly-sequential reasoning about method bodies, through *fenced refinement* $I, \theta \vdash \varphi \sqsubseteq \psi$. As above, I describes a specific object instance; we leave the location ℓ implicit. Fenced refinement says that φ refines ψ under the assumption that all memory is either part of the shared instance described by I, or private. Moreover, interference on the shared memory is bounded by the specification θ (the "rely" [12]). Here are the key rules:

Some laws of fenced refinement $\qquad I, \theta \vdash \varphi \sqsubseteq \psi$

INV
$$I, \theta \vdash \{I\} \equiv \mathsf{skip}$$

LIFT
$$\frac{\varphi \sqsubseteq \psi}{I, \theta \vdash \varphi \sqsubseteq \psi}$$

SEQL
$$I, \theta \vdash \langle \forall \overline{x} : p,\ q \rangle ; \langle \forall \overline{x} : q * p',\ r \rangle \sqsubseteq \{I * \exists \overline{x}.p\}; \langle \forall \overline{x} : p * p',\ r \rangle$$

STAB
$$\frac{\theta \sqsubseteq \langle \forall \overline{x} : q,\ q \rangle}{I, \theta \vdash \langle \forall \overline{x} : p,\ q \rangle ; \{\exists \overline{x}.q\} \sqsubseteq \langle \forall \overline{x} : p,\ q \rangle}$$

INV expresses that the representation invariant I always holds, so asserting it will always succeed. We check that the method itself maintains the invariant elsewhere, in the DATA$_2$ rule.

LIFT reflects the fact that fenced refinement is more permissive than standard refinement.

In fenced refinement, any data that is not part of the data structure instance fenced by I is private, and hence can be reasoned about sequentially. Suppose that, at some point, we know that $I * p$; that is, our view of the heap includes both a shared object instance described by I, and some other memory described by p. By our ownership discipline, we know that p is private. In this case, the execution of an action like $\langle p,\ q \rangle$ will *not* be visible to other threads, because it is operating on private memory. The SEQ rules permit atomicity abstraction: two sequenced atomic actions can be combined if one of them only operates on private data. We give one rule, SEQL, which combines two actions when the first (the Left) is unobservable. Notice that SEQL introduces the assertion $I * \exists \overline{x}.p$

as a verification condition for showing that the data operated on by the first action really is private. We omit the similar SEQR rule.

Finally, STAB allows as assertion $\{q\}$ to be removed if it is *stably* satisfied after an action. That is, the predicate q must be established by an action, and once established, must be maintained under any interference θ.

In addition to those rules, fenced refinement is also nearly a congruence: if $I, \theta \vdash \varphi \sqsubseteq \psi$ then $I, \theta \vdash C[\varphi] \sqsubseteq C[\psi]$ for any context C that does not contain parallel composition. It is not a congruence for parallel composition because of the sequential reasoning it permits on private data. Since we use fenced refinement only to reason about the method bodies implementing a concurrent object—which we assume to be sequential—this is all we need.

To give a semantics for fenced refinement, we want to *ignore* certain aspects of the denotational semantics. In some cases we throw away traces: for example, traces where the environment does not obey the rely θ should be discarded. A deeper issue is the distinction between private and shared memory. Because private memory cannot be observed or interfered with by concurrent threads, *we do not model it with a trace*. Instead, we view private memory in terms of "input-output" behavior, recording its value only at the beginning and end of a computation φ. As a consequence, sequential reasoning (*e.g.* SEQL) is valid for steps involving only private memory.

In short, the fenced semantics of method bodies is a hybrid between trace semantics and input-output relations. We thus define:

$$\text{FENCEDTRACE} \triangleq \Sigma^{\sharp} \times \text{TRACE} \times \Sigma^{\sharp}$$

A *fenced trace* contains three components: the initial state of private memory, a trace involving only the shared memory, and the final state of private memory. Suppose φ is a fragment of a method body. We calculate its fenced traces as follows:

Fenced projection $[\![I, \theta \vdash \varphi]\!]^\rho \subseteq \text{FENCEDTRACE}$

$$[\![I, \theta \vdash \varphi]\!]^\rho \triangleq \left\{ (\text{fst}(u),\ t,\ \text{lst}(u)) \ :\ \begin{array}{l} t \uplus u \in [\![\varphi]\!]^\rho \\ \sigma \in t \Rightarrow \sigma, \rho \models I \\ u \text{ seq} \\ t \text{ rely}(\theta, \rho) \end{array} \right\}$$

where

$$\epsilon \uplus \epsilon \triangleq \epsilon \qquad (o_1, o_1')t \uplus (o_2, o_2')u \triangleq (o_1 \uplus o_2, o_1' \uplus o_2')(t \uplus u)$$

$$\frac{}{(o, o') \text{ seq}} \qquad \frac{u \text{ seq}}{(o, \text{fst}(u))u \text{ seq}}$$

$$\frac{}{(o, o') \text{ rely}(\theta, \rho)} \qquad \frac{t \text{ rely}(\theta, \rho) \qquad (o', \text{fst}(t)) \in [\![\theta]\!]^\rho}{(o, o')t \text{ rely}(\theta, \rho)}$$

The functions $\text{fst}(u)$ and $\text{lst}(u)$ return the first and last heaps (or outcomes) that appear in a trace u.

Fenced projection separates each trace of φ into a part t dealing with the shared state and a part u dealing with the rest of the heap, which is presumed to be method-private. We do this by lifting $\uplus$ to traces. Notice that t and u have the same length and represent the same moves of φ; we are just splitting the heap involved in each move into a shared part and a private part.

The requirement that t satisfies I throughout is how we force t to represent precisely the shared state. This is where the fact that I is precise comes in: it means that, if (σ, σ') was a move in φ, there will be at most one subheap of σ and σ' satisfying I. If no such t can be found, the invariant must have been broken—and we do not include such traces. In other words, we are assuming that both φ and its environment maintain the invariant. Both of these assumptions are checked when we try to derive a standard refinement from a fenced refinement (DATA$_2$ below).

More generally, recall that the transition trace semantics generates traces where the environment may modify the heap in an arbitrary way between each atomic step. In fenced projection, we do not want to include all such traces, because we are making some assumptions about what the environment can and cannot do. Thus, we filter out any traces in which the environment breaks our rely assumption. The rely is, of course, only relevant to the shared state described by t. We also require the private trace u to be *sequential*, meaning that we filter any traces where the private data appears to have been interfered with at all.

Fenced projection gives us the fenced traces of a specification. *Fenced refinement* just compares these traces:

Semantics of fenced refinement $I, \theta \models \varphi \sqsubseteq \psi$

$$I, \theta \models \varphi \sqsubseteq \psi \quad \text{iff} \quad \forall \text{ closing } \rho.\ [\![I, \theta \vdash \varphi]\!]^\rho \subseteq [\![I, \theta \vdash \psi]\!]^\rho$$

Theorem 3. The laws of fenced refinement are sound: if $I, \theta \vdash \varphi \sqsubseteq \psi$ then $I, \theta \models \varphi \sqsubseteq \psi$.

Proof. The semantics of fenced refinement was designed to make this theorem straightforward. Each law corresponds to one of the basic assumptions of fenced projection: INV, that the invariant is satisfied; STAB, that the rely is obeyed; SEQL, that the private traces are sequential. The LIFT rule expresses that fenced projection is monotonic. □

5.3 Enforcing ownership: hiding

So far, fenced refinement is just a fantasy: we have no way to deduce standard refinements from fenced refinements. To do so, we have to justify the assumptions made in fenced projection. The ownership discipline described in §5.1 is the desired policy; now we turn to the mechanism for carrying it out.

Consider the following specification:

$$\textbf{let } f(\mathit{ret}) = \langle \mathsf{emp},\ \mathit{ret} \mapsto 0 \rangle \textbf{ in } \exists x. f(x); \mathsf{put}(x, 1)$$

Here we have a simple concurrent object with a constructor, f, and no methods. Its client, however, violates our ownership discipline by directly modifying an object instance, using put. If the assumptions of fenced refinement are to be met, such clients must somehow be ruled out.

Our solution is to introduce an abstraction barrier. We extend the language of specifications to include **abs** $\alpha.\varphi$, which binds the *abstraction variable* α in φ, and we likewise extend heaps to include *abstract cells* $\ell \overset{\alpha}{\mapsto} e$. Each location in the heap is either concrete ($\mapsto$) or abstract ($\overset{\alpha}{\mapsto}$), so the predicate $\ell \mapsto - \wedge \ell \overset{\alpha}{\mapsto} -$ is unsatisfiable. The purpose of abstract cells is to deny clients access to concurrent objects. Revisiting the misbehaved client above, we could instead write:

$$\textbf{abs } \alpha.\textbf{let } f(\mathit{ret}) = \left\langle \mathsf{emp},\ \mathit{ret} \overset{\alpha}{\mapsto} 0 \right\rangle \textbf{ in } \exists x. f(x); \mathsf{put}(x, 1)$$

In this case, the client will *fault* when attempting the put, because put operates on concrete cells, but f allocates an abstract cell.

In general, the concrete representation of an object may involve multiple memory cells. When we introduce an abstraction variable, we also have an opportunity to switch to a more abstract representation, as in the following rule (applicable when $r[\ell, -]$ is precise):

DATA$_1$

$$\begin{array}{rl} & \textbf{let } \overline{f_i(\ell, x) = \langle \forall y : p_i * r[\ell, e_i], \qquad q_i * r[\ell, e_i'] \rangle} \textbf{ in } \varphi \\ \sqsubseteq & \textbf{abs } \alpha.\ \textbf{let } f_i(\ell, x) = \left\langle \forall y : p_i * \ell \overset{\alpha}{\mapsto} e_i,\ q_i * \ell \overset{\alpha}{\mapsto} e_i' \right\rangle \textbf{ in } \varphi \end{array}$$

As before, the predicate $r[\ell, z]$ is meant to describe an object instance located at ℓ with abstract contents z. Notice that the client

$$\frac{\begin{array}{l}\textsc{Data}_2\\ r[\ell,-]\ \text{precise} \qquad p_i\ \text{pure} \qquad r[\ell,-],\ (\bigvee\overline{\theta_i}) \vdash \varphi_i \sqsubseteq \theta_i \qquad \theta_i \sqsubseteq \langle \forall y : r[\ell,y],\ r[\ell,e_i]\wedge p_i\rangle\end{array}}{\begin{array}{l}\textbf{let}\ f(\ell) = \langle \mathsf{emp},\ r[\ell,e]\rangle\ \textbf{in}\\ \textbf{let}\ \overline{g_i(\ell,x) = \varphi_i}\ \textbf{in}\ \psi\end{array} \sqsubseteq \begin{array}{l}\textbf{abs}\ \alpha.\textbf{let}\ f(\ell) = \left\langle \mathsf{emp},\ \ell \overset{\alpha}{\mapsto} e\right\rangle\ \textbf{in}\\ \textbf{let}\ \overline{g_i(\ell,x) = \left\langle \forall y : \ell \overset{\alpha}{\mapsto} y,\ \ell \overset{\alpha}{\mapsto} e_i \wedge p_i\right\rangle}\ \textbf{in}\ \psi\end{array}}$$

Figure 4. Rely/guarantee reasoning

φ is unconstrained, except for the implicit constraint that α cannot appear free in it. The idea is that, in applying this abstraction rule, we swap a concrete version of a concurrent object for an abstract one. Cases where the client might have accessed a concrete object instance directly will, in the context of the abstract object, result in a fault. Since a faulting specification is permissive (§3), this means that behaviors where the client performed such an access are irrelevant for this refinement; they are masked out by the fault. On the other hand, to ultimately verify that, say, **abs** $\alpha.\varphi \sqsubseteq \langle p,\ q\rangle$, it will be necessary to show that the client is well-behaved as long as the precondition p is satisfied. In short: when we introduce hiding, we can assume the client is well-behaved; when we verify a full program, we must check this assumption.

We define $[\![\textbf{abs}\ \alpha.\varphi]\!]^\rho$ as follows:

$$\begin{array}{l}\{s \uplus u \qquad\qquad : \quad s \uplus_\alpha t \in [\![\varphi]\!]^\rho,\ t, u\ \text{seq},\ \text{fst}(t,u) = \emptyset\}^\dagger\\ \cup \left\{ (s_1 \uplus u)(\sigma, \lightning) : s_1 s_2 \uplus_\alpha t \in [\![\varphi]\!]^\rho,\ t,u\ \text{seq},\ \begin{array}{l}\text{fst}(t,u) = \emptyset,\\ \text{lst}(u) \not\subseteq \sigma\end{array} \right\}^\dagger\end{array}$$

The notation $s \uplus_\alpha t$ denotes $s \uplus t$, but is only defined when *no* heap in s contains $\overset{\alpha}{\mapsto}$ cells, and *every* heap in t consists only of $\overset{\alpha}{\mapsto}$ cells (so it is not commutative!). Thus, as in fenced refinement, we are isolating in trace t the contents of the heap specific to an abstract variable.

We split out the α resources because an implementation should be able to use concrete resources instead. The role of **abs** is to ensure that these concrete resources are not interfered with:

- In the first clause of the definition, we swap the abstract trace t with a concrete trace u. We require that both traces are sequential (hence free from interference). We also require the traces to start with the empty heap, because initially no object instances are allocated.
- In the second clause of the definition, we introduce new fault steps whenever the concrete resources are interfered with. We truncate the trace at the point where interference occurs, and add a faulting move. Interference has occurred if $\text{lst}(u) \not\subseteq \sigma$, meaning that the subheap containing the concrete resources has changed.

The introduction of faults in this semantics is akin to runtime contract checking; an alternative would have been to introduce a type system or other hiding mechanism.

As before, the effect of the introduced faults is to mask out behavior when the client or environment is ill-behaved. To reiterate: **abs** allows reasoning about an object while assuming its client is well-behaved, but in order to verify a whole program (that includes the client) one must show that it does not fault, which entails showing that particular client to be well-behaved.

Finally, we have the $\textsc{Data}_2$ rule in Figure 4, which allows us to derive standard refinements from fenced refinements, ensuring that the assumed ownership discipline is actually followed; "ownership is in the eye of the asserter" [23].

In $\textsc{Data}_2$, we are working with a concurrent object with constructor f and methods g_i. The predicate $r[\ell,y]$ describes a concrete object instance at location ℓ with abstract contents y. As in $\textsc{Data}_1$, the abstracted version of the object works on abstract cells $\ell \overset{\alpha}{\mapsto} y$ instead.

We introduce θ_i as a midpoint between φ_i and the specification $\overline{\langle \forall y : r[\ell,y],\ r[\ell,e_i] \wedge p_i\rangle}$ because we want to keep the rely $\vee\overline{\theta_i}$ as simple as possible. In particular, methods usually operate on small portions of the object, while $r[\ell,y]$ refers to the entire object instance.

The definition of **abs** ensures that the client or environment of an object cannot interfere with it. But fenced refinement also assumes that (1) methods do not violate the invariant $r[\ell,-]$ and (2) methods do not violate the rely $\vee\overline{\theta_i}$.

- The invariant is maintained because each method body φ_i is a fenced refinement $\langle \forall y : r[\ell,y],\ r[\ell,e_i]\wedge p_i\rangle$, which clearly maintains the invariant.
- The rely is never violated because each method body φ_i is a fenced refinement of θ_i, and the rely permits the behavior of any of the θ_i's.

A key feature of $\textsc{Data}_2$ is its provision for modular and dynamic rely/guarantee reasoning. It is *modular* because we have isolated the memory involved (to $r[\ell,-]$) and the code involved (each φ_i)—we do not constrain the clients ψ, nor the contexts in which the data refinement holds. It is *dynamic* because it encompasses arbitrary allocation of new data structure instances—we get rely/guarantee reasoning for each individual instance, even though we do not know how many instances the client ψ will allocate.

As given, $\textsc{Data}_2$ only applies to methods whose action on the shared state is given in a single atomic step. The rule can easily be extended to allow methods which also take an arbitrary number of steps refining $\langle \forall y : r[\ell,y],\ r[\ell,y]\rangle$, which make internal adjustments to the object instance (often known as "helping" [10]) but look like skip to the client. We will not need this for our case study below, but it is necessary for related data structures such as queues.

The soundness proof for $\textsc{Data}_2$ is sketched in §7.

6. Case study: Treiber's nonblocking stack

Using fenced refinement, we will be able to verify a version of Treiber's nonblocking stack [26]:

```
newStk(ret)  =  new(0, ret)

push(s, x)   =  ∃n.new((x, 0), n);
                μX. ∃t.get(s, t);
                    put_2(n, t);
                    ∃b.cas(s, t, n, b);
                    if b = 0 then X

pop(s, ret)  =  μX. ∃t.get(s, t); {t ≠ 0};
                    ∃n.get_2(t, n);
                    ∃b.cas(s, t, n, b);
                    if b = 0 then X else get_1(t, ret)
```

The specifications get_i and put_i operate on the i^{th} component of pairs, *e.g.*, $\mathsf{put}_2(a,x) \triangleq \langle \forall y : a \mapsto (y,-),\ a \mapsto (y,x)\rangle$. We have simplified pop so that it asserts that the stack is nonempty.

Stacks provide two new challenges compared to counters. First, the loop in push modifies the heap *every* time it executes, by calling put_2, rather than just at a successful cas; this makes atomicity nontrivial to show. Second, pop has a potential "ABA" problem [10]: it assumes that if the head pointer s is unchanged (*i.e.* equal to t at the cas), then the tail of that cell is unchanged (*i.e.* equal to n at the cas). Intuitively this assumption is justified because stack cells are never changed or deallocated once they are introduced;[2] we must make such an argument within the logic.

First, we introduce the following procedures:

$$\begin{aligned}\mathsf{getHd}(s, \mathit{ret}) &= \langle \forall x : s \mapsto x,\ s \mapsto x \wedge \mathit{ret} = x\rangle\\ \mathsf{pushHd}(s,t,n) &= \langle \forall x : s \mapsto x,\ s \mapsto n \wedge x = t\rangle\\ \mathsf{popHd}(s,t,n) &= \langle \forall x : s \mapsto x,\ s \mapsto n \wedge x = t\rangle\end{aligned}$$

We apply Lemma 6 (Optimism) to push and pop:[3]

$$\begin{aligned}\mathsf{push}(s,x) \sqsubseteq\ & \exists n.\mathsf{new}((x,0),n);\\ & (\exists t.\mathsf{getHd}(s,t); \mathsf{put}_2(n,t))^*;\\ & \exists t.\mathsf{getHd}(s,t); \mathsf{put}_2(n,t);\\ & \mathsf{pushHd}(s,t,n)\end{aligned}$$

$$\begin{aligned}\mathsf{pop}(s,\mathit{ret}) \sqsubseteq\ & (\exists t.\mathsf{getHd}(s,t); \{t \neq 0\}; \exists n.\mathsf{get}_2(t,n))^*;\\ & \exists t.\mathsf{getHd}(s,t); \{t \neq 0\}; \exists n.\mathsf{get}_2(t,n);\\ & \mathsf{popHd}(s,t,n)\end{aligned}$$

Notice that pop, after recording a pointer t to the current head node and checking that it is non-null, attempts to read the *tail* of that node (using get_2). In a sequential setting this would be unproblematic; in our concurrent setting, we must worry about interference. What if, between taking the snapshot t and getting its tail, the node t was popped from the stack, or worse, deallocated? Clearly the push and pop methods will give no such interference, but we must find an appropriate representation invariant I and interference description θ to explain that to the logic.

It turns out to be slightly tricky to formulate an appropriate representation invariant, because part of what we want to assert—that cells, even when popped, are neither altered nor deallocated—can involve memory that is no longer reachable from the head of the stack. In order to state the invariant, we need some way of remembering the addresses of cells which *used* to be part of the stack, but no longer are. To do this, we will introduce an internal abstract value α which caries a "ghost" parameter A, using $\textsc{Data}_1$:

$$\begin{aligned}\mathsf{getHd}(s,\mathit{ret}) &= \left\langle \forall x, A : \begin{array}{l} s \overset{\alpha}{\mapsto} (x,A),\\ s \overset{\alpha}{\mapsto} (x,A) \wedge \mathit{ret} = x\end{array}\right\rangle\\ \mathsf{pushHd}(s,t,n) &= \left\langle \forall x, A : \begin{array}{l} s \overset{\alpha}{\mapsto} (x,A),\\ s \overset{\alpha}{\mapsto} (n,A) \wedge x = t\end{array}\right\rangle\\ \mathsf{popHd}(s,t,n) &= \left\langle \forall x, A : \begin{array}{l} s \overset{\alpha}{\mapsto} (x,A),\\ s \overset{\alpha}{\mapsto} (n,x\cdot A) \wedge x = t\end{array}\right\rangle\end{aligned}$$

In order to apply $\textsc{Data}_1$, we have used the predicate $p[\ell,(x,A)] = \ell \mapsto x$. Notice that the parameter A does not appear in the concrete predicate. The idea is that the abstract A is a sequence of addresses of popped cells, while the concrete representation of A is simply—nothing!

We can now give a representation invariant I for stacks, along with a bound on interference, θ:

$$I[\ell] \triangleq \exists n.\exists A.\ \ell \overset{\alpha}{\mapsto} (x,A) * \mathit{cells}(A) * \exists x.\mathit{list}(n,x)$$

$$\begin{aligned}\mathit{cells}(\epsilon) &\triangleq \mathsf{emp}\\ \mathit{cells}(x\cdot A) &\triangleq x \mapsto - * \mathit{cells}(A)\end{aligned}$$

[2] We assume garbage collection here, but we can also verify a stack that manages its own memory using hazard pointers [16].

[3] Technically, for pop, we need a slight variant of the lemma allowing the existential $\exists n$ to scope over the cas.

$$\begin{array}{rl} & \mathsf{push}(s,x)\\ 1 \sqsubseteq & \exists n.\mathsf{new}((x,0),n);\\ & (\exists t.\mathsf{getHd}(s,t); \mathsf{put}_2(n,t))^*;\\ & \exists t.\mathsf{getHd}(s,t); \mathsf{put}_2(n,t); \mathsf{pushHd}(s,t,n)\\ 2 \sqsubseteq & \exists n.\mathsf{new}((x,0),n); (\exists t.\mathsf{put}_2(n,t))^*;\\ & \exists t.\mathsf{put}_2(n,t); \mathsf{pushHd}(s,t,n)\\ 3 \sqsubseteq & \exists n.\mathsf{new}((x,-),n); \exists t.\mathsf{put}_2(n,t); \mathsf{pushHd}(s,t,n)\\ 4 \sqsubseteq & \exists n.\exists t.\mathsf{new}((x,-),n); \mathsf{put}_2(n,t); \mathsf{pushHd}(s,t,n)\\ 5 \sqsubseteq & \exists n.\exists t.\mathsf{new}((x,t),n); \mathsf{pushHd}(s,t,n)\\ 6 \sqsubseteq & \exists n.\exists t.\left\langle \forall z, A : \begin{array}{l} s \overset{\alpha}{\mapsto} (z,A),\\ s \overset{\alpha}{\mapsto} (n,A) * n \mapsto (x,t) \wedge z = t\end{array}\right\rangle\\ 7 \sqsubseteq & \left\langle \forall z, A : \begin{array}{l} s \overset{\alpha}{\mapsto} (z,A),\\ \exists n.\exists t.s \overset{\alpha}{\mapsto} (n,A) * n \mapsto (x,t) \wedge z = t\end{array}\right\rangle\\ 8 \sqsubseteq & \left\langle \forall z, A : \begin{array}{l} s \overset{\alpha}{\mapsto} (z,A),\\ \exists n.s \overset{\alpha}{\mapsto} (n,A) * n \mapsto (x,z)\end{array}\right\rangle \end{array}$$

Figure 5. High-level proof of push

$$\begin{aligned}\theta \triangleq\ & \left\langle \forall x, A : \begin{array}{l} \ell \overset{\alpha}{\mapsto} (x,A),\\ \exists n.\ \ell \overset{\alpha}{\mapsto} (n,A) * n \mapsto (-,x)\end{array}\right\rangle\\ \vee\ & \left\langle \forall x, x_1, x_2, A : \begin{array}{l} \ell \overset{\alpha}{\mapsto} (x,A) * x \mapsto (x_1,x_2),\\ \ell \overset{\alpha}{\mapsto} (x_2, x\cdot A) * x \mapsto (x_1,x_2)\end{array}\right\rangle\end{aligned}$$

Notice that θ describes interference only in terms of the effect on the head of the stack. Implicitly (by framing) this means that the rest of the stack described by I—including the heap cells given by A—remain invariant under interference.

6.1 Proving push

We verify push using the refinements in Figure 5, which are fenced by I and θ. Step (1) just restates what we have done already. In step (2), we abstract getHd into the assertion $\{s \mapsto -\}$ (as in §4.1), and then apply $\textsc{Inv}$ to abstract these assertions to skip (because they are implied by the invariant). In steps (3) and (5) we use $\textsc{SeqL}$ to merge the calls to new and put_2 into a single atomic action, new; this introduces an assertion $\{I * \mathsf{emp}\}$, again implied by the invariant. Step (4) is just $\textsc{Str}_2$. Step (6) again applies $\textsc{SeqL}$, again producing a trivial assertion which we remove with $\textsc{Inv}$. Step (7) applies $\textsc{Str}_1$, and step (8) is by $\textsc{Csq}_1$.

6.2 Proving pop

We do not give the full proof of pop, but instead we show one important step in very careful detail (Figure 6). Recall that in pop, we take *two* snapshots every time we loop: one of the address of the current head node, and one of its tail. We will show that, although these snapshots take place at distinct points in time, they may as well have happened in a single atomic step. The justification is that no interference could change the tail between the two steps.

We have labeled each step with the rule it applies, except for the step (*), which applies the derived rule $\langle \forall \overline{x} : p,\ q\rangle \sqsubseteq \langle \forall \overline{x} : p \wedge r,\ q \wedge r\rangle$ for r pure. This rule follows from the frame rule and $\textsc{Csq}_1$, because for pure r we have $p * r \iff p \wedge r$.

The proof mostly consists in structural steps which we give for illustrative purposes. These steps serve to introduce enough names and frame to actually say what we want to say about the snapshots. The (*) step strengthens the precondition of the second snapshot by asserting that it has a fixed value t_2. The $\textsc{Stab}$ step actually demonstrates that the tail of t has a stable value; strictly speaking, it requires a use of $\textsc{Csq}_2$ to strengthen the assertion so that it matches the postcondition.

$$
\begin{array}{rcl}
 & & \exists t.\mathsf{getHd}(s,t); \exists n.\mathsf{get}_2(t,n); \varphi \\
\text{defn} & \equiv & \exists t.\left\langle \forall x, A : s \overset{\alpha}{\mapsto} (x,A),\ s \overset{\alpha}{\mapsto} (x,A) \wedge t = x \right\rangle ;\ \exists n.\left\langle \forall y_1, y_2 : t \mapsto (y_1,y_2),\ t \mapsto (y_1,y_2) \wedge y_2 = n \right\rangle ; \varphi \\
\textsc{Frm} & \sqsubseteq & \exists t.\left\langle \forall x, x_1, x_2, A : p,\ p \wedge t = x \right\rangle ;\ \exists n.\left\langle \forall y_1, y_2 : t \mapsto (y_1,y_2),\ t \mapsto (y_1,y_2) \wedge y_2 = n \right\rangle ; \varphi \\
\textsc{Csq}_1 & \sqsubseteq & \exists t.\left\langle \forall x, x_1, x_2, A : p,\ p \wedge t = x \wedge x_2 = x_2 \right\rangle ;\ \exists n.\left\langle \forall y_1, y_2 : t \mapsto (y_1,y_2),\ t \mapsto (y_1,y_2) \wedge y_2 = n \right\rangle ; \varphi \\
\exists\ \text{intr} & \sqsubseteq & \exists t.\exists t_2.\left\langle \forall x, x_1, x_2, A : p,\ p \wedge t = x \wedge t_2 = x_2 \right\rangle ; \exists n.\left\langle \forall y_1, y_2 : t \mapsto (y_1,y_2),\ t \mapsto (y_1,y_2) \wedge y_2 = n \right\rangle ; \varphi \\
(*) & \sqsubseteq & \exists t.\exists t_2.\left\langle \forall x, x_1, x_2, A : p,\ p \wedge t = x \wedge t_2 = x_2 \right\rangle ; \exists n.\left\langle \forall y_1, y_2 : t \mapsto (y_1,y_2) \wedge y_2 = t_2,\ t \mapsto (y_1,y_2) \wedge y_2 = n \wedge y_2 = t_2 \right\rangle ; \varphi \\
\textsc{Csq}_1 & \sqsubseteq & \exists t.\exists t_2.\left\langle \forall x, x_1, x_2, A : p,\ p \wedge t = x \wedge t_2 = x_2 \right\rangle ; \exists n.\left\langle \forall y_1, y_2 : t \mapsto (y_1,y_2) \wedge y_2 = t_2,\ t \mapsto (y_1,y_2) \wedge y_2 = t_2 \wedge t_2 = n \right\rangle ; \varphi \\
\textsc{Asm} & \equiv & \exists t.\exists t_2.\left\langle \forall x, x_1, x_2, A : p,\ p \wedge t = x \wedge t_2 = x_2 \right\rangle ; \exists n.\left\langle \forall y_1, y_2 : t \mapsto (y_1,y_2) \wedge y_2 = t_2,\ t \mapsto (y_1,y_2) \wedge y_2 = t_2 \right\rangle ; [t_2 = n]; \varphi \\
\textsc{Ext} & \equiv & \exists t.\exists t_2.\left\langle \forall x, x_1, x_2, A : p,\ p \wedge t = x \wedge t_2 = x_2 \right\rangle ; \exists n.\{t \mapsto (-,t_2)\}; [t_2 = n]; \varphi \\
\textsc{Str} & \equiv & \exists t.\exists t_2.\exists n.\left\langle \forall x, x_1, x_2, A : p,\ p \wedge t = x \wedge t_2 = x_2 \right\rangle ; \{t \mapsto (-,t_2)\}; [t_2 = n]; \varphi \\
\textsc{Stab} & \equiv & \exists t.\exists t_2.\exists n.\left\langle \forall x, x_1, x_2, A : p,\ p \wedge t = x \wedge t_2 = x_2 \right\rangle ; [t_2 = n]; \varphi \\
\text{Asm} & \equiv & \exists t.\exists t_2.\exists n.\left\langle \forall x, x_1, x_2, A : p,\ p \wedge t = x \wedge t_2 = x_2 \wedge t_2 = n \right\rangle ; \varphi \\
\textsc{Csq}_1 & \sqsubseteq & \exists t.\exists t_2.\exists n.\left\langle \forall x, x_1, x_2, A : p,\ p \wedge t = x \wedge n = x_2 \right\rangle ; \varphi \\
\therefore & \sqsubseteq & \exists t.\exists n.\left\langle \forall x, x_1, x_2, A : p,\ p \wedge t = x \wedge n = x_2 \right\rangle ; \varphi \qquad \text{where } p \triangleq s \overset{\alpha}{\mapsto} (x,A) * x \mapsto (x_1,x_2)
\end{array}
$$

Figure 6. Detailed atomicity abstraction for pop

7. Soundness of $\textsc{Data}_2$

As with the other refinement laws, we prove $\textsc{Data}_2$ by means of the denotational model. However, unlike the other laws, this proof is nontrivial.

In order to prove the law, we first show that if the hypotheses of the law hold, then

$$\textbf{let } f(\ell) = \langle \mathsf{emp},\ r[\ell, e] \rangle \textbf{ in let } \overline{g_i(\ell, x) = \varphi_i} \textbf{ in } \psi$$

simulates

$$\textbf{let } f(\ell) = \left\langle \mathsf{emp},\ \ell \overset{\alpha}{\mapsto} (e) \right\rangle \textbf{ in}$$
$$\overline{\textbf{let } g_i(\ell, x) = \left\langle \forall y : \ell \overset{\alpha}{\mapsto} (y),\ \ell \overset{\alpha}{\mapsto} (e_i) \wedge p_i \right\rangle} \textbf{ in } \psi$$

Simulation relates the concrete traces of the first specification to the abstract traces of the second. It is a generalization of fenced refinement: instead of dealing with a single object instance, we must track the arbitrary instances that may arise as a specification executes. For each object instance, we assume that the environment obeys the rely.

We prove the simulation result by induction on the structure of the *client* ψ. The noninductive cases include method calls, which interact with the concrete objects in known ways, and client actions, which cannot interact with the concrete objects. For the latter, we use the fact that in the second specification, the concrete objects do not exist at all—they are held abstract. Thus, if the client (which does not mention α) attempts to interact with them, it will fault, which as usual is permissive.

For inductive cases, we are assured inductively that the client itself never breaks the rely.

This simulation tells us that the neither the methods nor the client can break the rely condition. To prove $\textsc{Data}_2$ we introduce **abs**, which further guarantees that the environment will not interfere at all.

An important lemma for showing simulation is *locality*:

Definition 2. T is *local* if, whenever $t(\sigma_1 \uplus \sigma_2, \sigma')u \in T$ either

- $\sigma' = \sigma_1' \uplus \sigma_2$ and $t(\sigma_1, \sigma_1')u \in T$ or
- $t(\sigma_1, \lightning) \in T$.

Lemma 7 (Locality)**.** For every φ, ρ, the set $[\![\varphi]\!]^\rho$ is local (assuming we take the fixpoint μ over only local trace sets).

Locality captures the idea that if, at some point, a specification is given fewer resources to execute with it will either fault, or it did not need those resources (so they are part of the frame for that step). We use this lemma when reasoning about steps that the client takes: we know that if we remove the concrete object instances, either the client would fault (and hence was ill-behaved) or else did not modify those instances.

The details of this and other proofs can be found online at

http://www.ccs.neu.edu/home/turon/sepref/

8. Evaluation and related work

In the last several years, there has been stunning progress in the formal verification of fine-grained concurrent data structures [4, 6, 9, 27], giving logics appropriate for hand verification as well as automated verification. We have sought in this work to clarify and consolidate this literature—the linchpin for us being the connection between ownership, atomicity, and interference. Our main contribution lies in making this connection, formalizing it with a new semantics for separation logic, and formulating proof rules based on the idea. While the logic resulting from our work offers some significant new features, more experience using it is required before its practicality or applicability can be compared to other approaches.

The most closely related work is "Concurrent Abstract Predicates," (CAP, [4]), a recent paper that also seeks to modularize reasoning about interference by using ownership and separation logic. CAP takes a radically different approach toward atomicity: rather than proving linearizability or refinement, in CAP one always uses *self-stable* predicates which are invariant under internal interference—thus sidestepping atomicity issues entirely. By contrast, we have focused on the semantic issues related to both atomicity and interference, and have shown both in terms of semantics and proof rules how these issues interact. It should be possible to apply our semantic insights to explain CAP in terms of contextual refinement as we have done.

Separation logic has also been used to localize rely/guarantee reasoning in Feng's work [7], from which we borrow the "fence" terminology. However, it does not appear possible to use that technique to both localize reasoning about interference to a group of methods and also let clients make use of the methods.

The *abstract predicates* in CAP are related to Bierman and Parkinson's work [24] investigating data abstraction in a sequential separation logic setting. In particular, Bierman and Parkinson use abstract predicates whose definition is known to the data structure implementation, but opaque to the client—a form of second-order existential quantification. There is clearly a close relationship to our abstract resources, but we do not define **abs** in terms of second-order quantification, because we need the ability to introduce faults and require sequentiality. The connections between these approaches certainly merits further investigation.

The basic form of our calculus clearly bears resemblance to Elmas *et al.*'s calculus of atomic actions [5, 6]. The key idea in that work is to combine Lipton's technique of *reduction* [14] for enlarging the grain of atomicity, with *abstraction* (*e.g.* weakening a Hoare triple) on atomic actions. The authors demonstrate that the combination of techniques is extremely powerful; they were able to automate their logic and use it to verify a significant number of data structures and other programs. A significant difference between their calculus and ours is what refinement entails, semantically. For them, refinement is ultimately about the *input-output relation* of a program, where for us, it is about the reactive, trace-based semantics. The distinction comes down to this: is refinement a congruence for parallel composition? For us it is, and this means that our system as a whole is compositional. We also demonstrate that reduction is not necessary for atomicity refinement, at least for the class of examples we have considered; we instead use ownership-based reasoning. Finally, aside from these points, we have shown how to incorporate separation logic into a calculus of atomic actions, enabling cleaner reasoning about the heap.

A significant departure in our work is that we have dispensed with linearizability, which is usually taken as the basic correctness condition for the data structures we are studying [11]. Here we are influenced by Filipović *et al.* [8], who point out that "programmers expect that the behavior of their program does not change whether they use experts' data structures or less-optimized but obviously-correct data structures." The authors go on to show that, if we take refinement as our basic goal, we can view linearizability as a sound (and sometimes complete) *proof technique*. But implicit in their proof of soundness—and indeed in the definition of linearizability—is the assumption that the heap data associated with data structures is never interfered with by clients. In a setting where clients are given pointers into those data structures, this is an assumption that should be checked. In contrast, we are able to give a comprehensive model including both data structures and their clients, and make explicit assumptions about ownership.

Our use of separation logic and rely/guarantee clearly derives from Vafeiadis *et al.*'s work [28], especially Vafeiadis's groundbreaking thesis [27]. In that line of work, it was shown how to combine separation logic and rely/guarantee reasoning, which provided a basis for verifying fine-grained concurrent data structures. While their logic for proving Hoare triples was proved sound, no formal connection to linearizability or refinement was made; there is only an implicit methodology for proving certain Hoare triples about data structures and concluding that those data structures are linearizable. We show how to make that methodology explicit, and moreover compositional, by tying it to data abstraction. As a byproduct, we get a modularized account of rely/guarantee. We also eliminate any need for explicit linearization points (which sometimes require prophecy variables) or annotations separating shared from private resources.

Finally, it should be noted that our semantics owes a debt both to Brookes [2] and to Calcagno *et al.* [3].

Acknowledgements Thanks to Dan Brown, Claudio Russo and Sam Tobin-Hochstadt for feedback and discussions. The first author was supported by a grant from Microsoft Research.

References

[1] R. J. Back and J. von Wright. *Refinement calculus: a systematic introduction*. Springer, 1998.

[2] S. Brookes. Full abstraction for a shared variable parallel language. *Information and Computation*, 127(2):145–163, 1996.

[3] C. Calcagno, P. W. O'Hearn, and H. Yang. Local action and abstract separation logic. In *LICS*, pages 366–378. IEEE Computer Society, 2007.

[4] T. Dinsdale-Young, M. Dodds, P. Gardner, M. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In *ECOOP*, June 2010.

[5] T. Elmas, S. Qadeer, and S. Tasiran. A calculus of atomic actions. In *POPL*, pages 2–15. ACM, 2009.

[6] T. Elmas, S. Qadeer, A. Sezgin, O. Subasi, and S. Tasiran. Simplifying linearizability proofs with reduction and abstraction. In *TACAS*, pages 296–311. Springer, 2010.

[7] X. Feng. Local rely-guarantee reasoning. In *POPL*, pages 315–327. ACM, 2009.

[8] I. Filipović, P. O'Hearn, N. Rinetzky, and H. Yang. Abstraction for concurrent objects. In *ESOP*, pages 252–266. Springer, 2009.

[9] L. Groves. Reasoning about nonblocking concurrency. *JUCS*, 15(1): 72–111, 2009.

[10] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[11] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *TOPLAS*, 12(3):463–492, 1990.

[12] C. B. Jones. Tentative steps toward a development method for interfering programs. *TOPLAS*, 5(4):596–619, 1983.

[13] L. Lamport. The temporal logic of actions. *TOPLAS*, 16(3):872–923, 1994.

[14] R. J. Lipton. Reduction: a method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.

[15] B. Liskov and S. Zilles. Programming with abstract data types. In *Symposium on Very high level languages*, pages 50–59. ACM, 1974.

[16] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15:491–504, 2004. ISSN 1045-9219.

[17] M. M. Michael and M. L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *J. Parallel Distrib. Comput.*, 51(1):1–26, 1998.

[18] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., 1982.

[19] J. C. Mitchell and G. D. Plotkin. Abstract types have existential type. *TOPLAS*, 10(3):470–502, 1988.

[20] M. Moir and N. Shavit. Concurrent data structures. In *Handbook of Data Structures and Applications, D. Metha and S. Sahni Editors*, pages 47–14–47–30, 2007. Chapman and Hall/CRC Press.

[21] C. Morgan and T. Vickers. *On the refinement calculus*. Springer, 1993.

[22] J. H. Morris, Jr. Protection in programming languages. *CACM*, 16(1): 15–21, 1973.

[23] P. W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.

[24] M. Parkinson and G. Bierman. Separation logic and abstraction. *POPL*, 40(1):247–258, 2005.

[25] D. Sangiorgi and D. Walker. *Pi-Calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.

[26] R. K. Treiber. Systems programming: coping with parallelism. Technical report, Almaden Research Center, 1986.

[27] V. Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, 2008.

[28] V. Vafeiadis and M. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*, pages 256–271. Springer, 2007.

[29] R. J. van Glabbeek. The linear time - branching time spectrum. In *CONCUR*, pages 278–297. Springer, 1990.