A separation logic for refining concurrent objects

Aaron Turon and Mitchell Wand

Northeastern University

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. The result is a comprehensive and tidy account of concurrent data refinement that clarifies and consolidates the existing approaches.

1 Introduction

To benefit from multiprocessing, shared-memory data structures must allow concurrent access from many clients, without unduly sequentializing that access. Yet despite this fine-grained interleaving, operations on the data structures must appear to take effect atomically. The algorithms implementing these operations are often short but subtle, making formal correctness proofs both feasible and useful.

What does it mean for a concurrent data structure to be correct? If we can express both the implementation D and the specification S of a data structure in the same language, we can take the perspective of refinement: D *refines* S if for every program context $C[-]$, each behavior of the program $C[D]$ is a possible behavior of $C[S]$.[1] That is, no client can detect that it is interacting with D rather than S, even if the client invokes many operations of D concurrently.

The appeal of refinement is that, to prove something about the behaviors of $C[D]$ for a particular client C, it suffices to consider the behaviors of $C[S]$ instead—and S is usually much simpler than D. For concurrent data structures, we hope to gain simplicity in two respects: a larger grain of atomicity, and a more abstract view of data. For example, a set implementation might use a linked-list representation and fine-grained locking to enable concurrent operations, but its specification treats the data as an abstract set, and updates that data atomically.

In this paper, we present a new approach for proving refinement of concurrent objects. The main idea is to incorporate the notion of ownership from concurrent separation logic (CSL [1]) into a model that captures atomicity and refinement. By making both atomicity and heap ownership explicit, we can clarify and exploit their relationship. In particular, *atomicity is relative to ownership.* In more detail, our contributions are as follows:

[1] In particular, we dispense with linearizability; see §5.

- We present a new specification language in the tradition of refinement calculi [2, 3], but tailored to separation logic-style reasoning (§2). An important innovation is the data abstraction operator **abs** $\alpha.\varphi$ (§3.1), which allows us to make explicit the assumption that clients of a data structure do not observe or interfere with its representation. Thus an instance of a data structure is considered to be jointly owned by its operations.
- We show how this simple ownership model can enable both atomicity and data refinement (§3.2). When reasoning about an operation on a data structure instance, any memory that is not part of that instance is considered private to that operation. Private memory cannot be observed or interfered with, so computations only dealing with private resources can be viewed as part of a larger atomic step. As with CSL, "ownership is in the eye of the asserter" [1] and is used to describe, not determine, runtime behavior. Because ownership is described locally, changes to the grain of atomicity are enabled by strictly local reasoning—in contrast to methods like reduction [4], which must consider interactions with all possibly concurrent actions.
- We show how to use rely/guarantee reasoning [5] in a modular and dynamic fashion (§3.2). Again, data abstraction and ownership is crucial: any interference with a dynamic instance of a data structure must be due to one of the data structure operations. Rely/guarantee reasoning is used during data refinement steps. The reasoning is modular in the sense that it involves only the data structure's operations, and neither affects nor involves its clients; the reasoning is dynamic in the sense that it deals with the arbitrarily many instances of the data structure that may appear at runtime.
- We adapt Brookes's transition trace model [6] for dynamic memory allocation, to give our specification language a simple denotational semantics (§4). The semantics is adequate for a corresponding operational model, which justifies our claims about refinement. Finally, we introduce the semantic notion of *fenced refinement* to justify our ownership-based reasoning.

This paper draws on several ideas from recent work, especially that of Elmas *et al.* [7, 8] and Vafeiadis *et al.* [9, 10]. We refer the reader to §5 for a detailed discussion of prior work.

2 Programs and their specifications

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

```
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" [11]. Fine-grained concurrent objects permit as many good interleavings as possible, without allowing any bad ones. The code

```
while (true) { tmp = *C; if (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 [12]. 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:

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

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.

Our specifications can be understood as logical formulas satisfied by some set of observable behaviors (*cf.* [13]), which for us are finite traces capturing safety properties. Thus the grammar of specifications

$$\varphi, \psi, \theta ::= \varphi; \psi \mid \varphi|\psi \mid \textbf{let } f(\overline{x}) = \varphi \textbf{ in } \psi \mid f(\overline{e}) \mid \mu X.\varphi \mid X \mid \varphi \vee \psi \mid \exists x.\varphi \mid \langle \overline{x} : p, q \rangle \mid \{p\}$$

includes a combination of programming constructs like sequential and parallel composition, let-binding of procedures, and recursion, and logical constructs like disjunction and quantification. The latter can be read operationally, however. For example, the behaviors admitted by $\varphi \vee \psi$ are just those admitted either by φ or by ψ, so disjunction corresponds to nondeterministic choice. Variables x are immutable—only the heap can be mutated—so existentials behave as usual.

To describe the last two forms of specifications, we need to take a brief look at predicates p, which are checked against a heap σ and environment ρ:

Predicate semantics $\sigma, \rho \models p$, where $\sigma \in \Sigma \triangleq \textsc{Loc} \rightharpoonup \textsc{Val}$, $\rho \in \textsc{Var} \rightharpoonup \textsc{Val}$

$$p, q, r \;::=\; \mathsf{tt} \mid \mathsf{ff} \mid \mathsf{emp} \mid e \mapsto e' \mid e = e' \mid \quad \mid p * q \mid p \wedge q \mid p \vee q \mid p \Rightarrow q \mid \forall x.p \mid \exists x.p$$

$$\sigma, \rho \models \mathsf{emp} \quad \text{iff } \sigma = \emptyset \qquad\qquad \sigma, \rho \models e = e' \text{ iff } [\![e]\!]^\rho = [\![e']\!]^\rho$$

$$\sigma, \rho \models e \mapsto e' \text{ iff } \sigma = [[\![e]\!]^\rho \mapsto [\![e']\!]^\rho] \qquad \sigma, \rho \models \forall x.p \quad \text{iff } \forall v.\ \sigma, \rho[x \mapsto v] \models p$$

$$\sigma, \rho \models p * q \quad \text{iff} \quad \exists \sigma_1, \sigma_2.\ \ \sigma = \sigma_1 \uplus \sigma_2, \quad \sigma_1, \rho \models p, \quad \sigma_2, \rho \models q$$

$$\text{Assume:} \quad e ::= x \mid n \mid e + e' \mid (e, e') \mid \cdots \qquad [\![e]\!]^\rho \in \textsc{Val} \qquad \textsc{Loc} \subseteq \textsc{Val}$$

We include the key connective of separation logic, the *separating conjunction* $p * q$, which is satisfied by any heap separable into one subheap satisfying p and one satisfying q. The operators $\wedge, \vee, \Rightarrow, \exists, \mathsf{tt}, \mathsf{ff}$ are defined as usual.

An *action* $\langle \overline{x} : p, q \rangle$ describes an atomic step of computation in terms of the strongest partial correctness assertion it satisfies (*cf.* Morgan's specification statements [2]). The variables $\overline{x}$ are in scope for both p and q, and are used to link them together (as in inc above). Starting from a heap σ, an action can either diverge (*i.e.* have no behavior), *fault* if its precondition is not satisfied, or atomically update the heap so that the postcondition is satisfied. More precisely, (σ, o) is a behavior of the action in environment ρ iff, for every σ_1, σ_2 and $\overline{v}$,

$$\sigma = \sigma_1 \uplus \sigma_2,\ \rho[\overline{x \mapsto v}], \sigma_1 \models p \quad \Longrightarrow \quad o = \sigma_1' \uplus \sigma_2,\ \rho[\overline{x \mapsto v}], \sigma_1' \models q$$

where $o \in \Sigma \cup \{\top\}$. The symbol $\top$ represents a fault. This semantics of actions enforces the locality intrinsic to separation logic [14]. If the precondition is satisfied by a (or multiple) subheap, then the postcondition must be satisfied *by changing only that portion of the heap*, leaving the rest (the *frame* σ_2) intact. If the precondition is nowhere satisfied, any outcome o is permitted—even $\top$. According to this semantics, $\mathsf{inc}(\mathsf{c}, \mathsf{ret})$ faults iff c is unallocated.

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

$$\mathsf{skip} \triangleq \langle \mathsf{emp},\ \mathsf{emp} \rangle \qquad \mathsf{get}(\mathsf{a}, \mathsf{ret}) \triangleq \langle x : \mathsf{a} \mapsto x, \mathsf{a} \mapsto x \wedge x = \mathsf{ret} \rangle$$
$$\mathsf{new}(\mathsf{x}, \mathsf{ret}) \triangleq \langle \mathsf{emp},\ \mathsf{ret} \mapsto \mathsf{x} \rangle \qquad \mathsf{put}(\mathsf{a}, \mathsf{x}) \triangleq \langle \mathsf{a} \mapsto -,\ \mathsf{a} \mapsto \mathsf{x} \rangle$$
$$\mathsf{cas}(\mathsf{a}, \mathsf{old}, \mathsf{new}, \mathsf{ret}) \triangleq \left\langle x : \mathsf{a} \mapsto x, \begin{array}{l} (x \neq \mathsf{old} \wedge \mathsf{a} \mapsto x \wedge \mathsf{ret} = 0) \\ \vee\ (x = \mathsf{old} \wedge \mathsf{a} \mapsto \mathsf{new} \wedge \mathsf{ret} = 1) \end{array} \right\rangle$$

where the predicate $\mathsf{a} \mapsto -$ is shorthand for $\exists z.\mathsf{a} \mapsto z$.

Assertions $\{p\}$ simply test a predicate: if the heap satisfies $p * \mathsf{tt}$ the heap is left fixed, and otherwise the assertion faults. Assertions provide a way to introduce a claim during verification while postponing its justification (*cf.* [7]).

An *assumption* $[P]$ is shorthand for the action $\langle \mathsf{emp},\ \mathsf{emp} \wedge P \rangle$, which is a guard: according to the semantics above, it cannot fault, and must make P true without changing the heap. Thus if P does not already hold, it must diverge (have no behavior). The metavariables P, Q, R stand for *pure* predicates, which constrain only the environment: for each ρ, either $\rho, \sigma \models P$ for all σ or for no σ.

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

$$\mathsf{inc}'(\mathsf{c}, \mathsf{ret}) = \mu X. \exists t.\mathsf{get}(\mathsf{c}, t); \exists s.\mathsf{cas}(\mathsf{c}, t, t+1, s); ([s = 1][\mathsf{ret} = t] \vee [s \neq 1]; X)$$

3 Proof theory

Our first goal is to show how to execute inc' but reason in terms of inc.

Specifications are related through refinement: we write $\varphi \sqsubseteq \psi$ if every behavior of φ is a possible behavior of ψ. Crucially, refinement is a congruence (§4). Under refinement, faulting is maximally permissive, so a specification allowing a fault at a given point allows *any* behavior at that point. Thus the specification $\langle \mathsf{ff},\ \mathsf{tt} \rangle$, which faults regardless of the state, is *trivial*: $\varphi \sqsubseteq \langle \mathsf{ff},\ \mathsf{tt} \rangle$ for all φ. Dually, $\langle \mathsf{tt},\ \mathsf{ff} \rangle$, whose behaviors take any heap to one satisfying ff, is *miraculous* [2]: *no* behaviors satisfy it, so $\langle \mathsf{tt},\ \mathsf{ff} \rangle \sqsubseteq \varphi$ for every φ. Miraculous specifications are akin to imaginary numbers: they are not the primary objects of our study, but they are indispensable for calculation. Ultimately, if we can prove that $\varphi \sqsubseteq \langle \overline{x} : p, q \rangle$, we know that φ faults iff $\exists \overline{x}.p$ does not hold.

Thinking of specifications as formulas in a logic, we have a model theory saying when a behavior satisfies a formula (§4), and the following sound proof theory for showing implications (refinements) between formulas:

A selection of sound refinements $\varphi \sqsubseteq \psi$, with $\varphi \equiv \psi$ iff $\varphi \sqsubseteq \psi$ and $\psi \sqsubseteq \varphi$

$$\langle \mathsf{tt},\ \mathsf{ff}\rangle \sqsubseteq \varphi \qquad \varphi \vee \varphi \sqsubseteq \varphi \qquad \exists x.\varphi \sqsubseteq \varphi \qquad \langle x, \overline{y} : p, q\rangle \sqsubseteq \langle \overline{y} : p[e/x], q[e/x]\rangle$$
$$\langle \mathsf{ff},\ \mathsf{tt}\rangle \sqsupseteq \varphi \qquad \varphi \vee \psi \sqsupseteq \varphi \qquad \exists x.\varphi \sqsupseteq \varphi[e/x] \qquad \langle x, \overline{y} : p, q\rangle \sqsupseteq \langle \overline{y} : p, q\rangle$$

DstL $(\varphi_1 \vee \varphi_2); \psi \equiv \varphi_1; \psi \vee \varphi_2; \psi$ Frm $\langle \overline{x} : p, q\rangle \sqsubseteq \langle \overline{x} : p * r, q * r\rangle$
DstR $\psi; (\varphi_1 \vee \varphi_2) \equiv \psi; \varphi_1 \vee \psi; \varphi_2$ Ext $\langle \overline{x} : p, p\rangle \equiv \{\exists \overline{x}.p\}$ (p exact)

Str rules
$\varphi; (\exists x.\psi) \equiv \exists x.\varphi; \psi$
$\langle \overline{y} : p, \exists x.q\rangle \equiv \exists x.\, \langle \overline{y} : p, q\rangle$

Idm$_1$ $\{p\}\{p\} \equiv \{p\}$
Idm$_2$ $\{\exists \overline{x}.p\}\, \langle \overline{x} : p, q\rangle \equiv \langle \overline{x} : p, q\rangle$
Asm $\langle \overline{x} : p, q \wedge R\rangle \equiv \langle \overline{x} : p, q\rangle\, [R]$

Ind
$$\frac{\varphi[\psi/X] \sqsubseteq \psi}{\mu X.\varphi \sqsubseteq \psi}$$

Csq$_1$
$$\frac{\forall \overline{x}.\ p \Rightarrow p' \qquad \forall \overline{x}.\ q' \Rightarrow q}{\langle \overline{x} : p', q'\rangle \sqsubseteq \langle \overline{x} : p, q\rangle}$$

Csq$_2$
$$\frac{(q{*}\mathsf{tt}) \Rightarrow (p{*}\mathsf{tt})}{\{p\} \sqsubseteq \{q\}}$$

NB: a predicate appearing both in and outside a binder for x cannot mention x.

Many of these rules are familiar. The top row 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 [15]. 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. 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. Csq$_2$ tells us that assertions are antitonic, which follows from the permissive nature of faulting.

To show $\mathsf{inc}' \sqsubseteq \mathsf{inc}$, we first prove two lemmas describing the effect of cas:

$$\begin{array}{rl} & \mathsf{cas}(\mathsf{a},\mathsf{o},\mathsf{n},\mathsf{r}); [\mathsf{r} = 1] \\ (\text{Asm, Csq}_1) & \sqsubseteq \langle x : \mathsf{a} \mapsto x, \mathsf{a} \mapsto \mathsf{n} \wedge x = \mathsf{o}\rangle \end{array} \qquad \begin{array}{rl} & \mathsf{cas}(\mathsf{a},\mathsf{o},\mathsf{n},\mathsf{r}); [\mathsf{r} \neq 1] \\ (\text{Asm, Csq}_1) & \sqsubseteq \langle x : \mathsf{a} \mapsto x, \mathsf{a} \mapsto x\rangle \equiv \{\mathsf{a} \mapsto -\} \end{array}$$

Next we begin abstracting the loop in inc', with X replaced by $\mathsf{inc}(\mathsf{c},\mathsf{ret})$:

$$\begin{array}{rl} & \exists t.\mathsf{get}(\mathsf{c},t); \exists s.\mathsf{cas}(\mathsf{c},t,t+1,s); ([s=1][\mathsf{ret}=t] \vee [s \neq 1]; \mathsf{inc}(\mathsf{c},\mathsf{ret})) \\ \text{DstR} & \sqsubseteq \exists t.\mathsf{get}(\mathsf{c},t); \exists s. \begin{pmatrix} \mathsf{cas}(\mathsf{c},t,t+1,s); [s=1][\mathsf{ret}=t] \\ \vee\ \mathsf{cas}(\mathsf{c},t,t+1,s); [s \neq 1]; \mathsf{inc}(\mathsf{c},\mathsf{ret}) \end{pmatrix} \\ \text{lemmas} & \sqsubseteq \exists t.\mathsf{get}(\mathsf{c},t); \exists s. \begin{pmatrix} \langle x : \mathsf{c} \mapsto x, \mathsf{c} \mapsto t+1 \wedge x = \mathsf{t}\rangle\, [\mathsf{ret}=t] \\ \vee\ \{\mathsf{c} \mapsto -\}; \mathsf{inc}(\mathsf{c},\mathsf{ret}) \end{pmatrix} \\ \begin{array}{c}\text{Asm,}\\ \text{Csq}_1\end{array} & \sqsubseteq \exists t.\mathsf{get}(\mathsf{c},t); \exists s.(\mathsf{inc}(\mathsf{c},\mathsf{ret}) \vee \{\mathsf{c} \mapsto -\}; \mathsf{inc}(\mathsf{c},\mathsf{ret})) \end{array}$$

Next is the key step: abstracting get into an assertion(*cf.* havoc abstraction [7]):

$$\begin{array}{rl}
\text{definition} & \equiv \exists t.\ \langle x : \mathsf{n} \mapsto x, \mathsf{n} \mapsto x \wedge x = t\rangle \,;\, \exists s.(\mathsf{inc}(\mathsf{c}, \mathsf{ret}) \vee \{\mathsf{c} \mapsto -\}; \mathsf{inc}(\mathsf{c}, \mathsf{ret})) \\
\textsc{Csq}_1 & \sqsubseteq \exists t.\ \langle x : \mathsf{n} \mapsto x, \mathsf{n} \mapsto x\rangle \,;\, \exists s.(\mathsf{inc}(\mathsf{c}, \mathsf{ret}) \vee \{\mathsf{c} \mapsto -\}; \mathsf{inc}(\mathsf{c}, \mathsf{ret})) \\
\textsc{Ext} & \sqsubseteq \exists t.\{\mathsf{c} \mapsto -\}; \exists s.(\mathsf{inc}(\mathsf{c}, \mathsf{ret}) \vee \{\mathsf{c} \mapsto -\}; \mathsf{inc}(\mathsf{c}, \mathsf{ret}))
\end{array}$$

revealing that the value t returned from get is unimportant for correctness. Finally the assertions $\{\mathsf{c} \mapsto -\}$ are absorbed using $\textsc{Idm}_2$, yielding one atomic action: $\mathsf{inc}(\mathsf{c}, \mathsf{ret})$.Using $\textsc{Ind}$, we conclude that $\mathsf{inc}' \sqsubseteq \mathsf{inc}$.

Notice that we do not give and did not use any rules for reasoning about parallel composition. We certainly could give such rules, but that would be beside the point. Our aim is to reason about concurrent data structure implementations, which are pieces of *sequential* code that clients may choose to execute in parallel. Having proved the refinement, we can conclude that even for a concurrent client $C[-]$ we have $C[\mathsf{inc}'] \sqsubseteq C[\mathsf{inc}]$.

3.1 Data abstraction

The specification inc is simpler than inc' in an important respect: it has a larger grain of atomicity. However, inc is still unsatisfying as a specification, because it reveals the representation of the counter data structure, and hence is not refined by implementations using a different representation. We need data abstraction.

We can think of programs as syntactic sugar for certain specifications, as in our definitions of get, cas, *etc.*, but the specification language need not be limited to program-like constructs. Thus, we take the radical step of introducing a *data abstraction* operator $\mathbf{abs}\ \alpha.\varphi$, which allows us to write

$$\begin{array}{l}
\mathbf{abs}\ \mathit{cnt}.\ \mathbf{let}\ \mathsf{newCnt}(\mathsf{ret}) = \langle \mathsf{emp},\ \mathit{cnt}(\mathsf{ret}, 0)\rangle\,, \\
\qquad\qquad\quad\ \mathsf{inc}(c, \mathsf{ret}) \quad = \langle x : \mathit{cnt}(\mathsf{c}, x), \mathit{cnt}(\mathsf{c}, x+1) \wedge \mathsf{ret} = x\rangle\ \ \mathbf{in}\ \varphi
\end{array}$$

where cnt does not appear in φ. The idea is that $\mathit{cnt}(\ell, x)$ is an *abstract resource* that the client φ can work with only by using the let-bound procedures (because none of its own actions mention cnt). Each abstract resource has two parameters: an abstract location ℓ and an abstract value x. We redefine $\Sigma \triangleq (\textsc{Loc} \rightharpoonup \textsc{Val}) \ \cup\ (\textsc{ResVar} \rightharpoonup 2^{\textsc{Loc}\times\textsc{Val}})$ so that $\sigma(\alpha)$ is the set of owned α resources. Then newCnt, for example, takes σ to $\sigma \uplus [\mathit{cnt} \mapsto \{(a, 0)\}]$ for some location a; we assume $\uplus$ does not permit two abstract resources with the same location. We introduce data abstraction via the following rule, reminiscent of $\exists$-introduction:[2]

$$\textsc{Data}_1\quad \mathbf{let}\ \overline{f(\overline{\mathsf{x}}) = \langle \overline{y} : p[r/\alpha], q[r/\alpha]\rangle}\ \mathbf{in}\ \varphi\ \sqsubseteq\ \mathbf{abs}\ \alpha.\mathbf{let}\ \overline{f(\overline{\mathsf{x}}) = \langle \overline{y} : p, q\rangle}\ \mathbf{in}\ \varphi$$

Here we are thinking of r as a predicate with two holes, so that $r[\ell, x]$ gives a concrete representation invariant for the abstract resource $\alpha(\ell, x)$. For inc, we would let $r[\ell, x] = \ell \mapsto x$.

[2] Our abstract resource variables also mirror Parkinson and Bierman's abstract predicates [16]. But the semantics of $\mathbf{abs}\ \alpha.\varphi$ is not that of second-order $\exists$; see §4.

The use of abstract resources allows us to localize interaction with the heap cells associated with a data structure, even though pointers to those cells escape. Crucially, we do this without restricting the client of the data structure—notice DATA$_1$ does not constrain φ. For example, we have the valid data refinement

$$\begin{array}{rl} \textsf{\textbf{let}}\ f(\textsf{ret}) = \textsf{new}(0, \textsf{ret})\ \textsf{\textbf{in}} & \exists x. f(x); \textsf{put}(x, 1) \\ \sqsubseteq \textsf{\textbf{abs}}\ \alpha.\ \textsf{\textbf{let}}\ f(\textsf{ret}) = \langle \textsf{emp},\ \alpha(\textsf{ret})\rangle\ \textsf{\textbf{in}} & \exists x. f(x); \textsf{put}(x, 1) \end{array}$$

even though the client improperly follows a pointer into, and then modifies, the data structure we are abstracting. Because the abstract f does not allocate x on the concrete heap, when the client attempts to $\textsf{put}(x, 1)$ it will fault—we have literally taken away the concrete resources associated with the data structure—and the permissive nature of faulting means that the refinement holds no matter what the concrete implementation does. In general, moving from a concrete to an abstract data representation causes the client to fault whenever it attempts to forge, access, or modify an instance of the data structure without using the let-bound procedures.

3.2 Ownership, fenced refinement, and rely/guarantee

The key step in developing our logic is to permit rely/guarantee and ownership-based reasoning through data abstraction. We motivate and illustrate that step by verifying a version of Treiber's stack [17]:

```
newStk(ret) = new(0, ret)              pop(s, ret) =
push(s, x) = $\exists n$.new((x, 0), $n$);      $\mu X. \exists t$.get(s, $t$);
  $\mu X. \exists t$.get(s, $t$);                   if $t = 0$ then [ret = emptyStack]
      put$_2(n, t)$;                          else $\exists n$.get$_2(t, n)$;
      $\exists b$.cas(s, $t, n, b$);                     $\exists b$.cas(s, $t, n, b$);
      if $b = 0$ then $X$                          if $b = 0$ then $X$ else get$_1(t$, ret)
```

The specifications $\textsf{get}_i$ and $\textsf{put}_i$ operate on the i^{th} component of pairs, *e.g.*, $\textsf{put}_2(\textsf{a}, \textsf{x}) \triangleq \langle y : \textsf{a} \mapsto (y, -), \textsf{a} \mapsto (y, \textsf{x})\rangle$. Stacks provide two new challenges compared to counters. First, the loop in push modifies the heap *every* time it executes, by calling $\textsf{put}_2$, rather than just at a successful cas; this makes atomicity nontrivial to show. Second, pop has a potential "ABA" problem [18]: 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;[3] we must make such an argument within the logic.

To deal with these challenges, we introduce the conceptual notion of ownership, captured by the formal notion of fenced refinement—which significantly enhances the power of the logic.

Where does ownership enter the picture? Informally, we want to think of the representation of stack instances (living in the heap) as being jointly owned by the stack procedures, while only the corresponding abstract stack resources

[3] We assume garbage collection here, but we can also verify a deallocating stack.

are owned by the client—thus, as already mentioned, the client cannot directly interfere with stack instances. Conversely, the stack does not interfere with its client's memory. But notice that the procedure push allocates memory for itself that, initially, belongs to neither the client nor any stack instance. Until that memory becomes part of a stack instance, it is exclusively owned by push—and hence, we can reason about it sequentially. Likewise, if memory is removed from a stack instance by a procedure (as in pop), it becomes private to that procedure, even if other threads have pointers to it. From the point of view of those other threads, the memory has effectively been deallocated by their environment.

We need a way to characterize the memory associated with a data structure instances. A *precise* predicate is one that, for any heap, it is satisfied by at most one subheap—that is, it unambiguously picks out ("*fences*" [19]) the part of the heap constituting the data structure instance. Our representation invariant $\ell \mapsto -$ for a counter located at ℓ is precise, as is the invariant $\exists x.\mathsf{list}(\ell, x)$ where

$$\mathsf{list}(\ell, \epsilon) \triangleq \ell \mapsto 0 \qquad \mathsf{list}(\ell, \mathsf{x} \cdot \mathsf{xs}) \triangleq \exists \ell'.\ell \mapsto (\mathsf{x}, \ell') * \mathsf{list}(\ell', \mathsf{xs})$$

These precise predicates in some sense describe a way of tracing the heap, starting from some location, to find all the memory associated with a data structure. We will use I as a metavariable for such precise predicates.

For simplicity, we will assume the procedures for a data structure either (1) allocate an instance of the data structure or (2) read or write an instance of the data structure, leading to the following specialized data abstraction rule:

$$\textsc{Data}_2\quad \frac{(\exists z.I[\ell, z]), (\bigvee \overline{\theta_i}) \vdash \varphi_i \sqsubseteq \theta_i \qquad \theta_i \sqsubseteq \langle y : I[\ell, y], I[\ell, e_i] \wedge P_i\rangle}{\begin{array}{l} \quad\ \ \mathbf{let}\ f(\ell) = \langle \mathsf{emp},\ I[\ell, e]\rangle\ ,\ \overline{g_i(\ell, x) = \varphi_i}\ \mathbf{in}\ \psi \\ \sqsubseteq \mathbf{abs}\ \alpha.\mathbf{let}\ f(\ell) = \langle \mathsf{emp},\ \alpha(\ell, e)\rangle, \overline{g_i(\ell, x) = \langle y : \alpha(\ell, y), \alpha(\ell, e_i) \wedge P_i\rangle}\ \mathbf{in}\ \psi \end{array}}$$

where I has a hole for a location ℓ and a value z (as in $\mathsf{list}(\ell, z)$). If we ignore the grayed portion of the rule, $\textsc{Data}_2$ is derivable from $\textsc{Data}_1$. The advantage of the new rule is its use of *fenced refinement*, which has the general form $I, \theta \vdash \varphi \sqsubseteq \psi$. In fenced refinement, we take the perspective of a procedure interacting with a data structure instance: our view of the heap consists only of a shared instance of the data structure (located at ℓ, fenced by $\exists z.I[\ell, z]$) and private memory. As the procedure makes changes to the data structure, memory moves in and out of the region described by I, and this corresponds to the procedure giving up or claiming, respectively, exclusive ownership of that memory. Each θ_i is an atomic abstraction of the procedure φ_i. Thus the specification $\bigvee \overline{\theta_i}$ characterizes the expected interference on the shared data structure instance fenced by I.

Fenced refinement is strictly more permissive than standard refinement, because it makes assumptions about the context—in particular, about the ownership of data—that do not always hold. $\textsc{Data}_2$ allows us to derive standard refinements from fenced refinements, by ensuring that the assumed ownership protocol is actually followed; "ownership is in the eye of the asserter" [1]. The second antecedent of the rule checks the ownership protocol at the level of each θ_i (an atomic abstraction) rather than each φ_i (a fine-grained implementation).

A key novel feature of $\textsc{Data}_2$ is its provision for modular and dynamic rely/guarantee reasoning. We use fenced refinement to prove that each procedure body φ_i guarantees to behave like θ_i, as long it can rely on its environment to interfere with the shared data only by taking the atomic steps described by $\bigvee \overline{\theta_i}$. This form of reasoning is *modular* because we have isolated the memory involved (to $I[\ell, -]$ for some ℓ) 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.

Below, we set out and explain some of the rules of fenced refinement, then illustrate them on fragments of the stack example.

A selection of sound fenced refinements $\qquad I, \theta \vdash \varphi \sqsubseteq \psi$

$\textsc{SeqL}$

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

$\textsc{Inv}$

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

$\textsc{SeqR}$

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

$\textsc{Lift}$

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

$\textsc{Stab}_1$

$$\frac{\langle \overline{x} : p, q \rangle \sqsubseteq \langle \overline{x} : (p * \mathsf{tt}) \wedge s, r \rangle \qquad \theta \sqsubseteq \langle r, \ r \rangle}{I, \theta \vdash \langle \overline{x} : p, q \rangle \, \{r\} \sqsubseteq \{s\} \, \langle \overline{x} : p, q \rangle}$$

$\textsc{Stab}_2$

$$\frac{\varphi \sqsubseteq \langle \overline{x} : p, I * q \rangle}{I, \theta \vdash \varphi; \{I * q\} \sqsubseteq \varphi}$$

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. The $\textsc{Seq}$ rules permit atomicity refinement: two atomic actions can be combined if one of them only operates on private data. $\textsc{SeqL}$ combines two actions when the first is private, introducing an assertion as a verification condition for showing that the data p is separate from I on the heap—and hence private. In $\textsc{SeqR}$, where the second action must be private, separability must be satisfied in between the two atomic actions; showing that verification condition may involve introducing a further assertion s.

$\textsc{Inv}$ expresses that the invariant I always holds, so asserting it will always succeed. $\textsc{Lift}$ reflects the fact that fenced refinement is more permissive than standard refinement.

Finally, $\textsc{Stab}_1$ allows as assertion $\{r\}$ to be removed if it is *stably* satisfied after an action (which may require a new assertion, $\{s\}$, to hold prior to the action). The predicate r holds stably if it holds immediately after the action (the first antecedent of the rule) and if it is invariant under any interference θ. $\textsc{Stab}_2$ is a handy special case for assertions that solely constrain private data.

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 procedures implementing a data structure—which we argued are usually sequential—this is all we need.

Returning to Treiber's stack, we can make progress reasoning about push even before nailing down the invariant I. Below is a fragment of push, corresponding to a successful execution of the loop body. In this and the following examples, we assume a fixed $I, \theta \vdash$ part of the refinement, and elide uses of Lift.

$$\begin{array}{rl}
 & \mathsf{put}_2(n,t); \exists b.\mathsf{cas}(\mathsf{s},t,n,b); [b \neq 0] \\
\text{lemma} & \sqsubseteq \mathsf{put}_2(n,t); \langle z : \mathsf{s} \mapsto z, \mathsf{s} \mapsto n \wedge z = t \rangle \\
\text{Frm, Csq}_1 & \sqsubseteq \mathsf{put}_2(n,t); \langle z : \mathsf{s} \mapsto z * n \mapsto (\mathsf{x}, t), \mathsf{s} \mapsto n * n \mapsto (\mathsf{x}, z) \rangle \\
\text{SeqL} & \sqsubseteq \{I * n \mapsto (\mathsf{x}, -)\}; \langle z : \mathsf{s} \mapsto z * n \mapsto (\mathsf{x}, -), \mathsf{s} \mapsto n * n \mapsto (\mathsf{x}, t) \rangle
\end{array}$$

We use SeqL to enlarge the grain of atomicity, but it imposes the proof obligation (*i.e.*, assertion) that I, whatever it is, must be separable from the pair located at n. Intuitively, it will be separable, because n is private data which does not become part of a stack instance until *after* the successful cas. That is,

$$\begin{array}{rl}
 & \exists n.\mathsf{new}((\mathsf{x},0),n); \{I * n \mapsto (\mathsf{x}, -)\}; \langle z : \mathsf{s} \mapsto z * n \mapsto (\mathsf{x}, -), \mathsf{s} \mapsto n * n \mapsto (\mathsf{x}, t) \rangle \\
\text{Stab}_2 & \sqsubseteq \exists n.\{I\}; \mathsf{new}((\mathsf{x},0),n); \langle z : \mathsf{s} \mapsto z * n \mapsto (\mathsf{x}, -), \mathsf{s} \mapsto n * n \mapsto (\mathsf{x}, t) \rangle \\
\text{SeqL, Idm}_1 & \sqsubseteq \exists n.\{I\}; \langle z : \mathsf{s} \mapsto z, \mathsf{s} \mapsto n * n \mapsto (\mathsf{x}, t) \rangle \\
\text{Inv} & \sqsubseteq \langle z : \mathsf{s} \mapsto z, \exists n.\mathsf{s} \mapsto n * n \mapsto (\mathsf{x}, t) \rangle
\end{array}$$

The fact that n is privately allocated—even without a specific choice of stack invariant I—is enough to abstract push into a single atomic action. Of course, to subsequently use the Data$_2$ rule, we will need to choose such an invariant and show that the abstracted version of push maintains it.

To choose an appropriate I for a stack, we need to analyze its ownership protocol. Why not let $I[\ell, x]$ be $\exists z.\ell \mapsto z * \mathsf{list}(z, x)$? According to this invariant, when a cell is popped from the stack, the cell would no longer be part of stack representation, and would therefore be exclusively owned by the pop procedure that removed it. Meanwhile, threads concurrently executing pop might try to follow a pointer to that cell (the get_2 in pop), and fault from lack of access to that memory.[4] We want instead to treat any cell that has appeared in the stack as a permanent part of the stack's representation—after all, we never deallocate.

We therefore need some way to talk about cells that were reachable in the past, but may no longer be. We employ the device of *ghost abstractions*, just as one might introduce a ghost variable for this purpose in Hoare logic. A ghost abstraction is an abstract resource whose concrete representation is simply emp—concretely, it does not exist at all—and as such it is trivial to introduce using Data$_1$. We use Data$_1$ to introduce a ghost abstraction α into cas(a, old, new, ret) in pop, abstracting its definition to

$$\left\langle x, A : \mathsf{a} \mapsto x * \alpha(\mathsf{s}, A), \begin{array}{l} (\mathsf{a} \mapsto x * \alpha(\mathsf{s}, A) \wedge x \neq \mathsf{old} \wedge \mathsf{ret} = 0) \\ \vee\ (\mathsf{a} \mapsto \mathsf{new} * \alpha(\mathsf{s}, A \cdot \mathsf{a}) \wedge x = \mathsf{old} \wedge \mathsf{ret} = 1) \end{array} \right\rangle$$

The ghost abstraction α has two parameters: the stack location s it is associated with, and a sequence A of addresses of popped cells. On a successful pop, we

[4] This faulting behavior is part of the fenced semantics given in the next section.

add a new address. With α in place, we define the following:

$$
\begin{aligned}
I[\mathsf{s},\mathsf{xs}] &\triangleq \mathsf{list}(\mathsf{s},\mathsf{xs}) * \exists A.\alpha(\mathsf{s},A) * \mathsf{own}(A) & \mathsf{own}(\epsilon) &\triangleq \mathsf{emp}\\
\theta_{\mathsf{push}} &\triangleq \langle n : \mathsf{s} \mapsto n, \exists z.\mathsf{s} \mapsto z * z \mapsto (\mathsf{x}, n)\rangle & \mathsf{own}(\mathsf{a}\cdot A) &\triangleq \mathsf{a} \mapsto - * \mathsf{own}(A)\\
\theta_{\mathsf{pop}} &\triangleq \langle n, n_1, n_2, A : \mathsf{s} \mapsto n * n \mapsto (n_1, n_2) * \alpha(\mathsf{s}, A), \mathsf{s} \mapsto n_2 * \alpha(\mathsf{s}, n\cdot A) \wedge \mathsf{ret} = n_1\rangle
\end{aligned}
$$

and use $\textsc{Data}_2$ to abstract *e.g.* $\mathsf{push}(\mathsf{s},\mathsf{x})$ to $\langle \mathsf{xs} : \mathit{stack}(\mathsf{s},\mathsf{xs}), \mathit{stack}(\mathsf{s}, \mathsf{x}\cdot\mathsf{xs})\rangle$, where *stack* is the abstract resource for stacks.

4 Model theory

To justify our proof theory, we need a model theory saying when $\models \varphi \sqsubseteq \psi$. We accomplish this by introducing a denotational semantics for specifications. Our semantics is based on Brookes's transition trace model [6], which gave a fully abstract semantics for a simple parallel WHILE language. In this section we briefly recapitulate that model, making a few adjustments to support our specification language, and define data abstraction and fenced refinement.

The semantics of a specification is the set of its observable behaviors—its transition traces—and concurrency is modeled by nondeterministic interleaving. A *transition trace* t is a sequence of pairs in $(\Sigma^\top \times \Sigma^\top)^*$. Each pair represents an observable step of execution, which could consist of zero, one, or multiple atomic steps of computation, depending on the scheduler. For example, the trace $(\sigma_1, \sigma_1')(\sigma_2, \sigma_2')$ can be read as: the specification began with a heap σ_1, and by the end of its timeslice changed the heap to σ_1'; then its environment had a chance to execute, changing the heap to σ_2, and so on. A pair $(\sigma, \top)$ represents a faulting execution of the specification, while a pair $(\top, o)$ represents a fault on the part of the environment (in which case the second component is irrelevant).

An important subspace of traces are the *sequential* ones: we write t seq if the ending heap of each pair in t is equal to the starting heap of the next pair. Sequential traces are those in which the environment never observably interferes.

With this model of behavior, the semantics of most specifications is straightforward (fig 1); we explain **abs** below. Environments ρ map variables x to values v, specification variables X to sets of traces T, and procedure variables f to functions $\textsc{Val} \to 2^{\textsc{Trace}}$. Sequential composition is concatenation and parallel composition is nondeterministic interleaving.

Crucial to Brookes's model is a quotient on sets of traces, given by a closure operator $-^\dagger$. If T is a set of observable behaviors, then $T^\dagger$ adds further observations they give rise to. The unshaded closure rules were in Brookes's paper; they say that (1) each behavior gives rise to itself (2) scheduling the specification for an additional zero-length timeslice does not change the observable behavior and (3) scheduling the environment for a zero-length timeslice does not change the observable behavior. To these rules we add a rule saying that a faulting behavior leads to all possible behaviors with the same prefix—faults are permissive.

Data abstraction works by replacing the abstract resource α by an arbitrary concrete resource. We use trace merge $(s \uplus t)$ to separate a trace into abstract and concrete parts: the notation $s \uplus_\alpha t$ denotes a trace that splits into two traces

Atomic step semantics $\mathrm{act}(x,\rho,p,q) \subseteq \Sigma \times \Sigma^{\top}$

$$\begin{array}{l}(\sigma,o) \in \mathrm{act}(x,\rho,p,q) \text{ iff } \forall \sigma_1,\sigma_2,v. \\ \quad \sigma = \sigma_1 \uplus \sigma_2,\ \rho[x \mapsto v],\sigma_1 \models p \quad \Longrightarrow \quad o = \sigma_1' \uplus \sigma_2,\ \rho[x \mapsto v],\sigma_1' \models q\end{array}$$

Specification semantics $\llbracket\varphi\rrbracket^{\rho} \subseteq \textsc{Trace}$

$$\begin{array}{rcl}\llbracket f(e)\rrbracket^{\rho} & \triangleq & \rho(f)(\llbracket e\rrbracket^{\rho}) \\ \llbracket \varphi;\psi\rrbracket^{\rho} & \triangleq & (\llbracket\varphi\rrbracket^{\rho};\llbracket\psi\rrbracket^{\rho})^{\dagger} \\ \llbracket \varphi \parallel \psi\rrbracket^{\rho} & \triangleq & (\llbracket\varphi\rrbracket^{\rho} \parallel \llbracket\psi\rrbracket^{\rho})^{\dagger} \\ \llbracket \varphi \vee \psi\rrbracket^{\rho} & \triangleq & \llbracket\varphi\rrbracket^{\rho} \cup \llbracket\psi\rrbracket^{\rho} \\ \llbracket \exists x.\varphi\rrbracket^{\rho} & \triangleq & \bigcup_v \llbracket\varphi\rrbracket^{\rho[x\mapsto v]}\end{array} \qquad \begin{array}{rcl}\llbracket \mathsf{let}\ f(x)=\varphi\ \mathsf{in}\ \psi\rrbracket^{\rho} & \triangleq & \llbracket\psi\rrbracket^{\rho[f\mapsto \lambda v.\llbracket\varphi\rrbracket^{\rho[x\mapsto v]}]} \\ \llbracket \mu X.\varphi\rrbracket^{\rho} & \triangleq & \bigcap\{T^{\dagger} : \llbracket\varphi\rrbracket^{\rho[X\mapsto T^{\dagger}]} \subseteq T^{\dagger}\} \\ \llbracket \langle x:p,q\rangle\rrbracket^{\rho} & \triangleq & \mathrm{act}(x,\rho,p,q)^{\dagger} \\ \llbracket \{p\}\rrbracket^{\rho} & \triangleq & \{(\sigma,\sigma) \ :\ \sigma \in \llbracket p * \mathsf{tt}\rrbracket^{\rho}\}^{\dagger} \\ & \cup & \{(o,\top) \ :\ o \notin \llbracket p * \mathsf{tt}\rrbracket^{\rho}\}^{\dagger}\end{array}$$

$$\begin{array}{rcl}\llbracket \mathsf{abs}\ \alpha.\varphi\rrbracket^{\rho} & \triangleq & \{\ s \uplus u \ :\ s \uplus_{\alpha} t \in \llbracket\varphi\rrbracket^{\rho},\ t,u\ \mathrm{seq}_{\emptyset}\ \}^{\dagger} \\ & \cup & \{\ (s_1 \uplus u)(o,\top) \ :\ (s_1 s_2) \uplus_{\alpha} t \in \llbracket\varphi\rrbracket^{\rho},\ t,u\ \mathrm{seq}_{\emptyset},\ \mathrm{last}(u) \not\subseteq o\ \}^{\dagger}\end{array}$$

Closure $t \in T^{\dagger}$

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

Trace merge

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

Entailment

$$\begin{array}{rcl}\models \varphi \sqsubseteq \psi & \triangleq & \forall \rho\ .\ \llbracket\varphi\rrbracket^{\rho} \subseteq \llbracket\psi\rrbracket^{\rho} \\ I,\theta \models \varphi \sqsubseteq \psi & \triangleq & \forall \rho,o,o'\ .\ \mathrm{rcl}_{\llbracket\theta^*\rrbracket^{\rho}}(\llbracket I\rrbracket^{\rho} \triangleright_o^{o'} \llbracket\varphi\rrbracket^{\rho}) \ \subseteq\ \mathrm{rcl}_{\llbracket\theta^*\rrbracket^{\rho}}(\llbracket I\rrbracket^{\rho} \triangleright_o^{o'} \llbracket\psi\rrbracket^{\rho})\end{array}$$

Data structure projection $H \triangleright_o^{o'} T \subseteq \textsc{Trace}$

$$H \triangleright_o^{o'} T \triangleq \{s \ :\ s \uplus t \in T,\ \ s \in (H \times H)^*,\ \ t\ \mathrm{seq}_o^{o'}\}$$

Rely closure

$$\frac{t \in T}{t \in \mathrm{rcl}_S(T)} \qquad \frac{(\sigma',o) \notin S \qquad s(\sigma,\sigma')t \in \mathrm{rcl}_S(T)}{s(\sigma,\sigma')(o,\top) \in \mathrm{rcl}_S(T)} \qquad \frac{s(o,\top)t \in \mathrm{rcl}_S(T)}{s(o,o')u \in \mathrm{rcl}_S(T)}$$

Fig. 1. The model theory

s and t, where s contains no α resources and t contains only α resources. The requirement that t $\mathrm{seq}_{\emptyset}$ (that is, t is sequential and begins with no resources) reflects the fact that no interference on α from the environment of the abstraction is possible, and initially no α resources are allocated.

In the first clause defining data abstraction, we simply replace the abstract resource trace by an arbitrary $\mathrm{seq}_{\emptyset}$ trace u—swapping concrete for abstract resources. Because u may simply be a sequence of empty heaps $(\emptyset,\emptyset)^*$, the environment of the abstraction cannot assume that the concrete resources have any particular shape—and if it does make such an assumption, it will fault. Consequently, in the second clause, we respond to interference on the concrete representation by faulting. Because faulting is permissive, this relieves us from showing anything about the implementation in the presence of an interfering environment. The second clause enables modular rely/guarantee reasoning.

We define fenced refinement in two steps. First, we use *data structure projection* to project the relevant data structure instance from traces. The projection takes the denotation H (a set of heaps) of the predicate I fencing a data structure, along with a set of traces T, and yields those traces without the resources outside of the data structure. In addition, it filters the traces to a particular start and end state (o and o') of the private resources. This reflects the fact that in fenced refinement, we view behavior using a trace semantics for the shared data (so that we can handle interference) and a sequential semantics for the private data (since it cannot be interfered with). Finally, we apply *rely closure* to express our assumption about interference on the shared data. Any interference outside of those assumptions results in a fault, just as in data abstraction.

Two important theorems hold about the system we have presented. First, the denotational semantics of specifications (without **abs**) is *adequate* for the natural operational semantics, where we observe termination of a closed program. It follows that $\varphi \sqsubseteq \cdots \sqsubseteq \langle \overline{x} : p, q \rangle$ for a program φ means the program will not fault if p is satisfied, and will leave the heap satisfying q—even if the $\cdots$ steps included uses of data abstraction. Second, the proof rules we have presented—including congruence—are sound for the model. The proofs of these theorems will appear in a forthcoming technical report.

5 Evaluation and related work

In the last several years, there has been stunning progress in the formal verification of fine-grained concurrent data structures [7–10], 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 data abstraction. While the logic resulting from our work does offer some significant new features, we do not claim that it is more practical or applicable than the extant approaches—only that it provides some additional insight, and more modularity.

The basic form of our calculus clearly bears resemblance to Elmas *et al.*'s calculus of atomic actions [7, 8]. The key idea in that work is to combine Lipton's technique of *reduction* [4] 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, which makes reasoning about the heap much more pleasant.

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 [20]. Here we are influenced by Filipović *et al.* [21], 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 the necessary assumptions about ownership.

Our use of separation logic and rely/guarantee clearly derives from Vafeiadis *et al.*'s work [9], especially Vafeiadis's groundbreaking thesis [10]. 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.

Separation logic has already been used to localize rely/guarantee reasoning in Feng's work [19], from which we borrow the "fence" terminology. In Feng's paper, rely/guarantee reasoning for a memory region can be tied to a particular part of the program where interference may occur. After proving that part of the program, the rely and guarantee are removed, and no larger context using that memory in parallel can be adjoined. In contrast, we localize rely/guarantee reasoning to a data structure definition, even though the memory associated with that data structure is used in arbitrary parallel contexts. We believe these use cases are complementary. A further difference, though, is that our rely condition grows to encompass arbitrarily many instances of a data structure, by describing the expected interference on an instance in general.

Bierman and Parkinson [16], and Filipović *et al.* [22] have investigated 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 merit further investigation.

References

1. O'Hearn, P.W.: Resources, concurrency, and local reasoning. Theor. Comput. Sci. **375**(1-3) (2007) 271–307
2. Morgan, C., Vickers, T.e.: On the refinement calculus. Springer (1993)
3. Back, R.J., von Wright, J.: Refinement calculus: a systematic introduction. Springer (1998)
4. Lipton, R.J.: Reduction: a method of proving properties of parallel programs. Commun. ACM **18**(12) (1975) 717–721
5. Jones, C.B.: Tentative steps toward a development method for interfering programs. ACM Trans. Program. Lang. Syst. **5**(4) (1983) 596–619
6. Brookes, S.: Full abstraction for a shared variable parallel language. Information and computation **127**(2) (1996) 145–163
7. Elmas, T., Qadeer, S., Tasiran, S.: A calculus of atomic actions. In: POPL, ACM (2009) 2–15
8. Elmas, T., Qadeer, S., Sezgin, A., Subasi, O., Tasiran, S.: Simplifying linearizability proofs with reduction and abstraction. In: TACAS, Springer (2010) 296–311
9. Vafeiadis, V., Parkinson, M.: A marriage of rely/guarantee and separation logic. In: CONCUR, Springer (2007) 256–271
10. Vafeiadis, V.: Modular fine-grained concurrency verification. PhD thesis, University of Cambridge (2008)
11. Moir, M., Shavit, N.: Concurrent data structures. In: Handbook of Data Structures and Applications, D. Metha and S. Sahni Editors. (2007) 47–14 47–30 Chapman and Hall/CRC Press.
12. Michael, M.M., Scott, M.L.: Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. J. Parallel Distrib. Comput. **51**(1) (1998) 1–26
13. Hoare, C.A.R., Hanna, F.K.: Programs are predicates [and discussion]. Philosophical Transactions of the Royal Society of London. Series A, Mathematical and Physical Sciences **312**(1522) (1984) 475–489
14. Calcagno, C., O'Hearn, P.W., Yang, H.: Local action and abstract separation logic. In: LICS, IEEE Computer Society (2007) 366–378
15. van Glabbeek, R.J.: The linear time - branching time spectrum. In: CONCUR, Springer (1990) 278–297
16. Parkinson, M., Bierman, G.: Separation logic and abstraction. POPL **40**(1) (2005) 247–258
17. Treiber, R.K.: Systems programming: coping with parallelism. Technical report, Almaden Research Center (1986)
18. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann (2008)
19. Feng, X.: Local rely-guarantee reasoning. SIGPLAN Not. **44**(1) (2009) 315–327
20. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3) (1990) 463–492
21. Filipović, I., O'Hearn, P., Rinetzky, N., Yang, H.: Abstraction for concurrent objects. In: ESOP, Springer (2009) 252–266
22. Filipović, I., OHearn, P., Torp-Smith, N., Yang, H.: Blaming the client: on data refinement in the presence of pointers. In: FACS, Springer (2009)