Expressive Modular Fine-Grained Concurrency Specification

Bart Jacobs * Frank Piessens

DistriNet Research Group, Department of Computer Science, Katholieke Universiteit Leuven, Belgium

{bart.jacobs,frank.piessens}@cs.kuleuven.be

Abstract

Compared to coarse-grained external synchronization of operations on data structures shared between concurrent threads, fine-grained, internal synchronization can offer stronger progress guarantees and better performance. However, fully specifying operations that perform internal synchronization modularly is a hard, open problem. The state of the art approaches, based on linearizability or on concurrent abstract predicates, have important limitations on the expressiveness of specifications. Linearizability does not support ownership transfer, and the concurrent abstract predicates-based specification approach requires hardcoding a particular usage protocol. In this paper, we propose a novel approach that lifts these limitations and enables fully general specification of fine-grained concurrent data structures. The basic idea is that clients pass the ghost code required to instantiate an operation's specification for a specific client scenario into the operation in a simple form of higher-order programming.

We machine-checked the theory of the paper using the Coq proof assistant. Furthermore, we implemented the approach in our program verifier VeriFast and used it to verify two challenging fine-grained concurrent data structures from the literature: a multiple-compare-and-swap algorithm and a lock-coupling list.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Specification techniques

General Terms Verification

Keywords fine-grained concurrency, separation logic

1. Introduction

34 years after Owicki and Gries (O&G) proposed their resource-invariants-based (RI) method [12] and their interference-freedom-checks-based (IF) method [13] for the verification of parallel programs, doing so fully modularly is still an area of active research. For parallel programs, two kinds of modularity can be distinguished: thread-modularity and procedure-modularity.

Thread-modularity means that each thread can be verified separately, under a well-defined, concise set of assumptions on its environment. The RI method satisfies this criterion, since the resource invariants are the only shared assumptions among the threads. The IF method does not, since it requires each command of each thread to be checked for interference with each command of each other thread. The latter problem was solved by the rely-guarantee (RG) method [9], which summarizes each thread's interference assumptions and guarantees in a single rely, resp. guarantee condition.

Procedure-modularity means that each procedure, or each group of procedures that cooperate to implement an abstract data type, can be verified separately, again under a well-defined, concise set of assumptions on its environment that performs proper abstraction over implementation aspects. Neither the RI method nor RG satisfy this criterion. RI, because it requires auxiliary variable annotations that break modularity, as we will show; and RG, because it does not allow as-if-atomic operations on data structures to be treated just like atomic machine instructions, although this has been addressed recently with work on linearizability-based verification [6, 15].

In this paper, we propose an extension of the RI method that achieves procedure-modularity. The basic idea is simple: at each procedure call, the auxiliary variable updates required to enable verification of the client program are passed into the procedure as an extra argument. Correspondingly, the procedure's specification is parameterized by a precondition and a postcondition for the updates, and imposes a correctness condition on the updates in the form of a Hoare triple.

We first describe the approach informally in the context of an informal extension of the RI method with procedures. However, since the RI method imposes syntactic restrictions on which threads may mention specific variables, an extension with procedures requires more involved bookkeeping of variable occurrences. These details are not very interesting, and we do not develop this setting formally; rather, our formal system, like our implementation, uses separation logic [11, 14], where these issues do not occur.

We implemented the approach in our program verifier VeriFast and verified two challenging fine-grained concurrent data structures from the literature: a multiple-compare-and-swap (MCAS) algorithm [5], and a lock-coupling list.

The remainder of the paper is structured as follows. In §2 we recall the RI method. In §3 we show that this method is not procedure-modular. In §4, we informally present our approach for extending RI to achieve procedure-modularity. In §5, we describe how our approach may be lifted to a dynamic setting using separation logic and permission accounting. In §6, we introduce *ghost objects*, data structures constructed from auxiliary heap cells that enable partial information sharing. In §7, we describe how programs that use atomic machine instructions can be encoded straightforwardly into the system of this paper. In §8, we describe our proof of a concurrent set algorithm. In §9 we describe the verification tool. Finally, we discuss related work in §10.

2. The Owicki-Gries Method

Consider the simple parallel program used by O&G [12] to introduce their resource-invariant-based method, reproduced in Fig-

* This work was performed while Bart Jacobs was a Postdoctoral Fellow of the Research Foundation - Flanders (FWO)

```
**resource** $r(x)$ : **cobegin**
    **with** $r$ **when** $true$ **do** $x := x + 1$
  //
    **with** $r$ **when** $true$ **do** $x := x + 1$
**coend**
```

Figure 1. A simple parallel program; can be verified with resource invariants and auxiliary variables

ure 1. It uses the parallel execution command

$$\textbf{cobegin } S_1 // \ldots // S_n \textbf{ coend}$$

to run two threads that each increment variable x. Variable x is protected by resource r: the critical section command **with** r **when** B **do** S blocks until B is true and no other thread is using r.

In this simple language, data races can be avoided by imposing the following simple syntactic restrictions:

- An assignment to a variable x that belongs to a resource r is allowed only inside a critical section for r.
- An occurrence of a variable x outside of a critical section for a resource to which it belongs, if any, is allowed only if no other thread modifies the variable.

O&G propose the following axioms for parallel executions and critical sections, similar to the ones proposed by Hoare but with relaxed conditions on variable occurrences. The axioms assume that an assertion $I(r)$ has been defined for each resource r, called the resource's *resource invariant*.

Parallel Execution Axiom. If $\{P_1\}\ S_1\ \{Q_1\}$ and $\{P_2\}\ S_2\ \{Q_2\}$ and …and $\{P_n\}\ S_n\ \{Q_n\}$ and no variable free in P_i or Q_i is changed in S_j ($i \neq j$) and all variables in $I(r)$ belong to resource r, then $\{P_1 \wedge \cdots \wedge P_n \wedge I(r)\}$ **resource** r : **cobegin** $S_1 // \ldots // S_n$ **coend** $\{Q_1 \wedge \cdots \wedge Q_n \wedge I(r)\}$.

Critical Section Axiom. If $\{I(r) \wedge P \wedge B\}\ S\ \{I(r) \wedge Q\}$, and $I(r)$ is the invariant from the **cobegin** command, and no variable free in P or Q is changed in another process, then $\{P\}$ **with** r **when** B **do** S $\{Q\}$.

We would like to prove that if $x = 0$ before the program executes, then $x = 2$ after the program terminates. As O&G point out, this property cannot be verified using the above axioms directly. They propose to augment the program with *auxiliary variables* y and z, that track each thread's contribution to the value of x. Figure 2 shows the proof outline given by O&G for the augmented program.

Notice that the auxiliary variables are mentioned in assertions outside of the **with do** commands, even though they are protected by resource r; this is allowed provided that the thread that mentions a given variable is the only one that modifies that variable, per the Parallel Execution Axiom.

To regulate reasoning using auxiliary variables, O&G propose auxiliary variable sets and the Auxiliary Variable Axiom.

Definition. A set AV of program variables is an *auxiliary variable set* for a given program if variables in AV appear in the program only in assignments to variables in AV.

Auxiliary Variable Axiom. If AV is an auxiliary variable set for S, let S' be obtained from S by deleting all assignments to variables in AV. Then if $\{P\}\ S\ \{Q\}$ is true and P and Q do not refer to any variables from AV, $\{P\}\ S'\ \{Q\}$ is also true.

3. The Procedure-Modularity Problem

Now consider again the un-augmented program of Figure 1. Suppose we wish to encapsulate x and the operation on it, i.e. the in-

```
$\{x = 0\}$
**begin** $y := 0; z := 0;$
  $\{y = 0 \wedge z = 0 \wedge I(r)\}$
  **resource** $r(x, y, z)$ : **cobegin**
    $\{y = 0\}$
    **with** $r$ **when** $true$ **do**
      $\{y = 0 \wedge I(r)\}$
      **begin** $x := x + 1; y := 1$ **end**
      $\{y = 1 \wedge I(r)\}$
    $\{y = 1\}$
  //
    $\{z = 0\}$
    **with** $r$ **when** $true$ **do**
      $\{z = 0 \wedge I(r)\}$
      **begin** $x := x + 1; z := 1$ **end**
      $\{z = 1 \wedge I(r)\}$
    $\{z = 1\}$
  **coend**
  $\{y = 1 \wedge z = 1 \wedge I(r)\}$
**end**
$\{x = 2\}$
$I(r) = \{x = y + z\}$
```

Figure 2. Owicki and Gries' proof of the program of Figure 1. y and z are auxiliary variables.

```
**procedure** $incr()$ **do** $x := x + 1$
**resource** $r(x)$ : **cobegin**
    **with** $r$ **when** $true$ **do** $incr()$
  //
    **with** $r$ **when** $true$ **do** $incr()$
**coend**
```

(a) External synchronization

```
**procedure** $incr(r)$ **do with** $r$ **when** $true$ **do** $x := x + 1$
**resource** $r(x)$ : **cobegin**
    $incr(r)$
  //
    $incr(r)$
**coend**
```

(b) Internal synchronization

Figure 3. Two modularized versions of the program of Figure 1

crement operation, into a separate module. (Note: neither the programming language of O&G, nor their proof system, support procedures, since both impose syntactic conditions on the threads where variables occur, and this is not well-defined in the presence of procedures. In this section and the next, we informally introduce our approach, while glossing over this issue. In Section 5 we present our approach formally, using heap cells instead of global variables, thus eliminating this issue.) There are two ways to do this: using external synchronization (Figure 3a) and using internal synchronization (Figure 3b).

In the version that uses external synchronization, the module is easy to specify: procedure $incr$ satisfies the following specification: $\{x = X\}\ incr()\ \{x = X + 1\}$ where X is a logical variable; the specification holds for all values of X. Using this specification, both module and client program are easy to verify; the proof outline of Figure 2 is mostly unchanged. The reason is that the auxiliary variables can be added in the client program; no augmentation of the module is required.

```
procedure $incr(r, \rho)$ do
  with $r$ when $true$ do begin $x := x + 1;\ \rho$ end
begin $y := 0; z := 0;$
  resource $r(x, y, z)$ : cobegin
    $incr(r, y := 1)$
  //
    $incr(r, z := 1)$
  coend
end
```

Figure 4. The program of Figure 3b, augmented per our approach

For the version that uses internal synchronization, this is not the case. The updates of y and z need to be added inside the **with do** command, but this command is in the module and furthermore different updates need to be added for different call sites. One might then wonder whether insisting on internal synchronization is worthwhile; it is, because delegating synchronization to the module allows the module to perform *fine-grained synchronization*, for example by acquiring locks multiple times for smaller amounts of time, or by using atomic machine instructions such as compare-and-swap.

One can easily see that verifying this program with the Owicki-Gries method is impossible. Indeed, consider any augmentation of the program with auxiliary variable assignments, and any proof outline for the augmented program. Since the assignments inside the critical section occur in both threads, no variable modified inside the critical section may be mentioned by any thread's proof outline outside the critical section. Therefore, removing the critical section from the program does not invalidate the proof outline. Consequently, the proof outline cannot verify the triple $\{x = 0\} \cdot \{x = 2\}$.

4. Achieving Procedure-Modularity

In order to enable a modular specification of the module of Figure 3b, we propose to augment the program not just with auxiliary variables, but with a simple form of higher-order programming to allow the client program to pass auxiliary variable updates into the module. Specifically, we augment procedure $incr$ with a parameter ρ that ranges over *commands*, and its body so that it executes ρ after the update of x inside the critical section. In the client program, at each call of $incr$, the appropriate auxiliary variable update is specified as the value for parameter ρ. The augmented program is shown in Figure 4.

The specification of $incr$ is now more involved:

$$\frac{\begin{array}{c} x \notin FV(P, U, Q) \\ P \wedge I(r) \Rightarrow U(x + 1) \qquad \{U(x)\}\ \rho\ \{Q \wedge I(r)\} \end{array}}{\{P\}\ incr(r, \rho)\ \{Q\}}$$

The specification is universally quantified over the predicates P, Q, and U; it can be instantiated with appropriate predicates at each call site. Notice also that the specification is generic in the resource invariant. The resource invariant for the resource that protects a fine-grained concurrent data structure is chosen by the client of the data structure. This enables the client to specify the relationship between the state of the data structure and the auxiliary variables introduced by the client.

It is important to point out that although the specification of $incr$ looks like a proof rule, it is not part of the proof system and it does not affect the soundness of the proof system. Rather, it is a derived proof rule that must be verified starting from the proof rules of the proof system.

```
$\{P\}$
with $r$ when $true$ do begin
  $\{P \wedge I(r)\}$
  $\{U(x + 1)\}$
  $x := x + 1;$
  $\{U(x)\}$
  $\rho$
  $\{Q \wedge I(r)\}$
end
$\{Q\}$
```

Figure 5. Proof outline for procedure $incr$

```
$\{x = 0\}$
begin $y := 0; z := 0;$
  $\{y = 0 \wedge z = 0 \wedge I(r)\}$
  resource $r(x, y, z)$ : cobegin
    $\{y = 0\}$
    $incr(r, y := 1)$
      with $P \equiv y = 0; Q \equiv y = 1; U(X) \equiv X = 1 + z$
    $\{y = 1\}$
  //
    $\{z = 0\}$
    $incr(r, z := 1)$
      with $P \equiv z = 0; Q \equiv z = 1; U(X) \equiv X = y + 1$
    $\{z = 1\}$
  coend
  $\{y = 1 \wedge z = 1 \wedge I(r)\}$
end
$\{x = 2\}$
$I(r) = \{x = y + z\}$
```

Figure 6. Proof outline for the client program

It is easy to see that the implementation of $incr$ satisfies the specification; a proof outline is shown in Figure 5.

The proof of the client program is equally easy; see the proof outline in Figure 6.

5. Formal System

We presented our approach informally in the preceding sections. In order to achieve a well-defined approach, we need to resolve the problem of O&G's syntactic restrictions on which threads may mention which variables; these are not compatible with procedures. To do so, we move to a programming language without global variables, where threads share data only through the heap; and we use separation logic to reason about such programs. Specifically, we adopt the programming language and program logic of Gotsman et al. [4] for storable locks and threads, with a few modifications:

- We add support for auxiliary heap cells and passing closed commands into procedures as argument values.
- We do not treat local variables as resources.

A translation of the modularized Owicki-Gries example with internal synchronization of Figure 3 (b) to the more dynamic programming language is shown in Figure 7. The program consists of a procedure $incr$ and a main program. The main program allocates two consecutive memory cells and it initializes the first one (at address ℓ) for use as a lock, and releases it. (After a thread initializes a lock, it initially holds it.) The second cell (at address $\ell{+}1$) corresponds to the global variable x in the original program. The program then starts two threads, both of which increment the sec-

```
procedure $incr(\ell)$ =
    acquire($\ell$); $r := [\ell + 1]$; $[\ell + 1] := r + 1$; release($\ell$)
$\ell :=$ cons(1, 0); init($\ell$); release($\ell$);
$t_1 :=$ fork $incr(\ell)$;
$t_2 :=$ fork $incr(\ell)$;
join($t_1$); join($t_2$);
acquire($\ell$); finalize($\ell$)
```

Figure 7. The Owicki-Gries example, translated into the dynamic programming language

ond cell, under protection of the lock. Finally, it joins both threads, acquires the lock, and decommissions it. (A thread may only decommission locks that it holds.) We wish to prove that when the program terminates, we have $\ell + 1 \mapsto 2 * \mathbf{true}$.

The original proof by O&G used auxiliary global variables. In this section, we use *auxiliary heap cells* instead. Specifically, with each allocated real heap cell, say at address ℓ, we associate an infinite number of auxiliary heap cells, whose address is given by a pair of integers $\ell.\ell'$, where ℓ is the real address and ℓ' is the *ghost offset*.[1] In the example, we use the auxiliary heap cells at addresses $\ell.0$ and $\ell.1$ to track the contributions of thread 1 and 2 to the value of the cell at $\ell + 1$, corresponding to auxiliary variables y and z in the original proof.

The proof system of Gotsman et al. requires that a *tag* A be associated with each lock, and a *lock invariant* I_A with each tag. We will associate the tag mylock with the lock of the example, and the following lock invariant with the tag:

$$I_{\text{mylock}}(\ell) \ = \\ \exists C_0, C_1 \bullet \ell.0 \overset{1/2}{\mapsto} C_0 * \ell.1 \overset{1/2}{\mapsto} C_1 * \ell + 1 \mapsto C_0 + C_1$$

The lock invariant corresponds closely to the resource invariant of the original example. It states that the value of $\ell + 1$ is the sum of the value of $\ell.0$ and $\ell.1$. In the original proof, syntactic restrictions ensured that auxiliary variable y could be modified only inside a critical section and only by the first thread; in the current proof, fractional permissions [1] achieve the same goal. Specifically, one half of the permission for each auxiliary heap cell becomes owned by the lock; the other half is retained by the corresponding thread.

As in the previous section, to verify procedure *incr*, we start by augmenting it with an auxiliary parameter ρ that ranges over auxiliary commands, and by augmenting its body with an occurrence of ρ after the update of $\ell + 1$ but before the lock is released. As before, this parameter will serve to perform the auxiliary state updates required to preserve the lock invariant. The specification of procedure *incr* enforces that it does so:

$$\frac{\begin{array}{c} I_A(\ell) * P \Rightarrow \exists X \bullet \ell + 1 \mapsto X * U(X) \\ \forall X \bullet \{\ell + 1 \mapsto X + 1 * U(X)\}\ \rho\ \{I_A(\ell) * Q\} \end{array}}{\{\pi A(\ell) * P\}\ incr(\ell, \rho)\ \{\pi A(\ell) * Q\}}$$

The specification is universally quantified over the address ℓ of the lock, the tag A of the lock, the fraction π of the lock permission available to the procedure (any fraction will do), an additional precondition P and postcondition Q, and a predicate $U(X)$, parameterized over an integer X, that describes the resources (specifically, the auxiliary heap cells) owned by the lock besides the heap cell at address $\ell + 1$, and states that those resources are in a state corresponding to value X of the heap cell at address $\ell + 1$.

The specification has two premises. The first one states that the lock invariant $I_A(\ell)$ combined with the additional precondition P implies full permission to access the heap cell at address $\ell + 1$, plus some extra state $U(X)$, where X is the value of the heap cell. The second premise states the correctness of the parameter ρ: it states that executing command ρ must re-establish the lock invariant after the update of $\ell + 1$, and the remaining state must satisfy Q.

We again point out that this specification has the shape of a proof rule, but is not part of the proof system; as always when verifying programs with procedures, the specifications of the procedures must be derived using the proof rules of the system as part of the verification of the program.

A proof outline of the program is shown in Figure 8. Notice the following:

- The release of the lock consumes the lock invariant.
- A thread specification and thread specification arguments are associated with each **fork** operation for verification purposes. A fork with thread specification τ and arguments $\overline{n}$ consumes the precondition of τ and produces a thread permission $\mathsf{tid}_\tau(t, \overline{n})$, where t is the thread identifier.
- Joining a thread consumes the thread permission and produces the thread specification's postcondition.
- The acquisition of the lock produces the lock invariant. Merging the fractions of the auxiliary heap cells yields full information about $\ell + 1$, per the following law:

$$(a \overset{1/2}{\mapsto} v * \exists C \bullet a \overset{1/2}{\mapsto} C * P(C)) \Rightarrow a \mapsto v * P(v)$$

5.1 Programming Language

The syntax of arithmetic expressions e, boolean expressions b, and commands c is given below. All commands return a value. Local variables are scoped, using **let** commands. The syntax $x := c; y := c'; c''$ is syntactic sugar for $\mathbf{let}\ x\ :=\ c\ \mathbf{in}\ \mathbf{let}\ y\ :=\ c'\ \mathbf{in}\ c''$. We assume a global table $\overline{pdef}$ of procedure definitions. The recursion operator $(\mu f(\overline{x}) \bullet c)(\overline{e})$ applies the recursive function f with parameters $\overline{x}$ and body c to arguments $\overline{e}$. The scope of f is c, minus any command expressions in c. The syntax $\mathbf{letrec}\ f(\overline{x}) = c\ \mathbf{in}\ c'$ is syntactic sugar for $c'[(\mu f(\overline{x}) \bullet c)/f]$. All substitutions are capture-avoiding. A command is *closed* if it has no free variables $x \in \mathsf{Vars}$ and no free functions $f \in \mathsf{FuncNames}$. We assume a bijective encoding $\lfloor\cdot\rfloor$ of closed commands into integers. A command expression c denotes the encoding of c as an integer. A closure execution command $\mathbf{exec}(e)$ executes the closed command obtained by decoding the value of e. In the examples, we simply write ρ instead of $\mathbf{exec}(\rho)$. That is, a variable name used as a command denotes a closure execution.

$$\begin{array}{rl}
 & n \in \mathbb{Z}, x \in \mathsf{Vars}, p \in \mathsf{ProcNames}, f \in \mathsf{FuncNames} \\
e ::= & n \mid x \mid e + e \mid e - e \mid c \\
b ::= & e = e \mid e < e \\
c ::= & \mathbf{cons}(\overline{e}) \mid \mathbf{gcons}(e) \mid [e] \mid [e.e] \mid [e] := e \mid [e.e] := e \\
 & \mid \mathbf{dispose}(e) \mid \mathbf{if}\ b\ \mathbf{then}\ c\ \mathbf{else}\ c \mid \mathbf{return}\ e \\
 & \mid p(\overline{e}) \mid \mathbf{exec}(e) \mid \mathbf{let}\ x := c\ \mathbf{in}\ c \\
 & \mid (\mu f(\overline{x}) \bullet c)(\overline{e}) \mid f(\overline{e}) \mid \mathbf{fork}\ c \mid \mathbf{join}(e) \\
 & \mid \mathbf{init}_A(e) \mid \mathbf{acquire}(e) \mid \mathbf{release}(e) \mid \mathbf{finalize}(e) \\
pdef ::= & \mathbf{procedure}\ p(\overline{x}) = c
\end{array}$$

The evaluation $[\![e]\!]$ of a closed expression e is defined as follows:

$$[\![n]\!] = n \quad [\![e{+}e']\!] = [\![e]\!]{+}[\![e']\!] \quad [\![e{-}e']\!] = [\![e]\!]{-}[\![e']\!] \quad [\![c]\!] = \lfloor c \rfloor$$

We define a small-step interleaving semantics. A configuration consists of a real heap h, a ghost heap g, and a thread map T. A real heap is a finite partial function from positive integers to integers. A ghost heap is a partial function from pairs of integers to integers. A thread map is a finite partial function from thread identifiers to closed *continuations*. The continuations κ and *contexts* ξ are

[1] We use the terms *auxiliary* and *ghost* interchangeably.

```
**procedure** $incr(\ell, \rho)$ =
  $\{\pi A(\ell) * P\}$
  **acquire**($\ell$);
  $\{\pi A(\ell) * \mathsf{locked}_A(\ell) * I_A(\ell) * P\}$
  $\{\pi A(\ell) * \mathsf{locked}_A(\ell) * \ell + 1 \mapsto X * U(X)\}$
  $r := [\ell + 1]; [\ell + 1] := r + 1;$
  $\{\pi A(\ell) * \mathsf{locked}_A(\ell) * \ell + 1 \mapsto X + 1 * U(X)\}$
  $\rho;$
  $\{\pi A(\ell) * \mathsf{locked}_A(\ell) * I_A(\ell) * Q\}$
  **release**($\ell$),
  $\{\pi A(\ell) * Q\}$
**threadspec** $thread1(\ell)$
  **req** $\frac{1}{2}\mathsf{mylock}(\ell) * \ell.0 \overset{1/2}{\mapsto} 0$
  **ens** $\frac{1}{2}\mathsf{mylock}(\ell) * \ell.0 \overset{1/2}{\mapsto} 1$
**threadspec** $thread2(\ell)$
  **req** $\frac{1}{2}\mathsf{mylock}(\ell) * \ell.1 \overset{1/2}{\mapsto} 0$
  **ens** $\frac{1}{2}\mathsf{mylock}(\ell) * \ell.1 \overset{1/2}{\mapsto} 1$

$\{\mathbf{emp}\}$
$\ell :=$ **cons**$(1, 0)$;
$\{\ell \mapsto 1 * (\circledast_{\ell' \in \mathbb{N}} \ell.\ell' \mapsto 0) * \ell + 1 \mapsto 0 * (\circledast_{\ell' \in \mathbb{N}} (\ell + 1).\ell' \mapsto 0)\}$
$\{\ell \mapsto 1 * \ell.0 \mapsto 0 * \ell.1 \mapsto 0 * \ell + 1 \mapsto 0 * \mathbf{true}\}$
**init**$_{\mathsf{mylock}}(\ell)$; **release**$(\ell)$;
$\{\mathsf{mylock}(\ell) * \ell.0 \overset{1/2}{\mapsto} 0 * \ell.1 \overset{1/2}{\mapsto} 0 * \mathbf{true}\}$
$t_1 :=$ **fork**
  $\{\frac{1}{2}\mathsf{mylock}(\ell) * \ell.0 \overset{1/2}{\mapsto} 0\}$
  $incr(\ell, [\ell.0] := 1);$
    with $U(X) = \ell.0 \mapsto 0 * \ell.1 \overset{1/2}{\mapsto} X$
  $\{\frac{1}{2}\mathsf{mylock}(\ell) * \ell.0 \overset{1/2}{\mapsto} 1\}$
$\{\frac{1}{2}\mathsf{mylock}(\ell) * \ell.1 \overset{1/2}{\mapsto} 0 * \mathsf{tid}_{thread1}(t_1, \ell) * \mathbf{true}\}$
$t_2 :=$ **fork**
  $\{\frac{1}{2}\mathsf{mylock}(\ell) * \ell.1 \overset{1/2}{\mapsto} 0\}$
  $incr(\ell, [\ell.1] := 1);$
    with $U(X) = \ell.0 \overset{1/2}{\mapsto} X * \ell.1 \mapsto 0$
  $\{\frac{1}{2}\mathsf{mylock}(\ell) * \ell.1 \overset{1/2}{\mapsto} 1\}$
$\{\mathsf{tid}_{thread1}(t_1, \ell) * \mathsf{tid}_{thread2}(t_2, \ell) * \mathbf{true}\}$
**join**$(t_1)$;
$\{\frac{1}{2}\mathsf{mylock}(\ell) * \ell.0 \overset{1/2}{\mapsto} 1 * \mathsf{tid}_{thread2}(t_2, \ell) * \mathbf{true}\}$
**join**$(t_2)$;
$\{\mathsf{mylock}(\ell) * \ell.0 \overset{1/2}{\mapsto} 1 * \ell.1 \overset{1/2}{\mapsto} 1 * \mathbf{true}\}$
**acquire**$(\ell)$; **finalize**$(\ell)$
$\{(\exists C_0, C_1 \bullet \ell.0 \overset{1/2}{\mapsto} C_0 * \ell.1 \overset{1/2}{\mapsto} C_1 * \ell + 1 \mapsto C_0 + C_1)$
  $* \ell.0 \overset{1/2}{\mapsto} 1 * \ell.1 \overset{1/2}{\mapsto} 1 * \ell \mapsto \_ * \mathbf{true}\}$
$\{\ell.0 \mapsto 1 * \ell.1 \mapsto 1 * \ell + 1 \mapsto 2 * \ell \mapsto \_ * \mathbf{true}\}$
```

Figure 8. Proof outline for the example program

defined as follows:

$$\begin{array}{lcl} \kappa & ::= & c; \xi \mid n; \xi \\ \xi & ::= & \mathbf{let}\ x := [\,]\ \mathbf{in}\ c; \xi \mid \mathbf{done} \end{array}$$

The step relation $\rightsquigarrow$ is defined in Figure 9. In the step rules, symbols n match not just integer literals but other closed expressions as well, and denote their value. Notice that locks are implemented as a single heap cell that holds either the value 0, if the lock is not held, or the value 1, if the lock is held. We omit rules for **init** and **finalize**; we define them as equivalent to **return** 0 (i.e., a no-op) for purposes of the step relation. Throughout, $f[x := y]$ denotes function update; i.e. $f[x := y](x) = y$ and $f[x := y](z) = f(z)$ for $z \neq x$.

5.2 Simple Closures

We say a program has *simple closures* if there exists a partitioning of procedure parameters into closure parameters and non-closure parameters such that all **exec** commands are of the form **exec**(x) where x is a closure parameter, and all procedure call argument expressions for closure parameters are either command expressions or closure parameters. Applying the specification approach of this paper requires only simple closures. As we will see, simple closures admit a very simple proof system.

5.3 Proof System

The correctness of a command c is expressed in the form of a correctness judgment $\Gamma \vdash \{P\}\ c\ \{Q\}$, where Γ is a function environment and P and Q are *assertions*. An assertion describes a set of *permissions*. The set of permissions is defined as follows:

$$\hat{p} ::= \ell \mapsto v \mid \ell.\ell' \mapsto v \mid A(\ell) \mid \mathsf{locked}_A(\ell) \mid \mathsf{tid}_\tau(t, \overline{v})$$

A *permission bundle* is a total function from permissions to real numbers between 0, inclusive and 1, inclusive. We identify assertions with sets of permission bundles. That is, we treat assertions semantically. We denote the empty permission bundle (that maps all permissions to 0) as $\mathbf{0}$.

We define some syntax for assertions:

$$\begin{array}{l}
\mathbf{emp} = \{\mathbf{0}\} \\
\ell \overset{\pi}{\mapsto} v = \{\mathbf{0}[\ell \mapsto v := \pi]\} \\
\ell.\ell' \overset{\pi}{\mapsto} v = \{\mathbf{0}[\ell.\ell' \mapsto v := \pi]\} \\
\pi A(\ell) = \{\mathbf{0}[A(\ell) := \pi]\} \\
\pi \mathsf{tid}_\tau(t, \overline{v}) = \{\mathbf{0}[\mathsf{tid}_\tau(t, \overline{v}) := \pi]\} \\
P * Q = \{b \mid \exists b_1, b_2 \bullet b = b_1 + b_2 \wedge b_1 \in P \wedge b_2 \in Q\} \\
(\exists X \bullet P(X)) = \{b \mid \exists X \bullet b \in P(X)\} \\
(\circledast_{i \in \mathbb{N}}\ P(i)) = \{b \mid \exists B \bullet b = \Sigma_i B(i) \wedge \forall i \bullet B(i) \in P(i)\} \\
\quad \text{where } b = \Sigma_i B(i) \ \Leftrightarrow \\
\qquad (\forall \hat{p}, \varepsilon \bullet \exists n \bullet \forall i > n \bullet |b(\hat{p}) - \Sigma_{0 \leq j \leq i} B(j)(\hat{p})| < \varepsilon)
\end{array}$$

We say a permission bundle is *consistent* if there are no two points-to permissions with the same address and different values that both map to non-zero coefficients. We say one assertion A implies another one A', written $A \Rightarrow A'$, if for every *consistent* bundle $b \in A$, we have $b \in A'$.

The correctness judgment is defined inductively by the rules shown in Figure 10.

Notice that the proof rules for procedure calls and for closure executions simply require the correctness of the procedure or closure's body. It follows that a procedure that calls another procedure, or that executes a closure, does not, in isolation, have a closed proof tree. Rather, its proof tree is parameterized by the proof trees for the procedures called and closures executed. This simple approach is sufficient if the program has simple closures and an acyclic procedure call graph. Indeed, in that case, given a main command, one can inline all procedure calls to obtain an equivalent command that contains no procedure call or closure execution commands; the shape of the proof tree for the original main command will reflect the shape of the main command after inlining.

5.4 Soundness

We sketch how one can prove soundness of the program logic used. More details are in the technical report [7]; also, a machine-checked proof is at http://www.cs.kuleuven.be/~bartj/finegrained/.

$$\begin{array}{lll}
\multicolumn{3}{c}{\textbf{Continuation step rules} \text{ for relation } \langle h, g, \kappa\rangle \leadsto \langle h', g', \kappa'\rangle \mid \textbf{abort}} \\
\langle h, g, \textbf{cons}(n_1, \ldots, n_m); \xi\rangle & \leadsto \langle h \cup \{\ell \mapsto n_1, \ldots, \ell + m - 1 \mapsto n_m\}, g \cup \{(\ell_1, \ell_2) \mapsto 0 \mid \ell \leq \ell_1 < \ell + m\}, \ell; \xi\rangle & \\
 & & \text{if } 0 < \ell \text{ and } \{\ell, \ldots, \ell + m - 1\} \cap \mathrm{dom}(h) = \emptyset \\
\langle h, g, \textbf{gcons}(n); \xi\rangle & \leadsto \langle h, g \cup \{(0, \ell') \mapsto n\}, \ell'; \xi\rangle & \text{if } 0 < \ell' \wedge (0, \ell') \notin \mathrm{dom}(g) \\
\langle h, g, [n]; \xi\rangle & \leadsto \text{if } n \in \mathrm{dom}(h) \text{ then } \langle h, g, h(n); \xi\rangle \text{ else } \textbf{abort} & \\
\langle h, g, [n.n']; \xi\rangle & \leadsto \text{if } (n, n') \in \mathrm{dom}(g) \text{ then } \langle h, g, g((n, n')); \xi\rangle \text{ else } \textbf{abort} & \\
\langle h, g, [n] := v; \xi\rangle & \leadsto \text{if } n \in \mathrm{dom}(h) \text{ then } \langle h[n := v], g, 0; \xi\rangle \text{ else } \textbf{abort} & \\
\langle h, g, [n.n'] := v; \xi\rangle & \leadsto \text{if } (n, n') \in \mathrm{dom}(g) \text{ then } \langle h, g[(n, n') := v], 0; \xi\rangle \text{ else } \textbf{abort} & \\
\langle h, g, \textbf{dispose}(n); \xi\rangle & \leadsto \text{if } n \in \mathrm{dom}(h) \text{ then } \langle h \setminus_{\mathrm{dom}} \{n\}, g \setminus_{\mathrm{dom}} \{(n, \ell') | \textbf{true}\}, 0; \xi\rangle \text{ else } \textbf{abort} & \\
\langle h, g, \textbf{if } b \textbf{ then } c \textbf{ else } c'; \xi\rangle & \leadsto \text{if } b = \mathsf{true} \text{ then } \langle h, g, c; \xi\rangle \text{ else } \langle h, g, c'; \xi\rangle & \\
\langle h, g, p(\overline{v}); \xi\rangle & \leadsto \langle h, g, c[\overline{v}/\overline{x}]; \xi\rangle & \text{if } \textbf{procedure } p(\overline{x}) = c \\
\langle h, g, \textbf{exec}(\lfloor c \rfloor); \xi\rangle & \leadsto \langle h, g, c; \xi\rangle & \\
\langle h, g, \textbf{return } n; \xi\rangle & \leadsto \langle h, g, n; \xi\rangle & \\
\langle h, g, \textbf{let } x := c \textbf{ in } c'; \xi\rangle & \leadsto \langle h, g, c; \textbf{let } x := [\,] \textbf{ in } c'; \xi\rangle & \\
\langle h, g, (\mu f(\overline{x}) \bullet c)(\overline{v}); \xi\rangle & \leadsto \langle h, g, c[(\mu f(\overline{x}) \bullet c)/f, \overline{v}/\overline{x}]; \xi\rangle & \\
\langle h, g, \textbf{acquire}(n); \xi\rangle & \leadsto \langle h[n := 1], g, 0; \xi\rangle & \text{if } (n, 0) \in h \\
\langle h, g, \textbf{acquire}(n); \xi\rangle & \leadsto \textbf{abort} & \text{if } n \notin \mathrm{dom}(h) \\
\langle h, g, \textbf{release}(n); \xi\rangle & \leadsto \text{if } n \in \mathrm{dom}(h) \text{ then } \langle h[n := 0], g, 0; \xi\rangle \text{ else } \textbf{abort} & \\
\langle h, g, v; \textbf{let } x := [\,] \textbf{ in } c; \xi\rangle & \leadsto \langle h, g, c[v/x]; \xi\rangle & \\
\multicolumn{3}{c}{\textbf{Thread map step rules} \text{ for relation } \langle h, g, T\rangle \leadsto \langle h', g', T'\rangle \mid \textbf{abort}} \\
\langle h, g, T\rangle & \leadsto \langle h', g', T[t := \kappa']\rangle & \text{if } (t, \kappa) \in T \text{ and } \langle h, g, \kappa\rangle \leadsto \langle h', g', \kappa'\rangle \\
\langle h, g, T\rangle & \leadsto \textbf{abort} & \text{if } (t, \kappa) \in T \text{ and } \langle h, g, \kappa\rangle \leadsto \textbf{abort} \\
\langle h, g, T\rangle & \leadsto \langle h, g, T[t := t'; \xi, t' := c; \textbf{done}]\rangle & \text{if } (t, \textbf{fork } c; \xi) \in T \text{ and } t' \notin \mathrm{dom}(T) \\
\langle h, g, T\rangle & \leadsto \langle h, g, T[t := v; \xi] \setminus_{\mathrm{dom}} \{t'\}\rangle & \text{if } (t, \textbf{join}(t'); \xi) \in T \text{ and } (t', v; \textbf{done}) \in T \\
\langle h, g, T\rangle & \leadsto \textbf{abort} & \text{if } (t, \textbf{join}(n); \xi) \in T \text{ and } n \notin \mathrm{dom}(T)
\end{array}$$

Figure 9. Step rules for continuations κ and thread maps T. Note: **init** and **finalize** are no-ops.

$$\{\textbf{emp}\}\ r := \textbf{cons}(v_0, \ldots, v_n)\ \{\circledast_i\, r + i \mapsto v_i * (\circledast_{\ell' \in \mathbb{N}}\, (r + i).\ell' \mapsto 0)\} \qquad \{\textbf{emp}\}\ r := \textbf{gcons}(v)\ \{0.r \mapsto v\}$$

$$\{\ell \overset{\pi}{\mapsto} v\}\ r := [\ell]\ \{\ell \overset{\pi}{\mapsto} v \wedge r = v\} \qquad \{\ell.\ell' \overset{\pi}{\mapsto} v\}\ r := [\ell.\ell']\ \{\ell.\ell' \overset{\pi}{\mapsto} v \wedge r = v\} \qquad \{\ell \mapsto _\}\ [\ell] := v\ \{\ell \mapsto v\}$$

$$\{\ell.\ell' \mapsto _\}\ [\ell.\ell'] := v\ \{\ell.\ell' \mapsto v\} \qquad \{\ell \mapsto _ * \circledast_{\ell' \in \mathbb{N}} \ell.\ell' \mapsto _\}\ \textbf{dispose}(\ell)\ \{\textbf{emp}\} \qquad \{P\}\ r := \textbf{return}\ v\ \{P \wedge r = v\}$$

$$\frac{\Gamma \vdash \{P \wedge b\}\ c\ \{Q\} \qquad \Gamma \vdash \{P \wedge \neg b\}\ c'\ \{Q\}}{\Gamma \vdash \{P\}\ \textbf{if } b \textbf{ then } c \textbf{ else } c'\ \{Q\}} \qquad \frac{\textbf{procedure } p(\overline{x}) = c \qquad \{P\}\ c[\overline{v}/\overline{x}]\ \{Q\}}{\{P\}\ p(\overline{v})\ \{Q\}} \qquad \frac{\{P\}\ c\ \{Q\}}{\{P\}\ \textbf{exec}(c)\ \{Q\}}$$

$$\frac{\Gamma \vdash \{P\}\ r := c\ \{Q(r)\} \qquad \forall X \bullet \Gamma \vdash \{Q(X)\}\ c'[X/x]\ \{Q'\}}{\Gamma \vdash \{P\}\ \textbf{let } x := c \textbf{ in } c'\ \{Q'\}} \qquad \frac{\textbf{threadspec } \tau(\overline{x}) \textbf{ req } P \textbf{ ens } Q \qquad \Gamma \vdash \{P[\overline{v}/\overline{x}]\}\ c\ \{Q[\overline{v}/\overline{x}]\}}{\Gamma \vdash \{P[\overline{v}/\overline{x}]\}\ r := \textbf{fork}\ c\ \{\mathsf{tid}_\tau(r, \overline{v})\}}$$

$$\frac{\Gamma, \{P\}\ f(\overline{x})\ \{Q\} \vdash \{P\}\ c\ \{Q\}}{\Gamma \vdash \{P[\overline{v}/\overline{x}]\}\ (\mu f(\overline{x}) \bullet c)(\overline{v})\ \{Q[\overline{v}/\overline{x}]\}} \qquad \frac{\textbf{threadspec } \tau(\overline{x}) \textbf{ req } P \textbf{ ens } Q}{\{\mathsf{tid}_\tau(t, \overline{v})\}\ \textbf{join}(t)\ \{Q[\overline{v}/\overline{x}]\}} \qquad \frac{\forall X \bullet \Gamma \vdash \{P(X)\}\ c\ \{Q(X)\}}{\Gamma \vdash \{\exists X \bullet P(X)\}\ c\ \{\exists X \bullet Q(X)\}}$$

$$\Gamma, \{P\}\ f(\overline{x})\ \{Q\}, \Gamma' \ \vdash \{P[\overline{v}/\overline{x}]\}\ f(\overline{v})\ \{Q[\overline{v}/\overline{x}]\} \qquad \{s \mapsto 1\}\ \textbf{init}_A(s)\ \{A(s) * \mathsf{locked}_A(s)\}$$

$$\{\pi A(s)\}\ \textbf{acquire}(s)\ \{\pi A(s) * \mathsf{locked}_A(s) * I_A(s)\} \qquad \{\mathsf{locked}_A(s) * I_A(s)\}\ \textbf{release}(s)\ \{\textbf{emp}\} \qquad \frac{\{P\}\ c\ \{Q\}}{\Gamma \vdash \{P\}\ c\ \{Q\}}$$

$$\{A(s) * \mathsf{locked}_A(s)\}\ \textbf{finalize}(s)\ \{s \mapsto _\} \qquad \frac{P \Rightarrow P' \qquad \Gamma \vdash \{P'\}\ c\ \{Q\} \qquad Q \Rightarrow Q'}{\Gamma \vdash \{P\}\ c\ \{Q'\}} \qquad \frac{\Gamma \vdash \{P\}\ c\ \{Q\}}{\Gamma \vdash \{P * R\}\ c\ \{Q * R\}}$$

Figure 10. Proof rules

The soundness theorem states that if for a command c we have $\{\textbf{emp}\}\ c\ \{\textbf{true}\}$, then $\langle \emptyset, \emptyset, \{(t, c; \textbf{done})\}\rangle \not\leadsto^* \textbf{abort}$ for any thread identifier t.

We do not define the semantics of the correctness judgment directly. Rather, we define an assertion transformer $\mathsf{valid}_\Gamma(c, Q)$ (similar to a weakest precondition operator, but we don't worry about whether it is the *weakest* precondition or not). We prove

$$(\Gamma \vdash \{P\}\ c\ \{Q\}) \Rightarrow (P \Rightarrow \mathsf{valid}_\Gamma(c, Q))$$

We can then define validity of a configuration. A configuration $\langle h, g, T\rangle$ is valid iff there exists a set of permissions P such that

- for every (ℓ, ℓ') with $0 < \ell$, if $(\ell, \ell') \in \mathrm{dom}(g)$ then $\ell \in \mathrm{dom}(h)$
- h equals the set of non-ghost points-to permissions in P plus one element $\ell \mapsto 0$ or $\ell \mapsto 1$ for each $A(\ell)$ permission in P
- g equals the set of ghost points-to permissions in P
- there is exactly one $\mathsf{tid}_\tau(t, \overline{v})$ permission for each thread $t \in \mathrm{dom}(T)$, and
- there is exactly one $\mathsf{locked}_A(\ell)$ permission for each $A(\ell)$ permission whose corresponding heap element equals 1, and
- there exists a permission bundle b_t for each thread t, and a permission bundle b_ℓ for each $A(\ell)$ permission for which there is no $\mathsf{locked}_A(\ell)$ permission, and a permission bundle b_{C} for the program's environment, which in particular contains the tid permission for the main thread, such that
  - the sum of all b_t and all b_ℓ and b_{C} equals $\{(\hat{p}, 1) \mid \hat{p} \in P\}$, and
  - for each thread $(t, \kappa) \in T$, $b_t \in \mathsf{valid}(\kappa, Q[\overline{v}/\overline{x}])$ where **threadspec** $\tau(\overline{x})$ **req** $\cdots$ **ens** Q and $\mathsf{tid}_\tau(t, \overline{v}) \in P$.
  - for each lock $A(\ell) \in P$ for which there is no $\mathsf{locked}_A(\ell) \in P$, $b_\ell \in I_A(\ell)$.

By correctness of the main program, the initial configuration is valid. We then prove that each execution step preserves configuration validity. The theorem then follows from the fact that **abort** is not a valid configuration.

5.5 Ghost erasure

After a program is verified, ghost code can be removed without invalidating the proof. Specifically, if all code that is removed is side-effect-free and terminates, then if the program after erasure aborts, the original program aborts. If the program has simple closures and the procedure call graph is acyclic, then non-termination can result only from non-terminating recursive functions. Removed code is side-effect-free if it affects only the ghost heap, provided that all ghost heap accesses are removed.

6. Ghost Objects

In the previous section we used fractional points-to assertions to enable a thread to maintain information about a shared object. The location is read-only while no thread has full permission, and the thread has full information: it knows the exact value.

Often, proofs require a more fine-grained type of tracking. A thread needs to maintain partial information about a value, while allowing other threads to modify the value in ways that preserve all threads' assumptions.

A general approach to this problem is rely-guarantee reasoning. However, in this paper we propose a different strategy. We propose the use of *ghost objects*. A ghost object is a data structure built from auxiliary heap cells, that represents some mathematical value, and that allows clients to obtain *handles* on the object that represent a condition on the value of the object. Handles represent partial information about the object. Correspondingly, they represent the permission to violate the condition, in the sense that the object does not allow violating the condition without handing in the handle.

A basic ghost object is a *ghost bag*. The abstract predicate $\mathsf{gbag}(b, B)$ represents a ghost bag with identifier b, currently holding the bag of integers B. The object provides the following operations:

$$\{\mathbf{emp}\}\ r := \mathit{create\_gbag}()\ \{\mathsf{gbag}(r, \emptyset)\}$$

$$\{\mathsf{gbag}(b, B)\}\ \mathit{gbag\_add}(b, v)\ \{\mathsf{gbag}(b, B \uplus \{v\}) * \mathsf{gbagh}(b, v)\}$$

$$\begin{array}{c}\{\mathsf{gbag}(b, B) * \mathsf{gbagh}(b, v)\} \\ \mathit{gbag\_remove}(b, v) \\ \{v \in B \wedge \mathsf{gbag}(b, B - \{v\})\}\end{array}$$

The predicate $\mathsf{gbagh}(b, v)$ represents the knowledge that the ghost bag b currently contains element v. It furthermore represents the permission to remove this element.

This ghost object can be implemented in terms of simple auxiliary heap cells. It does not need to be built into the proof system. A verified ghost bag implementation comes with our verification tool (see Section 9). Furthermore, based on ghost bags, a wide variety of ghost objects can be implemented easily.

7. Atomic Instructions

The programming language of the previous section does not include atomic machine instructions such as atomic compare-and-swap (CAS) instructions, which are available on most platforms. However, one can easily translate a program that uses atomics to a behaviorally equivalent (but less efficient) program of the formal language that uses locks by introducing an extra lock for each data structure of the program that is accessed using atomics, and then translating the atomic operations into code sequences that acquire the corresponding lock, perform the operation, and then release the lock. We will call such a lock an *atomic space* and its address an *atomic space identifier*, ranged over by s. (Note: our verification tool supports atomics and atomic spaces directly, and does not require a translation.)

A procedure corresponding to a CAS operation could look as follows, augmented with two ghost parameters ρ and ρ' for verification purposes:

```
procedure $cas(s, \ell, o, n, \rho, \rho') =$
  acquire$(s)$;
  $v := [\ell]$;
  if $v = o$ then $([\ell] := n; \rho)$ else $\rho'$;
  release$(s)$;
  return $v$
```

We can prove the following specification for it:

$$\frac{\begin{array}{c}I_A(s) * P \Rightarrow \exists X \bullet \ell \mapsto X * S(X) \\ \{S(o) * \ell \mapsto n\}\ \rho\ \{I_A(s) * Q(o)\} \\ \forall X \bullet X \neq o \Rightarrow \{S(X) * \ell \mapsto X\}\ \rho'\ \{I_A(s) * Q(X)\}\end{array}}{\{\pi A(s) * P\}\ r := \mathit{cas}(s, \ell, o, n, \rho, \rho')\ \{\pi A(s) * Q(r)\}}$$

We will use this procedure in the examples below, as well as procedures *load* and *store* corresponding to atomic loads and stores, respectively, specified as follows:

$$\frac{\begin{array}{c}I_A(s) * P \Rightarrow \exists X \bullet \ell \mapsto X * S(X) \\ \forall X \bullet \{S(X) * \ell \mapsto X\}\ \rho\ \{I_A(s) * Q(X)\}\end{array}}{\{\pi A(s) * P\}\ r := \mathit{load}(s, \ell, \rho)\ \{\pi A(s) * Q(r)\}}$$

$$\frac{\begin{array}{c}I_A(s) * P \Rightarrow \exists X \bullet \ell \mapsto X * S(X) \\ \forall X \bullet \{S(X) * \ell \mapsto v\}\ \rho\ \{I_A(s) * Q\}\end{array}}{\{\pi A(s) * P\}\ \mathit{store}(s, \ell, v, \rho)\ \{\pi A(s) * Q\}}$$

8. Abstraction: A Concurrent Set

In the example of the preceding sections, the data structure being manipulated was a simple cell, and its memory representation, $\ell + 1 \mapsto X$, was disclosed in the specification of the operation, *incr*. In this section, we show that our approach supports specifications that abstract over the representation of the concurrent data structure. Furthermore, we show that the approach allows abstract specification and verification of concurrent data structures built on top of other concurrent data structures. We do so by first showing a specification and implementation of a binary semaphore module implemented in terms of atomic machine instructions. We then show a specification, implementation, and proof of a concurrent set module implemented by hand-over-hand locking of a sorted linked list, that uses the semaphore module.

8.1 Semaphore Specification

The semaphore module defines an abstract predicate $\mathsf{sema}(\ell, v)$ which represents a semaphore at address ℓ whose value is v (either 0 or 1). The specification of the functions exported by the module can be given in terms of this predicate as follows:

$$\{\ell \mapsto 0\}\ \mathit{init\_sema}(\ell)\ \{\mathsf{sema}(\ell, 0)\}$$

$$\frac{\begin{array}{c} I_A(s) * P \Leftrightarrow \exists v \bullet \mathsf{sema}(\ell, v) * S(v) \\ \{S(0) * \mathsf{sema}(\ell, 1)\}\ \rho\ \{I_A(s) * Q\} \end{array}}{\{\pi A(s) * P\}\ \mathit{sema\_acquire}(s, \ell, \rho)\ \{\pi A(s) * Q\}}$$

$$\frac{\begin{array}{c} I_A(s) * P \Rightarrow \exists v \bullet \mathsf{sema}(\ell, v) * S(v) \\ \forall v \bullet \{S(v) * \mathsf{sema}(\ell, 0)\}\ \rho\ \{I_A(s) * Q\} \end{array}}{\{\pi A(s) * P\}\ \mathit{sema\_release}(s, \ell, \rho)\ \{\pi A(s) * Q\}}$$

$$\{\mathsf{sema}(\ell, \_)\}\ \mathit{finalize\_sema}(\ell)\ \{\ell \mapsto \_\}$$

Actually, to enable sharing of information about the state of a semaphore, the semaphore module defines a slightly different predicate $[\pi]\mathsf{sema}(\ell, v)$. Just $\mathsf{sema}(\ell, v)$ is shorthand for $[1]\mathsf{sema}(\ell, v)$. The module further exports the following lemmas:

$$[\pi_1 + \pi_2]\mathsf{sema}(\ell, v) \Rightarrow [\pi_1]\mathsf{sema}(\ell, v) * [\pi_2]\mathsf{sema}(\ell, v)$$
$$[\pi_1]\mathsf{sema}(\ell, v) * [\pi_2]\mathsf{sema}(\ell, v') \Rightarrow [\pi_1 + \pi_2]\mathsf{sema}(\ell, v) \land v' = v$$

where π_1 and π_2 range over positive real numbers.

8.2 Semaphore Implementation

The implementation of the semaphore module is straightforward:

```
predicate $[\pi]\mathsf{sema}(\ell, v) = \ell \overset{\pi}{\mapsto} v$
procedure $\mathit{init\_sema}(\ell)$ = skip
procedure $\mathit{sema\_acquire}(s, \ell, \rho)$ =
  letrec $\mathit{iter}()$ =
    $r := \mathit{cas}(s, \ell, 0, 1, \rho, \mathbf{skip})$;
    if $r \neq 0$ then $\mathit{iter}()$
  in $\mathit{iter}()$
procedure $\mathit{sema\_release}(s, \ell, \rho)$ =
  $\mathit{store}(s, \ell, 0, \rho)$
procedure $\mathit{finalize\_sema}(\ell)$ = skip
```

We omit the proof; it, too, is straightforward.

8.3 Set Specification

The set module exports three procedures:

$$\{\mathbf{emp}\}\ r := \mathit{create\_set}()\ \{\mathsf{set}(r, \emptyset)\}$$

$$\frac{\begin{array}{c} I_A(s) * S \Leftrightarrow \exists V \bullet \mathsf{set}(o, V) * U(V) \\ \forall V \bullet v \notin V \Rightarrow \{U(V) * P\}\ \rho\ \{U(V \cup \{v\}) * Q(1)\} \\ \forall V \bullet v \in V \Rightarrow \{U(V) * P\}\ \rho'\ \{U(V) * Q(0)\} \end{array}}{\{\pi A(s) * S * P\}\ r := \mathit{add}(s, o, v, \rho, \rho')\ \{\pi A(s) * S * Q(r)\}}$$

$$\frac{\begin{array}{c} I_A(s) * S \Leftrightarrow \exists V \bullet \mathsf{set}(o, V) * U(V) \\ \forall V \bullet v \in V \Rightarrow \{U(V) * P\}\ \rho\ \{U(V \setminus \{v\}) * Q(1)\} \\ \forall V \bullet v \notin V \Rightarrow \{U(V) * P\}\ \rho'\ \{U(V) * Q(0)\} \end{array}}{\{\pi A(s) * S * P\}\ r := \mathit{remove}(s, o, v, \rho, \rho')\ \{\pi A(s) * S * Q(r)\}}$$

Procedure add returns 1 if the element was not yet present, and 0 otherwise. Analogously, procedure remove returns 1 if the element was present, and 0 otherwise.

Notice that in the specification of add and remove, the first premise, which enables the procedure to separate the set out of the lock invariant, uses S instead of P. In earlier specifications, there was no separate S predicate and P was used for simplicity; however, using P means the procedure cannot perform further atomic operations on the set after the closure ρ or ρ' has been executed, since it consumes P and produces Q. Using a separate predicate S means the procedure can access the set both before and after executing the closure.

To simplify the presentation, the example module does not offer a procedure for disposing a set object. An implementation that supports disposal, verified using our verification tool, is available online.

8.4 Sugared Specifications

In the above specifications of procedures add and remove, the functional behavior is obscured somewhat by the fine-grained concurrency scaffolding. Fortunately, however, we can easily define an abbreviated notation for typical fine-grained specifications, which makes them look just like sequential specifications. We introduce the notation

$$\{\mathsf{object}(o, v)\}\ r := \mathit{foo}_{\mathrm{FG}}(o)\ \{\mathsf{object}(o, \mathrm{post}(v)) \land r = \mathrm{res}(v)\}$$

as an abbrevation for

$$\frac{\begin{array}{c} I_A(s) * S \Leftrightarrow \exists v \bullet \mathsf{object}(o, v) * U(v) \\ \forall v \bullet \{U(v) * P\}\ \rho\ \{U(\mathrm{post}(v)) * Q(\mathrm{res}(v))\} \end{array}}{\{\pi A(s) * S * P\}\ r := \mathit{foo}(s, o, \rho)\ \{\pi A(s) * S * Q(r)\}}$$

Often, it is convenient to split the postcondition into multiple cases, with corresponding ghost command parameters and corresponding premises. We introduce

$$\begin{array}{l} \{\mathsf{object}(o, v)\} \\ r := \mathit{foo}_{\mathrm{FG}}(o) \\ \left\{ \begin{array}{l} G_1(v) \land \mathsf{object}(o, \mathrm{post}_1(v)) \land r = \mathrm{res}_1(v) \lor \\ G_2(v) \land \mathsf{object}(o, \mathrm{post}_2(v)) \land r = \mathrm{res}_2(v) \end{array} \right\} \end{array}$$

where G_i are pure assertions, as an abbreviation for

$$\frac{\begin{array}{c} I_A(s) * S \Leftrightarrow \exists v \bullet \mathsf{object}(o, v) * U(v) \\ \forall v \bullet G_1(v) \Rightarrow \{U(v) * P\}\ \rho_1\ \{U(\mathrm{post}_1(v)) * Q(\mathrm{res}_1(v))\} \\ \forall v \bullet G_2(v) \Rightarrow \{U(v) * P\}\ \rho_2\ \{U(\mathrm{post}_2(v)) * Q(\mathrm{res}_2(v))\} \end{array}}{\{\pi A(s) * S * P\}\ r := \mathit{foo}(s, o, \rho_1, \rho_2)\ \{\pi A(s) * S * Q(r)\}}$$

This notation allows the above specification for procedure add to be written as

$$\begin{array}{l} \{\mathsf{set}(o, V)\} \\ r := \mathit{add}_{\mathrm{FG}}(o, v) \\ \{v \notin V \land \mathsf{set}(o, V \cup \{v\}) \land r = 1 \lor v \in V \land \mathsf{set}(o, V) \land r = 0\} \end{array}$$

8.5 Set Implementation

The implementation of the set module is shown in Figure 11.

For node values, we implicitly perform an encoding of $\mathbb{Z} \cup \{-\infty, +\infty\}$ into $\mathbb{Z}$.

We use the following syntactic sugar: $n.\mathsf{next} = n + 1$, and $n.\mathsf{value} = n + 2$.

The set is implemented as a sorted linked list. For synchronization, one field of each node is converted into a semaphore that is used to perform hand-over-hand locking of consecutive nodes. This affords some degree of parallelism for concurrent operations on the set.

8.6 Set proof

The core of the proof of the set module is the definition of the set predicate; it serves as the invariant of the data structure, which holds before and after each atomic operation. This invariant must

```
procedure $create\_set() \ =$
    $lastNode := \mathbf{cons}(0, 0, +\infty);$
    $firstNode := \mathbf{cons}(0, lastNode, -\infty);$
    $init\_sema(firstNode);$ return $firstNode$

letrec $locate(n) \ =$
    $n' := [n.\mathsf{next}]; v' := [n'.\mathsf{value}];$
    if $v' < v$ then (
        $sema\_acquire(s, n'); sema\_release(s, n);$
        return $locate(n')$
    ) else return $n$
in

procedure $add(s, o, v) \ =$
    $sema\_acquire(s, o); n := locate(o);$
    $n' := [n.\mathsf{next}]; v' := [n'.\mathsf{value}];$
    if $v' = v$ then (
        $sema\_release(s, n);$ return $0$
    ) else (
        $n'' := \mathbf{cons}(0, n', v); init\_sema(n'');$
        $[n.\mathsf{next}] := n''; sema\_release(s, n);$ return $1$
    )

procedure $remove(s, o, v) \ =$
    $sema\_acquire(s, o); n := locate(o);$
    $n' := [n.\mathsf{next}]; v' := [n'.\mathsf{value}];$
    if $v' = v$ then (
        $sema\_acquire(s, n');$
        $n'' := [n'.\mathsf{next}]; [n.\mathsf{next}] := n'';$
        $sema\_release(s, n);$ return $1$
    ) else ($sema\_release(s, n);$ return $0$)
```

Figure 11. Implementation of the set module. Note: desugaring inlines *locate* into *add* and *remove*

enable each thread to retain, between the atomic operations that constitute a set operation, the information it needs about the state of the data structure.

For example, after locating a node, a thread must know this node will remain in the data structure. For this purpose, we track the set of nodes in the linked list using a ghost bag (see Section 6). We keep the identifier of the ghost bag in a ghost field of the first node. To refer to this ghost field, we use the syntactic sugar $o.\mathsf{bag} = o.1$.

Another consideration when defining the invariant is that we wish to retain the shape of the linked list even when a thread has taken ownership of a node's next field in preparation for inserting or removing the next node. Therefore, we use the ghost field at ghost offset 0 of each node as the oldNext field: $n.\mathsf{oldNext} = n.0$.

As usual, we use a recursive predicate lseg to describe the linked list:

$$\begin{array}{l}\mathsf{lseg}(b, f, v_f, \ell, v_\ell, \alpha, \beta) \ = \\ \quad (\alpha = \epsilon \wedge \beta = \epsilon \wedge f = \ell \wedge v_f = v_\ell) \vee \\ \quad (\exists \alpha', \beta', v_s, n, v_n \bullet \alpha = f \cdot \alpha' \wedge \beta = v_f \cdot \beta' \wedge \\ \qquad \mathsf{node}(b, f, v_s, v_f, n, v_n) * \mathsf{lseg}(b, n, v_n, \ell, v_\ell, \alpha', \beta'))\end{array}$$

where

$$\begin{array}{l}\mathsf{node}(b, n, v_s, v, n', v') \ = \\ \quad (v_s = 0 \wedge \mathsf{sema}(n, v_s) * n.\mathsf{next} \mapsto n' * n.\mathsf{oldNext} \mapsto n' * \\ \quad\ n.\mathsf{value} \overset{1/2}{\mapsto} v * n'.\mathsf{value} \overset{1/2}{\mapsto} v' * \mathsf{gbagh}(b, n) \wedge v < v') \vee \\ \quad (v_s = 1 \wedge [\frac{1}{2}]\mathsf{sema}(n, v_s) * n.\mathsf{oldNext} \overset{1/2}{\mapsto} n' * \\ \quad\ n.\mathsf{value} \overset{1/4}{\mapsto} v * n'.\mathsf{value} \overset{1/4}{\mapsto} v' \wedge v < v')\end{array}$$

The predicate $\mathsf{lseg}(b, f, v_f, \ell, v_\ell, \alpha, \beta)$ denotes the section of the sorted linked list from node f to node ℓ, excluding node ℓ. The other parameters are the ghost bag identifier b, the first value v_f, the last value v_ℓ, the list of nodes α, and the list of values β. The body of the predicate is a disjunction. The first disjunct describes the case where the first node equals the last node and therefore the section is empty.

The second disjunct describes the non-empty case. Specifically, it describes the first node using the predicate node and recursively calls the predicate to describe the subsection from the second node to the last node. This disjunct quantifies existentially over the tail α' of α, the tail β' of β, the value of the semaphore v_s of the first node, the next node n, and the value of the next node v_n.

Predicate node's body, too, is a disjunction; the first disjunct describes the case where the node is not locked; the second disjunct describes the case where the node is locked. In the latter case, full ownership of the n.next field and fractional ownership of the semaphore and the n.oldNext, n.value, and n'.value fields has been transferred to the thread that acquired the lock.

Notice that each node owns half of the value field of the next node. This means that when a thread locks a node, it knows not only that node's value but also the next node's value. This allows it to safely insert a new node in between, while maintaining the sortedness of the list.

The definition of the set predicate itself is now straightforward:

$$\begin{array}{l}\mathsf{set}(o, V) \ = \\ \quad \exists b, \ell, \alpha, \beta \bullet \mathsf{lseg}(b, o, -\infty, \ell, +\infty, \alpha, -\infty \cdot \beta) * \\ \qquad o.\mathsf{bag} \overset{-}{\mapsto} b * \mathsf{gbag}(b, \mathsf{elems}(\alpha)) * \mathbf{true} \wedge V = \mathsf{elems}(\beta);\end{array}$$

The definition uses the mathematical function $\mathsf{elems}(\alpha)$ which denotes the bag of the elements of the list α. It states that there is a sorted linked list starting at o, that starts with value $-\infty$ and ends with value $+\infty$ (which means that an insertion point can be found within the list for any finite value). It further states that the nodes of the list are exactly the elements in the ghost bag at $o.\mathbf{bag}$, and that abstract value V of the set is exactly the bag of the values of the list.

The syntax $\ell \overset{-}{\mapsto} v$ is shorthand for $\exists \pi \bullet \ell \overset{\pi}{\mapsto} v$. That is, it denotes an unspecified fraction of the points-to permission. As applied in the set predicate, this allows threads to remember the connection between o and b.

The $\mathbf{true}$ conjunct allows us to leak memory locations (or fractions thereof) that we do not use; specifically, of the last node ℓ we use only one-half of field ℓ.value. We would need to be more precise if we wanted to support disposal of the set object.

The specification of local recursive function *locate* is as follows:

$$\left\{\begin{array}{l}\pi A(s) * S * o.\mathsf{bag} \overset{-}{\mapsto} b * \mathsf{gbagh}(b, n) * \\ [\frac{1}{2}]\mathsf{sema}(n, 1) * n.\mathsf{oldNext} \overset{1/2}{\mapsto} n' * n.\mathsf{next} \mapsto n' * \\ n.\mathsf{value} \overset{1/4}{\mapsto} v_n * n'.\mathsf{value} \overset{1/4}{\mapsto} v_{n'} \wedge v_n < v\end{array}\right\}$$
$$r := locate(n)$$
$$\left\{\begin{array}{l}\pi A(s) * S * o.\mathsf{bag} \overset{-}{\mapsto} b * \mathsf{gbagh}(b, r) * [\frac{1}{2}]\mathsf{sema}(r, 1) * \\ \exists n, v_r, v_n \bullet r.\mathsf{oldNext} \overset{1/2}{\mapsto} n * r.\mathsf{next} \mapsto n * \\ \quad r.\mathsf{value} \overset{1/4}{\mapsto} v_r * n.\mathsf{value} \overset{1/4}{\mapsto} v_n \wedge v_r < v \wedge v \leq v_n\end{array}\right\}$$

Note that even though *locate* is shown outside of *add* and *remove*, after desugaring it is within the scope of the parameters of these procedures, and furthermore its proof can use the premises of these procedures' specifications, and in particular the first premise.

We show in Figure 12 the set implementation annotated with ghost commands. A full proof outline is in the technical report [7].

```
procedure $create\_set() =$
    $lastNode := \mathbf{cons}(0, 0, +\infty)$;
    $firstNode := \mathbf{cons}(0, lastNode, -\infty)$;
    $[firstNode.\mathsf{oldNext}] := lastNode$;
    $init\_sema(firstNode)$;
    $b := create\_gbag(); gbag\_add(b, f); [firstNode.\mathsf{bag}] := b$;
    return $firstNode$

letrec $locate(n) =$
    $n' := [n.\mathsf{next}]; v' := [n'.\mathsf{value}]$;
    if $v' < v$ then (
        $sema\_acquire(s, n', \mathbf{skip})$;
        $sema\_release(s, n, \mathbf{skip})$;
        return $locate(n')$
    ) else
        return $n$
in

procedure $add(s, o, v, \rho, \rho') =$
    $sema\_acquire(s, o, \mathbf{skip})$;
    $b := [o.\mathsf{bag}]; n := locate(o)$;
    $n' := [n.\mathsf{next}]; v' := [n'.\mathsf{value}]$;
    if $v' = v$ then (
        $sema\_release(s, n, \rho')$;
        return $0$
    ) else (
        $n'' := \mathbf{cons}(0, n', v); [n''.\mathsf{oldNext}] := n'$;
        $init\_sema(n''); [n.\mathsf{next}] := n''$;
        $sema\_release(s, n,$
            $(gbag\_add(b, n''); [n.\mathsf{oldNext}] := n''; \rho))$;
        return $1$
    )

procedure $remove(s, o, v, \rho, \rho') =$
    $sema\_acquire(s, o, \mathbf{skip})$;
    $b := [o.\mathsf{bag}]; n := locate(o)$;
    $n' := [n.\mathsf{next}]; v' := [n'.\mathsf{value}]$;
    if $v' = v$ then (
        $sema\_acquire(s, n', \mathbf{skip})$;
        $n'' := [n'.\mathsf{next}]; [n.\mathsf{next}] := n''$;
        $sema\_release(s, n,$
            $(gbag\_remove(b, n'); [n.\mathsf{oldNext}] := n''; \rho))$;
        return $1$
    ) else (
        $sema\_release(s, n, \rho')$;
        return $0$
    )
```

Figure 12. The set module, with ghost commands (highlighted)

8.7 Client program

In this subsection, to illustrate how the specification of the set module can be used to verify rich properties of client programs, we verify the example client program shown in Figure 13. The program starts by creating a set object o and a lock s for use by the set module to emulate its atomic operations. (Remember that this lock can be erased after verification if real atomic operations are used; see Section 7.) Then, the lock is initialized. From this time, the lock protects the set data structure. Finally, a producer thread is forked and the main thread turns into a consumer thread. The producer thread simply adds 1,2,3,... to the set. The consumer thread repeatedly performs the following experiment: it picks an arbitrary number (by allocating a heap cell and disposing it, just

```
letrec
    $producerThread(s, o, x) =$
        $add(s, o, x); producerThread(s, o, x + 1)$
    $consumer(s, o, x) =$
        $r := remove(s, o, x)$;
        if $r = 1$ then $(r := remove(s, o, x); \mathbf{assert}(r = 0))$
    $consumerThread(s, o) =$
        // pick random number $x$
        $x := \mathbf{cons}(0); \mathbf{dispose}(x)$;
        $consumer(s, o, x); consumerThread(s, o)$
in
$o := create\_set(); s := \mathbf{cons}(1); \mathbf{init}_{\mathsf{space}}(s); \mathbf{release}(s)$;
fork $producerThread(s, o, 1)$;
$consumerThread(s, o)$
```

Figure 13. Example client program for the concurrent set module

for the address) and tries to remove it. If the remove operation succeeds, it tries to remove it again and asserts that the latter remove operation fails. It always does, since the producer thread never adds the same number twice.

Here is how our approach succeeds in verifying the **assert** command of this program. A proof outline for this program, including ghost commands, is shown in Figure 14. As always, the crucial step is coming up with an invariant; specifically, a lock invariant for the lock s. It is shown at the bottom of Figure 14. The proof uses three ghost fields of s: $s.\mathsf{set}$ (sugar for $s.0$) connects the lock to the set o; $s.\mathsf{prod}$ (sugar for $s.1$) records the last number added by the producer; and $s.\mathsf{cons}$ (sugar for $s.2$) records the last number removed by the consumer. The invariant states that the last value added by the producer is an upper bound for the set's elements; that the last value removed by the consumer is not greater than the last value added by the producer; and that the last value removed by the consumer is not in the set.

Once the invariant is established, the proof outline follows easily. As usual, each thread retains half of its associated ghost field: the producer thread retains half of $s.\mathsf{prod}$ and the consumer thread retains half of $s.\mathsf{cons}$. The producer passes the required update of $s.\mathsf{prod}$ into add as a ghost argument; analogously, the consumer passes the required update of $s.\mathsf{cons}$ into $remove$ as a ghost argument.

The specifications of add and $remove$ are instantiated as follows. For all calls, predicate S is instantiated with $s.\mathsf{set} \overset{-}{\mapsto} o$ and predicate $U(V)$ is instantiated with the invariant minus the set data structure itself:

$$\begin{aligned} U(V) = \exists p, c \bullet s.\mathsf{set} \overset{-}{\mapsto} o * s.\mathsf{prod} \overset{1/2}{\mapsto} p * s.\mathsf{cons} \overset{1/2}{\mapsto} c \\ \wedge (\forall v \in V \bullet v \leq p) \wedge c \leq p \wedge c \notin V \end{aligned}$$

where variables s and o are bound at the call site. The instantiations of P and Q are shown at the call sites in Figure 14. Given these instantiations, the premises of add and $remove$'s specifications can be verified easily.

9. Verification Tool

We implemented our approach in our program verification tool, VeriFast [8], and we used the tool to verify two challenging fine-grained concurrent data structures from the literature: a multiple-compare-and-swap algorithm [5] and a lock-coupling list [15].

VeriFast is a general-purpose verifier prototype for C programs, based on separation logic. It takes source code annotated with function specifications, loop invariants, predicate definitions, and other annotations, and reports either that the program is memory-safe, data-race-free, and complies with function specifications, or

```
$\textbf{threadspec}\ \mathit{producerThread}(s, o, x) =$
  $\textbf{req}\ \frac{1}{2}\mathsf{space}(s) * s.\mathsf{set} \overset{-}{\mapsto} o * \exists p \bullet s.\mathsf{prod} \overset{1/2}{\mapsto} p \land p < x$
  $\textbf{ens false}$
$\textbf{threadspec}\ \mathit{consumerThread}(s, o) =$
  $\textbf{req}\ \frac{1}{2}\mathsf{space}(s) * s.\mathsf{set} \overset{-}{\mapsto} o * \exists c \bullet s.\mathsf{cons} \overset{1/2}{\mapsto} c$
  $\textbf{ens false}$
$\textbf{letrec}$
  $\mathit{producerThread}(s, o, x) =$
    $\{\frac{1}{2}\mathsf{space}(s) * s.\mathsf{set} \overset{-}{\mapsto} o * \exists p \bullet s.\mathsf{prod} \overset{1/2}{\mapsto} p \land p < x\}$
    $\mathit{add}(s, o, x, [s.\mathsf{prod}] := x, \textbf{skip});$
      $P = \exists p \bullet s.\mathsf{prod} \overset{1/2}{\mapsto} p \land p < x$
      $Q(r) = s.\mathsf{prod} \overset{1/2}{\mapsto} x$
    $\{\frac{1}{2}\mathsf{space}(s) * s.\mathsf{set} \overset{-}{\mapsto} o * s.\mathsf{prod} \overset{1/2}{\mapsto} x\}$
    $\mathit{producerThread}(s, o, x + 1)$
  $\mathit{consumer}(s, o, x) =$
    $\{\frac{1}{2}\mathsf{space}(s) * s.\mathsf{set} \overset{-}{\mapsto} o * \exists c \bullet s.\mathsf{cons} \overset{1/2}{\mapsto} c\}$
    $r := \mathit{remove}(s, o, x, [s.\mathsf{cons}] := x, \textbf{skip});$
      $P = \exists c \bullet s.\mathsf{cons} \overset{1/2}{\mapsto} c$
      $Q(r) = \exists c \bullet s.\mathsf{cons} \overset{1/2}{\mapsto} c \land (r = 1 \Rightarrow c = x)$
    $\{\frac{1}{2}\mathsf{space}(s) * s.\mathsf{set} \overset{-}{\mapsto} o * \exists c \bullet s.\mathsf{cons} \overset{1/2}{\mapsto} c \land (r = 1 \Rightarrow c = x)\}$
    $\textbf{if}\ r = 1\ \textbf{then}\ ($
      $r := \mathit{remove}(s, o, x, \textbf{skip}, \textbf{skip});$
        $P = s.\mathsf{cons} \overset{1/2}{\mapsto} x$
        $Q(r) = s.\mathsf{cons} \overset{1/2}{\mapsto} x \land r = 0$
      $\{\frac{1}{2}\mathsf{space}(s) * s.\mathsf{set} \overset{-}{\mapsto} o * s.\mathsf{cons} \overset{1/2}{\mapsto} x \land r = 0\}$
      $\textbf{assert}(r = 0))$
  $\mathit{consumerThread}(s, o) =$
    $\{\frac{1}{2}\mathsf{space}(s) * s.\mathsf{set} \overset{-}{\mapsto} o * \exists c \bullet s.\mathsf{cons} \overset{1/2}{\mapsto} c\}$
    // pick random number $x$
    $x := \textbf{cons}(0); \textbf{dispose}(x);$
    $\mathit{consumer}(s, o, x);$
    $\mathit{consumerThread}(s, o)$
$\textbf{in}$
$\{\textbf{emp}\}$
$o := \mathit{create\_set}();$
$\{\mathsf{set}(o, \emptyset)\}$
$s := \textbf{cons}(1);$
$\{\mathsf{set}(o, \emptyset) * s \mapsto 1 * \circledast_{\ell' \in \mathbb{N}} s.\ell' \mapsto 0\}$
$[s.\mathsf{set}] := o;$
$[s.\mathsf{prod}] := 0;$
$[s.\mathsf{cons}] := 0;$
$\{\mathsf{set}(o, \emptyset) * s \mapsto 1 * s.\mathsf{set} \mapsto o * s.\mathsf{prod} \mapsto 0 * s.\mathsf{cons} \mapsto 0 * \textbf{true}\}$
$\textbf{init}_{\mathsf{space}}(s); \textbf{release}(s);$
$\{\mathsf{space}(s) * s.\mathsf{set} \overset{-}{\mapsto} o * s.\mathsf{prod} \overset{1/2}{\mapsto} 0 * s.\mathsf{cons} \overset{1/2}{\mapsto} 0 * \textbf{true}\}$
$\textbf{fork}\ \mathit{producerThread}(s, o, 1);$
$\{\frac{1}{2}\mathsf{space}(s) * s.\mathsf{set} \overset{-}{\mapsto} o * s.\mathsf{cons} \overset{1/2}{\mapsto} 0 * \textbf{true}\}$
$\mathit{consumerThread}(s, o)$

$I_{\mathsf{space}}(s) \ =$
  $\exists o, p, c, V \bullet$
    $s.\mathsf{set} \overset{-}{\mapsto} o * s.\mathsf{prod} \overset{1/2}{\mapsto} p * s.\mathsf{cons} \overset{1/2}{\mapsto} c *$
    $\mathsf{set}(o, V) \land (\forall v \in V \bullet v \leq p) \land c \leq p \land c \notin V$
```

Figure 14. Proof outline for the client program

it shows a symbolic execution trace that leads to a potential error. It symbolically executes each function in turn, using a separation logic formula as the symbolic representation of memory.

VeriFast supports ghost commands for creating and updating ghost cells. It also supports *lemma functions*, which are like ordinary C functions except they may contain only ghost commands and VeriFast checks that they terminate. It follows that calls of lemma functions are ghost commands. Thirdly, it supports *lemma function pointers* and *lemma function pointer calls*. These features are all that was needed to make it possible to apply our approach in VeriFast. More generally, any verification tool that supports ghost variables, ghost functions, and dynamic binding of ghost functions supports our verification approach. This means it should be easy to extend other verification tools, such as VCC [2] and Chalice [10], to support our approach.

We have used VeriFast to verify the concurrent set module used as the example for this paper. We also verified a multiple-compare-and-swap (MCAS) algorithm proposed by Harris et al. [5]. MCAS is built on top of a restricted-double-compare-single-swap (RDCSS) algorithm by the same authors. Our MCAS proof consists of a proof of RDCSS with respect to an abstract specification of RDCSS, and a proof of MCAS based on the abstract specification of RDCSS. We also verified a simple example client program for each algorithm. The annotation overhead, consisting of specifications as well as proof steps, is shown in the following table:

Program	LOC	LOAnn	Overhead	Time
lcset.c	72	610	847%	0.37s
lcset_client.c	27	266	985%	0.13s
rdcss.c	51	528	1035%	0.5s
mcas.c	63	1111	1763%	1.33s
mcas_client.c	34	230	676%	0.22s

In each case, the annotation overhead is in the order of 10 to 20 lines of annotation per line of code. Three things should be kept in mind when considering the overhead. Firstly, these are probably some of the most complex algorithms in existence. Secondly, we did not optimize the annotation requirements for lemma function pointers; it currently involves significant boilerplate. Thirdly, we show these results only as evidence that the specification approach is applicable to challenging algorithms; this paper is not about VeriFast.

Notice that the run-time of the verification tool is very acceptable: on the order of one second. This enables an interactive annotation insertion process.

The tool and the annotated example programs are available online at http://www.cs.kuleuven.be/~bartj/verifast/.

10. Related Work

To the best of our knowledge, our approach is the first that enables fully general modular specification and verification of fine-grained concurrent modules and their clients.

We are aware of two existing approaches for specification of fine-grained concurrent data structures, both based on a marriage of rely-guarantee and separation logic [17]: a linearizability-based approach, initially proposed in Vafeiadis' PhD thesis [15], and concurrent abstract predicates [3].

In the linearizability-based approach, the specification for a data structure operation is in the form of a piece of sequential code that operates on a ghost variable that holds the abstract state of the data structure. An implementation complies with the specification if for each execution trace, there is a total ordering of the operation invocations in the trace such that their return values equal the return values that would result if the operations' specifications were executed sequentially in this total order. In other words, there exists a linearization point between the start and end of each operation invocation such that the result values are as if each operation's specification was executed atomically at the linearization point.

Linearizability-based verification verifies that the data structure is linearizable, by verifying that there exists a linearization point for each operation. The approach can then verify client code as if the operations executed atomically.

A limitation of the linearizability-based approach is that it does not support the transfer of ownership of memory locations or other resources between the data structure and its client. For example, a queue implemented as a linked list where nodes are allocated by the client, passed into the module on enqueue, and passed back to the client on dequeue, cannot be specified by the linearizability-based approach. This is because the definition of linearizability assumes no memory is shared across the module boundary, and all interaction is in the form of invocation arguments and results. In contrast, in our approach ownership transfer is supported. For example, here is a specification for the enqueue and dequeue operations of the queue module suggested above:

$$\frac{\begin{array}{c} I_A(s) * S \Leftrightarrow \exists \alpha \bullet \mathsf{queue}(q, \alpha) * U(\alpha) \\ \forall \alpha \bullet \{U(\alpha) * P\}\ \rho\ \{U(\alpha \cdot n) * Q * \mathsf{node}(n)\} \end{array}}{\{\pi A(s) * S * P\}\ \mathit{enqueue}(s, q, n, \rho)\ \{\pi A(s) * S * Q\}}$$

$$\frac{\begin{array}{c} I_A(s) * S \Leftrightarrow \exists \alpha \bullet \mathsf{queue}(q, \alpha) * U(\alpha) \\ \forall n, \alpha \bullet \{U(n \cdot \alpha) * P * \mathsf{node}(n)\}\ \rho\ \{U(\alpha) * Q(n)\} \\ \{U(\epsilon) * P\}\ \rho'\ \{U(\epsilon) * Q(0)\} \end{array}}{\{\pi A(s) * S * P\}\ r := \mathit{dequeue}(s, q, \rho, \rho')\ \{\pi A(s) * S * Q(r)\}}$$

An important advantage of linearizability, however, is that powerful automation techniques have been built for it, e.g. [16].

Concurrent abstract predicates (CAP) extend separation logic with *shared regions*. Each shared region is associated with an *interference specification*, which is a set of action names with associated pre- and postconditions. A piece of local state can be converted into a shared region. This gives the thread full permission to perform the actions associated with the region. It may then pass fractions of these *action permissions* to other threads. Each assertion about a shared region must be stable with respect to the actions that other threads may perform.

The CAP authors [3] propose the following approach for modular specification of a fine-grained data structure. The module exposes the data structure to clients in the form of a number of concurrent abstract predicates, each of which give permission to perform a particular type of operation. For example, their example lock module exposes predicates $\mathsf{isLock}(x)$ and $\mathsf{Locked}(x)$, which give permission to acquire, resp. release lock x. Their example set module exposes predicates $\mathsf{in}(h, v)$ and $\mathsf{out}(h, v)$, which give permission to remove, resp. add element v.

This specification approach does not subsume ours. Whereas our approach enables fully general specification of data structure operations, this approach enforces restrictions on how the data structure may be used. Specifically, while the set module specification allows threads to concurrently add or remove distinct elements, it does not allow them to race to concurrently add or remove the same element. More generally, the choice of which predicates to expose is a trade-off between the restrictions on usage and the type of information a client can track. Indeed: the information content of a predicate imposes a restriction on what other threads can do. For example, if one thread holds an $\mathsf{in}(h, v)$ permission, other threads cannot remove this element.

To achieve a fully general specification, the choice of stable permissions must be done by the client, not by the module designer. In order to enable this, the client must be able to do atomic or unstable observations, not just non-atomic or stable ones. This is what linearizability enables by allowing operations to be treated like atomic instructions, and what our approach enables by allowing the insertion of ghost code into the critical section, and by allowing the client to choose the auxiliary variables and the lock invariant.

Note that our comparison is with the way the CAP logic is used in [3], not with other specification approaches based on the CAP logic that may be proposed in the future.

However, the CAP logic is a convenient alternative to the use of ghost objects to track partial information, and as such is complementary to our specification approach. Specifically, one could have a single auxiliary variable that holds the precise abstract state of the data structure, and then insert this variable into a shared region. For example, the ghost bags of Section 6 could be implemented more straightforwardly using shared regions than using a data structure built from auxiliary heap cells.

Acknowledgments

The authors would like to thank Cristiano Calcagno, Mike Dodds, Peter O'Hearn, Matthew Parkinson, Viktor Vafeiadis, and Hongseok Yang for helpful comments on drafts of this paper. This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, and by the Research Fund K.U.Leuven.

References

[1] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *POPL*, 2005.

[2] Markus Dahlweid, Michał Moskal, Thomas Santen, Stephan Tobies, and Wolfram Schulte. VCC: Contract-based modular verification of concurrent C. In *ICSE*, 2009.

[3] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In *ECOOP*, 2010.

[4] Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. Local reasoning for storable locks and threads. In *APLAS*, 2007.

[5] Tim Harris, Keir Fraser, and Ian A. Pratt. A practical multi-word compare-and-swap operation. In *16th International Symposium on Distributed Computing*, 2002.

[6] Maurice Herlihy and Jeanette Wing. Linearizability: A correctness condition for concurrent objects. *ACM TOPLAS*, 12(3), 1990.

[7] Bart Jacobs and Frank Piessens. Expressive modular fine-grained concurrency specification (extended version). Technical Report CW590, Dept. CS, K.U.Leuven, 2010.

[8] Bart Jacobs, Jan Smans, and Frank Piessens. A quick tour of the VeriFast program verifier. In *APLAS*, 2010.

[9] C. B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, 1983.

[10] K. Rustan M. Leino, Peter Müller, and Jan Smans. *Foundations of Security Analysis and Design V, FOSAD 2007/2008/2009 Tutorial Lectures*, volume 5705 of *LNCS*, chapter Verification of concurrent programs with Chalice. Springer, 2009.

[11] Peter W. O'Hearn, John Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *CSL*, 2001.

[12] Susan Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *CACM*, 19(5):279–285, May 1976.

[13] Susan Owicki and David Gries. An axiomatic proof technique for parallel programs i. *Acta Inf.*, 6, 1976.

[14] J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In *LICS*, 2002.

[15] Viktor Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, Computer Laboratory, University of Cambridge, July 2007.

[16] Viktor Vafeiadis. Automatically proving linearizability. In *CAV*, 2010.

[17] Viktor Vafeiadis and Matthew Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*, 2007.