Structuring the Verification of Heap-Manipulating Programs

Aleksandar Nanevski
IMDEA Software, Madrid
aleks.nanevski@imdea.org

Viktor Vafeiadis
Microsoft Research, Cambridge
viktorva@microsoft.com

Josh Berdine
Microsoft Research, Cambridge
jjb@microsoft.com

Abstract

Most systems based on separation logic consider only restricted forms of implication or non-separating conjunction, as full support for these connectives requires a non-trivial notion of variable context, inherited from the logic of bunched implications (BI). We show that in an expressive type theory such as Coq, one can avoid the intricacies of BI, and support full separation logic very efficiently, using the native structuring primitives of the type theory.

Our proposal uses reflection to enable equational reasoning about heaps, and Hoare triples with binary postconditions to further facilitate it. We apply these ideas to Hoare Type Theory, to obtain a new proof technique for verification of higher-order imperative programs that is general, extendable, and supports very short proofs, even without significant use of automation by tactics. We demonstrate the usability of the technique by verifying the fast congruence closure algorithm of Nieuwenhuis and Oliveras, employed in the state-of-the-art Barcelogic SAT solver.

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

General Terms Languages, Verification

Keywords Type Theory, Hoare Logic, Separation Logic, Monads

1. Introduction

While separation logic [25, 26] has proved to be extremely effective in reasoning about heap-manipulating programs in the presence of aliasing, most practical systems such as Smallfoot [6], HIP [22], SLAyer [5], Space Invader [8] or Xisa [10] address only a restricted fragment of assertions, roughly described by the grammar:

$$P := \mathsf{atomic} \mid \mathsf{emp} \mid \top \mid x \mapsto y \mid P_1 * P_2 \mid \exists x.\, P. \qquad (*)$$

Here emp is an assertion which holds of the empty heap, the "points-to" predicate $x \mapsto y$ holds of the singleton heap with location x whose contents is y, and $P_1 * P_2$ holds of a heap if it can be split into disjoint subheaps satisfying P_1 and P_2, respectively.

One important omission in (*) is the customary non-separating connectives such as implication, conjunction and universal quantification. To see why these are omitted, consider the entailments (1) $\Gamma \vdash P_1 * P_2 \to Q$ and (2) $\Gamma \vdash P_1 \wedge P_2 \to Q$, in the sequent calculus for BI [24]. Separation assertion logic is a theory of BI, obtained by specializing the model of heaps and adding the $\mapsto$ predicate; thus, all the above tools perform proof search by using some form of a sequent calculus for BI. Proving either of the sequents has to break up the implication at some point, and move P_1 and P_2 into the context Γ. But then, one needs two different context constructors in order to record that P_1 and P_2 are conjoined by $*$ in the first case, and by $\wedge$ in the second. This is semantically important, because in the first case, P_1 and P_2 hold of separate heaps, while in the second case, they hold of the same heap. Thus, contexts in the presence of both $*$ and $\wedge$ cannot be implemented in the usual manner as lists of hypotheses, but must be more involved and subject to much more complicated rules for context manipulation.

There have been a few systems that consider proofs and proof search in BI [4, 13], but to the best of our knowledge, none has been extended to support general-purpose reasoning about heaps. Instead, separation logic systems simply restrict conjunction $P \wedge Q$ and implication $P \to Q$ to assertions P that are *pure*; that is, independent of the underlying heap. If P is pure, then list-like contexts suffice. Of course, this comes at the expense of the generality of the implemented logic.

An alternative is to explicitly introduce an abstract type of heaps into the formal logic, and represent separation assertions as predicates over this type. Then heap variables can explicitly name the various heaps during proving. For example, the entailments (1) and (2) can be transformed to $\Gamma\,(h_1 \bullet h_2), P_1\,h_1, P_2\,h_2 \vdash Q\,(h_1 \bullet h_2)$ and $\Gamma\,h, P_1\,h, P_2\,h \vdash Q\,h$, respectively. Here, the variable contexts are list-like, h, h_1, h_2 are fresh heap variables, and $h_1 \bullet h_2$ is a *disjoint union* of h_1 and h_2.

To someone working in separation logic, adding the type of heaps as above may look like a significant loss of abstraction, and explicit reasoning about disjointness and heap union may be difficult to automate. Even in interactive provers like Coq, where automation is not always a priority, this may lead to large and tedious proof obligations. Thus, all Coq embeddings of separation logic that we know of [1, 11, 12, 17, 18] effectively focus only on the (*) fragment, extended with pure predicates.

Our first contribution in this paper is to show that by choosing somewhat less straightforward definitions of heaps and of heap union in Coq, we can obtain effective reasoning in the presence of abstract heap variables, and hence support full separation assertion logic while using only native hypothesis contexts, and without excessive proof obligations. The definition uses dependently-typed programming, and the idea of *reflection*, whereby decidable operations on a type are implemented as functions with codomain bool. Its important aspect is to make heaps satisfy the algebraic properties of partial commutative monoids [9].

To test our new definition in practice, we apply it to the implementation of Hoare Type Theory (HTT) [11, 20, 21], which extends the type theory of Coq to integrate separation logic into it. In such a setting, one can develop *higher-order stateful* programs, carry out proofs of their full functional correctness, and check the proofs mechanically. Programs and proofs can be organized into verified li-

POPL'10, January 17–23, 2010, Madrid, Spain.
Copyright © 2010 ACM 978-1-60558-479-9/10/01...$10.00

braries, with interfaces at an arbitrary level of abstraction, thus enabling code and proof reuse. The existing implementations of HTT, however, either allowed general separation logic [21], but lead to a prohibitive overhead in the size and number of proof obligations about heap disjointness [15], or provided aggressive proof automation by tactics (and hence very short proof scripts), but sacrificed expressiveness by focusing on the (*) fragment and omitting most structural rules of separation logic [11].

As our second contribution, we reformulate HTT to support both properties. We rely on the new definition of heaps to avoid generating excessive obligations, and keep the proofs short in the presence of non-separating connectives. We rely on Hoare triples with postconditions that are *binary*, rather than unary relations on heaps, to make the system general and extendable. We show that the binary setting supports the standard structural rules of separation logic, but also that the user can extend the system with her own auxiliary structural rules – typically, after proving a simple lemma – thus implementing new proving strategies. We develop one such strategy, and confirm that it behaves well in practice. For example, for the linked data structures such as stacks, queues and hash tables, we derive explicit full correctness proofs that are of comparable size to the proof scripts or proof hints for similar examples in related systems for full verification such as Ynot [11] and Jahob [27]. This despite the fact that the related systems allow large parts of the proofs to be omitted by the user, as these will be recovered by the proving automation.

As our third contribution, we demonstrate that the technique can be effectively applied to more realistic and complex examples. We verify the fast congruence closure algorithm of Nieuwenhuis and Oliveras [23], deployed in the state-of-the-art Barcelogic SAT solver. Our developments are carried out in Ssreflect [14], which is a recent extension of Coq that simplifies dealing with reflection. All our files are available on the web at `http://software.imdea.org/~aleks/htt.tgz`.

2. Reflecting heap disjointness

The most natural – and we argue, naive – semantic definition represents heaps as functions from locations to some kind of values. For example, in [21], heaps are defined as $\mathsf{loc}\rightarrow\mathsf{option\ dynamic}$, where the type of locations loc is isomorphic to natural numbers and $\mathsf{dynamic}$ is the record type $\{\mathsf{tp{:}Type}, \mathsf{val{:}tp}\}$, packaging a value val with its type tp. The main problem with this definition shows up when one considers heap union.

$$h_1 \bullet h_2 = \mathsf{fun}\ x. \begin{cases} h_2\,x & \text{if } h_1\,x = \mathsf{None} \\ \mathsf{Some}\,v & \text{if } h_1\,x = \mathsf{Some}\,v \text{ and } h_2\,x = \mathsf{None} \\ \mathsf{None} & \text{if } h_1\,x = \mathsf{Some}\,v \text{ and } h_2\,x = \mathsf{Some}\,w \end{cases}$$

We could make a different choice and instead of returning None when h_1 and h_2 overlap, give preference to the value stored in one of them [11, 12, 17]. In either case, we are immediately faced with proving some basic algebraic properties.

$$\begin{array}{lll} \mathsf{commute} & : & h_1 \bullet h_2 = h_2 \bullet h_1 \\ \mathsf{assoc} & : & \mathsf{disjoint}\, h_1\, h_2 \vee \mathsf{disjoint}\, h_2\, h_3 \vee \mathsf{disjoint}\, h_3\, h_1 \rightarrow \\ & & \quad h_1 \bullet (h_2 \bullet h_3) = (h_1 \bullet h_2) \bullet h_3 \end{array}$$

where $\mathsf{disjoint}\, h_1\, h_2 = \forall x\ v.\, h_1\, x = \mathsf{Some}\ v \rightarrow h_2\, x = \mathsf{None}$. An inadequacy of this definition lies in the disjointness conditions that prefix the associativity law. Associativity is used so frequently in practice that discharging its preconditions quickly becomes a serious burden. If we choose the alternative definition of $\bullet$ which gives preference to one heap over the other when they overlap, then commutativity becomes conditional, which is even worse.

Most of this inadequacy can be hidden if one avoids explicit heap variables and $\bullet$ and uses only separating conjunction $*$ instead. Assertions conjoined by $*$ are explicitly made to operate on disjoint heaps, so $*$ is commutative and associative, unconditionally. However, it is unclear how to avoid explicit heaps and $\bullet$ in the presence of non-separating connectives, so it is worth finding definitions that support unconditional algebraic laws for $\bullet$.

The main problem is that $\bullet$ is a partial operation which is not really supposed to be applied to overlapping heaps. The common way of dealing with partial operations, of course, is to complete them. We will thus adjoin a new element to the type of heaps – call this element Undef – which will be used as a default result of $\bullet$ in case we try to union non-disjoint heaps.

For the latter to work smoothly in Coq, it has to be possible to decide if two heaps are disjoint. We need a terminating procedure $\mathsf{disj{:}heap{\rightarrow}heap{\rightarrow}bool}$, which *reflects* disjointness; that is, $\mathsf{disj}\, h_1\, h_2$ evaluates to true if and only if $\mathsf{disjoint}\ h_1\ h_2$ holds. The difference between the two expressions is that $\mathsf{disj}\, h_1\, h_2$ is a boolean, while $\mathsf{disjoint}\, h_1\, h_2$ is a proposition. The first can be branched on in conditionals, while the second cannot. We will use this property of disj to give a new definition of $\bullet$ below. We also need heaps to be canonical, in the sense that two heaps are equal iff they store equal values into equal locations. These two requirements can be satisfied in many ways, but here we choose to model heaps as lists of location-value pairs, sorted in some strictly increasing order with respect to locations. In this case, disj is conceptually easy to define; it merely traverses the lists of location-value pairs, returning false if it finds an overlap in the location components, and true when it reaches the end. The definition of heaps and heap operations then takes roughly the following form.

$$\begin{array}{lll} \mathsf{heap} & = & \mathsf{Undef} \mid \mathsf{Def\ of}\ \{l : \mathsf{list}\,(\mathsf{loc}{\times}\mathsf{dynamic}), \\ & & \qquad\qquad\qquad \_ : \mathsf{sorted}\ l\} \\ \mathsf{empty} & = & \mathsf{Def}\,(\mathsf{nil},\ \mathsf{sorted\_nil}) \\ [x \mapsto v] & = & \mathsf{if}\ x == \mathsf{null\ then\ Undef} \\ & & \mathsf{else\ Def}\,((x, v){::}\mathsf{nil},\ \mathsf{sorted\_cons}\ x\ v) \\ h_1 \bullet h_2 & = & \mathsf{if}\ (h_1, h_2)\ \mathsf{is}\ (\mathsf{Def}\,(l_1, \_), \mathsf{Def}\,(l_2, \_))\ \mathsf{then} \\ & & \quad \mathsf{if\ disj}\ l_1\ l_2\ \mathsf{then} \\ & & \qquad \mathsf{Def}\,(\mathsf{sort}\,(l_1 \mathbin{++} l_2), \mathsf{sorted\_cat}\ l_1\ l_2) \\ & & \quad \mathsf{else\ Undef} \\ & & \mathsf{else\ Undef} \\ \mathsf{def}\ h & = & \mathsf{if}\ h\ \mathsf{is\ Undef\ then\ false\ else\ true} \end{array}$$

Since the definition packages each list l with a proof of $\mathsf{sorted}\ l$, the operations require *dependently-typed programming* in order to produce various sortedness proofs on-the-fly. For example, the definition of $\bullet$ applies the lemma $\mathsf{sorted\_cat}{:}\forall l_1\ l_2.\, \mathsf{sorted}(\mathsf{sort}(l_1 \mathbin{++} l_2))$ to l_1 and l_2 to convince the typechecker that $\mathsf{sort}(l_1 \mathbin{++} l_2)$ is indeed sorted. Similarly, the definitions of empty heap and singleton heap $[x \mapsto v]$, require lemmas $\mathsf{sorted\_nil{:}sorted\ nil}$ and $\mathsf{sorted\_cons} : \forall x\ v.\, \mathsf{sorted}((x, v){::}\mathsf{nil})$.

Of course, we will hide the intricacies of this definition, and keep heaps as an abstract type, only exposing several algebraic properties. Main among them are the following *unconditional* equations which, together with the def predicate, show that heaps with $\bullet$ form a *partial commutative monoid*. We will use the equations as rewrite rules for reordering heap unions during proofs.

$$\begin{array}{lll} \mathsf{unC} & : & h_1 \bullet h_2 = h_2 \bullet h_1 \\ \mathsf{unCA} & : & h_1 \bullet (h_2 \bullet h_3) = h_2 \bullet (h_1 \bullet h_3) \\ \mathsf{unAC} & : & (h_1 \bullet h_2) \bullet h_3 = (h_1 \bullet h_3) \bullet h_2 \\ \mathsf{unA} & : & (h_1 \bullet h_2) \bullet h_3 = h_1 \bullet (h_2 \bullet h_3) \\ \mathsf{un0h} & : & \mathsf{empty} \bullet h = h \\ \mathsf{unh0} & : & h \bullet \mathsf{empty} = h \end{array}$$

For example, iterated rewriting by unCA or unAC can bring a heap expression from the middle of a large union to the front or the end of it, without the steep price of proving disjointness at every step.

Even more important is the def predicate, which we use to state disjointness of heaps. For example, we can define

$$P_1 * P_2 = \mathsf{fun}\ h.\, \exists h_1\ h_2.\, h = h_1 \bullet h_2 \wedge \mathsf{def}\ h\ \wedge P_1\ h_1 \wedge P_2\ h_2.$$

The real significance of def, however, is that it can operate on arbitrary heap expressions, and can thus state *simultaneous* disjointness of a series of heaps in a union. This will allow us to freely move between assertions in separation logic, to assertions with explicit heaps, without incurring a significant blowup in size. Indeed, consider the separation logic assertion $P_1 * P_2 * (P_3 \wedge \forall x.\, P_4\ x) \rightarrow P_5$, which is outside the (*) fragment. If we want to destruct this implication and move $P_1, \ldots, P_4$ into the Coq hypothesis context, we can make the heap variables explicit and write

$$\begin{array}{l} P_1\ h_1 \wedge P_2\ h_2 \wedge P_3\ h_3 \wedge (\forall x.\, P_4\ x\ h_3) \rightarrow \\ \quad \mathsf{def}\ (h_1 \bullet h_2 \bullet h_3) \rightarrow P_5\ (h_1 \bullet h_2 \bullet h_3) \end{array}$$

This is more verbose than the original, *but only slightly*, as we have to keep track of *only one* def predicate per sequence of iterated $*$'s. With the naive definition, exposing the heap variables is a non-starter, as we would have to separately assert that each pair of heaps in the series h_1, h_2, h_3 is disjoint, and possibly later prove disjointness for any *partitioning* of the series (e.g., h_1 is disjoint from $h_2 \bullet h_3$, h_2 is disjoint from $h_1 \bullet h_3$, etc.). This leads to an exponential blowup, whereas with the new definition, propositions and proofs are proportional to their separation logic originals. Of course, we will have to devise methods to make inferences from and about def predicates.

Example 1. A frequently used law related to non-separating conjunction is the following.

$$(x \mapsto v_1) * P_1 \wedge (x \mapsto v_2) * P_2 \rightarrow v_1 = v_2 \wedge (x \mapsto v_1) * (P_1 \wedge P_2)$$

The law can be proved in our setting as well, but we have found that a somewhat different formulation, which states a variant of the cancellation property for $\bullet$, is much more convenient to use.

$$\begin{array}{l} \mathsf{cancel} : \mathsf{def}([x \rightarrow v_1] \bullet h_1) \rightarrow ([x \rightarrow v_1] \bullet h_1 = [x \rightarrow v_2] \bullet h_2) \\ \qquad\quad \rightarrow v_1 = v_2 \wedge \mathsf{def}\, h_1 \wedge h_1 = h_2. \end{array}$$

The conclusion of cancel produces new facts def h_1 and $h_1 = h_2$, to which cancel can be applied again. This way, we can iterate and chain several cancellations in one line of proof, obtaining definedness of sub-unions, out of definedness of larger heap unions.

Example 2. Consider the predicate lseq $p\ l\ h$, which states that the heap h contains a singly-linked list headed at pointer p, and stores elements from the purely-functional list l.

$$\begin{array}{l} \mathsf{Fixpoint\ lseq}\ (p : \mathsf{loc})\ (l : \mathsf{list}\ T) : \mathsf{heap} \rightarrow \mathsf{Prop} = \\ \quad \mathsf{if}\ l\ \mathsf{is}\ x{::}xt\ \mathsf{then} \\ \qquad \mathsf{fun}\ h.\, \exists q\ h'.\, h = [p \rightarrow x] \bullet [p{+}1 \rightarrow q] \bullet h' \wedge \\ \qquad\qquad\qquad\quad \mathsf{lseq}\ q\ xt\ h' \wedge \mathsf{def}\ h \\ \quad \mathsf{else\ fun}\ h.\, p = \mathsf{null} \wedge h = \mathsf{empty} \end{array}$$

Imagine we want to prove that lseq is functional in the l argument; that is lseq_func : $\forall l_1\ l_2\ p\ h.$ lseq $p\ l_1\ h \rightarrow$ lseq $p\ l_2\ h \rightarrow l_1 = l_2$. We will use the following easy helper lemmas.

$$\begin{array}{ll} \mathsf{lseq\_nil} & : \forall l\ h.\, \mathsf{lseq\ null}\ l\ h \rightarrow l = \mathsf{nil} \wedge h = \mathsf{empty} \\ \mathsf{lseq\_cons} & : \forall l\ p\ h.\, p \neq \mathsf{null} \rightarrow \mathsf{lseq}\ p\ l\ h \rightarrow \\ & \qquad \exists x\ q\ h'.\, l = x{::}\mathsf{tail}\ l\ \wedge \\ & \qquad\qquad h = [p \rightarrow x] \bullet [p{+}1 \rightarrow q] \bullet h' \wedge \\ & \qquad\qquad \mathsf{lseq}\ q\ (\mathsf{tail}\ l)\ h' \wedge \mathsf{def}\ h \\ \mathsf{def\_null} & : \forall p\ x\ h.\, \mathsf{def}\ ([p \rightarrow x] \bullet h) \rightarrow p \neq \mathsf{null} \end{array}$$

The proof is by induction on l_1. If $l_1 = \mathsf{nil}$, then lseq $p\ l_1\ h$ implies $p =$ null and the result follows by applying the lemma lseq_nil to the hypothesis lseq $p\ l_2\ h$. Otherwise, let $l_1 = x_1 :: xt$, let *IH* be the induction hypothesis $\forall l_2\, p\, h.$ lseq $p\ xt\ h \rightarrow$ lseq $p\ l_2\ h \rightarrow xt = l_2$. From lseq $p\ l_1\ h$ and the definition of lseq, we know that there exist q_1, h_1 such that $h = [p \rightarrow x_1] \bullet [p{+}1 \rightarrow q_1] \bullet h_1$, and lseq $q_1\ xt\ h_1$ and def $([p \rightarrow x_1] \bullet [p{+}1 \rightarrow q_1] \bullet h_1)$. Call the last two facts H and D, respectively. It suffices to show

$$\mathsf{lseq}\ p\ l_2\ ([p \rightarrow x_1] \bullet [p{+}1 \rightarrow q_1] \bullet h_1) \rightarrow x_1 :: xt = l_2.$$

The hypothesis lseq $p\ l_2\ ([p \rightarrow x_1] \bullet [p{+}1 \rightarrow q_1] \bullet h_1)$ and the fact that $p \neq$ null (proved by def_null and D) can now be used with the lemma lseq_cons, to obtain x_2, q_2 and h_2, and reduce the goal to

$$\begin{array}{l} [p \rightarrow x_1] \bullet [p{+}1 \rightarrow q_1] \bullet h_1 = [p \rightarrow x_2] \bullet [p{+}1 \rightarrow q_2] \bullet h_2 \\ \quad \rightarrow \mathsf{lseq}\ q_2\ (\mathsf{tail}\ l_2)\ h_2 \rightarrow x_1 :: xt = x_2 :: \mathsf{tail}\ l_2 \end{array}$$

By applying cancel to D and the antecedent of this implication, we get $x_1 = x_2$ as well as def $([p{+}1 \rightarrow q_1] \bullet h_1)$ and $[p{+}1 \rightarrow q_1] \bullet h_1 = [p{+}1 \rightarrow q_2] \bullet h_2$. By chaining cancel again over this def predicate and equation, we further get $q_1 = q_2$ and $h_1 = h_2$, reducing the goal to lseq q_1 (tail l_2) $h_1 \rightarrow x_1 :: xt = x_1 :: \mathsf{tail}\, l_2$. But, if lseq q_1 (tail l_2) h_1, then by *IH* and H, it must be $xt =$ tail l_2, and thus $x_1 :: xt = x_1 ::$ tail l_2.

Notice that the proof did not require any overwhelming reasoning about heap disjointness, despite the explicit heap variables. In fact, the whole argument can be captured by the following quite concise formal proof in Ssreflect.

```
elim ⇒ [ |x1 xt IH] l2 p h; first by case⇒→→; case/lseq_nil.
case ⇒ q1 [h1][→] H D.
case/(lseq_cons (def_null D)) ⇒ x2 [q2][h2][→].
do 2![case/(cancel D) ⇒← {D} D] ⇒← .
by case/(IH _ _ _ H) ⇒← .
```

In Section 4, we return to the issue of chaining the reasoning about def predicates, and show how it applies when proving properties of Hoare triples. But first, we describe the basic ideas behind our representation of Hoare triples in type theory.

3. Hoare type theory for separation logic

The most common approach to formalizing Hoare logic in proof assistants like Coq is by "deep embedding" where one reasons about the abstract syntax of the programming language in question [17, 18]. This reasoning indirection via syntax is often quite burdensome. For example, a deep embedding of a typed functional language will usually involve explicit manipulation of de Bruijn representation of bound variables, formalization of a type checker for the embedded language, etc.

In contrast, HTT formalizes separation logic via types; a triple $\{p\}\, e\, \{q\}$ in HTT becomes a type ascription e : STsep $A\ (p, q)$, where A is the type of the return value of the "command" e. The type STsep $A\ (p, q)$ is a monad [20], which makes it possible for commands e to perform side-effects, without compromising the soundness of the whole system. Moreover, commands can freely use the purely-functional programming fragment of Coq, including inductive types, higher-order functions, type abstraction and first-class modules, which removes a level of indirection and streamlines the programming and reasoning in HTT. Encoding via types, however, is not straightforward, and requires a reformulation of the inference rules of separation logic.

These inference rules are presented in Figure 1, and they come in two flavors. The first flavor includes rules that infer properties based on program's top command, where the commands are: move $x\ v$ for assigning a value v to the variable x; store $x\ v$ for writing v into the location x; load $y\ x$ for reading the value stored in location x and assigning it to variable y; alloc $y\ v$ for allocating a new location initialized with v, and storing the address into y; dealloc x for deallocating the location x; and $e_1; e_2$ for sequential composition of commands e_1 and e_2.

The second flavor includes the structural rules. These vary across systems, but here we take them to include the rules of

$$\{\mathsf{emp}\}\ \mathsf{move}\ x\ v\ \{x = v \wedge \mathsf{emp}\} \qquad \{x \mapsto -\}\ \mathsf{store}\ x\ v\ \{x \mapsto v\}$$

$$\{x \mapsto v\}\ \mathsf{load}\ y\ x\ \{x \mapsto v \wedge y = v\}$$

$$\{\mathsf{emp}\}\ \mathsf{alloc}\ y\ v\ \{y \mapsto v\} \qquad \{x \mapsto -\}\ \mathsf{dealloc}\ x\ \{\mathsf{emp}\}$$

$$\frac{\{p\}\, e_1\, \{q\} \quad \{q\}\, e_2\, \{r\}}{\{p\}\, e_1; e_2 \{r\}}\,[\mathsf{seq}]$$

$$\frac{\{p\}\, e\, \{q\}}{\{p * r\}\, e\, \{q * r\}}\,[\mathsf{frame}]$$

$$\frac{p \rightarrow p' \ \ \{p'\}\, e\, \{q'\} \ \ q' \rightarrow q}{\{p\}\, e\, \{q\}}\,[\mathsf{consequence}]$$

$$\frac{\{p\}\, e\, \{q_1\} \quad \{p\}\, e\, \{q_2\}}{\{p\}\, e\, \{q_1 \wedge q_2\}}\,[\wedge] \qquad \frac{\{p\}\, e\, \{q\} \quad x \notin \mathsf{FV}(e, p)}{\{p\}\, e\, \{\forall x.\, q\}}\,[\forall]$$

$$\frac{\{p_1\}\, e\, \{q\} \quad \{p_2\}\, e\, \{q\}}{\{p_1 \vee p_2\}\, e\, \{q\}}\,[\vee] \qquad \frac{\{p\, x\}\, e\, \{q\} \quad x \notin \mathsf{FV}(e, q)}{\{\exists x.\, p\}\, e\, \{q\}}\,[\exists]$$

Figure 1. Inference rules of separation logic.

frame, consequence, conjunction and disjunction in both binary and quantified (i.e., universal and existential) variants. Separation logic also includes the rule of substitution, which allows inferring $\{\sigma\, p\}\, \sigma\, e\, \{\sigma\, q\}$ out of $\{p\}\, e\, \{q\}$, for any variable substitution σ, but we will not explicitly consider such a rule in this paper, as we will inherit it from the underlying substitution principles of Coq.

Ignoring the rule of frame for a second, the role of the other structural rules is, informally, to present the view of commands as relations between the input and output heaps. Intuitively, if $\{p\}\ e\ \{q\}$, then e implements the relation $\{(h_1, h_2) \mid p\, h_1 \rightarrow q\, h_2\}$, and e does not crash. The structural rules then simply expose how logical connectives interact with the implication in this relation (e.g., implication distributes over conjunction in the consequent, and disjunction in the antecedent, etc.).

The difficulty with structural rules is that they cannot easily be encoded as typing rules. One problem is that the universal and existential rules require a side-condition that x is not a free variable of e, and this property of e cannot be expressed from within the system. Another problem is that the structural rules use the same e both in premisses and conclusions, thus making it impossible to define the typing judgment by induction on the structure of expressions, which is one of the main design principles of Coq.

Our proposal for solving these problems is to switch to binary postconditions. If Hoare triples have binary postconditions, this quite directly exposes the relational nature of commands, which is what the role of structural rules was to start with: intuitively, if a command e has a binary postcondition q, then it must implement a relation on heaps which is a subset of q. Then reasoning about e can be reduced to reasoning about q and can be carried out *in the logic of assertions*, rather than in the logic of Hoare triples. Of course, this only works smoothly if the assertion logic can express properties of relations, and quantify over them. This is not a problem for us, as Coq already includes higher-order logic.

To present the semantics of STsep, we briefly sketch a denotational model based on predicate transformers. The related proofs are carried out in Coq, and can be found on our web site. We represent preconditions as elements of the type heap→Prop, and postconditions as elements of A→heap→heap→Prop, for any given type A. In addition to abstracting over two heaps, the postconditions also abstract over values of type A, because commands in HTT are value-returning, so the postconditions must be able to relate the value to the input and the output heap of the computation. Despite this, we still refer to the postconditions as "binary", as the type A does not introduce any significant complications.

Given the type A, precondition p:heap→Prop, and binary postcondition q:A→heap→heap→Prop, our predicate transformers are elements of the type

$$\mathsf{model}\ p\ A = \mathsf{ideal}\ p \rightarrow A \rightarrow \mathsf{heap} \rightarrow \mathsf{Prop}.$$

The transformers should only "transform" predicates that are "stronger" than p, so we define ideal p as:

$$\mathsf{ideal}\ p = \{f : \mathsf{heap} \rightarrow \mathsf{Prop} \mid f \sqsubseteq p\}$$

where $r_1 \sqsubseteq r_2$ iff $\forall h{:}\mathsf{heap}.\, r_1\ h \rightarrow r_2\ h$. We further only need transformers that are monotone and bounded by q:

$$\mathsf{ST}\ A\ (p, q) = \{F{:}\mathsf{model}\ p\ A \mid \mathsf{monotone}\ F \wedge \mathsf{bounded}\ F\ q\}$$

where

$$\mathsf{monotone}\ F = \forall r_1\ r_2{:}\mathsf{ideal}\ p.\, r_1 \sqsubseteq r_2 \rightarrow \forall x.\, F\ r_1\ x \sqsubseteq F\ r_2\ x$$
$$\mathsf{bounded}\ F\ q = \forall r\ x.\, F\ r\ x \sqsubseteq \mathsf{fun}\ h.\, (\exists i.\, r\ i \wedge q\ x\ i\ h).$$

The elements of type ST $A\ (p, q)$ can be used to model programs that return values of type A, and have a precondition p and postcondition q *in ordinary Hoare logic*, where p and q describe the behavior of the program on the *whole heap*. But in separation logic, p and q only describe the part of the heap that the program actually reads from or modifies during execution; the information that the rest of the heap remains invariant is implicit in the semantics. To capture this aspect of separation logic, we next select a specific subset of predicate transformers out of ST. Given a pre/postcondition pair s, we define *spatial extension* $s^{\bullet}$, and a new STsep type, as follows.

$$\begin{array}{rcl} s^{\bullet} & = & (\mathsf{pre}\ s * \mathsf{fun}\ h.\, \top,\ \mathsf{fun}\ x.\, \mathsf{pre}\ s \multimap \mathsf{post}\ s\ x) \\ \mathsf{STsep}\ A\ s & = & \mathsf{ST}\ A\ s^{\bullet} \end{array}$$

where pre and post are the projections out of the pair, and

$$\begin{array}{l} p \multimap q = \{(i, m) \mid \forall\, i_1\ h.\, i = i_1 \bullet h \rightarrow \mathsf{def}\, i \rightarrow p\ i_1 \rightarrow \\ \qquad\qquad\qquad \exists m_1.\, m = m_1 \bullet h \wedge \mathsf{def}\, m \wedge q\ i_1\ m_1\}. \end{array}$$

Spatial extension allows that heaps on which a transformer is applied be extended with portions that the transformer keeps invariant. For example, transformers in STsep $A\ (p, q)$ take a predicate describing a heap i which contains a subheap i_1 satisfying p, and transform it into a predicate stating that the rest of i (here called h) remains unchanged. The unchanged heap h can be arbitrary, as the precondition only requires h to satisfy $\top$. We note that the definition of $\multimap$ is quite similar to the notion of "best local action" from [9], and has also been used previously in [19].

We can now transcribe the inference rules about commands as typing rules about elements of STsep. We only list the relevant types, and defer to the Coq scripts for the definitions and proofs. In all the types, i and m stand for the initial and ending heap of a computation, and y is the name for the return value. We further adopt names that are traditional in functional programming, and use return for move, ":=" for store and "!" for load.

$$\begin{array}{ll} \mathsf{return} & : \Pi v{:}A.\, \mathsf{STsep}\ A\ (\mathsf{emp}, \mathsf{fun}\ y\ i\ m.\, y = v \wedge \mathsf{emp}\ m) \\ := & : \Pi x{:}\mathsf{loc}\ v{:}A. \\ & \quad \mathsf{STsep}\ \mathsf{unit}\ (x \mapsto -, \mathsf{fun}\ y\ i\ m.\, (x \mapsto v)\ m \wedge y = (\,)) \\ ! & : \Pi x{:}\mathsf{loc}.\, \mathsf{STsep}\ A\ (x \mapsto -,\ \mathsf{fun}\ y\ i\ m.\, \forall v.\, (x \mapsto v)\ i \rightarrow \\ & \qquad\qquad\qquad\qquad (x \mapsto v)\ m \wedge y = v) \\ \mathsf{alloc} & : \Pi v{:}A.\, \mathsf{STsep}\ \mathsf{loc}\ (\mathsf{emp}, \mathsf{fun}\ y\ i\ m.\, (y \mapsto v)\ m) \\ \mathsf{dealloc} & : \Pi x{:}\mathsf{loc}.\, \mathsf{STsep}\ \mathsf{unit}\ (x \mapsto -, \\ & \qquad\qquad \mathsf{fun}\ y\ i\ m.\, \mathsf{emp}\ m \wedge y = (\,)) \end{array}$$

We also have a command for allocation of a block of n consecutive locations, initialized with the value v:

$$\begin{array}{l} \mathsf{allocb} : \Pi v{:}A.\, \Pi n{:}\mathsf{nat}.\, \mathsf{STsep}\ \mathsf{loc}\ (\mathsf{emp}, \\ \qquad\qquad\qquad\qquad \mathsf{fun}\ y\ i\ m.\, m = \mathsf{iter}\ n\ y\ v) \end{array}$$

$$\begin{array}{l} \mathsf{iter}\ n\ y\ v = \mathsf{if}\ n\ \mathsf{is}\ n' + 1\ \mathsf{then}\ [y \rightarrow v] \bullet \mathsf{iter}\ n'\ (y + 1)\ v \\ \qquad\qquad\quad \mathsf{else\ empty} \end{array}$$

And, we require a fixed-point combinator with the type below. In our ST model, this combinator computes the least fixed point of the monotone completion of the argument function.

$$\begin{aligned}\mathsf{fix} : &((\Pi x{:}A.\, \mathsf{STsep}\ (B\ x)\ (s\ x)) \to \Pi x{:}A.\, \mathsf{STsep}\ (B\ x)\ (s\ x))\\ &\to \Pi x{:}A.\, \mathsf{STsep}\ (B\ x)\ (s\ x)\end{aligned}$$

Transcribing the rule for sequential composition is somewhat more involved. The command e_1 now returns a value of type A_1, and thus e_2 must be a function which takes that value as an argument. We will have a typing rule as follows

$$\begin{aligned}\mathsf{bind} : \Pi e_1{:}\mathsf{STsep}\ A_1\ s_1.\, \Pi e_2{:}(\Pi x{:}A_1.\, \mathsf{STsep}\ A_2\ (s_2\ x)).\\ \mathsf{STsep}\ A_2\ (\mathsf{bind\_s}\ s_1\ s_2),\end{aligned}$$

where s_1 and $s_2\ x$ are pairs of pre/postconditions for e_1 and $e_2\ x$, respectively, and $\mathsf{bind\_s}\ s_1\ s_2$ is the following pre/postcondition pair.

$$\begin{aligned}&(\mathsf{fun}\ i.\, \mathsf{pre}\ s_1^\bullet\ i \wedge \forall x\, h.\, \mathsf{post}\ s_1^\bullet\ x\ i\ h \to \mathsf{pre}\ (s_2\ x)^\bullet\ h,\\ &\ \mathsf{fun}\ y\ i\ m.\, \exists x\, h.\, \mathsf{post}\ s_1^\bullet\ x\ i\ h \wedge \mathsf{post}\ (s_2\ x)^\bullet\ y\ h\ m).\end{aligned}$$

The precondition in this pair states that in order to execute the sequential composition, we must ensure that the precondition $\mathsf{pre}\ s_1^\bullet$ holds, so that e_1 can run in a subheap of the initial heap i. After e_1 is done, we will have an intermediate value x and heap h satisfying $\mathsf{post}\ s_1^\bullet\ x\ i\ h$, so we need to show $\mathsf{pre}\ (s_2\ x)^\bullet\ h$ in order to execute e_2. The postcondition states that there exists an intermediate value x and heap h, obtained after running e_1 but before running e_2. In the model of ST, bind is implemented as the functional composition of the transformers for e_1 and e_2.

We now turn to the structural rules. For a command $e : \mathsf{ST}\ A\ s$, we consider what can be inferred about e *just by looking at the type A and specification s*. Quite directly, it must be that $\mathsf{pre}\ s\ i$ and $\mathsf{post}\ s\ y\ i\ m$ hold of the initial heap i, final heap m and return value y. Thus, given a property $q{:}A{\to}\mathsf{heap}{\to}\mathsf{Prop}$, we can show that $q\ y\ m$ holds after running e if we can prove $\mathsf{verify}\ i\ s\ q$, where

$$\mathsf{verify}\ i\ s\ q\ =\ \mathsf{pre}\ s\ i \wedge \forall y\, m.\, \mathsf{post}\ s\ y\ i\ m \to q\ y\ m.$$

This definition assumes that s describes how e acts on the *whole* heap i. If $e{:}\mathsf{STsep}\ A\ s$, then s describes the action of e only on a subheap of i. Following the definition of STsep, in order to show that $q\ y\ m$ holds after running e, it then suffices to prove $\mathsf{verify}\ i\ s^\bullet\ q$.

The verify predicate can now be used to represent Hoare triples as assertions. For example, given $e{:}\mathsf{STsep}\ A\ s$, the separation logic triple $\{p\}\, e\, \{q\}$ can be written as $\forall i.\, p\ i \to \mathsf{verify}\ i\ s^\bullet\ q$. This property will let us encode the standard structural rules, as well as many other useful rules, as simple *derived lemmas* about the verify predicate. Hence, our system will be inherently extendable, as the user is free to derive her own structural rules, and thus design custom reasoning principles and strategies. Moreover, the definition of verify does not involve the command e, but only the specification s, making any lemma about verify *independent* of our particular model of ST. We will be able in the future to develop different models for HTT, while preserving the lemmas and the verification technique we describe here.

As a first illustration of working with verify, we show the following variants of the binary and quantified conjunction rules.

$$\begin{aligned}\mathsf{conj} : \mathsf{verify}\ i\ s\ q_1 \to \mathsf{verify}\ i\ s\ q_2 \to\\ \mathsf{verify}\ i\ s\ (\mathsf{fun}\ y\ h.\, q_1\ y\ h \wedge q_2\ y\ h)\\ \mathsf{all} : (\forall x{:}B.\, \mathsf{verify}\ i\ s\ (q\ x)) \to B \to\\ \mathsf{verify}\ i\ s\ (\mathsf{fun}\ y\ m.\, \forall x{:}B.\, q\ x\ y\ m)\end{aligned}$$

Several interesting twists appear here. First, the rules use implication and quantification, and cannot be stated in the (*) fragment alone. Thus, here we are making an essential use of our formulation of heaps from Section 2. Second, the rules omit the precondition $p\ i$ as it is invariant across implications. They also omit the side-condition $x \notin \mathsf{FV}\, s$, because s is declared outside of the scope of x. Finally, the all rule requires that B is a non-empty type. Otherwise $\forall x{:}B.\, q\ x\ y$ is trivially true, but this does not suffice to establish the verify predicate, as the latter additionally requires the precondition to hold of the initial state, no matter what the postcondition is. This makes the semantics of HTT *fault-avoiding* [9]; that is, it ensures that well-typed commands are safe to execute.

On the other hand, the binary and quantified disjunction rules do not require any special treatment. For example, we can prove

$$\begin{aligned}\mathsf{disj} : (p_1\ i \to \mathsf{verify}\ i\ s\ q) \to (p_2\ i \to \mathsf{verify}\ i\ s\ q) \to\\ p_1\ i \vee p_2\ i \to \mathsf{verify}\ i\ s\ q\\ \mathsf{exist} : (\forall x.\, p\ x \to \mathsf{verify}\ i\ s\ q) \to (\exists x.\, p\ x) \to \mathsf{verify}\ i\ s\ q\end{aligned}$$

but these are just instances of the usual elimination rules for $\vee$ and $\exists$, and therefore do not require separate lemmas.

The frame rule can be formulated in several different ways, but we choose the following:

$$\begin{aligned}\mathsf{frame} : \mathsf{verify}\ i\ s^\bullet\ (\mathsf{fun}\ y\ m.\ \mathsf{def}\ (m \bullet h) \to q\ y\ (m \bullet h)) \to\\ \mathsf{def}\ (i \bullet h) \to \mathsf{verify}\ (i \bullet h)\ s^\bullet\ q.\end{aligned}$$

When read bottom-up, this lemma replaces a goal about the heap $i \bullet h$ and a postcondition q, with a new goal involving the heap i alone, and a postcondition recording that q should eventually be proved of the ending heap m extended with h. We have chosen this formulation because it applies to goals where q is arbitrary, whereas the usual formulation from Figure 1 requires first rewriting q into a form $q' * r$, and this if often tedious in the presence of higher-order operations and binary postconditions.

Finally, we need to connect STsep types with the verify predicate. The structural rules all show how to change a specification of a command under certain conditions. We match that ability at the level of typing rules, by introducing a construct for changing an STsep type of a command, which essentially implements the rule of consequence.

$$\begin{aligned}&\mathsf{do} : \mathsf{STsep}\ A\ s_1 \to\\ &\quad (\forall i.\, \mathsf{pre}\ s_2\ i \to \mathsf{verify}\ i\ s_1^\bullet\ (\mathsf{fun}\ y\ m.\, \mathsf{post}\ s_2\ y\ i\ m)) \to\\ &\quad\ \mathsf{STsep}\ A\ s_2\end{aligned}$$

In our model of ST, do is an identity predicate transformer. With this connective, we have embedded all the rules of separation logic from the beginning of this section.

Example 3. It is possible to use Coq's purely-functional pattern-matching to build pattern-matching constructs with side-effectful branches. For example, in the case of booleans, we have:

$$\begin{aligned}\mathsf{If}\ :\ \Pi b{:}\mathsf{bool}.\, \mathsf{STsep}\ A\ s_1\ \to \mathsf{STsep}\ A\ s_2 \to\\ \mathsf{STsep}\ A\ (\mathsf{if}\ b\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2)\\ = \mathsf{fun}\ b\ e_1\ e_2.\, \mathsf{if}\ b\ \mathsf{then}\ (\mathsf{do}\ e_1)\ \mathsf{else}\ (\mathsf{do}\ e_2)\end{aligned}$$

The do's in the branches serve to weaken the types of e_i into the common type of the conditional. Both do e_1 and do e_2 require a (simple) proof that if b equals true (resp. false), then s_1 (resp. s_2) can be weakened into if b then s_1 else s_2. To reduce clutter, in the rest of the paper we blur the distinction between purely-functional if and side-effectful If, and use if for both.

Example 4. The following functions insert and remove an element from the head of a singly-linked list pointed to by p.

$$\begin{aligned}&\mathsf{insert}\ (p : \mathsf{loc})\ (x : T) :\\ &\quad \mathsf{STsep}\ \mathsf{loc}\ (\mathsf{fun}\ i.\, \exists l.\, \mathsf{lseq}\ p\ l\ i,\\ &\qquad\qquad \mathsf{fun}\ y\ i\ m.\, \forall l.\, \mathsf{lseq}\ p\ l\ i \to \mathsf{lseq}\ y\ (x :: l)\ m) =\\ &\quad \mathsf{do}\ (y \leftarrow \mathsf{allocb}\ p\ 2;\\ &\qquad\quad y := x;\\ &\qquad\quad \mathsf{return}\ y)\end{aligned}$$

$$\begin{array}{l}
\mathsf{remove}\ (p : \mathsf{loc}) : \mathsf{STsep\ loc}\ (\mathsf{fun}\ i.\, \exists l.\, \mathsf{lseq}\ p\ l\ i,\\
\qquad\qquad\qquad\qquad\qquad \mathsf{fun}\ y\ i\ m.\, \forall l.\, \mathsf{lseq}\ p\ l\ i \longrightarrow\\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{lseq}\ y\ (\mathsf{tail}\ l)\ m) =\\
\quad \mathsf{do}\ (\mathsf{if}\ p == \mathsf{null\ then\ return}\ p\\
\qquad \mathsf{else}\ y \leftarrow !(p+1);\\
\qquad\quad \mathsf{dealloc}\ p;\\
\qquad\quad \mathsf{dealloc}\ (p+1);\\
\qquad\quad \mathsf{return}\ y)
\end{array}$$

Here, we have used the standard abbreviation $x \leftarrow e_1; e_2$ for $\mathsf{bind}\ e_1\ (\mathsf{fun}\ x.\, e_2)$, and $e_1; e_2$ when $x \notin \mathsf{FV}(e_2)$. For both functions, the STsep type gives the specification that we want to prove about the functions. The preconditions show that the functions can execute safely, as long as the initial heap contains a valid linked list, no matter what values l are stored in it. The postconditions show that the new list now contains $x :: l$ and $\mathsf{tail}\ l$, respectively, and that the returned value y is a pointer to the new head.

The specification pattern seen in these examples, where the predicate from the precondition is, somewhat redundantly, repeated in the postcondition, is characteristic to the setting with binary postconditions, though it is by no means always used. Fortunately, this redundancy will not cause an explosion in proof obligations, and in Section 4, we show how to quickly remove it.

The typing rules are designed so that they can now generate the proof obligation required to verify the programs. For insert, we get

$$\begin{array}{l}
\forall p\ x\ i.\, (\exists l.\, \mathsf{lseq}\ p\ l\ i) \longrightarrow\\
\quad \mathsf{verify}\ i\ (\mathsf{bind\_s}\ (\mathsf{allocb\_s}\ p\ 2)\\
\qquad\qquad (\mathsf{fun}\ y.\, \mathsf{bind\_s}\ (\mathsf{write\_s}\ y\ x)\\
\qquad\qquad\quad \mathsf{fun}\ \_.\, \mathsf{return\_s}\ y))^{\bullet}\\
\qquad\quad (\mathsf{fun}\ y\ m.\, \forall l.\, \mathsf{lseq}\ p\ l\ i \longrightarrow \mathsf{lseq}\ y\ (x{::}l)\ m)
\end{array}$$

and for remove

$$\begin{array}{l}
\forall p\ i.\, (\exists l.\, \mathsf{lseq}\ p\ l\ i) \longrightarrow\\
\quad \mathsf{verify}\ i\ (\mathsf{if}\ p\ ==\ \mathsf{null\ then\ return\_s}\ p\\
\qquad\qquad \mathsf{else\ bind\_s}\ (\mathsf{read\_s}\ (p+1)\\
\qquad\qquad\quad (\mathsf{fun}\ x.\, \mathsf{bind\_s}\ (\mathsf{dealloc\_s}\ p)\\
\qquad\qquad\qquad (\mathsf{fun}\ \_.\, \mathsf{bind\_s}\ (\mathsf{dealloc\_s}\ (p+1))\\
\qquad\qquad\qquad\quad \mathsf{fun}\ \_.\, \mathsf{return\_s}\ x))))^{\bullet}\\
\qquad\quad (\mathsf{fun}\ q\ m.\, \forall l.\, \mathsf{lseq}\ p\ l\ i \longrightarrow \mathsf{lseq}\ q\ (\mathsf{tail}\ l)\ m)
\end{array}$$

The proof obligations essentially copy the original command, except that the various primitive commands are replaced by their pre/postcondition pairs from the beginning of this section. For example, $\mathsf{return\_s}\ p$ is the pair $(\mathsf{emp}, \mathsf{fun}\ y\ i\ m.\, y = p \wedge \mathsf{emp}\ m)$, $\mathsf{read\_s}\ x$ is $(x \mapsto -, \mathsf{fun}\ y\ i\ m.\, \forall v.\, (x \mapsto v)\ i \longrightarrow (x \mapsto v)\ m \wedge y = v)$, etc. In the case of a call to an already verified non-primitive side-effectful command (not used in insert and remove, but used in programs in Section 5), the command is not copied, but the pre/postcondition pair from the type of the called command is simply spliced in. Calls to fix are similar, except that a separate obligation is generated to prove that the body of fix satisfies the provided type. Thus, the type of the fixed point serves as the loop invariant.

4. Structural rules and verification

As structural rules are now simply lemmas over the verify predicate, one is free to prove and use additional ones, that may be useful for the proof at hand. For example, the following is a variant of the rule for universal quantifiers, which pulls a quantifier *and an implication* out of a postcondition, both at the same time.

$$\begin{array}{l}
\mathsf{all\_imp} : (\exists x{:}B.\, p\ x) \longrightarrow\\
\qquad (\forall x{:}B.\, p\ x \longrightarrow \mathsf{verify}\ i\ s\ (\mathsf{fun}\ y\ m.\ \ q\ x\ y\ m)) \longrightarrow\\
\qquad\quad \mathsf{verify}\ i\ s\ (\mathsf{fun}\ y\ m.\, \forall x{:}B.\, p\ x \longrightarrow q\ x\ y\ m)
\end{array}$$

This rule can be used to simplify the proof obligation from Example 4, by removing the occurrence of lseq from the postcondition. If p uniquely determines x in the current context of hypotheses, we may use the following rule to instantiate the quantifier with the unique value for x.

$$\begin{array}{l}
\mathsf{all\_imp}_1 : \forall t{:}B.\, (\forall x{:}B.\, p\ x \to t = x) \to \mathsf{verify}\ i\ s\ (q\ t) \to\\
\qquad\qquad \mathsf{verify}\ i\ s\ (\mathsf{fun}\ y\ m.\, \forall x{:}B.\, p\ x \to q\ x\ y\ m)
\end{array}$$

Sometimes, p may not uniquely determine x, but determines "just enough" of x to establish q. For example, p may force x to be in an equivalence relation to a predetermined t. Then we are justified in instantiating x with t, as long as q only makes statements about the common equivalence class of x and t.

$$\begin{array}{l}
\mathsf{all\_imp}_2 : \forall t{:}B.\, (\forall x{:}B\ y\ m.\, p\ x \to q\ t\ y\ m \to q\ x\ y\ m) \to\\
\qquad\qquad \mathsf{verify}\ i\ s\ (q\ t) \to\\
\qquad\qquad\quad \mathsf{verify}\ i\ s\ (\mathsf{fun}\ y\ m.\, \forall x{:}B.\, p\ x \to q\ x\ y\ m)
\end{array}$$

We also have additional rules to help us discharge the proof obligations generated by typechecking. As Example 4 shows, these should be lemmas about how verify interacts with pre/postcondition pairs such as bind_s, read_s, etc. The main lemma of the system serves to simplify proof obligations that are obtained when verifying commands of the form $\mathsf{bind}\ e_1\ e_2$ where e_1, e_2 are arbitrary commands, with types $e_1 : \mathsf{STsep}\ A_1\ s_1$ and $e_2 : \Pi x{:}A_1.\, \mathsf{STsep}\ A_2\ (s_2\ x)$, respectively.

$$\begin{array}{l}
\mathsf{bnd\_do} : \mathsf{pre}\ s_1\ i_1 \to\\
\qquad (\forall x\ i_1'.\, \mathsf{post}\ s_1\ x\ i_1\ i_1' \to\\
\qquad\quad \mathsf{def}\ (i_1' \bullet i_2) \to \mathsf{verify}\ (i_1' \bullet i_2)\ (s_2\ x)^{\bullet}\ r) \to\\
\qquad \mathsf{def}\ (i_1 \bullet i_2) \to \mathsf{verify}\ (i_1 \bullet i_2)\ (\mathsf{bind\_s}\ s_1\ s_2)^{\bullet}\ r
\end{array}$$

Applying this lemma to a goal of the form $\mathsf{def}\ (i_1 \bullet i_2) \to \mathsf{verify}\ (i_1 \bullet i_2)\ (\mathsf{bind\_s}\ s_1\ s_2)^{\bullet}\ r$ essentially corresponds to "symbolically executing" e_1 in the subheap i_1. The lemma first issues a proof obligation that the precondition $\mathsf{pre}\ s_1$ of e_1 is satisfied in i_1, then replaces i_1 with a fresh heap variable i_1', inserts the knowledge that i_1' satisfies the postcondition of e_1, and reduces to verifying the continuation e_2 in the changed heap.

We can further instantiate this lemma to exploit additional knowledge that we may have about e_1. For example, if e_1 starts with one of the primitive commands, we have the following instances, where we omit the def predicate if the command does not change the heap.

$$\mathsf{bnd\_ret}\ :\ \mathsf{verify}\ i\ (s_2\ v)^{\bullet}\ r \longrightarrow \mathsf{verify}\ i\ (\mathsf{bind\_s}\ (\mathsf{return\_s}\ v)\ s_2)^{\bullet}\ r$$

$$\begin{array}{l}
\mathsf{bnd\_read}\ :\ \mathsf{verify}\ ([x \to v] \bullet i)\ (s_2\ v)^{\bullet}\ r \longrightarrow\\
\qquad\qquad \mathsf{def}\ ([x \to v] \bullet i) \longrightarrow\\
\qquad\qquad\quad \mathsf{verify}\ ([x \to v] \bullet i)\ (\mathsf{bind\_s}\ (\mathsf{read\_s}\ A\ x)\ s_2)^{\bullet}\ r
\end{array}$$

$$\begin{array}{l}
\mathsf{bnd\_write}\ :\ (\mathsf{def}\ ([x \to v] \bullet i) \longrightarrow \mathsf{verify}\ ([x \to v] \bullet i)\ (s_2\ ())^{\bullet}\ r) \longrightarrow\\
\qquad\qquad \mathsf{def}\ ([x \to w] \bullet i) \longrightarrow\\
\qquad\qquad\quad \mathsf{verify}\ ([x \to w] \bullet i)\ (\mathsf{bind\_s}\ (\mathsf{write\_s}\ x\ v)\ s_2)^{\bullet}\ r
\end{array}$$

$$\begin{array}{l}
\mathsf{bnd\_alloc}\ :\ (\forall x{:}\mathsf{loc}.\, \mathsf{def}\ ([x \to v] \bullet i) \longrightarrow\\
\qquad\qquad\qquad \mathsf{verify}\ ([x \to v] \bullet i)\ (s_2\ x)^{\bullet}\ r) \longrightarrow\\
\qquad\qquad \mathsf{def}\ i \longrightarrow \mathsf{verify}\ i\ (\mathsf{bind\_s}\ (\mathsf{alloc\_s}\ v)\ s_2)^{\bullet}\ r
\end{array}$$

$$\begin{array}{l}
\mathsf{bnd\_allocb}\ :\ (\forall x{:}\mathsf{loc}.\, \mathsf{def}\ (\mathsf{iter}\ n\ x\ v \bullet i) \longrightarrow\\
\qquad\qquad\qquad \mathsf{verify}\ (\mathsf{iter}\ n\ x\ v \bullet i)\ (s_2\ x)^{\bullet}\ r) \longrightarrow\\
\qquad\qquad \mathsf{def}\ i \longrightarrow \mathsf{verify}\ i\ (\mathsf{bind\_s}\ (\mathsf{allocb\_s}\ v\ n)\ s_2)^{\bullet}\ r
\end{array}$$

$$\begin{array}{l}
\mathsf{bnd\_dealloc}\ :\ (\mathsf{def}\ i \longrightarrow \mathsf{verify}\ i\ (s_2\ ())^{\bullet}\ r) \longrightarrow\\
\qquad\qquad \mathsf{def}\ ([x \to v] \bullet i) \longrightarrow\\
\qquad\qquad\quad \mathsf{verify}\ ([x \to v] \bullet i)\ (\mathsf{bind\_s}\ (\mathsf{dealloc\_s}\ x)\ s_2)^{\bullet}\ r
\end{array}$$

$$\begin{array}{l}
\mathsf{bnd\_bnd}\ :\ \mathsf{verify}\ i\ (\mathsf{bind\_s}\ t_1\ (\mathsf{fun}\ x.\, \mathsf{bind\_s}\ (t_2\ x)\ s_2))^{\bullet}\ r \longrightarrow\\
\qquad\qquad \mathsf{verify}\ i\ (\mathsf{bind\_s}\ (\mathsf{bind\_s}\ t_1\ t_2)\ s_2)^{\bullet}\ r
\end{array}$$

The above lemmas apply only when verifying compound commands (i.e., command starting with a bind). We need another set of lemmas for atomic commands. For example:

$$\mathsf{val\_ret}\ :\ r\ v\ i \longrightarrow \mathsf{def}\ i \longrightarrow \mathsf{verify}\ i\ (\mathsf{return\_s}\ v)^{\bullet}\ r,$$

and similarly for the other commands.

Verification of any given command in HTT then works basically by applying one of the lemmas above, or one of the structural rules, as may be required, updating the heap accordingly, and stripping off the commands from the goal one at a time. This process interacts very well with the partiality of heap union from Section 2, as we have instrumented the lemmas to chain the def predicates from one application to the next, changing the predicates to reflect the changes to the heaps. During verification, it may be necessary to reorder the involved heap unions and bring the subheap required by the current command to the top of the expression, or else the corresponding lemma will not apply. The reordering, however, is quite inexpensive, using the unconditional rewrite rules from Section 2. Once the commands are exhausted, we have to show that the heap obtained at the end satisfies the desired postcondition. At this point, we usually require some mathematical knowledge that is specific to the problem at hand, and has to be developed separately.

Example 5. We now proceed to discharge the proof obligation for insert. We first break up the obligation into $p{:}\mathsf{loc}$, $x{:}T$, $l{:}\mathsf{list}\ T$, hypothesis $H{:}\mathsf{lseq}\ p\ l\ i$, and the goal

$$\begin{array}{l}
\mathsf{verify}\ i\ (\mathsf{bind\_s}\ (\mathsf{allocb\_s}\ p\ 2) \\
\qquad\qquad (\mathsf{fun}\ y.\ \mathsf{bind\_s}\ (\mathsf{write\_s}\ y\ x) \\
\qquad\qquad\quad \mathsf{fun}\ \_.\ \mathsf{return\_s}\ y))^{\bullet} \\
\qquad\quad (\mathsf{fun}\ y\ m.\ \forall l.\ \mathsf{lseq}\ p\ l\ i \to \mathsf{lseq}\ y\ (x{::}l)\ m).
\end{array}$$

We apply the lemma $\mathsf{all\_imp}_1$ to remove the quantifier over l and the antecedent $\mathsf{lseq}\ p\ l\ i$ from the postcondition, to obtain

$$\begin{array}{l}
\mathsf{verify}\ i\ (\mathsf{bind\_s}\ (\mathsf{allocb\_s}\ p\ 2) \\
\qquad\qquad (\mathsf{fun}\ y.\ \mathsf{bind\_s}\ (\mathsf{write\_s}\ y\ x) \\
\qquad\qquad\quad \mathsf{fun}\ \_.\ \mathsf{return\_s}\ y))^{\bullet} \\
\qquad\quad (\mathsf{fun}\ y\ m.\ \mathsf{lseq}\ y\ (x{::}l)\ m).
\end{array}$$

The hypothesis of $\mathsf{all\_imp}_1$ is easily satisfied, using H and the lemma $\mathsf{lseq\_func}$ proved in Example 2. Next, by hypothesis H and helper lemma $\mathsf{lseq\_def}{:}\mathsf{lseq}\ p\ l\ i \to \mathsf{def}\ i$, we obtain $\mathsf{def}\ i$. Using this and $\mathsf{bnd\_allocb}$, we reduce the goal to

$$\begin{array}{l}
\mathsf{def}\ (([y \to p] \bullet [y{+}1 \to p] \bullet \mathsf{empty}) \bullet i) \to \\
\mathsf{verify}\ (([y \to p] \bullet [y{+}1 \to p] \bullet \mathsf{empty}) \bullet i) \\
\qquad (\mathsf{bind\_s}\ (\mathsf{write\_s}\ y\ x)\ (\mathsf{fun}\ \_.\ \mathsf{return\_s}\ y))^{\bullet} \\
\qquad (\mathsf{fun}\ y\ m.\ \mathsf{lseq}\ y\ (x{::}l)\ m)
\end{array}$$

where y is a fresh variable. We next want to bring the singleton heap $[y \to p]$ to the top of the union, so we remove empty, and apply the associativity law. After that, we can apply $\mathsf{bnd\_write}$ to obtain

$$\begin{array}{l}
\mathsf{verify}\ ([y \to x] \bullet [y{+}1 \to p] \bullet i) \\
\qquad (\mathsf{return\_s}\ y)^{\bullet} \\
\qquad (\mathsf{fun}\ y\ m.\ \mathsf{lseq}\ y\ (x{::}l)\ m)
\end{array}$$

under hypothesis $D : \mathsf{def}\ ([y \to x] \bullet [y{+}1 \to p] \bullet i)$. By $\mathsf{val\_ret}$, it suffices to show $\mathsf{lseq}\ y\ (x{::}l)\ ([y \to x] \bullet [y{+}1 \to p] \bullet i)$, which by definition of lseq equals

$$\begin{array}{l}
\exists q\ h'.\ [y \to x] \bullet [y{+}1 \to p] \bullet i = [y \to x] \bullet [y{+}1 \to q] \bullet h' \\
\qquad \wedge\ \mathsf{lseq}\ q\ l\ h' \wedge \mathsf{def}\ ([y \to x] \bullet [y{+}1 \to p] \bullet i).
\end{array}$$

One can now instantiate q and h' with p and i, respectively, or alternatively, introduce unification variables, and let the system instantiate q and h' from the heap equation in the goal. The argument can be summarized by the following Ssreflect proof.

```
apply: (all_imp1 l) ⇒ [?|]; first by apply: lseq_func.
apply: bnd_allocb (lseq_def H) ⇒ y; rewrite unh0 unA.
apply: bnd_write ⇒ D; apply: val_ret ⇒ //.
by do !econstructor.
```

5. Fast congruence closure

To put our proof technique to the test, we implemented and verified in HTT one of the fastest practical algorithms for computing the congruence closure of a set of equations, designed by Nieuwenhuis and Oliveras [23], and used in the Barcelogic SAT Solver whose efficiency has been confirmed in various SAT-solving competitions [3]. The algorithm simultaneously uses several stateful data structures such as arrays, hash tables and linked lists, which all interact in very subtle ways, governed by highly non-trivial invariants.

The algorithm starts with a set of equations between expressions, all of which contain symbols drawn from a finite set symb. Each expression is either a constant symbol, or an application, i.e. our type of expressions is

$$\mathsf{exp} = \mathsf{const\ of\ symb} \mid \mathsf{app\ of\ exp} \times \mathsf{exp}.$$

Of course, we will use the customary shorthand and, for example, abbreviate $\mathsf{const}\ c = \mathsf{app}\ (\mathsf{const}\ c_1)\ (\mathsf{const}\ c_2)$ as $c = c_1\ c_2$.

Definition 6. A binary relation R on expressions is monotone iff $\forall f_1\ f_2\ e_1\ e_2.\ (f_1, f_2) \in R \to (e_1, e_2) \in R \to (f_1\ e_1, f_2\ e_2) \in R$. R is a congruence iff it is monotone and an equivalence. The congruence closure of R is the smallest congruence containing R, and is defined as $\mathsf{closure}\ R = \bigcap\{C | C \text{ is congruence and } R \subseteq C\}$.

The algorithm internally maintains a data structure that represents the congruence closure of a set of equations. Its interface consists of two methods: (1) $\mathsf{merge}\ (t_1 = t_2)$, extends the currently represented congruence with a new equation $t_1 = t_2$, that is, it combines the congruence classes of t_1 and t_2, and (2) $\mathsf{check}\ t_1\ t_2$ determines whether the pair (t_1, t_2) belongs to the represented congruence. Additionally, the algorithm assumes that the equations passed to merge are in *flattened form* in the sense that they are either *simple* equations of the form $c_1 = c_2$ or *compound* equations of the form $c = c_1\ c_2$, where c, c_1, c_2 are *symbols*, rather than general expressions. We will need a data type of equations to capture this distinction, which we define as

$$\mathsf{Eq} = \mathsf{simp\ of\ symb} \times \mathsf{symb} \mid \mathsf{comp\ of\ symb} \times \mathsf{symb} \times \mathsf{symb}.$$

Any system of equations can be brought into a flattened form. For example, the non-flat equation $c = c_1\ c_2\ c_3$ can be flattened by introducing a fresh symbol c_4, and then decomposing into two equations: $c = c_4\ c_3$ and $c_4 = c_1\ c_2$. It turns out that in the setting of SAT solvers, it suffices to flatten the expressions from the original SAT formula once and for all, as the intervening computations of congruence closure will not require additional flattening and generation of new symbols [23].

Knowing the number of symbols ahead of time makes it possible to improve the efficiency by storing some of the data into arrays rather than linked structures. For example, the algorithm stores: (1) The array r of *representatives*. For each symbol c, $r[c]$ is the selected representative of the congruence class of c. To reduce clutter, we will abbreviate $r[c]$ simply as c'. (2) The array *clist* of *class* lists: for each representative symbol c, *clist*$[c]$ is (a pointer to) the (singly-linked) list of symbols in the congruence class of c. (3) The array *ulist* of *use* lists: for each representative symbol c, *ulist*$[c]$ is (a pointer to) the (singly-linked) list of compound equations $c_1 = c_2\ c_3$, where $c = c'_1$ or $c = c'_3$ or both. If during the execution c stops being a representative because its congruence class is merged into another, the *use* list of c gives an upper bound on the set of expressions and equations affected by this change. To restore the internal soundness of the data structures, it will suffice to reprocess only the equations in *ulist*$[c]$. (4) The pointer p to the list of *pending* simple equations. If the equation $c_1 = c_2$ is in the pending list, it indicates that the congruence classes of c_1 and c_2 need to be merged in order to restore the internal soundness. When the pending list is empty, the data structures are in a consistent state. (5) The *lookup table htab*, is a hash table storing for each pair of representatives (r_1, r_2) some compound equation $c = c_1\ c_2$ such that $r_1 = c'_1$ and $r_2 = c'_2$. If no such equation exists, the lookup

```
Module Array
  array  : finType → Type → Type
  shape  : array I T → (I → T) → Prop
  read   : Πa:array I T. Πk:I.
              STsep T (fun i. ∃f. shape a f i,
                          fun y i m. ∀f. shape a f i →
                                       y = f k ∧ i = m)
  write  : Πa:array I T. Πk:I. Πx:T.
              STsep unit (fun i. ∃f. shape a f i,
                             fun y i m. ∀f. shape a f i →
                                          shape a f[k ↦ x] m)
Module Hashtab
  kvmap  : eqType → Type → Type
  shape  : kvmap K V → (K → option V) → Prop
  lookup : Πt:kvmap K V. Πk:K.
              STsep (option V) (fun i. ∃f. shape t f i,
                                  fun y i m. ∀f. shape f i →
                                              shape t f m ∧ y = f k)
  insert : Πt:kvmap K V. Πk:K. Πx:V.
              STsep unit (fun i. ∃f. shape t f i,
                             fun y i m. ∀f. shape t f i →
                                          shape t f[k ↦ Some x] m)
```

Figure 2. Relevant parts of array and hash table signatures.

table contains no entries for (r_1, r_2). This table is the main data structure from which one can read off the represented congruence. For example, to check if the pair $(c, c_1\ c_2)$ is in the congruence, it suffices to search the lookup table for the key (c_1', c_2'). If the lookup returns some equation $d = d_1\ d_2$, then d' is the representative symbol for $c_1\ c_2$, and $(c, c_1\ c_2)$ is in the congruence iff $d' = c'$.

Since we require arrays and hash tables, we implemented libraries for both, but here only summarize in Figure 2 the signatures of the type constructors, predicates and methods that we use in this section. The actual libraries are much more general, and are available on our web site. Each module exports a type representing the data structure. Both type array $I\ T$ and kvmap $K\ V$ are implemented as loc, but the signature hides that fact. Arrays expect the index type I to be finite, and hash tables expect the type of keys K to be eqType, that is, it supports a decidable equality function $== : K \to K \to \mathsf{bool}$. The later is also a property required of finType's. Both modules export an abstract predicate shape, which relates the layout of each data structure with a mathematical entity that the structure represents. In the case of arrays, this entity is a function of type $I \to T$, and in the case of hash tables, it is a function of type $K \to \mathsf{option}\ V$, reflecting the fact that the hash table need not contain a value for every key. In our libraries, we also capture the fact that the hash table can contain values for only finitely many keys, but for this discussion, the above weaker abstraction suffices. For both arrays and hash tables, we write $f[k \mapsto x]$ to describe a function obtained from f by changing the value at k into x. Now the stateful data structures described above can be declared as the following five variables which are global to the methods of the algorithm: r : array symb symb, $clist, ulist$: array symb loc, $htab$: kvmap (symb × symb) (symb × symb × symb), and p : loc.

Since we are interested in the functional verification of the algorithm, we need to capture the contents of these arrays, hash tables and linked lists as appropriate mathematical values. We do this with the following record type.

$$\begin{aligned}\mathsf{data} = \{&\mathsf{rep} : \mathsf{symb} \to \mathsf{symb};\ \mathsf{class} : \mathsf{symb} \to \mathsf{list\ symb};\\ &\mathsf{use} : \mathsf{symb} \to \mathsf{list}\ (\mathsf{symb} \times \mathsf{symb} \times \mathsf{symb});\\ &\mathsf{lookup} : \mathsf{symb}{\times}\mathsf{symb} \to \mathsf{option}\ (\mathsf{symb}{\times}\mathsf{symb}{\times}\mathsf{symb});\\ &\mathsf{pending} : \mathsf{list}\ (\mathsf{symb} \times \mathsf{symb}))\}\end{aligned}$$

The intention is that, given D:data, the function rep D represents the contents of the array r, and similarly class D, use D, lookup D and pending D capture the contents of *clist*, *ulist*, *htab* and p. The formal correspondence is established by the following predicate.

$$\begin{aligned}&\mathsf{shape'}\ (D : \mathsf{data})\ (h : \mathsf{heap}) : \mathsf{Prop} :=\\ &\quad \exists ct\ ut{:}\mathsf{symb} \to \mathsf{loc}.\ \exists q{:}\mathsf{loc}.\\ &\qquad \mathsf{Array.shape}\ r\ (\mathsf{rep}\ D) *\\ &\qquad \mathsf{Array.shape}\ \mathit{clist}\ ct * \circledast_{c \in \mathsf{symb}} \mathsf{lseq}\ (ct\ c)\ (\mathsf{class}\ D\ c) *\\ &\qquad \mathsf{Array.shape}\ \mathit{ulist}\ ut * \circledast_{c \in \mathsf{symb}} \mathsf{lseq}\ (ut\ c)\ (\mathsf{use}\ D\ c) *\\ &\qquad \mathsf{Hashtab.shape}\ \mathit{htab}\ (\mathsf{lookup}\ D) *\\ &\qquad p \mapsto q * \mathsf{lseq}\ q\ (\mathsf{pending}\ D))\ h\end{aligned}$$

Here we freely use the separation logic $*$ (as defined in Section 2) and its iterated version $\circledast$. In the proofs, we will unfold their definitions in terms of explicit heaps, when needed. The shape$'$ predicate captures the layout of the structures in the heap, but we also need to capture the relationships between these structures.

$$\begin{aligned}&\mathsf{shape}\ (R : \mathsf{exp} \times \mathsf{exp} \to \mathsf{Prop})\ (h : \mathsf{heap}) : \mathsf{Prop} =\\ &\quad \exists D{:}\mathsf{data}.\ \mathsf{shape'}\ D\ h \wedge \mathsf{rep\_idemp}\ D \wedge \mathsf{class\_inv}\ D \wedge\\ &\qquad \mathsf{use\_inv}\ D \wedge \mathsf{lkp\_inv}\ D \wedge \mathsf{use\_lkp\_inv}\ D \wedge\\ &\qquad \mathsf{lkp\_use\_inv}\ D\ \wedge \mathsf{pending}\ D = \mathsf{nil} \wedge \mathsf{CRel}\ D =_r R\end{aligned}$$

In shape, we list that the array r must be idempotent:

$$\mathsf{rep\_idemp}\ D = \forall c.\ \mathsf{rep}\ D\ (\mathsf{rep}\ D\ c) = \mathsf{rep}\ D\ c.$$

The class lists invert the representative array:

$$\mathsf{class\_inv}\ D = \forall x\, c.\ (\mathsf{rep}\ D\ x == c) = (x \in \mathsf{class}\ D\ c).$$

Use lists store only equations with appropriate representatives:

$$\begin{aligned}&\mathsf{use\_inv}\ D = \forall a\, c\, c_1\, c_2.\ a \in \mathsf{reps}\ D \to\\ &\qquad (c, c_1, c_2) \in \mathsf{use}\ D\ a \to \mathsf{rep}\ D\ c_1 = a \vee \mathsf{rep}\ D\ c_2 = a,\end{aligned}$$

where reps D is the list of symbols that are representatives, that is, they appear in the *range* of the function rep D. Next, the hash table stores equations with appropriate representatives:

$$\begin{aligned}&\mathsf{lkp\_inv}\ D = \forall a\, b\, c\, c_1\, c_2.\ a \in \mathsf{reps}\ D \to b \in \mathsf{reps}\ D \to\\ &\quad \mathsf{lookup}\ D\ (a, b) = \mathsf{Some}\ (c, c_1, c_2) \to \mathsf{rep}\ D\ c_1 = a \wedge \mathsf{rep}\ D\ c_2 = b.\end{aligned}$$

For each equation in a use list, there is an appropriate equation in the hash table, and vice versa:

$$\begin{aligned}&\mathsf{use\_lkp\_inv}\ D = \forall a\, c\, c_1\, c_2.\ a \in \mathsf{reps}\ D \to (c, c_1, c_2) \in \mathsf{use}\ D\ a \to\\ &\quad \exists d\, d_1\, d_2.\ \mathsf{lookup}\ D\ (\mathsf{rep}\ D\ c_1, \mathsf{rep}\ D\ c_2) = \mathsf{Some}\ (d, d_1, d_2) \wedge\\ &\quad \mathsf{rep}\ D\ c_1 = \mathsf{rep}\ D\ d_1 \wedge \mathsf{rep}\ D\ c_2 = \mathsf{rep}\ D\ d_2 \wedge \mathsf{rep}\ D\ c = \mathsf{rep}\ D\ d\end{aligned}$$

$$\begin{aligned}&\mathsf{lkp\_use\_inv}\ D = \forall a\, b\, d\, d_1\, d_2.\ a \in \mathsf{reps}\ D \to b \in \mathsf{reps}\ D \to\\ &\quad \mathsf{lookup}\ D\ (a, b) = \mathsf{Some}\ (d, d_1, d_2) \to\\ &\qquad (\exists c\, c_1\, c_2.\ (c, c_1, c_2) \in \mathsf{use}\ D\ a \wedge\\ &\qquad\quad \mathsf{rep}\ D\ c_1 = a \wedge \mathsf{rep}\ D\ c_2 = b \wedge \mathsf{rep}\ D\ c = \mathsf{rep}\ D\ d) \wedge\\ &\qquad (\exists c\, c_1\, c_2.\ (c, c_1, c_2) \in \mathsf{use}\ D\ b \wedge\\ &\qquad\quad \mathsf{rep}\ D\ c_1 = a \wedge \mathsf{rep}\ D\ c_2 = b \wedge \mathsf{rep}\ D\ c = \mathsf{rep}\ D\ d).\end{aligned}$$

The shape predicate will be used for the specification of the main methods of the algorithm. Hence it also requires that pending D = nil, i.e., the structures are in a consistent state, and CRel $D =_r R$, i.e., the relation R *is* the congruence represented by the structures. Here, CRel D is defined as the congruence closure of all the equations in lookup D, pending D as well as the equations c = rep $D\ c$, for all c. The operator $=_r$ is the equality on relations: $R_1 {=_r} R_2 = \forall t.\ R_1\ t \leftrightarrow R_2\ t$. On the other hand, shape$'$ will be used to specify the helper functions, where some of the above properties may be temporarily invalidated.

The main functions of the algorithm are now implemented as HTT code in Figure 3. The type of merge quite directly states that merge starts with the internal state representing some congruence relation R, and changes the internal state to represent the congruence closure of the extension of R with the argument equation eq. We emphasize that the code does not contain any other kind of annotations, such as for example framing conditions, and in general looks very close to what one would write in an ordinary imperative language. If merge is passed a simple equation $a = b$, it places the pair (a, b) onto the head of the pending list, and invokes the helper function hpropagate, defined in Figure 5, to merge the congruence

```
1.  merge ($eq$ : Eq) :
2.      STsep unit (fun $i$. $\exists R$. shape $p\ R\ i$,
3.                  fun $y\ i\ m$. $\forall R$. shape $p\ R\ i \to$
4.                  shape $p$ (closure ($R \cup$ rel_of $eq$)) $m$) =
5.    match $eq$ with
6.      simp $a\ b \Rightarrow$
7.        do ($q \leftarrow !p$;
8.            $x \leftarrow$ insert $q$ $(a, b)$;
9.            $p := x$;
10.           hpropagate)
11.   | comp $c\ c_1\ c_2 \Rightarrow$
12.       do ($c_1' \leftarrow$ Array.read $r\ c_1$;
13.           $c_2' \leftarrow$ Array.read $r\ c_2$;
14.           $v \leftarrow$ Hashtab.lookup $htab$ $(c_1', c_2')$;
15.           match $v$ with
16.             None $\Rightarrow$
17.               Hashtab.insert $htab$ $(c_1', c_2')$ $(c, c_1, c_2)$;
18.               $u_1 \leftarrow$ Array.read $ulist\ c_1'$;
19.               $x \leftarrow$ insert $u_1$ $(c, c_1, c_2)$;
20.               Array.write $ulist\ c_1'\ x$;
21.               $u_2 \leftarrow$ Array.read $ulist\ c_2'$;
22.               $x \leftarrow$ insert $u_2$ $(c, c_1, c_2)$;
23.               Array.write $ulist\ c_2'\ x$
24.           | Some $(b, b_1, b_2) \Rightarrow$
25.               $q \leftarrow !p$;
26.               $x \leftarrow$ insert $q$ $(c, b)$;
27.               $p := x$;
28.               hpropagate
29.           end)
30.   end

31. check ($t_1\ t_2$ : exp) :
32.     STsep bool (fun $i$. $\exists R$. shape $p\ R\ i$,
33.                 fun $y\ i\ m$. $\forall R$. shape $p\ R\ i \to$ shape $p\ R\ m\ \wedge$
34.                                  $y =$ true $\leftrightarrow R\ (t_1, t_2)$) =
35.   do ($u_1 \leftarrow$ hnorm $t_1$;
36.       $u_2 \leftarrow$ hnorm $t_2$;
37.       return ($u_1 == u_2$))
```

where

```
rel_of ($eq$ : Eq) : exp $\times$ exp $\to$ Prop :=
  match $eq$ with
    simp $a\ b \Rightarrow$ fun $t$. $t.1 =$ const $a \wedge t.2 =$ const $b$
  | comp $c\ c_1\ c_2 \Rightarrow$ fun $t$. $t.1 =$ const $c\ \wedge$
                                $t.2 =$ app (const $c_1$) (const $c_2$)
  end
```

Figure 3. The main functions of the fast congruence closure algorithm, and their specifications.

```
38. hnorm ($t$ : exp) =
39.   fix (fun hnorm ($t$:exp).
40.     do (match $t$ with
41.           const $a \Rightarrow$
42.             $a' \leftarrow$ Array.read $r\ a$;
43.             return (const $a'$)
44.         | app $t_1\ t_2 \Rightarrow$
45.             $u_1 \leftarrow$ hnorm $t_1$;
46.             $u_2 \leftarrow$ hnorm $t_2$;
47.             match $u_1, u_2$ with
48.               const $w_1$, const $w_2 \Rightarrow$
49.                 $v \leftarrow$ Hashtab.lookup $htab$ $(w_1, w_2)$;
50.                 match $v$ with
51.                   None $\Rightarrow$ return (app $u_1\ u_2$)
52.                 | Some $(b, \_, \_) \Rightarrow$
53.                     $b' \leftarrow$ Array.read $r\ b$;
54.                     return (const $b'$)
55.                 end
56.             | _, _ $\Rightarrow$ return (app $u_1\ u_2$)
57.             end
58.         end)) $t$
```

Figure 4. Helper function for normalizing expressions.

classes of a and b (lines 7–10). If merge is passed a compound equation $c = c_1\ c_2$, then the lookup table is queried for an equation v of the form $b = b_1\ b_2$, where b_i and c_i have the same representatives (lines 12–14). If such an equation exists, then to extend R with eq, it suffices simply to join the congruence classes of b and c. This is accomplished by putting the pair (b, c) on the top of the pending list, and again invoking hpropagate (lines 25–28). If an equation v does not exist, then it suffices to insert the equation $c = c_1\ c_2$ directly into the lookup table for future queries (line 21), and add the equation to the use lists of c_1' and c_2' (lines 18–23).

The type of check declares that the return boolean value y shows whether the pair (t_1, t_2) is in the congruence relation R represented by the internal state. check first "normalizes" t_1 and t_2; that is, it expresses t_1 and t_2 in terms of representatives, using the helper function hnorm defined in Figure 4. Then the obtained normal forms are compared for syntactic equality (lines 35–37).

Next we have to implement and verify the helper functions. There will be four of them: hpropagate and hnorm are directly used by the main functions, and hjoin_class (Figure 6) and hjoin_use (Figure 7), are called from within hpropagate. In the verification of the helper functions we adopt the following strategy. We first implement the purely-functional variants propagate, norm, join_class and join_use, which is possible since the logic of Coq already includes pure lambda calculus with terminating recursion, and all of the helper functions are terminating loops. The pure variants will operate on the values of the data record, rather than on the pointers themselves. Of course, the pure variants do not exhibit the desired run-time complexity and efficiency, so we only use them for specification and reasoning. In particular, as a first phase of verification, we prove that each helper method exhibits the same behavior on the underlying stateful structures as that described by its pure variant. The first phase takes care of all the reasoning about pointers, aliasing and heap disjointness. Then in the second phase, we show that the pure variants combine to correctly compute congruence closure, but our task will be simplified by not having to worry about pointers anymore.

In Figures 4-7, we present the helper functions, but omit the definitions of the pure variants, as these – we hope – can easily be reconstructed from our discussion of the code. To reduce clutter, we also omit the types and the various loop invariants, since at this first phase these are not particularly involved: they all basically state that the helper function and its pure variant correspond to each other. For example, the types of hnorm and hpropagate are

$$\begin{aligned}\mathsf{normT} = \Pi t{:}\mathsf{exp}.\ \mathsf{STsep\ exp}\ (&\mathsf{fun}\ i.\ \exists D.\ \mathsf{shape'}\ p\ D\ i,\\ &\mathsf{fun}\ y\ i\ m.\ \forall D.\ \mathsf{shape'}\ p\ D\ i \to\\ &\quad \mathsf{shape'}\ p\ D\ m \wedge y = \mathsf{norm}\ D\ t)\end{aligned}$$

$$\begin{aligned}\mathsf{propagateT} = \mathsf{STsep\ unit}\ (&\mathsf{fun}\ i.\ \exists D.\ \mathsf{shape'}\ p\ D\ i,\\ &\mathsf{fun}\ y\ i\ m.\ \forall D.\ \mathsf{shape'}\ p\ D\ i \to\\ &\quad \mathsf{shape'}\ p\ (\mathsf{propagate}\ D)\ m)\end{aligned}$$

which show that the result of hnorm is specified by norm, and the behavior of hpropagate is specified by propagate.

We start our description with the function hnorm for computing normal forms of expressions, given in Figure 4. If the expression t is a constant symbol a, then the normal form of t is the representative a', as read from the array of representatives (lines 42-43). Otherwise, t is an expression of the form $t_1\ t_2$. To compute its normal form, we recursively compute the normal forms u_1 and u_2 of t_1 and t_2, respectively (lines 45-46). In case u_1 and u_2 are themselves constant symbols w_1 and w_2, then the lookup table may contain an equation of the form $b = w_1\ w_2$ which would imply that the normal form should be b' (lines 53-54). Otherwise, we return the application $u_1\ u_2$ as the result (lines 51 and 56).

```
59.  hpropagate =
60.    fix (fun loop (x:unit).
61.      do (q ← !p;
62.          if q == null then return ( )
63.          else
64.            eq ← !q;
65.            next ← !(q + 1);
66.            p := next;
67.            dealloc q;
68.            dealloc (q + 1);
69.            a' ← Array.read r (eq.1);
70.            b' ← Array.read r (eq.2);
71.            if a' == b' then loop ( )
72.            else
73.              hjoin_class a' b';
74.              hjoin_use a' b';
75.              loop ( ))) ( )
```

Figure 5. Helper function for propagating the pending equations.

```
76.  hjoin_class (a' b' : symb) =
77.    fix (fun loop (x : unit).
78.      do (ua ← Array.read clist a';
79.          ub ← Array.read clist b';
80.          if ua == null then return( )
81.          else
82.            s ← !ua;
83.            next ← !(ua + 1);
84.            ua + 1 := ub;
85.            Array.write clist b' ua;
86.            Array.write clist a' next;
87.            Array.write r s b';
88.            loop ( ))) ( )
```

Figure 6. Helper function for merging the class lists of a' and b'.

```
89.  hjoin_use (a' b' : symb) =
90.    fix (fun loop (x:unit).
91.      do (ua ← Array.read ulist a';
92.          if ua == null then return ( )
93.          else
94.            eqc ← !ua;
95.            next ← !(ua + 1);
96.            Array.write ulist a' next;
97.            c2' ← Array.read r eqc.2
98.            c3' ← Array.read r eqc.3
99.            v ← Hashtab.lookup htab (c2', c3');
100.           match v with
101.             None ⇒
102.               Hashtab.insert htab (c2', c3') eqc;
103.               ub ← Array.read ulist b';
104.               ua + 1 := ub;
105.               Array.write ulist b' ua;
106.               loop ( )
107.           | Some eqd ⇒
108.               dealloc ua;
109.               dealloc (ua + 1);
110.               p' ← !p;
111.               q ← insert p' (eqc.1, eqd.1);
112.               p := q;
113.               loop ( )
114.           end)) ( ))
```

Figure 7. Helper function for adjusting the use lists and the lookup table, after the class lists of of a' and b' have been merged.

The function hpropagate from Figure 5 is the main loop of merge. Its role is to "empty" the list of pending simple equations, by merging these equation into the other structures. Each pending equation is represented as a pair of symbols $eq = (a, b)$, denoting that the congruence classes of a and b should be merged. hpropagate reads off the equations from the pending list one-by-one (lines 61–68), computes the representatives a' and b' of the first and second elements of eq, respectively (lines 69-70). If a' and b' are equal, then the equation is redundant. Otherwise, hpropagate calls helper functions hjoin_class and hjoin_use to merge the classes of a' and b' and adjust the various pointers and array fields accordingly (lines 71-75).

The function hjoin_class takes two distinct symbols a' and b' and modifies the state of the algorithm so that the congruence class of a' is appended onto the congruence class of b'. This involves obtaining the pointers to the class list of a' and b' (lines 78-79), then iterating to remove the head symbols s from the class list for a', push s onto the class list of b' (lines 82-86), and then change the representative of s to b' (line 87). A call to hjoin_class joins the immediate data representing the congruence classes of a' and b', but a bit more work has to be done. For example, if the lookup table stores equations of the form $a'\ b = c$ and $b'\ b = d$, then merging a' and b' must be followed by a merge of c and d, in order to restore internal consistency. This is the job of hjoin_use.

A naive implementation of hjoin_use may be simply to traverse the lookup table, merging outstanding classes as they are discovered. A more efficient implementation, shown in Figure 7, exploits the property that it suffices to revisit only the equations stored in the *use* list of a'. If the use list of a' contains the equation $c_1 = c_2\ c_3$, represented as a triple $eqc = (c_1, c_2, c_3)$, we query the lookup table for the key (c_2', c_3') (lines 97–99). If some equation $eqd = (d_1, d_2, d_3)$ is discovered, then $c_2' = d_2'$, $c_3' = d_3'$, by the invariants of the algorithm, but there is no guarantee that c_1 and d_1 are congruent. Thus, we schedule the pair (c_1, d_1) for merging, by placing it onto the pending list (lines 110–112). If the query returns no equations, then we simply insert the equation eqc into the lookup table (line 102). We also move eqc onto the *use* list of b', to be considered in the future, when and if b' is equated to some other symbol (lines 103–105). Either way, eqc has to be removed from the use list of a' (lines 96 and 108–109).

The first phase of verification now closely follows the approach outlined in Section 4, of applying the various structural lemmas and reordering heap unions so as to indicate the subheap that the current command modifies. For all the six methods in this section, it took 276 lines of proof to complete. One minor hurdle was defining the iterated operator $\circledast$ from the shape$'$ predicate. It is best to iterate $\circledast$ over finite *sets*, rather than lists, which was our first attempt. If s is a set of symbols, one can show

$$x \in s \to \mathop{\circledast}_{i \in s} P\ i =_r P\ x * \mathop{\circledast}_{i \in s \setminus \{x\}} P\ i.$$

We used this lemma to expose the heaps storing the class and use lists of concrete symbols. If s were a list, the corresponding lemma requires a spurious condition that s contains x only once. In our development, we were able to reuse Ssreflect's extensive library of finite sets over types with decidable equality.

The second verification phase mainly involves showing that the various properties listed in the shape predicate hold after the execution of the pure variants of the helper functions. For example, one of the easier properties was that the predicate class_inv is preserved between the calls to the helper functions in hpropagate (lines 73-74), and after the call to hpropagate in merge (lines 10 and 28). It is established by the following lemmas.

1. $a' \neq b' \to$ class_inv $D \to$ class_inv (join_class $D\ a'\ b'$)
2. $a' \neq b' \to$ class_inv $D \to$ class_inv (join_use $D\ a'\ b'$)
3. class_inv $D \to$ class_inv (propagate D)

Most of the other predicates from the definition of shape were much more difficult to establish, primarily because they are actually invalidated at various point of the execution, but are then re-established at the end. Thus, we needed to generalize these predicates to properly capture how the code works at all stages, and then show that at the end of merge, the more general versions imply the original definitions.

This was, of course, the most difficult part of the whole development, as the dependencies between the congruence data structures are extremely subtle. The generalizations ended up being very involved, and took about 120 lines of Coq definitions, just to state. For example, it turns out that in cases when the pending list is not empty, the appropriate generalization of the use_lkp_inv property which relates the use lists with the lookup table is:

$$\begin{array}{l}\mathsf{use\_lkp\_inv0}\ D = \forall a\, c\, c_1\, c_2.\ a \in \mathsf{reps}\ D \to (c, c_1, c_2) \in \mathsf{use}\ D\ a \to \\ \quad \exists d\, d_1\, d_2.\, \mathsf{lookup}\ D\ (\mathsf{rep}\ D\ c_1, \mathsf{rep}\ D\ c_2) = \mathsf{Some}\ (d, d_1, d_2) \wedge \\ \quad \mathsf{rep}\ D\ c_1 = \mathsf{rep}\ D\ d_1 \wedge \mathsf{rep}\ D\ c_2 = \mathsf{rep}\ D\ d_2 \wedge \mathsf{similar}\ D\ c\ d\end{array}$$

Here, similar $D\ c\ d$ holds if the symbols c and d are in the congruence relation generated by the equations $x = \mathsf{rep}\ D\ x$ for all x, *as well as the equations in the pending list*. The property of similarity justifies the algorithm to save time when processing the use lists, and sometimes omit equations as redundant, on the grounds that their involved symbols will eventually be equated once the pending list is emptied.

After an equation $a' = b'$ is removed from the pending list in hpropagate, and before a call to hjoin_class $a'\ b'$ (line 73), another property use_lkp_inv1 D is required. This one replaces similar $D\ c\ d$ in the definition of use_lkp_inv0 with similar1 $D\ c\ d$ which makes it possible that c and d are related via an equation $a' = b'$ as well. Yet another property use_lkp_inv2 is required to describe the relation between the use lists and the lookup table after a call to hjoin_class, and during the call to hjoin_use, etc. Similar generalizations have to be made to lkp_use_inv as well, and then one has to prove that these properties indeed hold in the various stages of the program. In these proofs, we may need to rely on some of the other invariants. For example, we have a lemma

$$\begin{array}{l}\mathsf{join\_classP}\ (D : \mathsf{data})\ (a' b' : \mathsf{symb}) : \\ \quad a' \in \mathsf{reps}\ D \to b' \in \mathsf{reps}\ D \to a' \neq b' \to \\ \quad \mathsf{rep\_idemp}\ D \to \mathsf{use\_inv}\ D \to \mathsf{lkp\_inv}\ D \to \\ \quad \mathsf{use\_lkp\_inv1}\ D\ a'\ b' \to \mathsf{lkp\_use\_inv1}\ D\ a'\ b' \to \\ \quad \mathsf{use\_lkp\_inv2}\ (\mathsf{join\_class}\ D\ a'\ b')\ a'\ b' \wedge \\ \quad \mathsf{lkp\_use\_inv2}\ (\mathsf{join\_class}\ D\ a'\ b')\ a'\ b',\end{array}$$

which states that the above properties hold after a call to hjoin_class, assuming that appropriate properties held before the call. Then similar lemmas have to be proved for join_use and propagate in all combinations with the properties from the definition of shape.

Altogether, these proofs took 645 lines of proof, reflecting the subtlety of the invariants of the fast congruence closure algorithm, which is required for its practical efficiency. Of course, before we were able to carry out these proofs, we first had to develop a number of facts about congruences and closures, define the data types, define the pure variants of the helper function and prove them terminating, and define the generalized invariants themselves. This background development took another 632 lines.

6. Using Coq and Ssreflect

In our developments, we have kept the proofs fully explicit, always naming hypotheses as they are introduced, destructed, or modified. We have found this explicitness to be quite helpful when refactoring larger developments, such as our verification of fast congruence closure. When proofs are explicit in this sense, making changes to the definitions and lemmas usually causes the proofs to break exactly at the point where the error introduced by the changes actually is, rather than somewhere at random later in the proof.

Furthermore, we have used only a few simple custom-made tactics that we describe below, and have otherwise relied on only the standard primitives of Coq and Ssreflect for introduction and destruction of hypotheses, lemma application and rewriting. All of these have direct analogues in the natural deduction rules for Coq. Despite the full explicitness and general absence of automation, our proofs are – perhaps somewhat surprisingly – still quite short and comparable in size with other approaches, such as Ynot and Jahob, which use very aggressive automation (we discuss the relation to Ynot and Jahob in Section 7). Even in the case of congruence closure, whose full proof was quite large, the phase of the proof related to pointers and aliasing was proportional in size to the verified program. We attribute these properties not only to our new techniques, but also to the very prudent design of the Ssreflect language and libraries.

The tactics that we have used are the following.

1. heap_cancel takes a *hypothesis* in the form of an equation between heaps, such as for example $[x \to v_1] \bullet h_1 = [x \to v_2] \bullet h_2$, and derives consequences from it, like $v_1 = v_2$ and $h_1 = h_2$, which it prepends onto the goal of the sequent. In example 2, we have used a simple iteration of the cancellation lemma for this purpose, but heap_cancel is more general, as it does not rely on the order of heaps in the union.
2. heap_congr is dual to heap_cancel. It takes a *goal* in the form of a heap equation, and produces subgoals needed to discharge it. In the above example, it would produce exactly the subgoals $v_1 = v_2$ and $h_1 = h_2$.
3. defcheck takes an implication of the form def $h_1 \to$ def h_2, where h_1 and h_2 are unions of heaps, and tries to discharge it by matching all the locations in the heaps in h_2 to locations in the heaps in h_1, irrespectively of the order in which they appear. Thus, it effectively checks if the domain of the union h_2 is a subdomain of h_1.
4. hauto combines the generation of unification variables (the econstructor primitive of Coq), with heap_cancel and defcheck.
5. heval pattern-matches against the goal in the form of a verify predicate, to determine the first command appearing in it, so that it can choose which bnd_command or val_command lemma from Section 4 to apply.

All of these tactics are conceptually simple, and only modify goals of sequents, but not the hypotheses; thus they do not break the explicit nature of our proofs. However, because Coq's tactic language is interpreted and untyped, we have still found them to be somewhat slow in practice, and quite difficult to debug and maintain. In future work, we plan to remove even these tactics, and replace them with equivalent lemmas and rewrite rules, which could possibly be built using ideas based on reflection.

7. Related work

HTT and Ynot Just like the current paper, the original implementations of HTT and Ynot [20, 21] used Hoare triples with binary postconditions. However, those papers did not recognize the connection between binary postconditions and structural rules – which we proposed here. In particular, they used a different definition of the verify predicate from the one we used in Section 3, and which in our current notation can be presented roughly as follows.

$$\begin{array}{l}\mathsf{verify}\ i\ s\ q = \exists h\, i_1.\, i = i_1 \bullet h \wedge \mathsf{def}\ i \wedge \mathsf{pre}\ s\ i_1 \wedge \\ \qquad\qquad \forall y\, m\, m_1.\, m = m_1 \bullet h \to \mathsf{def}\ m \to \\ \qquad\qquad\quad \mathsf{post}\ s\ y\ i_1\ m_1 \to q\ y\ i\ m\end{array}$$

This definition existentially abstracts over the invariant part h of the heap, and thus directly "bakes in" the frame rule into the semantics of Hoare triples. In this sense, it is closely related to the recent semantic models of separation logic by Birkedal et al. [7]. However, abstracting h on the outside causes this definition to not support the rules of conjunction (binary or quantified), without additional requirements such as, for example, that $\mathsf{pre}\ s$ determines a unique subheap of i (i.e., that $\mathsf{pre}\ s$ is a *precise* predicate, in separation logic terminology). Our definition from Section 3 does not impose such additional requirements.

Furthermore, the implementation in [21] relied on a naive definition of heaps from Section 2, which caused an explosion in proof obligations. This problem was already observed by Krishnaswami et al. [15], who attempted to use the system to verify the "flyweight" OO-design pattern, but could not finish the proof.

This motivated Chlipala et al. [11] to revert to the (*) fragment, unary postconditions and no explicit heap variables, as well as to develop a number of tactics for automating the reasoning in separation logic. This is an appealing idea, as binary postconditions come with a redundancy exhibited in our Example 4, where the type of remove had to repeat the precondition as an antecedent of an implication in the postcondition. With unary postconditions, one could write this type simply as

$$\mathsf{remove}\ p : \mathsf{STsep\ loc}\ (\mathsf{lseq}\ p\ l, \mathsf{fun}\ q.\ \mathsf{lseq}\ q\ (\mathsf{tail}\ l)).$$

The latter, however, opens the question of where and how the variable l should be bound. One cannot use the ordinary dependent function type and write

$$\Pi l{:}\mathsf{list}\ T.\ \mathsf{STsep\ loc}\ (\mathsf{lseq}\ p\ l, \mathsf{fun}\ q.\ \mathsf{lseq}\ q\ (\mathsf{tail}\ l)),$$

because this allows l to be used in commands of the above type, and l is supposed to only be a *logical* variable; that is, it can appear in specifications, but not in the commands. Chlipala et al. propose that logical variables be coerced into *proofs*, and write roughly

$$\begin{array}{l}\Pi l{:}\mathsf{inhabited}(\mathsf{list}\ T).\\ \quad \mathsf{STsep\ loc}\ (\mathsf{let\ pack}\ l = x\ \mathsf{in\ lseq}\ p\ l,\\ \qquad\qquad \mathsf{fun}\ q.\ \mathsf{let\ pack}\ l = x\ \mathsf{in\ lseq}\ q\ (\mathsf{tail}\ l))\end{array}$$

where $\mathsf{inhabited}\ A$ is the *proposition* $\exists x{:}A.\top$, and $\mathsf{pack} : A \to \mathsf{inhabited}\ A$ is the single constructor of proofs of this proposition. Coq's type theory makes it impossible to "unpack" l within an executable program, and Coq's extraction mechanism for remove would, appropriately, not produce a closure which abstracts over l.

This coercion, however, comes with significant logical complexity. Even if l cannot be unpacked in a command e, it does not prevent l from being used in e, albeit packed. Thus, it is not clear that the technique can support structural rules where one needs to test if $l \notin \mathsf{FV}(e)$, such as the rule for existentials. Indeed, the system in [11] does not support this rule (nor any other structural rule beyond frame and consequence), which is a restriction that leads to loss of abstraction. The existential rule is frequently used to push a logical variable into the pre/post-conditions, so that it can be removed later by applying the rule of consequence. Without the existential rule, it seems that logical variables must remain bound in the type, even if they are not needed anymore. Working with the coercions further requires adding an axiom

$$\mathsf{pack\_injective} : \forall T{:}\mathsf{Set}.\ \forall x\, y{:}T.\ \mathsf{pack}\ x = \mathsf{pack}\ y \to x = y,$$

which compares proofs for equality, and is thus unsound in the presence of important features such as proof irrelevance or classical logic.

It may be possible that the recent extension of the calculus of constructions with a variant of intersection types [2], may offer a way out of these logical problems, and allow structural rules to be encoded as typing rules, rather than as logical formulas. But even then, questions remain as to the practicality of such an encoding. For example, Chlipala et al. encode the frame rule as a typing rule; as a result, programs written in their system often have to be explicitly annotated with framing predicates, as well as with instantiations for various ghost variables. In our opinion, this significantly obscures the structure of the programs. Our approach with binary postconditions does not require such annotations, as witnessed by our examples in Section 5. Binary postconditions also allow the user to derive auxiliary structural rules, thus implementing custom verification strategies, while it is not clear that this can be done in the alternative approach.

Moreover, binary postconditions do not lead to proof explosion, as the redundancy that they exhibit can be removed *in proofs* merely by one application of all_imp, or one of the related lemmas. And indeed, on the examples that we have implemented in common with [11], our developments are of comparable size, even if we do not use significant automation by tactics. For example, in the release current at the time of our writing, the verifications of stacks, queues and hash tables in the system of [11] take respectively 86, 199 and 397 lines of code, specifications, lemmas, tactics and proofs, whereas in our system, these numbers are 66, 116 and 160.

Separation logic in type theory Appel [1] defines heaps as finite lists of location-value pairs, just like we do, but does not reflect the disjointness predicate. As a consequence, he observes that "... the nonlinear conjunction of separation logic is not well suited to the assumptions of tactical provers...", and restricts to the (*) fragment. Marty et al. [17] define heaps in a similar way too, but they use a union operator which is not commutative, and thus also treat only the (*) fragment. McCreight [18] defines heaps following the memory model of Leroy et al. [16], which allows him to define a union operator that is commutative and associative, but his operator does not propagate the disjointness information, and hence there is no equivalent of our def predicate, which is crucial for efficient work. Thus, McCreight too admits only the (*) fragment. All of these systems target deeply embedded programs and languages, unlike HTT which uses shallow embedding.

Verification of linked data structures Jahob [27] is another higher-order system in which verification of interesting pointer-based data structures has been performed. Jahob computes the verification conditions for Java programs, and then feeds the conditions to automatic provers for discharging. The programmer has the option of including proof hints with the code, which can be used to guide the automation. In this respect, the proof hints in Jahob are similar to our explicit proofs. In the case of hash tables, Jahob takes 343 lines of proof hints and invariants, which is comparable in size to our proofs. One important difference between HTT and Jahob is that Java, unlike Coq, has not been designed with proofs in mind, and thus lacks the ability to package together programs, properties and proofs, and parametrize libraries with respect to such packages. We have used this in Section 5 to parametrize the implementation of congruence closure with respect to the signatures for arrays and hash tables. This makes it possible for us to freely plug in any verified implementation of these signatures, without changing the code or the *proofs* of congruence closure. We have not found a discussion or a theorem in [27] of whether similar substitutability is possible in Jahob as well.

Higher-order separation logic Krishnaswami et al. [15] has recently developed a higher-order separation logic for programs written in the core fragment of an ML-like language, and applied it to a verification of several object-oriented patterns. One difference from HTT is that the language in [15] is simply typed, and thus does not support first-class structures and functors that come for free with the dependent types of Coq, and are important for programming and proving in-the-large. Birkedal et al. [7] consider a higher-order separation logic and its interaction with higher-order frame rules and parametricity. In the current paper, we have not considered these issues, but believe that it is an important future work to build models for HTT that reconcile these features with dependent types.

8. Conclusions

The most common approach to program verification in separation or other logics is to investigate how to automate the discharging of the proof obligations in order to reduce the burden on the human verifier. Automation works very well when the properties of interest are relatively simple, but in the case of full functional verification, it is frequently insufficient. In this paper, we instead investigate how to exploit the structuring primitives of type theory, to prevent the proof obligations from being generated in the first place.

Our first example was a new definition of heaps, which let us work efficiently with ordinary logical connectives, without inducing a blowup in the proof obligations about heap disjointness. The definition involved advanced type theoretic features, such as dependently-typed programming and reflection, but its main point was to ensure that heaps satisfy the algebraic properties of a partial commutative monoid (PCM). PCMs have been considered before in the semantics of separation logic [9], but here we show that if heaps are PCMs, then it becomes quite practical to unfold the definitions of separating connectives such as $*$, and work directly with heap variables and disjoint unions. The latter was necessary for supporting non-separating connectives such as conjunction, implication and universal quantification.

Our second example was embedding and reformulating a separation logic for partial correctness into type theory with the use of binary postconditions. This made it possible to derive custom structural rules that helped in proofs. Moreover, stating the rules in this way essentially depends on our definition of heaps, because it requires a logic that efficiently supports implication and universal quantification.

We have used our approach successfully to verify a number of smaller programs such as modules for arrays, linked lists, stacks, queues and hash tables. In all the cases, we were able to produce correctness proofs of size proportional to the size of the programs. We have shown that the approach scales to larger examples as well, by verifying one of the fastest known congruence closure algorithms, used in the Barcelogic SAT solver.

9. Acknowledgment

We thank Georges Gonthier for introducing us to Ssreflect, and Nick Benton and Martin Hofmann for discussions regarding dependent types.

References

[1] A. W. Appel. Tactics for separation logic. Available at http://www.cs.princeton.edu/˜appel/papers/septacs.pdf, 2006.

[2] B. Barras and B. Bernardo. The implicit calculus of constructions as a programming language with dependent types. In *FoSSaCS'08*, pages 365–379.

[3] C. Barrett, M. Deters, A. Oliveras, and A. Stump. Design and results of the 4th annual satisfiability modulo theories competition (SMT-COMP 2008). To appear.

[4] J. M. L. Bean. *Ribbon Proofs – A Proof System for the Logic of Bunched Implications*. PhD thesis, Queen Mary University of London, 2006.

[5] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. W. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *CAV'07*, pages 178–192.

[6] J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *Formal Methods for Components and Objects*, pages 115–137, 2006.

[7] L. Birkedal and H. Yang. Relational parametricity and separation logic. *Logical Methods in Computer Science*, 4(2:6):1–27, 2008.

[8] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *POPL'09*, pages 289–300.

[9] C. Calcagno, P. W. O'Hearn, and H. Yang. Local action and abstract separation logic. In *LICS'07*, pages 366–368.

[10] B.-Y. E. Chang and X. Rival. Relational inductive shape analysis. In *POPL'08*, pages 247–260.

[11] A. J. Chlipala, J. G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. Effective interactive proofs for higher-order imperative programs. In *ICFP'09*, pages 79–90.

[12] X. Feng, Z. Shao, Y. Dong, and Y. Guo. Certifying low-level programs with hardware interrupts and preemptive threads. In *PLDI'08*, pages 170–182.

[13] D. Galmiche and D. Méry. Semantic labelled tableaux for propositional BI. *Journal of Logic and Computation*, 13(5):707–753, 2003.

[14] G. Gonthier and A. Mahboubi. A small scale reflection extension for the Coq system. Technical Report 6455, INRIA, 2007.

[15] N. R. Krishnaswami, J. Aldrich, L. Birkedal, K. Svendsen, and A. Buisse. Design patterns in separation logic. In *TLDI'09*, pages 105–116.

[16] X. Leroy and S. Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *J. Autom. Reason.*, 41(1):1–31, 2008.

[17] N. Marty and R. Affeldt. A certified verifier for a fragment of separation logic. *Computer Software*, 25(3):135–147, 2008.

[18] A. McCreight. Practical tactics for separation logic. In *TPHOL'09*, pages 343–358.

[19] A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and separation in Hoare Type Theory. In *ICFP'06*, pages 62–73.

[20] A. Nanevski, G. Morrisett, and L. Birkedal. Hoare type theory, polymorphism and separation. *Journal of Functional Programming*, 18(5&6):865–911, 2008.

[21] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: Dependent types for imperative programs. In *ICFP'08*, pages 229–240.

[22] H. H. Nguyen and W.-N. Chin. Enhancing program verification with lemmas. In *CAV'08*, pages 355–369.

[23] R. Nieuwenhuis and A. Oliveras. Fast congruence closure and extensions. *Information and Computation*, 205(4):557–580, 2007.

[24] P. O'Hearn. On bunched typing. *Journal of Functional Programming*, 13(4):747–796, 2003.

[25] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL'01*, pages 1–19.

[26] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS'02*, pages 55–74.

[27] K. Zee, V. Kuncak, and M. Rinard. An integrated proof language for imperative programs. In *PLDI'09*, pages 338–351.