

Structuring the verification of heap-manipulating programs

Aleksandar Nanevski Viktor Vafeiadis Josh Berdine

`{aleksn, viktorva, jjb}@microsoft.com`

Microsoft Research

July 2009
Technical Report
MSR-TR-2009-104

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
<http://www.research.microsoft.com>

This page intentionally left blank.

Structuring the verification of heap-manipulating programs

Aleksandar Nanevski Viktor Vafeiadis Josh Berdine

Microsoft Research

{aleksn, viktorva, 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 F3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs

General Terms Languages, Verification

Keywords Type Theory, Hoare Logic, Separation Logic, Monads

1. Introduction

While separation logic [23, 24] has proved to be extremely effective in reasoning about heap-manipulating programs in the presence of aliasing, most practical systems such as Smallfoot [5], HIP [20], SLAyer [4], Space Invader [8] or Xisa [10] address only a restricted fragment of assertions, roughly described by the grammar:

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

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 connectives such as implication, conjunction and universal quantification. To see why these are omitted, consider the entailments (1) $\Gamma \Longrightarrow P_1 * P_2 \rightarrow Q$ and (2) $\Gamma \Longrightarrow P_1 \wedge P_2 \rightarrow Q$, in the sequent calculus for BI [22]. 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 [3, 12], 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 $P \rightarrow Q$ 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 [16, 17, 1, 11] develop methods for dealing only with 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) [18, 19, 11], 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-

barriers, 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 [19], but lead to a prohibitive overhead in the size and number of proof obligations about heap disjointness [14], 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. We rely on Hoare triples with postconditions that are *binary*, rather than unary relations on heaps, to obtain generality. We show that this setting allows the user to extend the system with his or her own structural rules, 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 like Ynot [11] and Jahob [25]. 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 [21], deployed in the state-of-the-art Barcelog SAT solver. Our developments are carried out in Ssreflect [13], which is a recent extension of Coq that simplifies dealing with reflection. All our files are available on the web at <http://research.microsoft.com/~aleksn/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 [19], heaps are defined as $\text{loc} \rightarrow \text{option dynamic}$, where the type of locations loc is isomorphic to natural numbers and dynamic is the record type $\{\text{tp}:\text{Type}, \text{val}:\text{tp}\}$, packaging a value val with its type tp . The main problem with this definition shows up when one considers heap union.

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

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 [16, 11]. In either case, we are immediately faced with proving some basic algebraic properties.

$$\begin{aligned} \text{commute} &: h_1 \bullet h_2 = h_2 \bullet h_1 \\ \text{assoc} &: \text{disjoint } h_1 h_2 \vee \text{disjoint } h_2 h_3 \vee \text{disjoint } h_3 h_1 \rightarrow \\ & \quad h_1 \bullet (h_2 \bullet h_3) = (h_1 \bullet h_2) \bullet h_3 \end{aligned}$$

where $\text{disjoint } h_1 h_2 = \forall x v. h_1 x = \text{Some } v \rightarrow h_2 x = \text{None}$. 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 $\text{disj}:\text{heap} \rightarrow \text{heap} \rightarrow \text{bool}$, which *reflects* disjointness; that is, $\text{disj } h_1 h_2$ evaluates to true if and only if $\text{disjoint } h_1 h_2$ holds. The difference between the two expressions is that $\text{disj } h_1 h_2$ is a boolean, while $\text{disjoint } h_1 h_2$ 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.

$$\begin{aligned} \text{heap} &= \text{Undef} \mid \text{Def of } \{l : \text{list } (\text{loc} \times \text{dynamic}), \\ & \quad _ : \text{sorted } l\} \\ \text{empty} &= \text{Def } (\text{nil}, \text{sorted_nil}) \\ [x \rightarrow v] &= \text{if } x == \text{null then Undef} \\ & \quad \text{else Def } ((x, v) :: \text{nil}, \text{sorted_cons } x v) \\ h_1 \bullet h_2 &= \text{if } (h_1, h_2) \text{ is } (\text{Def } (l_1, _), \text{Def } (l_2, _)) \text{ then} \\ & \quad \text{if } \text{disj } l_1 l_2 \text{ then} \\ & \quad \quad \text{Def } (\text{sort } (l_1 ++ l_2), \text{sorted_cat } l_1 l_2) \\ & \quad \text{else Undef} \\ & \quad \text{else Undef} \\ \text{def } h &= \text{if } h \text{ is Undef then false else true} \end{aligned}$$

Since the definition packages each list l with a proof of 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 $\text{sorted_cat}:\forall l_1 l_2. \text{sorted}(\text{sort}(l_1 ++ l_2))$ to l_1 and l_2 to convince the typechecker that $\text{sort}(l_1 ++ l_2)$ is indeed sorted. Similarly, the definitions of empty heap and singleton heap $[x \rightarrow v]$, require lemmas $\text{sorted_nil}:\text{sorted nil}$ and $\text{sorted_cons}:\forall x v. \text{sorted}((x, v) :: \text{nil})$.

Of course, we will hide the intricacies of this definition, and keep heaps as an abstract type, only exposing several algebraic properties. Chief 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, without the steep price of proving disjointness at every step.

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

Even more important is the def predicate, which we use to state disjointness of heaps. For example, we can define $P_1 * P_2 = \text{fun } h. \exists h_1 h_2. h = h_1 \bullet h_2 \wedge \text{def } h \wedge P_1 h_1 \wedge P_2 h_2$. 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, \dots, P_4 into the Coq hypothesis context, we can make the heap variables explicit and write

$$P_1 h_1 \wedge P_2 h_2 \wedge P_3 h_3 \wedge (\forall x. P_4 x h_3) \rightarrow \\ \text{def } (h_1 \bullet h_2 \bullet h_3) \rightarrow P_5 (h_1 \bullet h_2 \bullet h_3)$$

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 $h_2 \bullet h_3$, 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 essentially states the cancellation property for \bullet , is used more often.

$$\text{cancel} : \text{def}([x \mapsto v_1] \bullet h_1) \rightarrow ([x \mapsto v_1] \bullet h_1 = [x \mapsto v_2] \bullet h_2) \\ \rightarrow v_1 = v_2 \wedge \text{def } h_1 \wedge h_1 = h_2.$$

The conclusion of `cancel` produces new facts `def h1` and `h1 = h2` which can be passed to `cancel` 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 .

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

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

$$\text{lseq_nil} : \forall l h. \text{lseq } \text{null } l h \rightarrow l = \text{nil} \wedge h = \text{empty} \\ \text{lseq_cons} : \forall l p h. p \neq \text{null} \rightarrow \text{lseq } p l h \rightarrow \\ \exists x q h'. l = x::\text{tail } l \wedge h = [p \rightarrow x] \bullet [p+1 \rightarrow q] \bullet h' \wedge \\ \text{lseq } q (\text{tail } l) h' \wedge \text{def } h \\ \text{def_null} : \forall p x h. \text{def } ([p \rightarrow x] \bullet h) \rightarrow p \neq \text{null}$$

The proof is by induction on l_1 . If $l_1 = \text{nil}$, then `lseq p l1 h` implies $p = \text{null}$, and the result follows by applying the lemma `lseq_nil` to `lseq null l2 h`. Otherwise, let $l_1 = x_1::xt$, let IH be the induction hypothesis $\forall l_2 p h. \text{lseq } p xt h \rightarrow \text{lseq } p l_2 h \rightarrow xt = l_2$, and let q_1, h_1 be such that `lseq q1 xt h1` and `def ([p → x1] • [p+1 → q1] • h1)`. Call the last two facts H and D , respectively. It suffices to show `lseq p l2 ([p → x1] • [p+1 → q1] • h1) → x1::xt = l2`. Now, $p \neq \text{null}$ by `def.null` and D , so `lseq.cons` and `lseq p l2 ([p → x1] • [p+1 → q1] • h1)` obtain x_2, q_2 and h_2 , and reduce the goal to

$$[p \rightarrow x_1] \bullet [p+1 \rightarrow q_1] \bullet h_1 = [p \rightarrow x_2] \bullet [p+1 \rightarrow q_2] \bullet h_2 \\ \rightarrow \text{lseq } q_2 (\text{tail } l_2) h_2 \rightarrow x_1::xt = x_2::\text{tail } l_2$$

By applying `cancel` to D and the antecedent of this implication, and then doing so again on the obtained def predicate and heap equation, we get $x_1 = x_2, q_1 = q_2, h_1 = h_2$, further reducing the

goal to `lseq q1 (tail l2) h1 → x1::xt = x1::tail l2`. Now, by IH and H , we get $xt = \text{tail } l_2$, which concludes the proof.

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`.

$$\text{elim} \Rightarrow [|x_1 xt IH| l_2 p h; \text{first by case} \Rightarrow \rightarrow \rightarrow; \text{case/lseq_nil.} \\ \text{move} \Rightarrow [q_1][h_1][\rightarrow] H D. \\ \text{case/(lseq_cons (def_null } D)) \Rightarrow x_2 [q_2][h_2][\rightarrow]. \\ \text{do } 2![\text{case/(cancel } D) \Rightarrow \leftarrow \{D\} D] \Rightarrow \leftarrow . \\ \text{by case/(IH } \dots H) \Rightarrow \leftarrow .$$

In Section 4, we will show how similar chaining of def predicates can be done when reasoning about Hoare triples as well.

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 [16, 17]. One burden with deep embedding is that it requires explicit manipulation of de Bruijn representation of variables. In contrast, HTT formalizes separation logic via types; a triple $\{p\} e \{q\}$ in HTT becomes a type ascription $e : \text{STsep } A (p, q)$, where A is the type of the return value of the effectful “command” e . The language from which commands are drawn is now Coq itself, which removes a level of indirection, streamlines the reasoning, and enables HTT to inherit the powerful features of Coq such as inductive types, higher-order functions, type abstraction and first-class modules. Encoding via types, however, is not straightforward, and requires a reformulation of the inference rules of separation logic.

These inference rules come in two flavors. The first set includes rules that infer properties based on the command’s top command.

$$\{\text{emp}\} \text{move } x v \{x = v \wedge \text{emp}\} \quad \{x \mapsto -\} \text{store } x v \{x \mapsto v\} \\ \{x \mapsto v\} \text{load } y x \{x \mapsto v \wedge y = v\} \\ \{\text{emp}\} \text{alloc } y v \{y \mapsto v\} \quad \{x \mapsto -\} \text{dealloc } x \{\text{emp}\} \\ \frac{\{p\} e_1 \{q\} \quad \{q\} e_2 \{r\}}{\{p\} e_1; e_2 \{r\}}$$

The second set includes the structural rules. This set varies across systems, but here we take it to include the rules of frame, consequence, conjunction and disjunction in both binary and quantified (i.e., universal and existential) variants.

$$\frac{\{p\} e \{q\}}{\{p * r\} e \{q * r\}} \quad \frac{p \rightarrow p' \quad \{p'\} e \{q'\} \quad q' \rightarrow q}{\{p\} e \{q\}} \\ \frac{\{p\} e \{q_1\} \quad \{p\} e \{q_2\}}{\{p\} e \{q_1 \wedge q_2\}} \quad \frac{\{p\} e \{q\} \quad x \notin \text{FV}(e, p)}{\{p\} e \{\forall x. q\}} \\ \frac{\{p_1\} e \{q\} \quad \{p_2\} e \{q\}}{\{p_1 \vee p_2\} e \{q\}} \quad \frac{\{p x\} e \{q\} \quad x \notin \text{FV}(e, q)}{\{\exists x. p\} e \{q\}}$$

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 . We will not need 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, 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. Given the type A , precondition $p:\text{heap} \rightarrow \text{Prop}$, and binary postcondition $q:A \rightarrow \text{heap} \rightarrow \text{heap} \rightarrow \text{Prop}$, our predicate transformers are elements of the type

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

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

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

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

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

where $\text{monotone } F = \forall r_1 \ r_2:\text{ideal } p. r_1 \sqsubseteq r_2 \rightarrow \forall x. F \ r_1 \ x \sqsubseteq F \ r_2 \ x$, and $\text{bounded } F \ q = \forall r \ x. F \ r \ x \sqsubseteq \text{fun } h. (\exists i. r \ i \wedge q \ x \ i \ h)$. Out of this type, we now select transformers that satisfy the frame rule, similar to [7, 19]. Given a pre/postcondition pair s , we define *spacial extension* s^\bullet , and a new STsep type, as follows.

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

where $p \multimap q = \{(i, m) \mid \forall i_1 \ h. i = i_1 \bullet h \rightarrow \text{def } i \rightarrow p \ i_1 \rightarrow \exists m_1. m = m_1 \bullet h \wedge \text{def } m \wedge q \ i_1 \ m_1\}$, and *pre* and *post* are the projections out of the pair. Spacial extension allows that heaps on which a transformer is applied be extended with portions that the transformer keeps invariant. For example, transformers in $\text{STsep } A(p, q)$ 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.

We can now transcribe the inference rules of separation logic 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.

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

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

$$\text{allocb} : \Pi v:A. \Pi n:\text{nat}. \text{STsep loc } (\text{emp}, \text{fun } y \ i \ m. m = \text{iter } n \ y \ v)$$

where $\text{iter } (n:\text{nat}) \ (y:\text{loc}) \ (v:A):\text{heap} = \text{if } n \text{ is } n'+1 \text{ then } [y \rightarrow v] \bullet \text{iter } n' \ (y+1) \ v \text{ else empty}$. 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.

$$\begin{aligned} \text{fix} &: ((\Pi x:A. \text{STsep } (B \ x) \ (s \ x)) \rightarrow \Pi x:A. \text{STsep } (B \ x) \ (s \ x)) \\ &\rightarrow \Pi x:A. \text{STsep } (B \ x) \ (s \ x) \end{aligned}$$

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

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

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

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

The precondition in this pair states that in order to execute the sequential composition, we must ensure that the precondition $\text{pre } s_1^\bullet$ 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 $\text{post } s_1^\bullet \ x \ i \ h$, so we need to show $\text{pre } (s_2 \ x)^\bullet \ h$ 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 $e:\text{STsep } A \ s$, we consider what can be inferred about e just by looking at the type A and specification s . Quite directly, it must be that $\text{pre } s \ i$ and $\text{post } s \ y \ i \ m$ hold of the initial heap i , final heap m and return value y . Thus, given a property $q:A \rightarrow \text{heap} \rightarrow \text{heap} \rightarrow \text{Prop}$, we can show that $q \ y \ m$ holds after running e if we can prove $\text{verify } i \ s \ q$, where

$$\text{verify } i \ s \ q = \text{pre } s \ i \wedge \forall y \ m. \text{post } s \ y \ m \rightarrow q \ y \ m.$$

This definition assumes that s describes how e acts on the *full* heap i . But sometimes s will be a tighter specification, describing the action of e only on a subheap of i , assuming the invariance of the rest. In those situations, we can show that $q \ y \ m$ holds after running e , if we can prove $\text{verify } i \ s^\bullet \ q$.

The *verify* predicate can now be used to represent Hoare triples as assertions. For example, given $e:\text{STsep } s$, the separation logic triple $\{p\} e \{q\}$ can be written as $\forall i. p \ i \rightarrow \text{verify } i \ s^\bullet \ q$. This property will let us encode the standard structural rules, as well as many other useful rules, simply as *derived lemmas* about the *verify* predicate. Hence, our system will be inherently extendable, as the user is free to derive 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.

$$\begin{aligned} \text{conj} &: \text{verify } i \ s \ q_1 \rightarrow \text{verify } i \ s \ q_2 \rightarrow \\ &\quad \text{verify } i \ s \ (\text{fun } y \ h. q_1 \ y \ h \wedge q_2 \ y \ h) \end{aligned}$$

$$\begin{aligned} \text{all} &: (\forall x:B. \text{verify } i \ s \ (q \ x)) \rightarrow B \rightarrow \\ &\quad \text{verify } i \ s \ (\text{fun } y \ m. \forall x:B. q \ x \ y \ m) \end{aligned}$$

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 $x \notin \text{FV } s$, 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

$$\text{disj} : (p_1 \ i \rightarrow \text{verify } i \ s \ q) \rightarrow (p_2 \ i \rightarrow \text{verify } i \ s \ q) \rightarrow p_1 \ i \vee p_2 \ i \rightarrow \text{verify } i \ s \ q$$

but this is just an instance of the usual elimination rule for disjunction, and therefore does not require a separate lemma.

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

$$\text{frame} : \text{verify } i \ s^* (\text{fun } y \ m. \text{def } (m \bullet h) \rightarrow q \ y \ (m \bullet h)) \rightarrow \text{def } (i \bullet h) \rightarrow \text{verify } (i \bullet h) \ s^* q.$$

In bottom-up proving, this formulation makes the lemma apply to goals where the verify predicate has an arbitrary postcondition q , whereas the usual formulation from the beginning of Section 3 requires first rewriting q into a form $q' * r$, which may be 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.

$$\text{do} : \text{STsep } A \ s_1 \rightarrow (\forall i. \text{pre } s_2 \ i \rightarrow \text{verify } i \ s_1^* (\text{fun } y \ m. \text{post } s_2 \ y \ i \ m)) \rightarrow \text{STsep } A \ s_2$$

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:

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

The do 's in the branches serve to weaken the types of e_i into the common type of the conditional. Each do requires a 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 . We omit the proofs here, as Coq can discharge them by itself. To reduce clutter, in the rest of the paper we blur the distinction between matches with pure and side-effectful bodies.

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

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

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

Here, we have used the standard abbreviation $x \leftarrow e_1; e_2$ for $\text{bind } e_1 \ (\text{fun } x. e_2)$, and $e_1; e_2$ when $x \notin \text{FV}(e_2)$. 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 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 commands. For insert, we get

$$\begin{aligned} \forall p \ x \ i. (\exists l. \text{lseq } p \ l \ i) \rightarrow \\ \text{verify } i \ (\text{bind_s } (\text{allocb_s } p \ 2) \\ (\text{fun } y. \text{bind_s } (\text{write_s } y \ x) \\ \text{fun } _ . \text{return_s } y)))^* \\ (\text{fun } y \ m. \forall l. \text{lseq } p \ l \ i \rightarrow \text{lseq } y \ (x :: l) \ m) \end{aligned}$$

and for remove

$$\begin{aligned} \forall p \ i. (\exists l. \text{lseq } p \ l \ i) \rightarrow \\ \text{verify } i \ (\text{if } p == \text{null then return_s } p \\ \text{else bind_s } (\text{read_s } (p + 1) \\ (\text{fun } x. \text{bind_s } (\text{dealloc_s } p) \\ (\text{fun } _ . \text{bind_s } (\text{dealloc_s } (p + 1)) \\ \text{fun } _ . \text{return_s } x))))^* \\ (\text{fun } q \ m. \forall l. \text{lseq } p \ l \ i \rightarrow \text{lseq } q \ (\text{tail } l) \ m) \end{aligned}$$

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, $\text{return_s } p$ is the pair $(\text{emp}, \text{fun } y \ i \ m. y = p \wedge \text{emp } m)$, $\text{read_s } x$ is $(x \mapsto _, \text{fun } y \ i \ m. \forall v. (x \mapsto v) \ i \rightarrow (x \mapsto v) \ m \wedge y = v)$, 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 commands 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.

$$\begin{aligned} \text{all_imp} : (\exists x:B. p \ x) \rightarrow \\ (\forall x:B. p \ x \rightarrow \text{verify } i \ s \ (\text{fun } y \ m. q \ x \ y \ m)) \rightarrow \\ \text{verify } i \ s \ (\text{fun } y \ m. \forall x:B. p \ x \rightarrow q \ x \ y \ m) \end{aligned}$$

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 .

$$\text{all_imp}_1 : \forall t:B. (\forall x:B. p\ x \rightarrow t = x) \rightarrow \text{verify } i\ s\ (q\ t) \rightarrow \text{verify } i\ s\ (\text{fun } y\ m. \forall x:B. p\ x \rightarrow q\ x\ y\ m)$$

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 .

$$\text{all_imp}_2 : \forall t:B. (\forall x:B\ y\ m. p\ x \rightarrow q\ t\ y\ m \rightarrow q\ x\ y\ m) \rightarrow \text{verify } i\ s\ (q\ t) \rightarrow \text{verify } i\ s\ (\text{fun } y\ m. \forall x:B. p\ x \rightarrow q\ x\ y\ m)$$

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 `bind e_1 e_2` where e_1, e_2 are arbitrary commands, with types $e_1 : \text{STsep } A_1\ s_1$ and $e_2 : \Pi x:A_1. \text{STsep } A_2\ (s_2\ x)$, respectively.

$$\begin{aligned} \text{bnd_do} : & \text{pre } s_1\ i_1 \rightarrow \\ & (\forall x\ i'_1. \text{post } s_1\ x\ i_1\ i'_1 \rightarrow \\ & \text{def } (i'_1 \bullet i_2) \rightarrow \text{verify } (i'_1 \bullet i_2)\ (s_2\ x) \bullet r) \rightarrow \\ & \text{def } (i_1 \bullet i_2) \rightarrow \text{verify } (i_1 \bullet i_2)\ (\text{bind_s } s_1\ s_2) \bullet r \end{aligned}$$

Applying this lemma to a goal of the form `def ($i_1 \bullet i_2$) \rightarrow verify ($i_1 \bullet i_2$) (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 `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.

$$\begin{aligned} \text{bnd_ret} : & \text{verify } i\ (s_2\ v) \bullet r \rightarrow \text{verify } i\ (\text{bind_s } (\text{return_s } v)\ s_2) \bullet r \\ \text{bnd_read} : & \text{verify } ([x \rightarrow v] \bullet i)\ (s_2\ v) \bullet r \rightarrow \\ & \text{def } ([x \rightarrow v] \bullet i) \rightarrow \\ & \text{verify } ([x \rightarrow v] \bullet i)\ (\text{bind_s } (\text{read_s } A\ x)\ s_2) \bullet r \\ \text{bnd_write} : & (\text{def } ([x \rightarrow v] \bullet i) \rightarrow \text{verify } ([x \rightarrow v] \bullet i)\ (s_2\ ())) \bullet r \rightarrow \\ & \text{def } ([x \rightarrow w] \bullet i) \rightarrow \\ & \text{verify } ([x \rightarrow w] \bullet i)\ (\text{bind_s } (\text{write_s } x\ v)\ s_2) \bullet r \\ \text{bnd_alloc} : & (\forall x:\text{loc}. \text{def } ([x \rightarrow v] \bullet i) \rightarrow \\ & \text{verify } ([x \rightarrow v] \bullet i)\ (s_2\ x) \bullet r) \rightarrow \\ & \text{def } i \rightarrow \text{verify } i\ (\text{bind_s } (\text{alloc_s } v)\ s_2) \bullet r \\ \text{bnd_allocb} : & (\forall x:\text{loc}. \text{def } (\text{iter } n\ x\ v \bullet i) \rightarrow \\ & \text{verify } (\text{iter } n\ x\ v \bullet i)\ (s_2\ x) \bullet r) \rightarrow \\ & \text{def } i \rightarrow \text{verify } i\ (\text{bind_s } (\text{allocb_s } v\ n)\ s_2) \bullet r \\ \text{bnd_dealloc} : & (\text{def } i \rightarrow \text{verify } i\ (s_2\ ())) \bullet r \rightarrow \\ & \text{def } ([x \rightarrow v] \bullet i) \rightarrow \\ & \text{verify } ([x \rightarrow v] \bullet i)\ (\text{bind_s } (\text{dealloc_s } x)\ s_2) \bullet r \\ \text{bnd_bnd} : & \text{verify } i\ (\text{bind_s } t_1\ (\text{fun } x. \text{bind_s } (t_2\ x)\ s_2)) \bullet r \rightarrow \\ & \text{verify } i\ (\text{bind_s } (\text{bind_s } t_1\ t_2)\ s_2) \bullet r \end{aligned}$$

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:

$$\text{val_ret} : r\ v\ i \rightarrow \text{def } i \rightarrow \text{verify } i\ (\text{return_s } v) \bullet r,$$

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:loc`, `x:T`, `l:list T`, hypothesis $H:\text{lseq } p\ l\ i$, and the goal

$$\begin{aligned} & \text{verify } i\ (\text{bind_s } (\text{allocb_s } p\ 2) \\ & \quad (\text{fun } y. \text{bind_s } (\text{write_s } y\ x) \\ & \quad \quad \text{fun } _. \text{return_s } y)) \bullet \\ & \quad (\text{fun } y\ m. \forall l. \text{lseq } p\ l\ i \rightarrow \text{lseq } y\ (x::l)\ m). \end{aligned}$$

We apply the lemma `all_imp1` to remove the quantifier over l and the antecedent `lseq $p\ l\ i$` from the postcondition, to obtain

$$\begin{aligned} & \text{verify } i\ (\text{bind_s } (\text{allocb_s } p\ 2) \\ & \quad (\text{fun } y. \text{bind_s } (\text{write_s } y\ x) \\ & \quad \quad \text{fun } _. \text{return_s } y)) \bullet \\ & \quad (\text{fun } y\ m. \text{lseq } y\ (x::l)\ m). \end{aligned}$$

The hypothesis of `all_imp1` is easily satisfied, using H and the lemma `lseq.func` proved in Example 2. Next, by hypothesis H and helper lemma `lseq.def:lseq $p\ l\ i \rightarrow$ def i` , we obtain `def i` . Using this and `bnd_allocb`, we reduce the goal to

$$\begin{aligned} & \text{def } (([y \rightarrow p] \bullet [y+1 \rightarrow p] \bullet \text{empty}) \bullet i) \rightarrow \\ & \text{verify } (([y \rightarrow p] \bullet [y+1 \rightarrow p] \bullet \text{empty}) \bullet i) \\ & \quad (\text{bind_s } (\text{write_s } y\ x)\ (\text{fun } _. \text{return_s } y)) \bullet \\ & \quad (\text{fun } y\ m. \text{lseq } y\ (x::l)\ m) \end{aligned}$$

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

$$\begin{aligned} & \text{verify } ([y \rightarrow x] \bullet [y+1 \rightarrow p] \bullet i) \\ & \quad (\text{return_s } y) \bullet \\ & \quad (\text{fun } y\ m. \text{lseq } y\ (x::l)\ m) \end{aligned}$$

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

$$\begin{aligned} & \exists q\ h'. [y \rightarrow x] \bullet [y+1 \rightarrow p] \bullet i = [y \rightarrow x] \bullet [y+1 \rightarrow q] \bullet h' \\ & \quad \wedge \text{lseq } q\ l\ h' \wedge \text{def } ([y \rightarrow x] \bullet [y+1 \rightarrow p] \bullet i). \end{aligned}$$

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.

$$\begin{aligned} & \text{apply: (all_imp}_1\ l) \Rightarrow [?]; \text{first by apply: lseq.func.} \\ & \text{apply: bnd_allocb (lseq.def } H) \Rightarrow y; \text{rewrite unh0 unA.} \\ & \text{apply: bnd.write } \Rightarrow D; \text{apply: val.ret } \Rightarrow // \\ & \text{by do !constructor.} \end{aligned}$$

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 [21], and used in the Barcelogic SAT Solver whose efficiency has been confirmed in various SAT-solving competitions [2]. 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

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

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

Definition 6. A binary relation R on expressions is monotone iff $\forall f_1 f_2 e_1 e_2. (f_1, f_2) \in R \rightarrow (e_1, e_2) \in R \rightarrow (f_1 e_1, f_2 e_2) \in R$. 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 closure $R = \bigcap \{C \mid C \text{ is congruence and } R \subseteq C\}$.

The algorithm internally maintains a data structure that represents the congruence closure of a set of equations. Its interface consists of two methods: (1) *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) *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

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

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 [21].

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
kvmmap : eqType → Type → Type
shape : kvmmap K V → (K → option V) → Prop
lookup : Πt:kvmmap K V. Πk:K.
      STsep (option V) (fun i. ∃f. shape t f i,
                      fun y i m. ∀f. shape t f i →
                          shape t f m ∧ y = f k)

insert : Πt:kvmmap 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 1. 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 1 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 $\text{array } I T$ and $\text{kvmmap } K V$ 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 $== : K \rightarrow K \rightarrow \text{bool}$. 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 $I \rightarrow T$, and in the case of hash tables, it is a function of type $K \rightarrow \text{option } V$, 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: $r : \text{array symb symb}$, $clist, ulist : \text{array symb loc}$, $htab : \text{kvmmap (symb} \times \text{symb) (symb} \times \text{symb} \times \text{symb)}$, and $p : \text{loc}$.

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.

```
data = { rep : symb → symb; class : symb → list symb;
        use : symb → list (symb × symb × symb);
        lookup : symb × symb → option (symb × symb × symb);
        pending : list (symb × symb) }
```

The intention is that, given $D:\text{data}$, the function $\text{rep } D$ represents the contents of the array r , and similarly $\text{class } D$, $\text{use } D$, $\text{lookup } D$ and $\text{pending } D$ capture the contents of $clist$, $ulist$, $htab$ and p . The

formal correspondence is established by the following predicate.

```

shape' (D : data) (h : heap) : Prop :=
  ∃ ct ut : symb → loc. ∃ q : loc.
    Array.shape r (rep D) *
    Array.shape clist ct *  $\bigotimes_{c \in \text{sybm}}$  lseq (ct c) (class D c) *
    Array.shape ulist ut *  $\bigotimes_{c \in \text{sybm}}$  lseq (ut c) (use D c) *
    Hashtab.shape htab (lookup D) *
    p  $\mapsto$  q * lseq q (pending D)) h

```

Here we freely use the separation logic $*$ (as defined in Section 2) and its iterated version \bigotimes . 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.

```

shape (R : exp  $\times$  exp  $\rightarrow$  Prop) (h : heap) : Prop =
  ∃ D : data. shape' D h  $\wedge$  rep_idemp D  $\wedge$  class_inv D  $\wedge$ 
    use_inv D  $\wedge$  lkp_inv D  $\wedge$  use_lkp_inv D  $\wedge$ 
    lkp_use_inv D  $\wedge$  pending D = nil  $\wedge$  CRel D =r R

```

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

```
rep_idemp D =  $\forall c$ . rep D (rep D c) = rep D c.
```

The class lists invert the representative array:

```
class_inv D =  $\forall x c$ . (rep D x == c) = (x  $\in$  class D c).
```

Use lists store only equations with appropriate representatives:

```
use_inv D =  $\forall a c_1 c_2$ . a  $\in$  reps D  $\rightarrow$ 
  (c, c1, c2)  $\in$  use D a  $\rightarrow$  rep D c1 = a  $\vee$  rep D c2 = a,
```

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

```
lkp_inv D =  $\forall a b c_1 c_2$ . a  $\in$  reps D  $\rightarrow$  b  $\in$  reps D  $\rightarrow$ 
  lookup D (a, b) = Some (c, c1, c2)  $\rightarrow$  rep D c1 = a  $\wedge$  rep D c2 = b.
```

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

```
use_lkp_inv D =  $\forall a c c_1 c_2$ . a  $\in$  reps D  $\rightarrow$  (c, c1, c2)  $\in$  use D a  $\rightarrow$ 
  ∃ d d1 d2. lookup D (rep D c1, rep D c2) = Some (d, d1, d2)  $\wedge$ 
  rep D c1 = rep D d1  $\wedge$  rep D c2 = rep D d2  $\wedge$  rep D c = rep D d
```

```
lkp_use_inv D =  $\forall a b d d_1 d_2$ . a  $\in$  reps D  $\rightarrow$  b  $\in$  reps D  $\rightarrow$ 
  lookup D (a, b) = Some (d, d1, d2)  $\rightarrow$ 
  (∃ c c1 c2. (c, c1, c2)  $\in$  use D a  $\wedge$ 
    rep D c1 = a  $\wedge$  rep D c2 = b  $\wedge$  rep D c = rep D d)  $\wedge$ 
  (∃ c c1 c2. (c, c1, c2)  $\in$  use D b  $\wedge$ 
    rep D c1 = a  $\wedge$  rep D c2 = b  $\wedge$  rep D c = rep D d).
```

The shape predicate will be used for the specification of the main methods of the algorithm. Hence it also requires that $\text{pending } D = \text{nil}$, i.e., the structures are in a consistent state, and $\text{CRel } D =_r R$, i.e., the relation R is the congruence represented by the structures. Here, $\text{CRel } D$ is defined as the congruence closure of all the equations in $\text{lookup } D$, $\text{pending } D$ as well as the equations $c = \text{rep } D c$, 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 2. 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 4, to merge the congruence

```

1. merge (eq : Eq) :
2.   STsep unit (fun i. ∃ R. shape p R i,
3.     fun y i m. ∀ R. shape p R i  $\rightarrow$ 
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 c1 c2  $\Rightarrow$ 
12.   do (c'1  $\leftarrow$  Array.read r c1;
13.     c'2  $\leftarrow$  Array.read r c2;
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, c1, c2);
18.       u1  $\leftarrow$  Array.read ulist c'1;
19.       x  $\leftarrow$  insert u1 (c, c1, c2);
20.       Array.write ulist c'1 x;
21.       u2  $\leftarrow$  Array.read ulist c'2;
22.       x  $\leftarrow$  insert u2 (c, c1, c2);
23.       Array.write ulist c'2 x
24. | Some (b, b1, b2)  $\Rightarrow$ 
25.   q  $\leftarrow$  !p;
26.   x  $\leftarrow$  insert q (c, b);
27.   p := x;
28.   hpropagate
29. end)
30. end

31. check (t1 t2 : exp) :
32.   STsep bool (fun i. ∃ R. shape p R i,
33.     fun y i m. ∀ R. shape p R i  $\rightarrow$  shape p R m  $\wedge$ 
34.       y = true  $\leftrightarrow$  R (t1, t2)) =
35.   do (u1  $\leftarrow$  hnorm t1;
36.     u2  $\leftarrow$  hnorm t2;
37.     return (u1 == u2))

where

rel_of (eq : Eq) : exp  $\times$  exp  $\rightarrow$  Prop :=
  match eq with
  simp a b  $\Rightarrow$  fun t. t.1 = const a  $\wedge$  t.2 = const b
  | comp c c1 c2  $\Rightarrow$  fun t. t.1 = const c  $\wedge$ 
    t.2 = app (const c1) (const c2)
  end

```

Figure 2. The main functions of the fast congruence closure algorithm, and their specifications.

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 3. 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 5) and hjoin_use (Figure 6), are called from within hpropagate . In the verification

```

38. hnorm (t : exp) =
39.   fix (fun hnorm (t:exp).
40.     do (match t with
41.       const a =>
42.         a' ← Array.read r a;
43.         return (const a')
44.     | app t1 t2 =>
45.       u1 ← hnorm t1;
46.       u2 ← hnorm t2;
47.       match u1, u2 with
48.       const w1, const w2 =>
49.         v ← Hashtab.lookup htab (w1, w2);
50.         match v with
51.         None => return (app u1 u2)
52.         | Some (b, -, -) =>
53.           b' ← Array.read r b;
54.           return (const b')
55.       end
56.     | -, - => return (app u1 u2)
57.   end
58. end)) t

```

Figure 3. Helper function for normalizing expressions.

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 3-6, 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

```

normT =  $\Pi t:\text{exp}. \text{STsep exp (fun } i. \exists D. \text{shape}' p D i,$ 
 $\text{fun } y i m. \forall D. \text{shape}' p D i \rightarrow$ 
 $\text{shape}' p D m \wedge y = \text{norm } D t)$ 

propagateT =  $\text{STsep unit (fun } i. \exists D. \text{shape}' p D i,$ 
 $\text{fun } y i m. \forall D. \text{shape}' p D i \rightarrow$ 
 $\text{shape}' p (\text{propagate } D) m)$ 

```

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 3. 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 4. 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 5. Helper function for merging the class lists of a' and b' .

The function `hpropagate` from Figure 4 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 6, 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

```

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.         c'_2 ← Array.read r eqc.2
98.         c'_3 ← Array.read r eqc.3
99.         v ← Hashtab.lookup htab (c'_2, c'_3);
100.        match v with
101.        None ⇒
102.          Hashtab.insert htab (c'_2, c'_3) 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 6. Helper function for adjusting the use lists and the lookup table, after the class lists of a' and b' have been merged.

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 \otimes from the shape' predicate. It is best to iterate \otimes over finite *sets*, rather than lists, which was our first attempt. If s is a set of symbols, one can show

$$x \in s \rightarrow \bigotimes_{i \in s} P i =_r P x * \bigotimes_{i \in s \setminus \{x\}} P i.$$

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. $a' \neq b' \rightarrow \text{class.inv } D \rightarrow \text{class.inv } (\text{join_class } D a' b')$
2. $a' \neq b' \rightarrow \text{class.inv } D \rightarrow \text{class.inv } (\text{join_use } D a' b')$
3. $\text{class.inv } D \rightarrow \text{class.inv } (\text{propagate } D)$

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 predi-

cates 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:

$$\text{use_lkp.inv0 } D = \forall a c c_1 c_2. a \in \text{reps } D \rightarrow (c, c_1, c_2) \in \text{use } D a \rightarrow \\ \exists d d_1 d_2. \text{lookup } D (\text{rep } D c_1, \text{rep } D c_2) = \text{Some } (d, d_1, d_2) \wedge \\ \text{rep } D c_1 = \text{rep } D d_1 \wedge \text{rep } D c_2 = \text{rep } D d_2 \wedge \text{similar } D c d$$

Here, *similar* $D c d$ holds if the symbols c and d are in the congruence relation generated by the equations $x = \text{rep } D x$ 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

$$\text{join_classP } (D : \text{data}) (a' b' : \text{symb}) : \\ a' \in \text{reps } D \rightarrow b' \in \text{reps } D \rightarrow a' \neq b' \rightarrow \\ \text{rep.idemp } D \rightarrow \text{use.inv } D \rightarrow \text{lkp.inv } D \rightarrow \\ \text{use_lkp.inv1 } D a' b' \rightarrow \text{lkp.use.inv1 } D a' b' \rightarrow \\ \text{use_lkp.inv2 } (\text{join_class } D a' b') a' b' \wedge \\ \text{lkp.use.inv2 } (\text{join_class } D a' b') a' b',$$

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. Related work

HTT and Ynot Just like the current paper, the original implementations of *HTT* and *Ynot* [18, 19] used Hoare triples with binary postconditions. However, those papers did not recognize the connection between binary postconditions and structural rules – which we proposed here – and ended up developing lemmas that did not lead to particularly streamlined proofs. In addition, that work relied on a naive definition of heaps from Section 2, which further caused an explosion in proof obligations about disjointness. 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 redun-

dancy 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 `remove p : STsep loc (lseq p l, fun q. lseq q (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 `III: list T. STsep loc (lseq p l, fun q. lseq q (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

```
III: inhabited (list T).
  STsep loc (let pack l = x in lseq p l,
             fun q. let pack l = x in lseq q (tail l))
```

where `inhabited A` is the *proposition* $\exists x:A. \top$, and `pack : A → 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 $l \notin \text{FV}(e)$, 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 `pack.injective: ∀T:Set. ∀x y:T. pack x = pack y → 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 [6], may offer a way out of these problems, and allow structural rules to be encoded as typing rules. But even then, the question remains which rules should be included. Our formulation of HTT with binary postconditions completely avoids all of the above problems. It is also very extendable, as it allows the user to derive his or her own structural rules, to implement custom verification strategies. 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. [16] 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 [17] defines heaps following the memory model of Leroy et al. [15], 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 [25] 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 [25] of whether similar substitutability is possible in Jahob as well.

Higher-order separation logic Krishnaswami et al. [14] 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 [14] 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.

7. 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 full separation logic.

Our second example was embedding a separation logic for partial correctness into type theory with the use of binary postconditions, which 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.

In our developments, we have kept our 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, 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. This was the case even in our proof of congruence closure, which was large, but the phase of the proof related to pointers and aliasing was proportional to the verified program. We attribute this property 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 $[x \rightarrow v_1] \bullet h_1 = [x \rightarrow v_2] \bullet h_2$, 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 $\text{def } h_1 \rightarrow \text{def } h_2$, 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 constructor 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 again ideas based on reflection.

8. 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] 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.
- [3] J. M. L. Bean. *Ribbon Proofs – A Proof System for the Logic of Bunched Implications*. PhD thesis, Queen Mary University of London, 2006.
- [4] 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.
- [5] 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.
- [6] B. Bernardo. Towards an implicit calculus of inductive constructions. In *TPHOL'09*. To appear.
- [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. Chlipala, G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. Effective interactive proofs for higher-order imperative programs. In *ICFP'09*. To appear.
- [12] D. Galmiche and D. Méry. Semantic labelled tableaux for propositional BI. *Journal of Logic and Computation*, 13(5):707–753, 2003.
- [13] G. Gonthier and A. Mahboubi. A small scale reflection extension for the Coq system. Technical Report 6455, INRIA, 2007.
- [14] N. R. Krishnaswami, J. Aldrich, L. Birkedal, K. Svendsen, and A. Buisse. Design patterns in separation logic. In *TLDI'09*, pages 105–116.
- [15] 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.
- [16] N. Marty and R. Affeldt. A certified verifier for a fragment of separation logic. *Computer Software*, 25(3):135–147, 2008.
- [17] A. McCreight. Practical tactics for separation logic. In *TPHOL'09*. To appear.
- [18] A. Nanevski, G. Morrisett, and L. Birkedal. Hoare type theory, polymorphism and separation. *Journal of Functional Programming*, 18(5&6):865–911, 2008.
- [19] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: Dependent types for imperative programs. In *ICFP'08*, pages 229–240.
- [20] H. H. Nguyen and W.-N. Chin. Enhancing program verification with lemmas. In *CAV'08*, pages 355–369.
- [21] R. Nieuwenhuis and A. Oliveras. Fast congruence closure and extensions. *Information and Computation*, 205(4):557–580, 2007.
- [22] P. O'Hearn. On bunched typing. *Journal of Functional Programming*, 13(4):747–796, 2003.
- [23] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL'01*, pages 1–19.
- [24] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS'02*, pages 55–74.
- [25] K. Zee, V. Kuncak, and M. Rinard. An integrated proof language for imperative programs. In *PLDI'09*, pages 338–351.