# Verified Heap Theorem Prover by Paramodulation

Gordon Stewart    Lennart Beringer    Andrew W. Appel

Princeton University, Princeton, NJ, USA
{jsseven, eberinge, appel}@cs.princeton.edu

## Abstract

We present *VeriStar*, a verified theorem prover for a decidable subset of separation logic. Together with VeriSmall [2], a proved-sound Smallfoot-style program analysis for C minor, VeriStar demonstrates that fully machine-checked static analyses equipped with efficient theorem provers are now within the reach of formal methods. As a pair, VeriStar and VeriSmall represent the first application of the *Verified Software Toolchain* [3], a tightly integrated collection of machine-verified program logics and compilers giving foundational correctness guarantees.

VeriStar is (1) *purely functional*, (2) *machine-checked*, (3) *end-to-end*, (4) *efficient* and (5) *modular*. By purely functional, we mean it is implemented in Gallina, the pure functional programming language embedded in the Coq theorem prover. By machine-checked, we mean it has a proof in Coq that when the prover says "valid", the checked entailment holds in a proved-sound separation logic for C minor. By end-to-end, we mean that when the static analysis+theorem prover says a C minor program is safe, the program will be compiled to a semantically equivalent assembly program that runs on real hardware. By efficient, we mean that the prover implements a state-of-the-art algorithm for deciding heap entailments and uses highly tuned verified functional data structures. By modular, we mean that VeriStar can be retrofitted to other static analyses as a plug-compatible entailment checker and its soundness proof can easily be ported to other separation logics.

***Categories and Subject Descriptors*** F.3.1 [*Specifying and Verifying and Reasoning about Programs*]: Mechanical verification

***General Terms*** Verification

***Keywords*** Separation Logic, Paramodulation, Theorem Proving

## 1. Introduction

Can you trust your decision procedure? When your memory analysis that calls upon this decision procedure returns "safe", how confident can you be that your C program won't dereference a null pointer? If you're writing safety- or security-critical code then such questions are crucial, but often difficult to answer: state-of-the-art theorem provers are large, intricate programs (Z3, for example, is over 300k lines of proprietary code [17]). A bug in the decision procedure might camouflage a bug in your static analysis regime, which may itself hide a disastrous bug in your safety-critical program.

To bridge the trust gap, you can instrument the decision procedure to produce *witnesses*—as in proof-carrying code (PCC) [4, 10, 26]—or implement and verify the decision procedure directly in a proof assistant [5, 11, 32]. Although one might suspect that separating the prover from the checker is necessary for efficiency, modern proof assistants have advanced to the point that it is now feasible to implement and verify even sophisticated analyses in a foundational way.

As evidence of this claim, we present VeriStar, an efficient *machine-verified* decision procedure for entailments in separation logic, the *de facto* standard for reasoning about shape properties of heap data. Tools based on separation logic, such as SLAyer [9], SpaceInvader [14], Infer [13] and Xisa [15], have been successfully applied to industrial code bases but have lacked foundational certification. VeriStar integrates with VeriSmall [2], a machine-checked symbolic executor, to yield a fully verified shape analysis for separation logic. When connected to the CompCert certified C compiler [22], VeriSmall+VeriStar enables end-to-end automatic verification of shape properties all the way from C to x86 or PowerPC assembly. Because CompCert's correctness theorem makes a claim directly about the generated assembly, the user of our system need trust only the Coq typechecker and CompCert's model of either x86 or PowerPC assembly.

***Contributions.*** The VeriStar system is:

- **Purely functional.** We implemented VeriStar in Gallina, the pure functional language embedded in the interactive theorem prover Coq. The use of a pure functional implementation language gave us both an elegant programming environment and an attractive proof theory for reasoning about our code.
- **Machine-checked.** We proved VeriStar sound with a machine-checked proof in Coq. Soundness means that when the prover returns "valid", the entailment checked holds in an separation logic for C minor [21]. The separation logic is proved sound, in turn, with respect to C minor's operational semantics.
- **End-to-end.** C minor programs verified with VeriStar can be compiled to (PowerPC or x86) assembly by the semantics-preserving compiler CompCert. The end-to-end machine-checked proof ensures the absence of soundness bugs anywhere along the chain.
- **Efficient.** VeriStar implements a state-of-the-art decision procedure based on *paramodulation*, a variant of resolution (cf. Navarro Pérez and Rybalchenko [25]), and can be compiled using Coq's code extraction utility and the OCaml system to native code for nearly any architecture. It uses highly tuned verified functional data structures such as a new implementation of red-black trees to implement clause sets.
- **Modular.** Although VeriStar forms the core of the fully verified static analysis VeriSmall, its modular structure means it can be retargeted to third-party separation logics and retrofitted to ex-

To appear in *ICFP'12: The 17th ACM SIGPLAN International Conference on Functional Programming*, September 9–15, 2012, Copenhagen, Denmark.

isting static analysis tools such as Smallfoot [8]. As supporting evidence, we describe two alternative separation logics (Section 4) and demonstrate the integration of our prover into the original (Berdine et al.) Smallfoot system.

To the best of our knowledge, VeriStar is the first machine-checked theorem prover for separation logic that connects to a real-world operational semantics (CompCert C minor). The VeriStar architecture employs a novel abstraction of separation logic, the *Separation Logic Interface*, in order to separate the system's soundness proof from the details of the separation logic implementation, and thus increase modularity. More generally, the lessons we learned while building VeriStar—on the effectiveness of code extraction as an execution model for verified software, on the power of an elegant proof theory for reasoning about functional programs, and on the importance of modular interfaces to proofs—will inform the future construction of large, verified software toolchains from independent, machine-checked components.

We have evaluated VeriStar on a suite of separation logic entailments generated by the original Smallfoot tool during symbolic execution. On these "real-world" entailments, VeriStar's performance is comparable to that of Smallfoot's unverified entailment checker—both systems are fast enough. On a suite of artificial entailments designed to simulate the heap inconsistency checks often performed during symbolic execution, VeriStar actually outperforms Smallfoot on the majority of entailments. On the other hand, VeriStar is still a small system that lacks features found in more established theorem provers. The current implementation of VeriStar supports just four atomic predicates: the points-to predicate of separation logic describing the singleton heap, a predicate describing acyclic list segments, a predicate describing empty heaps and an equality predicate on program variables. This assertion language resembles Smallfoot's quite closely but does not yet permit general intermixing of predicates from other theories, as in SMT solvers. Finally, although VeriStar's performance is adequate for verification of small to medium-sized programs, it could be further improved by memoizing common terms in the clause database through techniques such as hash-consing, or by performing multiple inferences at once, as is done in some state-of-the-art equational theorem provers [23]. None of these limitations is insurmountable. We foresee few technical difficulties in adding support for user-defined nonspatial predicates and spatial predicates for other sorts of data structures such as trees. Switching to a more efficient term and clause representation will require straightforward engineering.

## 2. VeriStar by Example

Figure 1 presents the main components of the VeriStar theorem prover. To build an intuition for how the pieces fit together, consider the following (valid) VeriStar entailment

$$a \neq c \wedge b = d \wedge a \mapsto b * \mathsf{lseg}(b, c) * \mathsf{lseg}(b, d) \vdash \mathsf{lseg}(a, c)$$

which consists of two assertions separated by a *turnstile* ($\vdash$). The first assertion states that program variable $a$ does not equal $c$, $b$ equals $d$ and the heap contains a pointer from $a$ to $b$ and two list segments with heads $b$ and tails $c$ and $d$, while the assertion to the right of the turnstile states that the heap is just the list segment with head $a$ and tail $c$. The task of the theorem prover is either to show that this entailment is *valid*—that every model of the assertion on the left is a model of the assertion on the right—or to return a counterexample in the process.

Most existing theorem provers for separation logic (e.g., Smallfoot [8], SLAyer [9]) attack the entailment problem *top-down*, by systematically exploring proof trees rooted at the goal. Each step of a top-down proof is an entailment-level deduction justified by a validity-preserving inference rule.

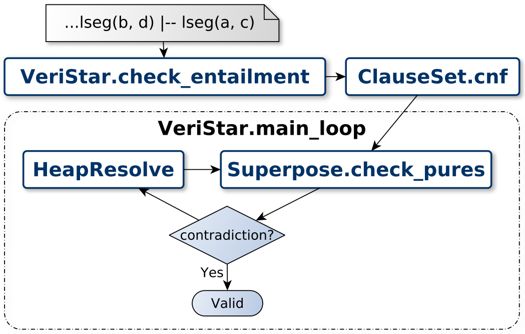


Figure 1: The main components of the VeriStar system, each of which is defined by a well-specified interface (Module Type) in Coq. Superpose and HeapResolve form the heart of the heap theorem prover, performing equational and spatial reasoning respectively. The ClauseSet module defines the clausal embedding of assertions as well as the prover's clause database using a tuned red-black tree implementation of the Coq MSets interface.

VeriStar, by contrast, is *bottom-up* and *indirect*. Instead of exploring proof trees rooted at the goal, it first decomposes the *negation* of the goal (hence indirect) into a logically equivalent set of *clauses* (its *clausal normal form*), then attempts to derive a contradiction from this set through the application of clausal inference rules. One can think of the clauses that form this initial set as a logically equivalent encoding of the original entailment into its atomic parts.

In particular, a VeriStar clause is a disjunction

$$(\pi_1 \vee \ldots \vee \pi_m) \vee (\overline{\pi'_1} \vee \ldots \vee \overline{\pi'_n}) \vee (\sigma_1 * \ldots * \sigma_r)$$

of positive pure literals $\pi$ (by *pure* we mean those that are heap-independent), negated pure literals $\overline{\pi'}$ and a spatial atom $\Sigma$ consisting of the star-conjoined simple spatial atoms $\sigma_1 * \ldots * \sigma_r$. The atom $\Sigma$ may be negated or may occur positively but not both: we never require clauses containing two atoms $\Sigma$ and $\Sigma'$ of different polarities. We write positive spatial clauses (those in which $\Sigma$ occurs positively) as $\Gamma \rightarrow \Delta, \Sigma$, where $\Gamma$ and $\Delta$ are sets of pure atoms and $\Sigma$ is a spatial atom, and use analogous notation for pure and negative spatial clauses. For example, in negative spatial clauses, $\Sigma$ appears to the left of the arrow ($\Gamma, \Sigma \rightarrow \Delta$), and in pure clauses $\Sigma$ does not appear at all ($\Gamma \rightarrow \Delta$). The *empty clause* $\emptyset \rightarrow \emptyset$ has no model because on the left, the conjunction of no clauses is True, and on the right, the disjunction of no clauses is False. Clauses such as $\Gamma \rightarrow a = a, \Delta$ and $\Gamma, a = b \rightarrow a = b, \Delta$ are tautologies.

To express the negation of the entailment as a set of clauses, VeriStar passes the entailment to ClauseSet.cnf (Figure 1), which takes advantage of the fact that it can encode any positive atom $\pi$ as the positive unit clause $\emptyset \rightarrow \pi$ and any negative atom $\overline{\pi'}$ as the negative unit clause $\pi' \rightarrow \emptyset$. It can do the same for negative and positive spatial atoms. Since the negation of any entailment $F \vdash G$ is equivalent, classically, to $F \wedge \neg G$, the original entailment becomes:

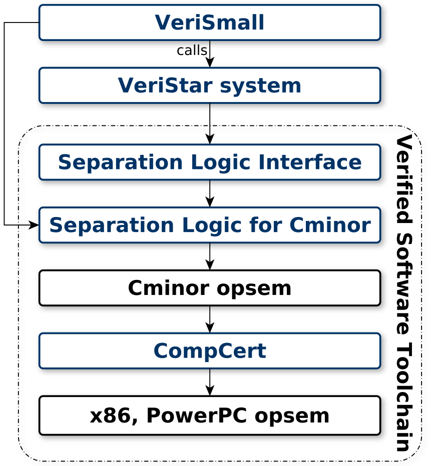
$$a = c \rightarrow \emptyset \tag{1}$$

$$\emptyset \rightarrow b = d \tag{2}$$

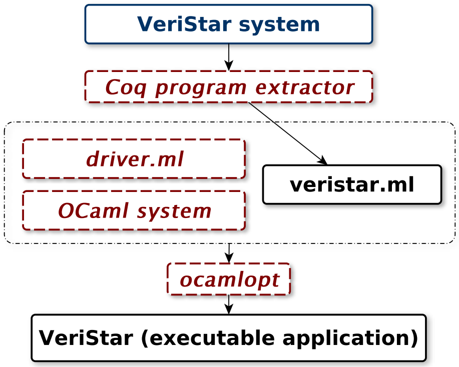
$$\emptyset \rightarrow a \mapsto b * \mathsf{lseg}(b, c) * \mathsf{lseg}(b, d) \tag{3}$$

$$\mathsf{lseg}(a, c) \rightarrow \emptyset \tag{4}$$

Here the spatial atom $\mathsf{lseg}(a, c)$ appears to the left of the arrow in clause (4) since it appears in the right-hand side of the original entailment. Likewise, the spatial atom $a \mapsto b * \mathsf{lseg}(b, c) * \mathsf{lseg}(b, d)$


VeriSmall
calls
VeriStar system
Separation Logic Interface
Separation Logic for Cminor
Cminor opsem
CompCert
x86, PowerPC opsem
Verified Software Toolchain

(a) Integration into the Verified Software Toolchain, yielding an end-to-end soundness proof



(b) Compilation toolchain

Figure 2: VeriStar's soundness proof (2a) and compilation toolchain (2b). Trusted components (dashed in red) are those that must be understood by the user to have confidence in the system. Verified components (solid in blue) have machine-checked correctness proofs. Cminor opsem and x86, PowerPC opsem are axiomatic definitions of the Cminor language and CompCert target languages respectively. When connected to the Verified Software Toolchain (2a), machine-checked proofs from VeriSmall/VeriStar through CompCert to assembly provide a foundational correctness guarantee with respect to the operational semantics of CompCert's target languages, x86 and PowerPC assembly. The modular construction of the soundness proof through the Separation Logic Interface facilitates retargeting VeriStar to third-party separation logics. VeriSmall is proved directly with respect to the C minor separation logic, and therefore is slightly less portable. In Figure 2b, we use Coq's extraction mechanism and the OCaml system to compile VeriStar to an executable application.

appears to the right of the arrow in clause (3) since it appears in the left-hand side of the original entailment.

After encoding the entailment as a set of clauses, VeriStar enters its main loop (VeriStar.main_loop in Figure 1). First, it filters the *pure* clauses from the initial clauseset (clauses (1) and (2) above), then passes these clauses to Superpose.check_pures, the pure prover. Superpose attempts to derive a contradiction from the pure clauses by equational reasoning. In this case, however, Superpose is unable to derive a contradiction, or indeed, any new clauses at all from the set, so it constructs a model of the pure clauses by setting $b$ equal to $d$ (completeness of the superposition calculus guarantees that this model exists) and passes the model, along with the current clauseset, to HeapResolve for spatial normalization and unfolding.

HeapResolve uses the fact that $b$ equals $d$ in the model as a hint to normalize the spatial clauses (3) and (4) by clause (2), resulting in the new spatial clause

$$\emptyset \to a \mapsto d * \mathsf{lseg}(d, c) * \mathsf{lseg}(d, d) \qquad (5)$$

in which $b$ has been rewritten to $d$ and therefore no longer appears. But now the spatial prover recognizes that since list segments are acyclic, $\mathsf{lseg}(d, d)$ can hold only if it denotes the empty heap. Thus $\mathsf{lseg}(d, d)$ can be simplified to emp, resulting in the new clause

$$\emptyset \to a \mapsto d * \mathsf{lseg}(d, c). \qquad (6)$$

This new clause can almost be resolved against clause (4) using *spatial resolution*—an inference rule allowing negative and positive occurrences of spatial atoms in two different clauses to be eliminated—but only if clause (4) is unfolded to accommodate the next atom $a \mapsto d$ in clause (6). Unfolding $\mathsf{lseg}(a, c)$ to $a \mapsto d * \mathsf{lseg}(d, c)$ is sound, in turn, only when $\mathsf{lseg}(a, c)$ is nonempty, i.e., when $a \neq c$. To encode this fact, HeapResolve generates the new clause

$$a \mapsto d * \mathsf{lseg}(d, c) \to a = c. \qquad (7)$$

Clause (7) can then be resolved with clause (6) to produce the positive unit clause

$$\emptyset \to a = c. \qquad (8)$$

Superpose now resolves clause (8) with clause (1) to derive the empty clause $\emptyset \to \emptyset$, which is unsatisfiable. Since the inference rules of the HeapResolve and Superpose systems preserve all models, the original set of clauses encoding the negation of the entailment VeriStar set out to prove is unsatisfiable; the entailment must therefore be valid.

### 2.1 Overview of the Rest of the Paper

In the next section, we introduce VeriStar in the context of the Verified Software Toolchain [3], a series of tightly integrated machine-verified components that connect end-to-end to yield foundational correctness guarantees. We also describe VeriStar's execution model. Sections 4 and 5 give the technical details of our model of separation logic and the VeriStar implementation and its soundness proof in Coq. Section 7 makes a case for machine-checked proofs, with examples from this case study. Section 8 describes our experience optimizing VeriStar. In Section 9, we evaluate the relative sizes in lines-of-code of the components of the prover and measure VeriStar's performance on a suite of benchmarks.

## 3. The Verified Software Toolchain

The Verified Software Toolchain [3] connects machine-checked program analyses to machine-checked program logics; the logics are connected to machine-checked compilers such as Leroy's CompCert, giving an end-to-end result.

Figure 2a puts VeriSmall and VeriStar in the context of the current instantiation of the Verified Software Toolchain: VeriStar is proved sound with respect to an abstract axiomatization of separation logic, the *Separation Logic Interface* (Section 4.2). VeriSmall is proved sound directly with respect to the C minor separation logic. Of course, VeriSmall's soundness proof must rely on that of VeriStar since VeriSmall frequently calls the prover to decide entailments during symbolic execution.

We instantiate the Separation Logic Interface with Hobor et al.'s separation logic for C minor [21], which has a machine-checked soundness proof in Coq with respect to CompCert C minor's operational semantics. Because CompCert preserves the semantics of safe C minor programs, properties proved at the source level using VeriSmall/VeriStar will hold of the generated assembly. Furthermore, although the operational semantics of C minor and those of CompCert's intermediate languages play a role in the end-to-end proof, only the operational semantics of the target languages, PowerPC and x86 assembly, must be trusted since the compiler's correctness theorem makes a claim directly about the behavior of the target program.

### 3.1 Execution via Extraction

We use Coq's extraction utility to generate OCaml code (veristar.ml, Figure 2b) for VeriStar. A small, trusted translation from Smallfoot-style entailments to VeriStar entailments (driver.ml) allows our modified Smallfoot to call VeriStar.check_entailment as a subroutine, thus replacing Smallfoot's standard entailment checker with a formally verified one and reducing the size of Smallfoot's trusted computing base by approximately 20% (modulo correctness of the Coq typechecker[1]). When connected to the machine-checked static analysis VeriSmall, VeriStar's trusted computing base is even smaller: just the Coq typechecker and CompCert's specifications of either PowerPC or x86 assembly.

## 4. Separation Logic Semantics

To ensure VeriStar can be retargeted to separation logics for a variety of languages and compiler frameworks, we proved the system sound with respect to an *abstract model* of separation logic. We first defined a generic Separation Logic Interface (Section 4.2) specifying the operators of separation logic on which the proof depends. We then constructed an abstract model of separation logic generically for any concrete implementation satisfying the interface (Section 4.3). We have instantiated the interface with two such implementations, Hobor et al.'s Separation Logic for C minor [21] (Appendix C) and a bare-bones implementation (Appendix B), but expect the interface, and hence VeriStar's soundness proof, to be general enough to be widely applicable.

### 4.1 The Assertion Language

Atomic assertions in VeriStar (Figure 3) denote equalities and inequalities of program variables, singleton heaps and acyclic list segments. The assertion emp denotes the empty heap. The assertion $a \mapsto b$ (Next $a$ $b$ in VeriStar syntax) denotes the heap containing just the value of variable $b$ at the location given by $a$ (and is empty everywhere else), while Lseg $a$ $b$ denotes the heap containing the acyclic list segment with head pointer $a$ and tail pointer $b$. Equalities and inequalities of variables are *pure* assertions because they make no reference to the heap, whereas $a \mapsto b$ and Lseg are *spatial* assertions.

A complex assertion $\Pi \wedge \Sigma$ is the conjunction of the pure atoms $\Pi$ with the *separating conjunction* of the spatial atoms $\Sigma$. The separating conjunction $\sigma_1 * \sigma_2$ (also called *star*) of two assertions—a notion from separation logic—is satisfied by any heap splittable into two disjoint subheaps satisfying $\sigma_1$ and $\sigma_2$, respectively. The assertion $\Pi \wedge \Sigma$ is satisfied by any environment $e$ and heap $h$ such that $e$ satisfies all the assertions in $\Pi$ and the pair $(e, h)$ satisfies the separating conjunction of the assertions in $\Sigma$. Entailments $F \vdash G$ are valid whenever all the models satisfying $F$ also satisfy $G$, i.e.: $\forall(e, h).\ F(e, h) \rightarrow G(e, h)$.

| | |
|---|---|
| **Expressions** $a$, $b$ | |
| Nil | null pointer |
| Var $x$ | Program variable |
| **Pure Atoms** $\pi$ (pn_atom) | |
| Equ $a$ $b$ | Expression $a$ equals $b$. |
| Nequ $a$ $b$ | Expression $a$ does *not* equal $b$. |
| **Spatial Atoms** $\sigma$ (space_atom) | |
| emp | Empty heap |
| Next $a$ $b$ | Singleton heap with $a \mapsto b$ |
| Lseg $a$ $b$ | Acyclic list segment from $a$ to $b$ |
| **Assertions** $F$, $G$ | |
| Assertion $\Pi$ $\Sigma$ | Pairs of pure atoms $\Pi$ and spatial atoms $\Sigma$ |
| **Entailments** $ent$ | |
| Entailment $F$ $G$ | Assertion $F$ implies $G$. |

Figure 3: VeriStar syntax

### 4.2 The Interface

We present a selection of the components of the Separation Logic Interface in Listing 1. The interface axiomatizes the types of locations loc and values val; the special values nil_val, corresponding to the null pointer, and empty_val, corresponding to undefined (i.e., not in the domain of a given heap); an injection val2loc from values to locations; the types of variable environments env and heaps (heap) and a points-to operator on heaps (rawnext).

We assume a *separation algebra* [19] on values (Sep_alg val), meaning that in addition to the operators on values specified in the interface (e.g., val2loc) we may use the *join* operator, written $\oplus$, to describe the union of two disjoint values. In simple separation logics, $v_1 \oplus v_2$ is defined only when either $v_1$ or $v_2$ is the empty_val (that is, two nonunit values are never disjoint). However, a more refined separation algebra on values, say with shares denoting read and write permissions, is often useful in concurrent separation logic. Our interface and the soundness proof are indifferent to the separation algebra actually used.

The parameter heap gives the type of program memories. As with val, we require a separation algebra on heaps. We also require two operators on heaps, rawnext, a low-level version of the $\mapsto$ predicate of separation logic, and emp_at $(l{:}loc)$ $(h{:}heap)$, which defines when a heap $h$ is empty at a location $l$. The behavior of these operators is defined by a series of axioms. For example, the axiom rawnext_out asserts that the heap rawnext $l$ $v$ is empty everywhere except at location $l$ (i.e., it is a singleton heap). The constructor mk_heap_rawnext allows one to construct new singleton heaps. In the definition of mk_heap_rawnext, comparable $h$ $h'$ means that $h$ and $h'$ share the same unit in our multi-unit separation algebras (thus they are *comparable*).[2] The assertion rawnext$'$ $l$ $v$ extends rawnext $l$ $v$ to any heap that contains $l \mapsto v$ as a subheap. The behavior of rawnext$'$ is given by a series of axioms not shown in Listing 1 but given in the code.

### 4.3 The Abstract Model

We defined our abstract separation logic model with respect to the opaque interface of Listing 1. In our Coq implementation, this model is literally a functor over modules satisfying the interface: we make

[1] and the Coq program-extractor, and the OCaml compiler that compiles both veristar.ml and driver.ml, and the C compiler that compiles OCaml's runtime system... For a discussion of these issues, see [3, Section 11].

[2] Dockins et al. [19] describes why multi-unit separation algebras are preferable to standard, single-unit ones.

```
Module Type VERISTAR_LOGIC.
(*Locations and values*)
Parameters loc val : Type.
Declare Instance Sep_val : Sep_alg val.
Parameter val2loc : val → option loc.
Parameter nil_val : val.
Parameter empty_val : val.

(*Environments*)
Parameter env : Type.
Parameter env_get : env → var → val.
Parameter env_set : var → val → env → env.
Axiom gss_env : ∀(x : var) (v:val) (e:env),
  env_get (env_set x v e) x = v.
Axiom gso_env : ∀(x y : var) (v:val) (e:env),
  x ≠ y → env_get (env_set x v e) y = env_get e y.
Parameter empty_env : env.

(*Heaps*)
Parameter heap : Type.
Declare Instance Sep_heap: Sep_alg heap.
Parameter rawnext : ∀(x:loc) (y:val) (e:heap), Prop.
Parameter emp_at : ∀(l:loc) (h:heap), Prop.
Definition nil_or_loc (v:val) :=
  v=nil_val ∨ ∃l:loc, val2loc v = Some l.
Axiom mk_heap_rawnext : ∀h x0 x y,
  val2loc x0 = Some x → nil_or_loc y →
  ∃h', rawnext x y h' ∧ comparable h h'.
Axiom rawnext_out : ∀x x0 x' y h,
  rawnext x y h → val2loc x0 = Some x' →
  x' ≠ x → emp_at x' h.
Definition rawnext' x y h :=
  ∃h0, join_sub h0 h ∧ rawnext x y h0.

(*Further parameters and axioms are elided.*)
End VERISTAR_LOGIC.
```

Listing 1: Selected values, operators and their properties from the Separation Logic Interface. The abstract types val, loc, var, env and heap are interface parameters.

no assumptions in the proof about the underlying module beyond those defined in the interface, thus increasing portability.

States are pairs of environments $e$ and heaps $h$.

```
Inductive state := State: ∀(e:env) (h:heap), state.
```

The Coq keyword **Inductive** declares a new inductively defined datatype with, in this case, a single constructor named State. State takes as parameters an environment $e$ and a heap $h$. In more conventional ML-like notation, this type is equivalent to the product type State **of** (env ∗ heap). Predicates on states, called spreds, are functions from states to Prop.

```
Notation spred := (state → Prop).
```

One can think of Prop as the type of truth values True and False, analogous to bool, except that predicates in Prop need not be decidable and are erased during program extraction. Thus, we use Prop in our proofs, but bool in the verified code. A Coq **Notation** simply defines syntactic sugar. The interpretations of expressions (expr_denote), expression equality (expr_eq) and pure atoms (pn_atom_denote) are standard.

List segments lseg are defined by an inductive type with two constructors.

```
Inductive lseg : val → val → heap → Prop :=
| lseg_nil : ∀x h, emp h → nil_or_loc x → lseg x x h
| lseg_cons : ∀x y z l h0 h1 h,
    x ≠ y → val2loc x = Some l → rawnext l z h0 →
    lseg z y h1 → join h0 h1 h → lseg x y h.
```

The lseg_nil constructor forms the trivial list segment whose head and tail pointers are equal and whose heap is emp. The lseg_cons constructor builds a list segment inductively when $x$ does not equal $y$, $x$ is injected to a location $l$ such that $l \mapsto z$, and there is a sub-list segment from $z$ to $y$.

The function space_atom_denote maps syntactic spatial assertions such as Lseg $x$ $y$ to their semantic counterparts (i.e., lseg $x$ $y$).

```
Definition space_atom_denote (a: space_atom) : spred :=
  match a with Next x y ⇒ fun s ⇒
    match val2loc (expr_denote x s) with
    | None ⇒ False
    | Some l ⇒
        rawnext l (expr_denote y s) (hp s) ∧
        nil_or_loc (expr_denote y s)
    end
  | Lseg x y ⇒ fun s ⇒
      lseg (expr_denote x s) (expr_denote y s) (hp s)
  end.
```

For Next $x$ $y$ assertions, it injects the value of the variable $x$ to a location $l$ and requires that the heap contain just the location $l$ with value $v$ (that is, the heap must be the singleton $l \mapsto v$), where $v$ is the interpretation of variable $y$. Coq's **match** syntax does case analysis on an inductively defined value (here the space atom $a$), defining a distinct result value for each constructor.

An Assertion $\Pi$ $\Sigma$ is the conjunction of the pure atoms $\pi \in \Pi$ with the separating conjunction of the spatial atoms $\sigma \in \Sigma$.

```
Definition assertion_denote (f:assertion) : spred :=
  match f with Assertion Π Σ ⇒
      fold pn_atom_denote andp (space_denote Σ) Π
  end.
```

The function space_denote interprets the list of spatial atoms $\Sigma$ as the *fold* of space_atom_denote over the list, with unit emp. Thus (space_denote $\Sigma$) is equivalent to

$$\left(\circledast_{\sigma\in\Sigma}\text{space\_atom\_denote}(\sigma)\right) * \text{emp}$$

(where $\circledast$ is iterated separating conjunction) and the denotation of Assertion $\Pi$ $\Sigma$ is

$$\bigwedge_{\pi\in\Pi}\text{pn\_atom\_denote}(\pi) \wedge \left(\circledast_{\sigma\in\Sigma}\text{space\_atom\_denote}(\sigma)\right)$$

if one simplifies $P * \text{emp}$ to $P$ (recognizing that emp is the unit for $*$). Here space_denote $\Sigma$ is the unit of the fold. Entailments from $F$ to $G$ are interpreted as the semantic entailment of the two assertions.

## 5. The VeriStar Algorithm

A key strength of the Navarro Pérez and Rybalchenko algorithm is that it splits the theorem prover into two modular components: the equational theorem prover for pure clauses (Superpose) and the spatial reasoning system HeapResolve, which calls Superpose as a subroutine in between rounds of spatial inference. This modular structure means well-studied techniques from equational theorem proving can be applied to the equational prover in isolation, while improving the performance of the heap theorem prover as a whole.

```
1   Function main_loop
2     ($n$: positive) ($\Sigma$: list space_atom) ($ncl$: clause) ($S$: M.t)
3     {measure nat_of_P $n$} :=
4     if Coqlib.peq $n$ 1 then Aborted (M.elements $S$) else
5     match Superpose.check_pures $S$ with
6     | (Superpose.Valid, $units$, _, _) $\Rightarrow$ Valid
7     | (Superpose.C_example $R$ $sel$, $units$, $S^*$, _) $\Rightarrow$
8       let $\Sigma'$ := simplify_atoms $units$ $\Sigma$ in
9       let $ncl'$ := simplify $units$ $ncl$ in
10      let $c$ := norm $sel$ (PosSpaceClause nil nil $\Sigma'$) $R$ in
11      let $S_1$ := incorp (do_wellformed $c$) $S^*$ in
12      if isEq (M.compare $S_1$ $S^*$)
13      then if is_model_of_$\Pi$ (List.rev $R$) $ncl'$
14           then let $c'$ := norm $sel$ $ncl'$ in
15                let $us$ := pures (unfolding $c$ $c'$) in
16                let $S_2$ := incorp $us$ $S_1$ in
17                if isEq (M.compare $S_1$ $S_2$) then C_example $R$
18                else main_loop (Ppred $n$) $\Sigma'$ $ncl'$ $S_2$ $c$
19           else C_example $R$
20      else main_loop (Ppred $n$) $\Sigma'$ $ncl'$ $S_1$ $c$
21    | (Superpose.Aborted $l$, $units$, _, _) $\Rightarrow$ Aborted $l$
22    end.
23  Proof.
24  (*Termination proof here, that $n$ decreases*)
25  Defined.
26
27  Definition check_entailment ($ent$: entailment) :=
28    let $S$ := pure_clauses (map order_eqv_clause (cnf $ent$)) in
29    match ent with
30    | Entailment (Assertion $\Pi$ $\Sigma$) (Assertion $\Pi'$ $\Sigma'$) $\Rightarrow$
31        match mk_pureR $\Pi$, mk_pureR $\Pi'$ with
32        | ($\Pi_+$, $\Pi_-$), ($\Pi'_+$, $\Pi'_-$) $\Rightarrow$
33            main_loop $m$ $\Sigma$ (NegSpaceClause $\Pi'_+$ $\Sigma'$ $\Pi'_-$)
34              (clause_list2set $S$)
35        end
36    end.
```

Listing 2: The main VeriStar procedures

In this section, we describe our verified implementation of the algorithm of Navarro Pérez and Rybalchenko and give an outline of its soundness proof in Coq.

### 5.1 Overview of the Algorithm

Listing 2 defines the main procedures of the VeriStar system, in slightly simplified form (we have commented out the termination proof for main_loop, line 24). The first step is to encode the entailment, $ent$, as a set of clauses (its *clausal normal form*, line 28). The algorithm then enters its main loop, first calling Superpose.check_pures (line 5) on the current set of pure clauses $S$, a subset of the clauses that encode $ent$, and checking whether the equational prover was able to derive the empty clause from this set. If it was, the algorithm terminates with Valid (line 6). Otherwise, Superpose returns with a model $R$ of the set of pure clauses (line 7) and a list of unit clauses $units$ derived during superposition inference (also line 7). VeriStar first rewrites the spatial atoms $\Sigma$ and spatial clause $ncl$ by $units$ (lines 8-9), then normalizes the rewritten positive spatial atom $\Sigma'$ using the model $R$ (line 10). It then adds any new pure clauses implied by the spatial wellformedness rules to the pure set (line 11). This process repeats until it converges on a fixed point (or the prover aborts abnormally; see Section 7 for details). Once a fixed point is reached, more normalization of spatial atoms is performed (line 14), and unfolding of lsegs is attempted (line 15), possibly generating new pure clauses to feed back into the loop. If no new pure clauses are generated during this process, the algorithm terminates with a counterexample.

### 5.2 HeapResolve for Spatial Reasoning

VeriStar divides spatial reasoning (lines 10-15 in Figure 2) into four major stages: normalization of spatial atoms, wellformedness inference, unfolding of list predicates and spatial resolution.

*Normalization* rules perform substitutions into spatial atoms based on pure facts inferred by the superposition system, as well as eliminate obviously redundant list segments of the form $\mathsf{lseg}(x, x)$.

*Wellformedness* rules generate new pure clauses from malformed spatial atoms. Consider, for example, the clause

$$\Gamma \to \Delta, \mathsf{lseg}(x, y) * \mathsf{lseg}(x, z)$$

which asserts that $\Gamma$ implies the disjunction of $\Delta$ and the spatial formula $\mathsf{lseg}(x, y) * \mathsf{lseg}(x, z)$. Since the separating conjunction in the spatial part requires that the two list segments be located in disjoint subheaps, we know that the list segments cannot both start at location $x$ unless one of the list segments is empty. However, we do not know which one is empty.[3] To formalize this line of reasoning, VeriStar generates the clause

$$\Gamma \to x = y, x = z, \Delta$$

whenever it sees a clause with two list segments of the form given above. This new clause states that $\Gamma$ implies either $\Delta$ (the positive pure atoms from the original clause) or $x = y \vee x = z$. The other wellformedness rules allow VeriStar to learn entirely pure facts from spatial facts in much the same way.

The spatial *unfolding* rules formalize the notion that nonempty list segments can be unfolded into their constituent parts: a points-to fact and a sub-list segment, or in some cases, two sub-list segments. List segments should not be unfolded *ad infinitum*, however—it would be sound to do so, but our algorithm would infinite-loop. Instead, VeriStar performs unfolding only when certain other spatial facts are present in the clause database. These *hints* or triggers for rule application are key to making the proof procedure tractable.

As an example, consider Navarro Pérez and Rybalchenko's inference rule U3

$$\frac{\Gamma \to \Delta, \mathsf{lseg}(x, y) * \Sigma \qquad \Gamma', \mathsf{lseg}(x, \mathsf{nil}) * \Sigma' \to \Delta'}{\Gamma', \mathsf{lseg}(x, y) * \mathsf{lseg}(y, \mathsf{nil}) * \Sigma' \to \Delta'}$$

which states that list segments $\mathsf{lseg}(x, \mathsf{nil})$ in negative positions should be unfolded to $\mathsf{lseg}(x, y) * \mathsf{lseg}(y, \mathsf{nil})$, but only when there is a positive spatial clause somewhere in the clause database that mentions $\mathsf{lseg}(x, y)$. In this rule, the left-hand side clause $\Gamma \to \Delta, \mathsf{lseg}(x, y) * \Sigma$ is unnecessary for soundness but necessary operationally for limiting when the rule is applied.

Our Coq implementation of this rule follows the declarative version rather closely.

```
Definition unfolding3 (sc1 sc2:clause) :=
  match sc1, sc2 with
  | PosSpaceClause $\Gamma$ $\Delta$ $\Sigma$, NegSpaceClause $\Gamma'$ $\Sigma'$ $\Delta'$ $\Rightarrow$
    let $l_0$ := unfolding3' nil $\Sigma$ $\Sigma'$ in
    let build_clause $\Sigma_0$ := NegSpaceClause $\Gamma'$ $\Sigma_0$ $\Delta'$ in
      map build_clause $l_0$
  | _, _ $\Rightarrow$ nil
  end.
```

Here unfolding3' is an auxiliary function that searches for and unfolds list segments from variable $x$ to Nil in $\Sigma'$ with counterpart lists of the appropriate form in $\Sigma$.

[3] The *spooky disjunction* of Berdine et al. [8].

Finally, VeriStar performs *spatial resolution* of spatial atoms that appear both negatively and positively in two different clauses.

$$\frac{\Gamma, \Sigma \to \Delta \qquad \Gamma' \to \Delta', \Sigma}{\Gamma, \Gamma' \to \Delta, \Delta'}$$

Like the wellformedness rules, spatial resolution makes it possible to infer new pure facts from clauses with spatial atoms, in the special case in which $\Sigma$ occurs both positively and negatively in two different clauses.

### 5.3 Superposition of Pure Clauses

In this section, we briefly describe our implementation of Bachmair and Ganzinger's System S [6], the superposition calculus with selection. We chose System S because it is a well-studied equational calculus that appears to perform well in practice but there are others (see, for instance, Nieuwenhuis and Rubio's System I [31]). System S operates by repeatedly applying inference rules of the form

$$\frac{\Gamma \to x = y, \Delta \qquad \Gamma' \to x = z, \Delta'}{\Gamma, \Gamma' \to y = z, \Delta, \Delta'}\ \text{PS}$$

to sets of clauses. The rule PS (positive superposition) implements the clausal form of replacement of equals with equals (i.e., substitution) in positive positions. System S includes rules for substitution in negative positions and equality factoring as well.

The main superposition procedure, check_pures, operates internally on two sets of clauses, the given set and the unselected set. The given set contains those clauses that were chosen to participate in superposition inference at least once in the past. The unselected set contains whichever clauses are left. At the beginning of the search, all clauses are in the unselected set and the given set is empty. At each step of the superposition procedure, a new clause is chosen from the unselected set (the given_clause). This clause may be chosen uniformly, but we instead apply a simple heuristic that greatly improves the search: choosing the smallest clause first. Intuitively, this optimization is profitable because it favors the generation of small clauses over large ones, and the ultimate goal of the search is to produce the empty clause. Once the given_clause has been chosen, we simplify it with respect to the current clauses in the given set, then perform all superposition inferences possible for $c$, the resulting simplified clause, and the given set extended with $c$. Simplification essentially rewrites the given clause by all the unit equalities in the given set. Any new clauses inferred in this process are added back to the unselected set and the process is repeated until either the empty clause is derived or a fixed point is reached.

## 6. Soundness End-to-end

One would hope that the modular structure of the prover lends itself to a modular soundness proof: that is, each component of the prover is shown sound in isolation and these verified modules are stitched together to prove the soundness of the entire system end-to-end. Of course, for this strategy to work the functionality and correctness of each component must be guarded by a narrow interface via a module type. Otherwise, maintenance to the prover and its soundness proof becomes overwhelming.

We employed exactly this strategy while proving the soundness of VeriStar and found that it greatly simplified the initial construction of the soundness proof and the rounds of optimizations we performed thereafter, each of which required changes both to the prover and to its soundness proof. To facilitate a modular structure, we divided the prover into the following major components:

- Clausal normal form encoding of entailments;
- Superposition;
- Spatial normalization;
- Spatial wellformedness inference rules;
- Spatial unfolding rules; and
- Model generation and selection of clauses for normalization.

Each of these components was then proved sound with respect to a minimal interface.

As an example of one such interface, the main soundness theorem for the clausal normal form encoding states that the negation of the clausal normal form of an entailment is equivalent to the original entailment before it was encoded as a clauseset.

```
Theorem cnf_correct: ∀(e:entailment),
  entailment_denote e ↔
  ∀(s:state), ¬(fold clause_denote andp TT (cnf e) s).
```

Here the notation fold $f$ andp TT $l$ $s$ means $\bigwedge_{x \in l}(f\ x\ s)$. TT is the *always true* predicate. The function clause_denote defines our interpretation of *clauses*, i.e., disjunctions of pure and spatial atoms. Theorem cnf_correct is the only theorem about the clausal normal form encoding that we expose to the rest of the soundness proof, thus limiting the exposure of the rest of the proof to isolated updates to the cnf component.

Likewise, the main soundness theorem for the superposition system states that if Superpose.check_pures was able to derive the empty clause from a set of clauses $init$, then the conjunction of the clauses in $init$ entails the empty_clause.

```
Theorem check_pures_Valid_sound: ∀init units g u,
  check_pures init = (Valid, units, g, u) →
  fold clause_denote andp TT (M.elements init)
    ⊢ clause_denote empty_clause.
```

We need an additional theorem for Superpose, however, since the pure prover may return C_example for some clausesets, in addition to those for which it returns Valid. In the counterexample case, VeriStar constructs a model for the pure clauses, then uses this model to normalize spatial ones. Any clauses inferred by the pure prover while it was searching for the empty clause must therefore be entailed by the initial set of clauses.

```
Theorem check_pures_Cexample_sound:
 ∀init units final empty R sel,
 check_pures init = (C_example R sel, units, final, empty) →
 fold clause_denote andp TT (M.elements init)
   ⊢ fold clause_denote andp TT (M.elements sel) &&
       fold clause_denote andp TT (M.elements final) &&
         fold clause_denote andp TT units.
```

***To prove the soundness*** of VeriStar.check_entailment, the main function exported by the prover (Listing 2), we made each of the components described above a functor over our abstract separation logic model, VERISTAR_MODEL. As we described in Section 4.3, our abstract model is itself a functor over modules satisfying the VERISTAR_LOGIC interface of Listing 1. VERISTAR_MODEL—and by extension, our soundness proof—is therefore entirely parametric in the low-level details of the target separation logic implementation (e.g., the definition of the *maps-to* operator).

In the main soundness proof for VeriStar.check_entailment, we imported the soundness proof for each component, instantiated each of the functors by Vsm:VERISTAR_MODEL, then composed the soundness theorems exported by each component to prove the main correctness theorem, check_entailment_sound.

```
Module VeriStarSound (Vsm:VERISTAR_MODEL).
  Module SPS := SP_Sound Vsm. (*Superposition*)
  Module NS  := Norm_Sound Vsm. (*Normalization*)
  ...
  Module WFS := WF_Sound Vsm. (*Wellformedness*)
  Module UFS := UF_Sound Vsm. (*Unfolding*)

  Theorem check_entailment_sound: ∀(ent:entailment),
      VeriStar . check_entailment ent = Valid →
      entailment_denote ent.
End VeriStarSound.
```

check_entailment_sound states that if the prover returns Valid, the original entailment is semantically valid in the Vsm model. Because of VeriStar's modular design, the proof of this theorem goes by a straightforward application of the soundness lemmas for each of the subcomponents.

### 6.1 Specialization to C minor

To target the soundness proof to C minor, we built an implementation of the VERISTAR_LOGIC interface for C minor addresses, values, local variable environments and heaps (CminLog). We instantiated our abstract separation logic by this module

```
Module Cmm:VERISTAR_MODEL:=VeriStarModel CminLog.
```

then applied VeriStarSound to Cmm,

```
Module Vss : VERISTAR_SOUND := VeriStarSound Cmm.
```

yielding an end-to-end proof. Here the module CminLog defines the operators and predicates on environments and heaps (env_get, env_set, rawnext, etc.) required by our soundness proof, and proves all of the required properties for these operators and predicates.

***The main soundness theorem*** for the VeriSmall static analyzer, check_sound,

```
Theorem check_sound : ∀Γ P c Q,
   check0 P c Q = true →
   semax Γ (assertion2wpred P) (erase_stmt c)
              (RET1 (assertion2wpred Q)).
```

relies on Vss to prove that calls made to VeriStar during symbolic execution are valid. The theorem states that when VeriSmall successfully checks a Hoare triple (check0 $P$ $c$ $Q$ = true), the triple is sound in our axiomatic semantics for C minor (semax).[4] The axiomatic semantics, in turn, has a machine-verified correctness proof with respect to the operational semantics of CompCert C minor. Thus we achieve an end-to-end correctness guarantee: C minor programs deemed safe by the static analyzer will be compiled by CompCert to observationally equivalent assembly programs.

## 7. Why bother with machine-checked proofs?

It takes some effort to encode an algorithm in a proof assistant like Coq and then prove it correct with a machine-checked proof. One might wonder whether all this effort is really worth it. That is, do we gain anything—over LaTeXproofs and unverified implementations—by formally proving an implementation of an algorithm correct?

***Soundness.*** In this case study, we can concretely say "yes". Formal verification of VeriStar uncovered two related soundness bugs in Navarro Pérez and Rybalchenko's spatial unfolding rules (specifically, rules U4 and U5 in [25, Section 4, Fig. 1]).

It appears likely that because of the interaction of rules U4 and U5 with the spatial resolution rule, these bugs did not result in unsoundness of Navarro's implemented system. However, we have been unable to verify that this is the case since we lack access to the source code (and, of course, the absence of such bugs cannot be confirmed even by extensive testing). We *have* verified the soundness of corrected forms of U4 and U5, discovered independently by us and Navarro Pérez and Rybalchenko. We present the corrected rules here.

$$\mathsf{U4}'\ \frac{\Gamma \to \Delta, \mathsf{lseg}(x,y) * \mathsf{next}(z,w) * \Sigma \qquad \Gamma', \mathsf{lseg}(x,z) * \Sigma' \to \Delta'}{\Gamma, \Gamma', \mathsf{lseg}(x,y) * \mathsf{lseg}(y,z) * \Sigma' \to \Delta, \Delta'}\ y \not\equiv z$$

$$\mathsf{U5}'\ \frac{\Gamma \to \Delta, \mathsf{lseg}(x,y) * \mathsf{lseg}(z,w) * \Sigma \qquad \Gamma', \mathsf{lseg}(x,z) * \Sigma' \to \Delta'}{\Gamma, \Gamma', \mathsf{lseg}(x,y) * \mathsf{lseg}(y,z) * \Sigma' \to z = w, \Delta, \Delta'}\ y \not\equiv z$$

The new U4 and U5 rules required adding $\Gamma$ and $\Delta$ to the conclusion of each rule so that the succedent of the first hypothetical clause ($\Delta, \mathsf{lseg}(x,y) * \mathsf{next}(z,w) * \Sigma$ and $\Delta, \mathsf{lseg}(x,y) * \mathsf{lseg}(z,w) * \Sigma$ resp. in U4 and U5) could be made to hold in the model. By ensuring that $\mathsf{next}(z,w)$ (resp. $\mathsf{lseg}(z,w)$) be disjoint from $\mathsf{lseg}(x,y)$, we avoid the counterexample we found for the original system (without $\Gamma, \Delta$) in which $\mathsf{lseg}(x,z)$ does not hold because $z$ points back into $\mathsf{lseg}(x,y)$ (lists must be acyclic). Appendix A presents this counterexample, which has been confirmed by the authors of [25] in an email exchange, in more detail.

This modification to rules U4 and U5 was not obvious to us initially, before we attempted to verify the rules, since in the other unfolding rules in Navarro Pérez and Rybalchenko's system, the first hypothetical clause acts only as an operational trigger for unfolding and is never necessary for soundness.

***Termination.*** All Coq functions are total, so a computable function implemented in Coq must terminate. One convinces the Coq system that a function terminates either by presenting a *structurally* recursive function (using Coq's **Fixpoint** notation) in which all recursive calls are clearly on substructures of the corresponding formal parameter; or by presenting a general function (using Coq's **Function** notation) along with a proof that one of the arguments decreases in some well-founded order.

Navarro Pérez and Rybalchenko state their termination proof as follows: "[T]he algorithm terminates since the growing set $S$ is bounded by ... the finite number of distinct pure clauses which can be written with the constant symbols occurring in $E$." Unfortunately this proof has some weaknesses. In real implementations, including Navarro's and including our own, the set $S$ does not grow monotonically, because optimizations are implemented to rewrite by unit equalities and remove redundant clauses. What does seem to grow is the closure of $S$ under the addition of certain kinds of redundant clauses, but the proof of this is not at all straightforward.

We have implemented a machine-checked termination proof of the superpose loop. Termination of check_entailment is much trickier and we have not yet implemented that proof. Instead we resort to a common hack: we provide VeriStar's main loop with an additional numeric argument, and after a set number of iterations it times out. As usual when this hack is applied, it does not compromise the soundness proof: time-out does not return a result that demands soundness. Then we pass a time-out parameter that is sufficiently large for all conceivable applications. Still, even though we have not implemented a proof, we believe the algorithm does terminate, i.e. on any input, given a large enough $n$, it will not time out.

***Completeness.*** Navarro Pérez and Rybalchenko [25] also proved completeness: when the algorithm returns *counterexample*, the original entailment is invalid. We have not yet done so for our Coq implementation of their algorithm. For many applications of verified software, completeness is not quite as important as soundness—an attacker could exploit a soundness bug in the verification toolset, but not a completeness bug. Nevertheless, to formally prove com-

[4] Since VeriSmall and VeriStar operate on syntax, we must lift the syntactic assertions $P$ and $Q$ to semantic assertions operating on worlds of the program logic (assertion2wpred).

| **optimization** | **speedup** | program | proof | ratio |
|---|---|---|---|---|
| clausesets | **1.21x** | 305 | 4,213 | 13.8x |
| priority heuristic | **3.43** | 35 | 50 | 1.43 |
| priority caching | **1.26** | 66 | 113 | 1.71 |
| int31 | **1.39** | 180 | 471 | 2.62 |
| set ops | **1.04** | 183 | 326 | 1.78 |
| redundancy elim. | **1.40** | 47 | 45 | 0.96 |
| model-based saturation | **2.09** | 338 | 732 | 2.17 |
| **total:** | **22.1** | 1,154 | 5,950 | 5.16 |

Figure 4: Geomean speedups across a suite of 9,000 random separation logic entailments for the last six optimizations we performed. Columns *program* and *proof* show how many lines of code were modified to implement the improvement; *ratio* is proof/program.

pleteness of our implementation would confirm that we have implemented the right algorithm.

## 8. Performance Tuning Verified Software

There is no secret to writing efficient programs [7]: (1) Take a baseline; (2) optimize; (3) evaluate the results; and (4) repeat. It is perhaps no surprise that the same methodology can be applied to verified software in much the same way, except now we are working with *machine-verified* software and must update soundness and termination proofs as we perform each optimization.

In this section, we report on some optimizations we performed while building VeriStar and measure (1) the speedup achieved by each optimization (second column of the table in Figure 4), and (2) the number of lines of code we modified—in both the program and the proof—to implement each optimization (last three columns of Figure 4). The total speedup for all optimizations was $22.1x$.

We report on these optimizations to show that verified software written in Coq is real software: when extracted and compiled with the OCaml system, it runs on real hardware, subject to the same performance constraints as all software. The corollary is that verified functional programs can be performance-tuned in predictable ways.

***Clausesets.*** We replaced Coq's standard-library AVL-tree implementation of the MSets interface for efficient finite sets, with a new red-black tree implementation. (We use MSets to store the clause database.) This new implementation included optimized routines for set insertion, lookup, and union; and we expanded the MSets interface with optimized versions of composite operations such as with minimum-element deletion and insertion-with-membership-query. This resulted in a relative speedup over our baseline VeriStar implementation of 1.21x.

***Priority Heuristic.*** The superpose one_inference_step picks a clause from the clauseset; any new clause will do (for soundness and completeness). An optimization is to pick the smallest new clause (see Section 5.3). This *priority* heuristic greatly winnows the size of the search space. We use the MSets not only to implement clause sets, but to simultaneously implement an efficient priority queue: the total ordering we supply for the red-black searchtree is a lexicographic ordering of clause priority and then clause content. We then use our efficient delete-min operation to pluck the smallest clause from the set in $\log N$ time. This heuristic gives speedup of $3.43x$, compared to selecting an arbitrary new clause.

***Priority Caching.*** Caching the priorities, as integers with the clauses, yields speedup of $1.26x$.

***OCaml Native Integers.*** Our initial implementation used Coq standard-library positive integers to represent variables and priorities. It is a data structure representing arbitrary precision binary numbers, so 101001 is xI(xO(xO(xI(xO(xI xH))))) in the datatype,

```
Inductive positive :=
| xI : positive → positive
| xO : positive → positive | xH : positive.
```

To make things faster, we now use OCaml native 31-bit integers for variables and priorities, yielding speedup of 1.39x.

When extracting to 31-bit OCaml integers, we had to be very careful about overflow. Coq positives are potentially unbounded (thus no overflow), whereas OCaml integers have mod-$2^{31}$ addition. However, VeriStar never performs addition or multiplication of variables, and never generates new variables.[5] VeriStar does some arithmetic on *priorities*, but the soundness and completeness proofs are oblivious to the specific priority function used, so our algorithm is still correct even if priorities happen to overflow. To ensure that our machine-checked soundness proof cannot rely on properties of int31 arithmetic, we do not even axiomatize arithmetic on variables and priorities—we define $+$ and $\times$ as unaxiomatized operators.

***Set Operations.*** Our paramodulation loop (Fig. 2) had used lists of clauses. We switched to set operations (MSets) primarily to make the code more elegant, but it also gives speedup of $1.04$.

***Redundancy Elimination.*** We remove redundant pure clauses before passing clausesets to Superpose. This yields a relatively large speedup of $1.40x$ since it often reduces the number of clauses passed to Superpose between rounds of spatial inference.

***Model-based Saturation.*** Our main implementation of the superposition calculus uses the given-clause algorithm to saturate clausesets. As a last optimization, we rewrote the superposition engine to employ a more intelligent saturation procedure: instead of finding the smallest clause at each saturation step, we attempt to construct a model of the current clauseset (whether saturated or not). Completeness of the superposition calculus implies that this model exists whenever the clauseset is saturated (and does not contain the empty clause). Moreover, for unsaturated sets the completeness proof tells us exactly which clause to select for inference, and for which type of inference (superposition right, superposition left, etc.) in order to bring the set closer to saturation. In this optimization pass, we also improved the global propagation of unit equalities. These two optimizations together resulted in speedup of about $9x$ over our original saturation procedure (the "clausesets" version, which did not use the priority heuristic or priority caching). In the table, $2.09x$ gives the pairwise speedup of model-based saturation over the next most recent version of the prover ("redundancy elimination").

***On optimization effort.*** Most optimizations required minimal updates to the proofs relative to the size of each change to the

[5] This is not true for VeriSmall since the static analysis must generate fresh variables during symbolic execution. But since VeriSmall—as opposed to VeriStar—is not complete in general, it can simply return "don't know" whenever generating fresh variables results in an overflow.

| **source file** | program | proof |
|---|---:|---:|
| compare.v | - | 253 |
| variables.v | 77 | 59 |
| datatypes.v | 60 | - |
| superpose.v | 342 | - |
| *superpose termination* | - | 1,904 |
| superpose_modelsat.v | 335 | - |
| heapresolve.v | 448 | - |
| veristar.v | 150 | - |
| model_type.v | - | 89 |
| model.v | - | 310 |
| clause_lemmas.v | - | 183 |
| cclosure.v | 169 | - |
| superpose_sound.v | - | 584 |
| superpose_modelsat_sound.v | - | 732 |
| spred_lemmas.v | - | 542 |
| clausify_sound.v | - | 474 |
| wellformed_sound.v | - | 399 |
| unfold_sound.v | - | 1,809 |
| norm_sound.v | - | 233 |
| veristar_sound.v | - | 354 |
| clauses.v | 308 | 594 |
| list_denote.v | - | 946 |
| driver.ml | 127 | - |
| **subtotals (excl. redblack.v):** | 2,016 | 9,465 |
| **ratio:** | 4.69 | |
| redblack.v | 281 | 4,110 |
| **totals:** | 2,297 | 13,575 |
| **ratio:** | 5.91 | |

Figure 5: Sizes of system components in lines-of-code. *Ratio* is proof/program.

| **prover** | Smallfoot | VeriStar | SLP | jStar |
|---|---|---|---|---|
| runtime (seconds) | 0.013 | 0.019 | 0.049 | 0.180 |

Figure 6: Average runtimes over 100 trials of Smallfoot, VeriStar, SLP and jStar on 209 entailments generated by Smallfoot.

program (often under $2x$, see Figure 4). Figure 5 gives the total sizes, in lines-of-code, of each of the VeriStar source files.

To prove the int31 optimization sound, we had to abstract all properties of variables on which the proof depended, resulting in more global changes. The clausesets optimization was somewhat of an outlier: to meet the existing Coq MSets interface, we had to prove many lemmas not even used by our soundness proof. On the other hand, the red-black tree implementation of Coq MSets is reusable even independently of VeriStar.

## 9. Measurements

We evaluated VeriStar's performance against that of jStar [18], Smallfoot and SLP on two suites of separation logic entailments. All evaluation was done on a SunFire X4100 server with two dual-core 2.2GHz Opteron 275 processors and 16GB of RAM running CentOS Linux. The first suite contains 209 separation logic entailments generated by Smallfoot during verification of 18 different list-manipulating programs found in the Smallfoot distribution. The second includes 22,000 synthetic entailments tuned from moderate to difficult. The synthetic entailments are the full suites used by Navarro Pérez and Rybalchenko to evaluate their Prolog-based theorem prover SLP against jStar and Smallfoot [25]. The Smallfoot entailments are Navarro Pérez and Rybalchenko's first *clone* set, slightly modified since our prover does not yet deal with arbitrary spatial predicates.

On the "real-world" entailments derived from programs in Smallfoot's test suite, VeriStar's performance was comparable to that of Smallfoot and SLP (on the order of hundredths of a second for the 209 entailments). jStar solved 101 of the 170 valid entailments in the suite in 0.180 seconds. We report the average runtime of each prover on these 209 entailments over 100 trials in Figure 6.

To test the provers at a finer granularity on more difficult entailments, we ran each prover on the 22,000 synthetic entailments in Navarro Pérez and Rybalchenko's Bolognesa and Spaguetti suites. The Spaguetti benchmarks consist of 11,000 entailments of the form $\Pi \wedge \Sigma \vdash \bot$, simulating the inconsistency checks that are often required during symbolic execution. The Bolognesa benchmarks consist of 11,000 general entailments of the form $\Pi \wedge \Sigma \vdash \Pi' \wedge \Sigma'$. For each prover, we measured the number of independent entailments solved within 0.01 to 5 seconds. Figure 7 shows the results of these measurements. Within 5 seconds, jStar checked 8,757 of the 22,000 entailments in the combined suites, VeriStar checked 18,610 entailments and Smallfoot checked 21,483. SLP checked 21,981 entailments within 5 seconds.

***Assessment.*** On real-world entailments generated by Smallfoot during symbolic execution (Figure 6), VeriStar's performance is more than adequate—it solved all 209 entailments in slightly less than two hundredths of a second, three hundredths of a second faster than SLP and only slightly slower than Smallfoot. VeriStar is also the only one of the four systems with a machine-checked soundness proof. On Navarro Pérez and Rybalchenko's synthetic entailments (Figure 7), VeriStar is, in a majority of cases, faster than Smallfoot when deciding heap inconsistency entailments (those of the form $\Pi \wedge \Sigma \vdash \bot$, the Spaguetti suite) and is almost as fast SLP. On general entailments (the Bolognesa suite), VeriStar is not quite as fast as Smallfoot, and not nearly as fast as SLP. Though it is certainly fast enough for Smallfoot-like applications, there is room for improvement. We believe the main issues are:

- VeriStar is a *pure* functional program. Functional programs have a clean proof theory that makes verification a breeze (or at most, a stiff wind). But it means that we pay a log $N$ penalty in some places, where we use red-black trees instead of arrays or hash tables. Using imperative techniques might speed things up, and yet still fit within the Coq framework [4].
- The paramodulation framework for resolution theorem-proving in its modern form is more than two decades old, and a prover such as SLP uses a large combination of time-tested heuristics. By comparison, VeriStar is still immature, and the incorporation of more of these standard techniques would likely improve performance significantly.

## 10. Related Work

Proof-carrying code (PCC) [26] demonstrated the effectiveness of *proof witnesses*—derivation trees in a core logic—as a means of incorporating large untrusted components into safe systems. But the problems with PCC were twofold: (1) the proof witnesses were unacceptably large in practice; and (2) the proof checkers, often running to tens of thousands of lines of code, had to be trusted.

Necula [27, 28] showed how to reduce the size of the proof by compressing common subterms and extending the proof checker to reconstruct these terms from the context. Foundational PCC [1] addressed (2) by proving the soundness of the proof checker from basic axioms but still required large proofs. In this project, we go further: the VeriStar system—when connected to VeriSmall and the

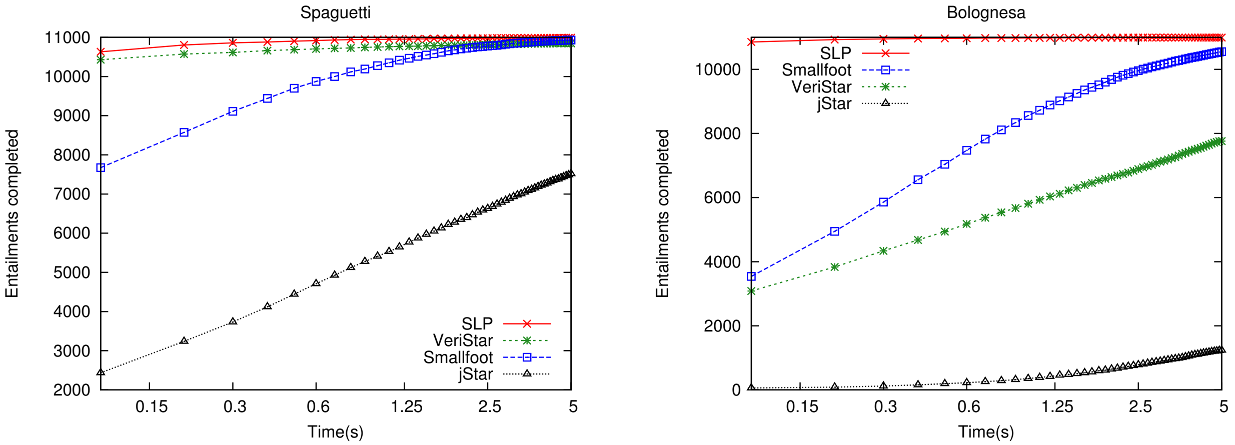


Figure 7: Number of independent entailments checked within 0.01 to 5 seconds by SLP, Smallfoot, VeriStar and jStar. (Higher is better.) The Spaguetti benchmark suite contains 11,000 entailments of the form, $\Pi \wedge \Sigma \vdash \bot$, simulating heap inconsistency checks. The Bolognesa benchmark suite contains 11,000 general separation logic entailments of the form $\Pi \wedge \Sigma \vdash \Pi' \wedge \Sigma'$.

rest of the Verified Software Toolchain—combines strong, foundational correctness guarantees (all the way down to the compiled assembly program) with minimal "proofs": just the program text itself, possibly annotated with light assertions such as loop invariants. These assertions guide the proved-sound static analyzer to an appropriate safety proof.

More recent work on integrating decision procedures into trusted systems has focused on efficiently translating and checking low-level certificates. Armand et al. [4] connect SAT (ZChaff, MiniSat) and SMT solvers (VeriT) to Coq and Isabelle/HOL by translating unsat cores and boolean models to efficient certificates. These certificates are then verified through a combination of small machine-verified proof checkers for resolution chains, linear arithmetic, congruence closure and other theories. Besson et al. describe a related system [10] that permits Nelson-Oppen-style theory combination and supports additional theories besides those supported by Armand et al. In Besson's system, a significant portion of total proof time is consumed by certificate generation and checking. Although VeriStar is significantly simpler than a state-of-the-art SMT solver, it demonstrates that for certain application domains, it is possible to verify the prover, not just the checker, and thus bypass low-level certificate generation and checking completely.

Chlipala's Bedrock system [16], an impressive toolkit for proving the correctness of low-level code, includes an ad hoc simplification procedure and entailment checker for separation logic that together appear to work well in practice. One advantage of Bedrock's checker is that it works on the unencoded implications generated by verification. Errors are therefore easier to communicate to the user in a transparent way. VeriStar entailments, by contrast, are encoded and checked at the clause level, both for efficiency and for interoperability with the Superposition system, and therefore are slightly less human-friendly.

Nguyen and Chin equip a Smallfoot-style entailment checker with a mechanism to integrate user-provided lemmas, complementing the folding/unfolding lemmas that are automatically generated from inductive definitions [29].

Brotherson et al. [12] present a heap theorem prover implemented in HOL that employs a notion of *cyclic proof.* Their work opens an avenue for the integration of user-defined inductive types and auxiliary lemmas that relate such definitions, but is apparently not yet integrated with a prover for the pure part, and is not presented in clausal form. We expect that our modular architecture will allow us to explore the integration of this and other spatial theorem provers into paramodulation-based reasoning tools comparatively easily, as any such prover can be substituted for (or complement) the present unfolding rules for singly-linked lists.

THOR [24] infers invariants by combining symbolic execution with abstraction. An alternative to reimplementing invariant inference in Coq is to use THOR, or some other tool like SLAyer, to annotate loops with invariants that are then confirmed by VeriSmall+VeriStar. The HIP/SLEEK project [30] employs an interesting "shallow embedding" technique for resource use verification that might also be adapted to our setting.

## 11. Lessons Learned

***Extraction for Execution and Profiling.*** Coq's code extraction facilities make it easy to write a Gallina program, extract it to OCaml (or Haskell or Scheme) and get decent performance by compiling using an optimizing compiler like `ocamlopt`. When the Gallina program is written in a straightforward functional style—forgoing extensive use of more advanced features of Coq such as dependent types[6]—code extraction followed by compilation is predictable enough even to support profiling in a traditional style, using a conventional tool like `gprof` [20]. One first extracts a verified Gallina program to OCaml, compiles the OCaml program with `ocamlopt -p`, then does a profiling run using `gprof`. Because the extracted OCaml code is very similar in structure to the Gallina source, the `gprof` profiling data can be used to optimize the Gallina program quite effectively. Figure 8 presents an excerpt of one such profiling run we performed using `gprof`, which showed us that demodulation (unit equality propagation) was a bottleneck for the prover on entailments in Navarro's Spaguetti suite.

[6] Such features are of course fair game when *proving* a Gallina program correct; Coq's type system ensures that only the program is ever extracted, never the proof.

```
time  seconds seconds      calls  name
20.81    7.63    7.63 943778606  demodulate_3526
10.42   11.45    3.82 1247887428 apply2
 9.27   14.85    3.40 343207163  pcompare_176
 5.40   16.83    1.98 297862037  pplus_111
 5.02   18.67    1.84 412584893  zplus_200
 4.12   20.18    1.51   2338512  fold_left_299
 4.09   21.68    1.50 943778606  fun_9033
 2.70   22.67    0.99 114722856  zlength_aux_322
```

Figure 8: Excerpt from the `gprof` trace of a single run of the extracted prover on 1000 entailments in the Spaguetti suite.

***On the Proof Theory of Functional Languages.*** Because Gallina programs are purely functional (one must even prove termination), Gallina has an elegant proof theory that is more tractable than that of ML or Haskell, and much nicer than that of C. This attractive proof theory made the difference, in our estimation, between a couple man-months to complete VeriStar's soundness proof and a couple man-years for a comparable machine-checked proof of a C implementation of the same algorithm.

***The Importance of Modular Proofs.*** The importance of module systems for the construction of large software projects is well-known. We were surprised at just how effective conventional, ML-style module and functor systems (as implemented in Coq) were for building and evolving modular *proofs* of programs as well. By protecting the soundness proof of each component of the prover with an opaque **Module Type** in Coq (Section 6), we saved a great deal of time and energy, especially as we evolved the prover through successive rounds of optimization (Section 8).

## 12. Conclusion

VeriStar is the first machine-verified theorem prover for separation logic that connects to a real-world operational semantics (CompCert C minor). Together with VeriSmall, VeriStar enables automatic *foundational* checking of shape properties with respect to the compiled x86 or PowerPC assembly. VeriStar implements an efficient decision procedure for separation logic [25] using highly tuned functional data structures. VeriStar's implementation and soundness proof can be retargeted to new domains through an opaque axiomatization of separation logic. Finally, VeriStar's design, and its integration with VeriSmall and our C minor separation logic through well-defined interfaces, provides a blueprint for the design of certified end-to-end systems more generally.

## Acknowledgments

Anindya Banerjee, members of the Princeton PL group and the anonymous referees provided helpful suggestions on prior drafts. Aleksandar Nanevski and David Walker gave the first author advice on appropriate venues for this work. This work was supported in part by AFOSR grant FA9550-09-1-0138 and NSF grant CNS-0910448.

## References

[1] A. W. Appel. Foundational proof-carrying code. In *LICS*, 2001.

[2] A. W. Appel. VeriSmall: Verified Smallfoot shape analysis. In *First International Conf. on Certified Programs and Proofs*, Dec. 2011.

[3] A. W. Appel. Verified Software Toolchain. In *ESOP*, pages 1–17, 2011.

[4] M. Armand, G. Faure, B. Grégoire, C. Keller, L. Théry, and B. Werner. A modular integration of SAT/SMT solvers to Coq through proof witnesses. In *First International Conf. on Certified Programs and Proofs*, 2011.

[5] R. Atkey. Amortised resource analysis with separation logic. *Logical Methods in Computer Science*, 7(2:17), 2011.

[6] L. Bachmair and H. Ganzinger. Equational Reasoning in Saturation-Based Theorem Proving. In W. Bibel and P. Schmitt, editors, *Automated Deduction - A Basis for Applications*, volume I, 1998.

[7] J. L. Bentley. *Writing Efficient Programs*. Prentice-Hall, 1982.

[8] 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 115–135, 2005.

[9] J. Berdine, B. Cook, and S. Ishtiaq. SLAyer: Memory safety for systems-level code. In *CAV*, pages 178–183, 2011.

[10] F. Besson, P.-E. Cornilleau, and D. Pichardie. Modular SMT proofs for fast reflexive checking inside Coq. In *First International Conf. on Certified Programs and Proofs*, 2011.

[11] T. Braibant and D. Pous. Tactics for reasoning modulo AC in Coq. In *First International Conf. on Certified Programs and Proofs*, 2011.

[12] J. Brotherston, D. Distefano, and R. L. Petersen. Automated cyclic entailment proofs in separation logic. In *Proceedings of CADE-23*, pages 131–146, 2011.

[13] C. Calcagno and D. Distefano. Infer: an automatic program verifier for memory safety of C programs. In *Third International Conference on NASA Formal Methods*, pages 459–465, 2011.

[14] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. *SIGPLAN Not.*, 44:289–300, January 2009.

[15] B.-Y. E. Chang and X. Rival. Relational inductive shape analysis. In *POPL*, pages 247–260, 2008.

[16] A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *PLDI'11*, pages 234–245, 2011.

[17] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[18] D. Distefano and M. J. Parkinson J. jStar: Towards practical verification for Java. In *OOPSLA*, 2008.

[19] R. Dockins, A. Hobor, and A. W. Appel. A fresh look at separation algebras and share accounting. In *APLAS: 7th Asian Symposium on Programming Languages and Systems*, pages 161–177, 2009.

[20] S. L. Graham, P. B. Kessler, and M. K. McKusick. Gprof: A call graph execution profiler. In *Proc. SIGPLAN '82 Symp. on Compiler Construction, SIGPLAN Notices*, pages 120–126. ACM Press, 1982.

[21] A. Hobor, A. W. Appel, and F. Zappa Nardelli. Oracle Semantics for Concurrent Separation Logic. In *ESOP*, pages 353 – 367, 2008.

[22] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[23] B. Löchner and S. Schulz. An evaluation of shared rewriting. In *Proceedings of the Second International Workshop on Implementation of Logics, Technical Report MPI-I-2001-2-006*, pages 33–48, 2001.

[24] S. Magill, A. Nanevski, E. Clarke, and P. Lee. Inferring invariants in separation logic for imperative list-processing programs. In *Third Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management (SPACE)*, 2006.

[25] J. A. Navarro Pérez and A. Rybalchenko. Separation logic + superposition calculus = heap theorem prover. In *PLDI*, pages 556–566, 2011.

[26] G. Necula. Proof-carrying code. In *POPL*, pages 106–119, 1997.

[27] G. C. Necula and P. Lee. Efficient representation and validation of proofs. In *LICS*, pages 93–104, 1998.

[28] G. C. Necula and S. P. Rahul. Oracle-based checking of untrusted software. In *POPL*, pages 142–154, 2001.

[29] H. H. Nguyen and W.-N. Chin. Enhancing program verification with lemmas. In *CAV*, pages 355–369, 2008.

[30] H. H. Nguyen, C. David, S. Qin, and W.-N. Chin. Automated verification of shape and size properties via separation logic. In *VMCAI*, pages 251–266, 2007.

[31] R. Nieuwenhuis and A. Rubio. Paramodulation-based theorem proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, 2001.

[32] T. Tuerk. A formalisation of Smallfoot in HOL. In *Theorem Proving in Higher Order Logics*, pages 469–484, 2009.

## A. U4 Countermodel

Inference rule U4, as presented in [25, Sec. 4, Figure 1] is:

$$\text{U4}\ \frac{\Gamma \to \Delta, \mathsf{lseg}(x,y) * \mathsf{next}(z,w) * \Sigma \qquad \Gamma', \mathsf{lseg}(x,z) * \Sigma' \to \Delta'}{\Gamma', \mathsf{lseg}(x,y) * \mathsf{lseg}(y,z) * \Sigma' \to \Delta'}\ y \not\equiv z$$

To construct a countermodel, let $\neg\Gamma$ hold, satisfying the first hypothetical clause (remember that the interpretation of a clause $\Gamma \to \Delta$ is $\neg\Gamma \vee \Delta$). We must prove $\Delta'$ assuming

$$\Gamma' \wedge \mathsf{lseg}(x,y) * \mathsf{lseg}(y,z) * \Sigma'$$

But

$$\mathsf{lseg}(x,y) * \mathsf{lseg}(y,z) \Rightarrow \mathsf{lseg}(x,z)$$

holds only when $z$ is *not* a pointer into the heap denoted by $\mathsf{lseg}(x,y)$ (list segments are required to be acyclic).

Note that the axiom *schema* U4 of [25, Figure 2] (not the rule presented in [25, Figure 1]) is sound since there, $\mathsf{lseg}(x,y) * \mathsf{lseg}(y,z)$ is starred with $\mathsf{next}(z,w)$, implying that $z$ does not equal any location in the domain of $\mathsf{lseg}(x,y)$. In Navarro's rule U4, we know only that $\mathsf{lseg}(x,y) * \mathsf{next}(z,w)$ holds hypothetically under $\Gamma$, and $\Gamma$ may be false. A similar argument applies to Navarro's U5 (again, the rule in [25, Figure 1], not the axiom schema).

## B. Bare-bones Model of VERISTAR_LOGIC

This appendix presents definitions from a bare-bones implementation of the Separation Logic Interface of Section 4.2. Coq's module system ensures that we prove the properties required by the interface.

```
Module Barebones : VERISTAR_LOGIC.
Definition loc := nat.
Instance Join_loc : Join loc := @Join_equiv _.
Instance Perm_loc : Perm_alg loc := _.
Instance Sep_loc : Sep_alg loc := _.
Instance Canc_loc : Canc_alg loc := _.
Instance Join_nat : Join nat := @Join_discrete _.
Instance Pos_nat : @Pos_alg nat (@Join_discrete _) :=
  @psa_discrete _.
Instance Canc_nat : @Canc_alg nat Join_nat.
Definition val := option nat.
Instance Join_val : Join val := Join_lower Join_nat.
Instance Perm_val : Perm_alg val := _.
Instance Sep_val : Sep_alg val := _.
Instance Canc_val : Canc_alg val :=
  @Canc_lower _ _ _ Canc_nat.
Definition val2loc (v: val) : option loc :=
  match v with Some (S n) ⇒ Some (S n)
  | _ ⇒ None end.
Definition nil_val : val := Some O.
Definition empty_val : val := None.
Definition full (v: val) := ∀v2, joins v v2 → identity v2.
Definition var : Type := Var.t.
Definition env : Type := var → val.
Definition env_get (ρ: env) : var → val := ρ.
Definition env_set (x: var) (v: val) (ρ: env) : env :=
  fun y ⇒ if Var.eq_dec x y then v else ρ y.
Definition empty_env : env := fun x ⇒ None.
Definition heap : Type := loc → val.
Instance Join_heap : Join heap := Join_fun loc val _.
Instance Perm_heap : Perm_alg heap := _.
Instance Sep_heap : Sep_alg heap := _.
Instance Canc_heap : Canc_alg heap :=
  @Canc_fun loc val  _ _.
Definition rawnext (x: loc) (y : val) (h : heap) :=
  y ≠ None ∧  x ≠ O ∧  h x = y ∧
  ∀x', x' ≠ x → h x' = None.
Definition emp_at (l: loc) (h: heap) := h l = None.
Definition nil_or_loc (v: val) :=
  v=nil_val ∨  ∃l, val2loc v = Some l.
Definition rawnext' x y h :=
  ∃h0, join_sub h0 h ∧  rawnext x y h0.
(∗Proofs about the definitions and operators above are elided.∗)
End Barebones.
```

## C. C minor Model of VERISTAR_LOGIC

This appendix presents definitions from our C minor implementation of the Separation Logic Interface of Section 4.2. Coq's module system ensures that we prove the properties required by the interface.

```
Module CminLog : VERISTAR_LOGIC.
Definition loc := address.
Definition val := option Values.val.
Instance Join_val: Join val := Join_val.
Instance Perm_val: Perm_alg val := _.
Instance Sep_val: Sep_alg val := _.
Instance Canc_val: Canc_alg val := _.
Definition val2loc (v: val) : option loc :=
  match v with
  | None ⇒  None
  | Some (Vptr b ofs) ⇒
     if Zdivide_dec (2∗align_chunk Mint32) (Int.signed ofs)
                    cell_size_pos
         then Some (b, Int.signed ofs)
         else None
  | _ ⇒  None
  end.
Definition nil_val : val := Some (Vint Int.zero).
Definition empty_val: val := None.
Definition full (v: val) := ∀ v2, joins v v2 → identity v2.
Definition var := Var.t.
Definition env: Type := env'.
Definition env_get (ρ: env) (x: var) : val := env'_get ρ (var2pos x).
Definition env_set (x: var) (v: val) (ρ: env) : env :=
   env'_set (var2pos x) v ρ.
Definition empty_env : env := empty_env'.
Definition heap : Type := rmap.
Instance Join_heap: Join heap := Join_rmap.
Instance Perm_heap: Perm_alg heap := _.
Instance Sep_heap: Sep_alg heap := _.
Instance Canc_heap: Canc_alg heap := _.
Definition rawnextp (x: loc) (y : val) : pred rmap :=
  (prop_and (2∗align_chunk Mint32 | snd x)
   (Ex h: Values.val, res_predicates.address_mapsto Mint32
                      h Share.top x)
    ∗ (Ex y': Values.val,
        prop_and (y = Some y')
        (res_predicates.address_mapsto Mint32
           y' Share.top (adr_add x 4)))).
Definition rawnext x y := app_pred (rawnextp x y).
Program Definition emp_atp (l: loc) : pred rmap :=
  fun h ⇒  ∀ l', adr_range l 8 l' → identity (h @ l').
Next Obligation. (∗Proof elided∗) Qed.
```

```
Definition emp_at ($l$: loc) := app_pred (emp_atp $l$).
Definition nil_or_loc ($v$: val) :=
  $v$=nil_val $\vee$ $\exists l$, val2loc $v$ = Some $l$.
Definition rawnext' $x$ $y$ $h$ :=
  $\exists h_0$, join_sub $h_0$ $h$ $\wedge$ rawnext $x$ $y$ $h_0$.
(*Proofs about the definitions and operators above are elided.*)
End CminLog.
```