VeriSmall: Verified Smallfoot Shape Analysis

Andrew W. Appel

Princeton University September 18, 2011

Abstract. We have implemented a version of the Smallfoot shape analyzer, calling upon a paramodulation-based heap theorem prover. Our implementation is done in Coq and is extractable to an efficient ML program. The program is verified correct in Coq with respect to our Separation Logic for C minor; this in turn is proved correct in Coq w.r.t. Leroy's operational semantics for C minor. Thus when our VeriSmall static analyzer claims some shape property of a program, an end-to-end machine-checked proof guarantees that the assembly language of the compiled program will actually have that property.

A static analysis algorithm or type checker takes as input a program, and checks that the program satisfies a certain assertion—or in some cases calculates an appropriate assertion. A static analysis algorithm is *sound* if, whenever it calculates syntactically that the program satisfies a certain assertion, then the corresponding property really does hold on executions of the program. One way to prove soundness is to demonstrate that whenever the static analysis makes a claim, then there is a derivation tree in a given program logic that the assertion is valid for the program. Some implementations of static analyses can produce proof witnesses; this is an example of *proof-carrying code (PCC),* i.e. the pairing of a program + the witness of some static analysis applied to it.

What is the form of a "proof" for PCC? One might think it must be a derivation in logic that can be checked by a proof checker. But such derivations are unacceptably large in practice. It is more practical to factor the static analysis into an untrusted "inference" part and a proved-correct "checker". The first infers invariants and annotates the input program with assertions, as often as once per extended basic block. The checker recomputes the static analysis applied to the program, but (because of the annotations) does not need to infer any invariants, so the checker is a much simpler program. The annotations—assertions—constitute the proof witness.

The checker simply *must* be correct, or else this scheme could not reasonably be called proof-carrying code. But such checkers are generally too complex to be trusted without proof. Therefore, *Foundational PCC* requires a machine-checked proof that the checker program is sound.

In 2003 we demonstrated this approach for safety checking of compiled ML programs [15]. The "inference" part was a type-preserving compiler for Standard ML, which output a program in Typed Assembly Language. The "checker" was a nonbacktracking Prolog program written in Twelf, with a soundness proof written in Higher-Order Logic embedded in Twelf. To absolutely minimize the "trusted computing base," we implemented a tiny proof-checker for LF with a tiny interpreter for deterministic Prolog; this "checker for the checker" was 1100 lines of C, and needed to be trusted, in the sense that bugs in that component could cause unsoundness of the system.

To appear in CPP 2011: First International Conference on Certified Programs and Proofs, December 2011. The original publication is available at www.springerlink.com.

In this paper we turn our attention beyond type systems to shape analysis based on separation logic. The state of the art in logics and proof assistants now makes it fairly straightforward to take *algorithms* from the scientific literature and implement them as *programs* with machine-checked proofs of correctness. We show that static analysis *programs* (not just *algorithms*), and decision-procedure *programs* (e.g., for entailments in separation logic) can be proved correct, and thus need not produce proof derivations. Our verified algorithm is a functional program with a proof of correctness, much like a "proof by reflection" in Coq. Because it is not just a witness-generating "program" specified as a collection of inference rules, we can more easily focus on efficiency, asymptotic complexity, and constant factors. It appears, from Cachera and Pichardie's survey of certified static analysis [6], that no one has done this before.

On the other hand, complex static analyses (that might be impractical to prove correct) can produce witnesses in the form of annotations that can be checked by our simple static analysis.

Our implementations are done in Gallina, the pure functional programming language embedded in the Coq theorem prover. Our proofs of correctness are done in the standard Coq tactic system. From the Gallina programs we use Coq's extraction to obtain ML programs, which we compile with the Ocaml system.

2 Smallfoot

Smallfoot [2, 3] is a shape analyzer based on a decidable fragment of separation logic. It takes as input a pointer-manipulating program in an imperative language with structured control flow, annotated with assertions in separation logic. The assertions specify the relation of program variables to *list segments* and *tree segments*, as well as equalities and inequalities on variables. Smallfoot does not infer loop invariants: the input to Smallfoot must be explicitly annotated with loop invariants and with assertions at the beginning and end of function bodies.

Deciding entailments. Smallfoot repeatedly calls upon a decision procedure for entailments in (a decidable fragment of) separation logic. We use our Gallina implementation of such a decision procedure, and its Coq soundness proof [13].

Isolating conjuncts. When Smallfoot encounters a load, store, or deallocate command that operates at some address e (where e is an expression of the source language), it must rearrange the current precondition to isolate a (separating) conjunct of the form $e \mapsto e'$. This may require unfolding a list segment or introducing a disjunction. We will describe our Gallina program to isolate conjuncts—the algorithms that Berdine et al. [2] call *rearrangement* and *exorcism*—and its soundness proof.

Symbolic execution. Static analysis proceeds by forward symbolic execution from each assertion, through straight-line commands and through if-statements until another assertion is reached. We will describe our Gallina implementation of symbolic execution, and its soundness proof.

Frame inference. Smallfoot infers separation-logic frames for function calls, but our current protype does not implement this.

Tuerk's *Holfoot* [14] is a Smallfoot-like tool implemented in the HOL4 proof assistant. It is *proof-generating* rather than *verified*. Holfoot moves smoothly from fully automatic "shape" proofs to semiautomatic functional correctness proofs, generating lemmas that a human being or an SMT solver must prove. Holfoot is not connected to the operational semantics of any particular programming language, but to an abstract local-action semantics. Holfoot is not so much a specific algorithm as the carefully ordered application of inference rules, along with a *consequence conversion* system.

Here, in contrast, we focus on an efficient and verifiable static analysis algorithm for a real programming language connected to a real compiler, but unlike Tuerk we do not (yet!) go beyond shape analysis into the realm of functional correctness.

3 Syntax of Separation Logic

```
Definition var := positive.
Inductive expr := Nil: expr | Var: var → expr.
Inductive pure_atom := Eqv : expr→ expr→ pure_atom | Neqv : expr→ expr→ pure_atom.
Inductive space_atom :=
  Next: expr → expr → space_atom | Lseg: expr → expr → space_atom.
Inductive assertion := Assertion: ∀ (Π: list pure_atom) (Σ: list space_atom), assertion.
Inductive entailment : Type := Entailment : assertion → assertion → entailment.
```

Above is our syntactic separation logic fragment. Variable-names are represented by positive numbers. An expression is either the literal Nil or a variable. A pure (non-spatial) atom is of the form $e_1 = e_2$ or $e_1 \neq e_2$; an assertion contains (the conjunction of) a list Π of pure atoms, and the *separating* conjunction of a list of *space atoms*. Each space atom describes either a list cell or a list segment (Smallfoot's space atoms also describe trees, which our current prototype does not handle). The list cell Next e1 e2 represents a cons cell at address e_1 whose tail-pointer contains the value e_2, or in primitive separation logic, $(e_1 \mapsto \_) * (e_1 + 1 \mapsto e_2)$. The list segment Lseg e1 e2 represents either $e_1 = e_2$ (meaning an empty segment) or a chain of one or more list cells, starting at address e_1, whose last tail-pointer is e_2, and where $e_1 \neq e_2$.

Smallfoot is a forward symbolic execution algorithm that takes a known precondition P in this fragment, along with a command c, and derives a postcondition Q such that $\{P\}c\{Q\}$. In cases where Q is a disjunction, the disjunction is always at top-level.

4 Semantics of Separation Logics

One application of our shape analysis is in our Verified Software Toolchain [1], where we have a separation logic for C minor, which is a source language for the CompCert verified C compiler [10]. Our higher-order impredicative Concurrent Separation Logic is proved sound with respect to the operational semantics of C minor; Leroy et al. have proved CompCert correct w.r.t. the same operational semantics.

We can also imagine many other uses for efficient, proved-correct decision procedures and shape analyses, so we do not want to tie our soundness result too closely to a particular model of separation logic. Figure 1 shows our general interface, specified as a Module Type, to practically any reasonable model of separation logic that could

Figure 1. Specification of models for separation logic

```
Require Import msl.sepalg.
Parameter loc : Type.
Parameter val: Type.
Declare Instance val_sa : sepalg val.
Parameter val2loc: val → option loc.
Parameter nil_val : val.
Axiom nil_not_loc: val2loc nil_val = None.
Parameter empty_val: val.
Axiom emp_empty_val: ∀ v,   identity v   ↔   v=empty_val.
Definition full (v: val) := ∀ v2, joins v v2 → identity v2.
Axiom val2loc_full: ∀ v l, val2loc v = Some l → full v.
Axiom nil_full: full nil_val.
Axiom empty_not_full: ∼full empty_val.
Axiom val2loc_inj: ∀ v1 v2 l, val2loc v1 = Some l → val2loc v2 = Some l → v1=v2.
Axiom loc_eq_dec: ∀ l1 l2: loc, Decidable.decidable (l1=l2).
Axiom nil_dec: ∀ v, Decidable.decidable (v=nil_val).

Definition var : Type := positive.
Parameter env : Type.
Parameter env_get: env → var → val.
Parameter env_set: var → val → env → env.
Axiom gss_env : ∀ (x : var) (v : val) (s : env), env_get (env_set x v s) x = v.
Axiom gso_env : ∀ (x y : var) (v : val) (s : env), x<>y →
                        env_get (env_set x v s) y = env_get s y.
Parameter empty_env : env.
Axiom env_gempty: ∀ x, env_get empty_env x = empty_val.

Parameter heap : Type.
Declare Instance heap_sa : sepalg heap.
Parameter rawnext: ∀ (x: loc) (y : val) (s : heap), Prop.
Parameter emp_at: ∀ (l: loc) (h: heap), Prop.
Axiom heap_gempty: ∀ h l, identity h → emp_at l h.
Definition nil_or_loc (v: val) := v=nil_val ∨ ∃ l, 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.
Axiom rawnext_at1: ∀ x y h1 h2 h, rawnext' x y h1 → join h1 h2 h →
       emp_at x h2 ∧ rawnext' x y h.
Axiom rawnext_at2: ∀ x y h1 h2 h, join h1 h2 h → rawnext' x y h →
       emp_at x h2 → rawnext' x y h1.
Axiom rawnext_not_emp: ∀ x y h, rawnext' x y h → ∼emp_at x h.
Axiom emp_at_join: ∀ h1 h2 h, join h1 h2 h → ∀ l, (emp_at l h1 ∧ emp_at l h2) ↔emp_at l h.
```

Figure 2. Denotations

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

Definition expr_denote (e : expr) (σ : state) : val :=
    match e , σ with Nil , _ ⇒ nil_val | Var x , State s _ ⇒ env_get s (Some x) end.
Definition expr_eq (x y : expr) (s : state) := expr_denote x s = expr_denote y s.

Definition spred := state → Prop.
Definition neg (P : spred) : spred := fun σ : state ⇒ ∼P σ.

Definition pure_atom_denote (a : pure_atom) : spred :=
  match a with Eqv e1 e2 ⇒ expr_eq e1 e2 | Neqv e1 e2 ⇒ neg (expr_eq e1 e2) end.

Inductive lseg : val → val → heap → Prop :=
| lseg_nil : ∀ x h, identity h → nil_or_loc x → lseg x x h
| lseg_cons : ∀ x x' h h0 h1 z, x<>y → val2loc x = Some x' →
    rawnext x' z h0 → lseg z y h1 → join h0 h1 h → lseg x y h.

Definition space_atom_denote (a : space_atom) : spred := fun σ ⇒
  match a, σ with
  | Next x y , State _ h ⇒
      fun σ ⇒ match val2loc (expr_denote x σ) with
      | Some l' ⇒ rawnext l' (expr_denote y σ) h ∧ nil_or_loc (expr_denote y σ)
      | None ⇒ False
      end
  | Lseg x y, State _ h ⇒ lseg (expr_denote x σ) (expr_denote y σ) h
  end.

Fixpoint list_denote {A T : Type} (f : A → T) (g : T → T → T) (b : T) l : T :=
  match l with nil ⇒ b | x :: l' ⇒ g (f x) (list_denote l') end.

Definition assertion_denote (f : assertion) : spred :=
match f with Assertion Π Σ ⇒ list_denote pure_atom_denote (@intersection state) T Π
                           ∧ list_denote space_atom_denote sep_con emp Σ
end.
```

support list segments. **Import** msl.sepalg refers to the notion of Separation Algebras [9] from our Mechanized Semantic Library (`msl.cs.princeton.edu`).

We prove that our C minor separation logic satisfies this specification. Separating out the interface in this way causes some local pain, compared to a direct nonabstract model, but the improvement in modularity is well worth it.

Based on this semantic specification of the operators, we can define the denotations of syntactic expressions and assertions, as shown in Figure 2.

Remark. Berdine et al. assume an abstract addressing model such that if $p \neq q$ then the fields of p cannot possibly overlap with the fields of q; other presentations of Separation Logic assume an address-arithmetic model, in which records might overlap; e.g., $100 \mapsto x * 101 \mapsto y * 102 \mapsto z$ might contain the pair (x,y) overlapping with

Figure 3. Freshness

```
Definition fresh {A} (f: A → positive) (a: A) (x: positive) : Prop := Zpos (f a) ≤ Zpos x.

Definition agree_except (x: var) (σ σ′: state) : Prop :=
 match σ, σ′ with State s h , State s' h' ⇒
   (∀ x', x' <> x → env_get s (Some x') = env_get s' (Some x')) ∧ h=h' end.

Definition existsv(x: var) (P: spred) : spred := fun σ ⇒ ∃ σ′, agree_except x σ σ′ ∧ P σ′.

Definition |−− (P Q: spred) := ∀ s, P s → Q s.        Infix "|−−".

Lemma pure_atom_denote_agree: ∀ a σ σ′ x, fresh freshmax_pure_atom a x →
  agree_except x σ σ′ → pure_atom_denote a σ → pure_atom_denote a σ′.

Lemma space_atom_denote_agree: ∀ a σ σ′ x, fresh freshmax_space_atom a x →
  agree_except x σ σ′ → space_atom_denote a σ → space_atom_denote a σ′.
```

the pair (y,z). We model Next so that it may be instantiated in either the abstract or the address-arithmetic style. But the Smallfoot inference rules assumed by Berdine et al. are sound only if such overlap cannot occur. The only way we know how to assure this, in an address-arithmetic setting, is to make the rather strong assumption that list cells are aligned on a multiple-of-size boundary.

5 Fresh variables

When symbolic execution rewrites separation-logic assertions, it sometimes uses fresh variables, i.e. new variables that are not free in the current program or precondition.

We have functions freshmax_expr, freshmax_pure_atom, freshmax_space_atom, freshmax_assertion, freshmax_stmt, that traverse assertions and commands to find the highest-numbered variable in use (highest-numbered nonfresh variable).

Figure 3 gives some definitions and lemmas regarding freshness of variables. Let a be an expression (or pure atom, space atom, assertion, statement) and let f be the freshmax_expr function (or respectively, freshmax_pure_atom, freshmax_space_atom, etc.). Then we say that some variable x is fresh for a by writing fresh f a x.

Zpos injects from positive to the integers; for efficiency our program *computes* on positives, but for convenience in *proofs* we use tactics and lemmas on the integers.

We can say that two states σ and σ' agree_except at x, and we define existsv x P to mean that P almost holds (on a given state)—that is, there exists a value v such that P would hold on the state if only we would set $x := v$. Finally, if x is fresh for a, and two states σ and σ' agree_except at x, then a at σ is equivalent to a at σ'.

6 Paramodulation

Smallfoot makes repeated calls to decide entailments in separation logic. Berdine et al. [2] sketch an algorithm for deciding entailments in their fragment of separation logic.

Navarro and Rybalchenko [11] apply *paramodulation*, a resolution theorem-proving algorithm, to this decision problem, and get a program that is significantly faster than the original Smallfoot implementation. Paramodulation [12] permits modular introduction of theories; a standard such theory to add is the *superposition calculus,* a theory of equalities and inequalities. Navarro and Rybalchenko extend paramodulation with superposition and with the *spatial terms* of Berdine et al.'s decidable fragment of separation logic, yielding a "heap theorem prover."

Gordon Stewart, Lennart Beringer, and I have built an implementation in Gallina of this paramodulation-based heap theorem prover. Our proof of soundness is nearly finished, and we intend to prove termination and perhaps completeness. Preliminary measurements of the extracted ML code are competitive with Smallfoot's entailment decider (also implemented in ML). This is not nearly as good as it might seem, because in fact Navarro and Rybalchenko's implementation (in Prolog) is about 100x faster than Smallfoot in solving large entailments. We expect that we can improve our program with more efficient data structures for term indexing and priority queues (with attended proofs of soundness). We will report on paramodulation in a separate paper [13].

7 Isolation

Consider the command `a:=b.next`, which loads a field of record b. (Similar issues pertain to storing a field or deallocating a record.) Suppose precondition P has the form $\mathsf{Next}\, b\, c\, *\, F$ for some frame F. Assuming that the variable a is not free in expressions b, c or formula F, it's easy to derive a postcondition $(a = c) * \mathsf{Next}\, b\, c * F$.

Suppose instead that P is $F_1 * \mathsf{Next}\, b\, c * F_2$. and the separation-logic Hoare rule for assignment has a syntactic form that requires $\mathsf{Next}\, b\, c * F$. Clearly by the associative law, $P \vdash \mathsf{Next}\, b\, c * (F_1 * F_2)$. We can use the rule of consequence to strengthen the precondition to match the desired form.

A harder case is one where the precondition P is $b \neq d * \mathsf{Lseg}\, b\, d * F$. Because the list segment is not empty ($b \neq d$), we can unfold it once; we insert a fresh variable x (not free in a, b, d, F) as follows: $P \vdash \mathsf{Next}\, b\, x * b \neq d * \mathsf{Lseg}\, x\, d * F$.

In each of the cases above, we rearrange the precondition to *isolate* one field as required by a load (or store); in the case of a deallocation command we would have to isolate all the fields of a particular record together, but the issues would be the same. An important component of the Smallfoot algorithm is this rearrangement.

In the case where P is $\mathsf{Lseg}\, b\, d * F$, such that $\mathsf{Lseg}\, b\, d * F \not\vdash b \neq d$, then the list segment might possibly be empty, so we cannot unfold it; symbolic execution will be stuck here, unable to prove by shape analysis that the program is safe.

The hardest case (as explained by Berdine et al.) is the "spooky disjunction." Suppose P is $d \neq e * \mathsf{Lseg}\, b\, d * \mathsf{Lseg}\, b\, e * F$. We know that exactly one of the two segments is nonempty; if both are empty, then $d = e$, and if both are nonempty, then the segments (b, d) and (b, e) would overlap (would not separate). Whichever segment is nonempty, we should be able to unfold it, but we do not know which. Therefore we can derive

$$P \vdash (\mathsf{Next}\, b\, x * d \neq e * \mathsf{Lseg}\, x\, d * b = e * F) \vee (\mathsf{Next}\, b\, x * d \neq e * b = d * \mathsf{Lseg}\, x\, e * F).$$

Figure 4. Exorcize and isolate

```
Fixpoint exorcize (e: expr) (Π: list pure_atom) (Σ0 Σ: list space_atom) (x: var)
                                                        : option(list assertion) :=
 match Σ with
 | nil ⇒ if incon (Assertion Π (rev Σ0)) then Some nil else None
 | Lseg f f' :: Σ1 ⇒
     if oracle (Entailment (Assertion Π (rev Σ0 ++ (Lseg f f') :: Σ1))
                           (Assertion (Eqv e f :: nil) (rev Σ0 ++ Lseg f f' :: Σ1)))
     then match exorcize e (Eqv f f' :: Π) (Lseg f f' :: Σ0) Σ1 x with
          | Some l ⇒ Some (Assertion Π (Next e (Var x) :: Lseg (Var x) f' :: rev Σ0 ++ Σ1) ::l)
          | None ⇒ None
          end
     else exorcize e Π (Lseg f f' :: Σ0) Σ1 x
 | a :: Σ1 ⇒ exorcize e Π (a :: Σ0) Σ1 x
 end.

Fixpoint isolate' (e: expr) (Π : list pure_atom) (Σ0 Σ: list space_atom) (x: var) (count: nat)
                                                        : option(list assertion) :=
match Σ with
| nil ⇒ if count < 2 then None
          else if incon (Assertion (Eqv e Nil :: Π) (rev Σ0))
                 then exorcize e Π nil (rev Σ0) x
          else None
| Next e1 e2 :: Σ1 ⇒
     if eq_expr e e1
     then Some [Assertion Π (Next e e2 :: rev Σ0 ++ Σ1)]
     else if oracle (Entailment (Assertion Π (rev Σ0 ++ (Next e1 e2) :: Σ1))
                                (Assertion (Eqv e e1 :: nil) (rev Σ0 ++ (Next e1 e2) :: Σ1)))
          then Some [Assertion Π (Next e e2 :: rev Σ0 ++ Σ1)
else isolate' e Π (Next e1 e2 :: Σ0) Σ1 x count
| Lseg f f' :: Σ1 ⇒
     if oracle (Entailment (Assertion Π (rev Σ0 ++ (Lseg f f') :: Σ1))
                           (Assertion (Eqv e f :: Neqv f f' :: nil ) (rev Σ0 ++ (Lseg f f') :: Σ1)))
     then Some [Assertion Π (Next e (Var x) :: Lseg (Var x) f' :: rev Σ0 ++ Σ1)]
     else if oracle (Entailment (Assertion Π (rev Σ0 ++ (Lseg f f') :: Σ1))
                                (Assertion (Eqv e f :: nil) nil (rev Σ0 ++ (Lseg f f') :: Σ1)))
          then isolate' e Π (Lseg f f' :: Σ0) Σ1 x (S count)
     else isolate' e Π (Lseg f f' :: Σ0) Σ1 x count
end.

Definition isolate (e: expr) (P: assertion) (x: var) : option (list assertion) :=
 match P with Assertion Π Σ ⇒ isolate' e Π nil Σ x 0 end.
```

The algorithm for eliminating the "spooky disjunctions" is called *exorcism* by Berdine et al., and their *entire* description of it is thus:

> To deal with this in the rearrangement phase we rely on a procedure for exorcising these spooky disjunctions. In essence, exor$(\Pi|\Sigma, e)$ is a collection of assertions obtained by doing enough case analysis (adding equalities and inequalities to Π) so that the location of e within a $*$-conjunct is determined. This makes the rearrangement rules complete. We omit a formal definition of exor for space reasons.

This function for isolating a field (preparatory to a load or store) we will name *isolate*. It calls upon an auxiliary function *exorcize*. Our assertion syntax has no disjunction operator, so we formulate the output of these functions as an option(list(assertion)). The result None indicates that it was not possible to isolate the given field; the result Some(l) gives a list l of assertions, the disjunction of which is implied by the input assertion P.

The isolate' function walks down a list Σ of space atoms, tossing them into Σ_0 as it passes by, as follows: • In the last **else** clauses of the Next and Lseg clauses, where e1 or f can't be proved equivalent to e, this Next or Lseg is an irrelevant conjunct—the recursive call to isolate' simply moves it from Σ to Σ_0 and continues. • If e is syntactically identical to e1, or we can prove $\Pi, \Sigma \vdash$ e $=$ e1, then the conjunct Next e1 e2 matches, and isolate succeeds. • If we can prove from Π, Σ that e=f and f≠f', then isolate succeeds by unfolding this list segment. • Finally, if $\Pi, \Sigma \vdash$ e $=$ f but we cannot also prove f≠f', then the conjunct is a candidate for a spooky disjunction, so we toss it into Σ_0 and increment the count variable, which counts the number of spooky disjuncts.

If isolate' reaches the end of Σ with count>1, then there is a spooky disjunction. exorcize handles it (Figure 4) by performing case-splitting (empty or nonempty) on each relevant Lseg. The two cases appear as Eqv f f' and Next e (Var x) :: Lseg (Var x) f' :: ..., respectively. In the Eqv case, we must also case-split on all the remaining relevant Lsegs, but in the non-Eqv case, all the others must be empty.

Discussion. This is just straightforward functional programming: nothing remarkable about it, except that we can now use Gallina's proof theory (i.e., CiC) to prove the soundness. Termination is already proved, because **Fixpoint** must terminate.

8 Soundness of isolate

```
Lemma exorcize_sound: ∀ e Π Σ x,
      fresh freshmax_expr e x → fresh freshmax_assertion (Assertion Π Σ) x →
      ∀ cl, (exorcize e Π nil Σ x) = Some cl →
(assertion_denote (Assertion Π Σ) |−−
  fold_right (fun P ⇒ union (existsv x (assertion_denote P))) FF cl) ∧
      (∀ Q, In Q cl →
            match Q with
            |Assertion _ (Next e0 _ :: _) ⇒ e=e0
            | _ ⇒ False
            end ∧ fresh freshmax_assertion Q (Psucc x)).
```

```
Lemma isolate_sound: ∀ e P x results,
      isolate e P x = Some results →
       fresh freshmax_expr e x → fresh freshmax_assertion P x →
      assertion_denote P |−−
        fold_right (fun Q ⇒ union (existsv x (assertion_denote Q))) FF results ∧
      ∀ Q, In Q results →
            match Q with
           |Assertion _ (Next e0 _ :: _) ⇒ e=e0
           | _ ⇒ False
           end ∧ fresh freshmax_assertion Q (Psucc x).
```

Given an assertion P and the desire to isolate a conjunct of the form Next e _ , and given x fresh for e and P, suppose isolate e P x returns Some results. Then we know:

- The denotation of P entails the union of all the disjuncts Q in results, provided that we set the variable x to some appropriate value.
- Every disjunct Q is of the form Assertion _ (Next e _ :: _).
- Every free variable in Q has name $\leq$ x, i.e., the *next* variable after x is fresh for Q.

9 Symbolic execution

Symbolic execution proceeds on a C minor syntax annotated with assertions. The shape analyses will not interpret many of the C minor expressions it sees, but simple expressions such as variables and and the constant 0 (interpreted as Nil) will be relevant to symbolic execution. Thus we define the function Cexpr2expr that translates simple expressions from C minor to the language of our syntactic assertions, and ignores others:

```
Definition Cexpr2expr (e: Cminor.expr) : option expr :=
  match e with Evar i ⇒ Some (Var i)
               | Eval (Vint z) ⇒ if Int.eq_dec z Int.zero then Some Nil else None
               | _ ⇒ None
  end.

Definition getSome {A} (x: option A) (f: A → bool):=
  match x with Some y ⇒ f y | None ⇒ false end.

Definition Cexpr2assertions(e:Cminor.expr)(a:assertion)(f:assertion→ assertion→ bool):=
 match a with Assertion Π Σ ⇒
  match e with
  | Ebinop (Cminor.Ocmp Ceq) a b ⇒
      getSome (Cexpr2expr a) (fun a' ⇒ getSome (Cexpr2expr b) (fun b' ⇒
        f (Assertion (Eqv a' b' ::Π) Σ) (Assertion (Neqv a' b' ::p) Σ)))
  | Ebinop (Cminor.Ocmp Cne) a b ⇒
      getSome (Cexpr2expr a) (fun a' ⇒ getSome (Cexpr2expr b) (fun b' ⇒
        f (Assertion (Neqv a' b' ::Π) Σ) (Assertion (Eqv a' b'::Π) Σ)))
  | _ ⇒ getSome (Cexpr2expr e) (fun a' ⇒
           f (Assertion (Neqv a' Nil ::Π) Σ) (Assertion (Eqv a' Nil ::Π) Σ))
  end
 end.
```

Figure 5. Symbolic execution

```
Fixpoint check (P: assertion) (BR: list assertion) (c: stmt) (x': positive)
                          (cont: assertion → positive → bool) : bool :=
 if incon P then true
 else match c with
  | Sskip ⇒ cont P x'
  | Sassert Q ⇒ oracle (Entailment P Q) && cont Q x'
  | Sassign x (Evar i) ⇒
     match P with Assertion Π Σ ⇒
     let P':=Assertion(Eqv (Var x) (subst_expr x (Var x') (Var i)) :: subst_pures x (Var x') Π)
                          (subst_spaces x (Var x') Σ)
      in cont P' (Psucc x')
     end
  | Sassign x (Eload Mint32 (Ebinop Cminor.Oadd (Evar i) (Eval (Vint ofs)))) ⇒
      Int.eq ofs (Int.repr 4) &&
      getSome (isolate (Var i) P x') (fun l ⇒
          forallb(fun P' ⇒
                match P' with
                | Assertion Π' (Next _ f :: Σ') ⇒
                  cont (Assertion (Eqv (Var x) (subst_expr x (Var (Psucc x')) f)
                                       :: subst_pures x (Var (Psucc x')) Π')
                          (subst_spaces x (Var (Psucc x')) (Next (Var i) f ::Σ')))
                               (Psucc (Psucc x'))
                | _ ⇒ false
               end)
              l)
  | Sstore Mint32 (Ebinop Cminor.Oadd e1 (Eval (Vint ofs))) e2 ⇒
       Int.eq ofs (Int.repr 4) &&
       getSome (Cexpr2expr e1) (fun e1' ⇒ getSome (Cexpr2expr e2) (fun e2' ⇒
       getSome (isolate e1' P x') (fun l ⇒
        forallb(fun P' ⇒
                match P' with
                | Assertion Π' (Next _ f :: Σ') ⇒
                       cont (Assertion Π' (Next e1' e2' :: Σ')) (Psucc x')
                | _ ⇒ false
                end)
             l)))
  | Sexit n ⇒ oracle (Entailment P (nth n BR false_assertion))
  | Sblock (Sloop (Sblock (Sifthenelse e c1 c2))) ⇒ (* while loop! *)
      Cexpr2assertions e P (fun P1 P2 ⇒
          check P1 (P::P2::BR) c1 x' (fun R y' ⇒ false) &&
          check P2 (P::P2::BR) c2 x' (fun R y' ⇒ false) &&
          cont P2 x')
  | Sifthenelse e c1 c2 ⇒
      Cexpr2assertions e P (fun P1 P2 ⇒
           check P1 BR c1 x' cont && check P2 BR c2 x' cont)
  | Sseq c1 c2 ⇒ check P BR c1 x' (fun P' y' ⇒ check P' BR c2 y' cont)
  | _ ⇒ false
end.
```

Symbolic execution is flow-sensitive, and when interpreting an **if** statement, "knows" in the **then** clause that the condition was true, and in the **else** clause that the condition was false. For this purpose we define a function Cexpr2assertions e a f that takes C-minor expression e and assertion a, and generates two new assertions equivalent (more or less) to e $\wedge$ a and $\sim$e $\wedge$ a, and applies the continuation f to both of these assertions. We write "more or less" because e $\wedge$ a is actually an illegitimate mixture of two different syntaxes; e must be properly translated into the assertion syntax, which is the purpose of Cexpr2assertions.

Symbolic execution relies on functions subst_expr x e e', subst_pures x e Π, and subst_spaces x e Σ that substitute expression e for the variable x in (respectively) an expression e', a pure term Π, or a space term Σ.

Smallfoot symbolic execution uses a restricted form of assertion without disjunction. Therefore when a disjunction would normally be needed, Smallfoot does multiple symbolic executions over the same commands. For example, for

(**if** e **then** c1 **else** c2); c3; c4; assert Q

with precondition P, Smallfoot executes the commands c1;c3;c4 with precondition e $\wedge$ P and then executes c2;c3;c4 with precondition $\sim$e $\wedge$ P. Because Berdine et al.'s original Smallfoot used only simple "if and while" control flow, this re-execution was easy to express.

C minor has a notion of nonlocal loop exit; that is, one can exit from any number of nested blocks (such as loops, loop bodies, or switch statements). One branch of an **if** statement might exit, while the other might continue normally. To handle this notion, the parameters of the check function include not only a precondition P but a break-condition list BR that gives exit-postconditions for all possible exit labels.

In order to handle re-execution mixed with multiple-level exit, we write the symbolic execution function in continuation-passing style. The argument cont is the check function's continuation. Once check has computed the postcondition Q for a given statement, it calls cont with Q. If it needs to call cont more than once, it may do so. For example, in the clause for Sifthenelse notice that cont is passed to two different recursive calls to check, each of which will perhaps call cont. Or perhaps not; the symbolic execution of Sexit n (to break out of n nested blocks) does not call cont at all, but looks up the nth item in BR.

The miracle of termination. In Coq, a **Fixpoint** function must have a structurally inductive parameter, such that in every recursive call the actual parameter is a substructure of the formal parameter. Here the structural parameter is the statement c. Most of the recursive calls are buried in continuations (lambda-expressions passed to the cont parameter)—and may not actually occur until much later, inside other calls to check. The miracle is that Coq still recognizes this function as structurally recursive.

10 Ghost variables

A variable mentioned in an assertion but not in the program is a *ghost variable*. In a Hoare logic with ghost variables, one has rules capable of proving such derivations as,

$$\frac{\{a = x\} \quad a \leftarrow a + 1 \ \{a = x + 1\}}{\{a = x - 1\} \ a \leftarrow a + 1 \ \{a = x\}}$$

That is, taking advantage of the fact that x is not free in the command $a := a + 1$, substitute for x in both the pre- and postcondition.

Our underlying Hoare logic does not handle ghost variables directly. We could add such a rule, as it is provable from the underlying operational model of C minor. But instead we find that our Concurrent Separation Logic is expressive enough to derive a new Separation Logic *with* a ghost-variable rule; its rules are proved sound as derived lemmas from the underlying rules. In the new logic, we add a separate namespace of *logical variables* (or ghost variables) visible to semantic assertions but not to ordinary commands. (Also, the underlying Separation Logic has variables-as-resources [5], but the top layer has a conventional (nonresource) treatment of variables; the underlying layer has fractional ownership shares [9], but the top layer is a conventional all-or-nothing separation logic.)

The Smallfoot algorithm would like to think that there's just one namespace of variables, so our *syntactic* separation logic (Section 3) has just one namespace. Let the variable ghost be the first one beyond the highest-numbered variabled used in the program. In our interpretation of Hoare triples during symbolic execution, all the variables beyond ghost in the syntactic Hoare logic will be interpreted as logical variables.

Then we do some predicate translation, as follows. Let P be a predicate on states. We define the *ghostly denotation* of P as a predicate on worlds:

$$[\![P]\!]_{\mathsf{ghost}} = \lambda w.\ P\,(\mathsf{State}\ (\mathsf{mix\_envs}\ 1\ \mathsf{ghost}\ (\mathrm{w\_rho}\ w)\ (\mathrm{w\_aux}\ w)\ )\ (\mathrm{w\_m}\ w))$$

where mix_envs lo hi ρ a is the environment that consults the "real" local-variable environment ρ on variables lo $\leq i <$ hi, otherwise consults the ghost environment a.

At the start of the symbolic execution, the check0 function computes the ghost boundary x for the given program by taking the max of all variable names in use:

```
Definition check0 (P: assertion) (c: stmt) (Q: assertion) : bool :=
 let x := Pmax (Pmax (freshmax_assertion P) (freshmax_stmt c)) (freshmax_assertion Q)
  in check P nil c x (fun Q' _ ⇒ oracle (Entailment Q' Q)).
```

11 Soundness of symbolic execution

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

Theorem [check_sound]. If the symbolic executor checks a Hoare triple (check0 P c Q) then that triple is semantically sound, according to our axiomatic semantics semax. Since check0 takes syntactic assertions and semax takes semantic assertions, in the statement of this theorem we must do world-to-state translations and take assertion-denotations, which is what assertion2wpred does.

Proof. By induction on the height of commands, using the following induction scheme.

```
Definition check_sound_scheme (c: stmt) :=
 ∀ ghost G P BR x cont
 (GHOST: Zpos ghost <= Zpos x)
 (H: check P BR c x cont = true)
 (FRESHc: fresh freshmax_stmt c ghost)
 (FRESHP: fresh freshmax_assertion P x)
 (FRESHBR: fresh (freshmax_list freshmax_assertion) BR x),
 semax G ⟦assertion_denote P⟧ghost (erase_stmt c)
      (RET ⟦cont_assert x cont⟧ghost ⟦map assertion_denote BR⟧ghost).
```

```
Lemma check_sound_helper: ∀ n c, (height c < n) → check_sound_scheme c.
```

We could *almost* do the induction directly on the structure of commands, except for one case involving C minor's block command.

The most difficult and annoying issues in the proof involve the treatment of fresh variables, especially in the case where some postcondition will hold provided that the fresh variable has an appropriate value.

The treatment of the continuation argument of the check function is one of the interesting parts of the proof. By the time symbolic execution calls cont with postcondition Q and a fresh variable x, that means variables <x may be used in Q. The semantic meaning of this cont function is the disjunction of all the arguments Q for which cont could return true, but the instantiation of fresh variables must also be taken into account, as follows:

```
Definition agree_except_range (lo hi: var) (σ σ′: state) : Prop :=
 match σ, σ′ with State s h , State s' h' ⇒
 (∀ x, lo <= x < hi ∨ env_get s (Some x) = env_get s (Some x)) ∧ h = h' end.

Definition existsvs (lo hi: var) (P: spred) : spred :=
   fun σ ⇒ ∃ σ′, agree_except_range lo hi σ σ′ ∧ P σ′.

Definition cont_assert (lo: positive) (cont: assertion → var → bool) : spred :=
 fun σ ⇒ ∃ (Q : assertion) (hi: var),
            cont Q hi = true ∧ lo <= hi ∧ fresh freshmax_assertion Q hi
            ∧ existsvs lo hi (assertion_denote Q) σ.
```

12 Conclusion

Yang et al. [16] improve on Smallfoot with a symbolic join operator ⊔ that reduces or eliminates the need to symbolically re-execute the same statements. The *bi-abduction* algorithm infers frames and anti-frames for function calls [7]. Both ⊔ and bi-abduction should be implementable and provable in Coq; even if not, an unverified static analyzer using them can generate annotations checkable by Smallfoot (and the extra ⊔-joined annotations would avoid symbolic re-execution in Smallfoot).

The SLAyer program analysis [4] infers loop invariants using external theorem provers; at this scale of software we might skip proving SLAyer correct, and just have it generate assertions checkable by our proved-correct tool.

Our implementation and verification is not much more than just competent engineering. Computer Science has reached the point where one can take a result from the literature, use conventional functional programming to write a purely functional program, and then use Coq or Isabelle (etc.) to prove correctness. The proofs were accomplished by the expedient of having a reasonably competent Coq hacker (the author) slog away in the tactical theorem prover for a few weeks; the paramodulation implementation and proofs are the work of Gordon Stewart, Lennart Beringer, and the author.

The soundness proofs are complete except for one issue: our C minor Hoare logic requires for the command M[p]:=q that q be an initialized variable; neither the original Smallfoot nor our implementation tracks initialized variables, but this would be trivial to add and prove sound.

Component	*Program Lines*	*Proof Lines*
Paramodulation	1096	∼4000
Isolate/exorcize	125	798
Symbolic execution	149	3606

We will not report performance benchmarks in this paper, as they depend critically on the performance of the entailment oracle (paramodulation), and our implementation of paramodulation is not yet tuned. However, the check function—when extracted to Caml—is about as clean and efficient as one could imagine an implementation of Smallfoot to be in any language, and even our preliminary untuned implementation of paramodulation is competitive with Smallfoot on Navarro's benchmark suite [11].

Here we proved Smallfoot sound w.r.t. a separation logic; the deeper and more difficult scientific results are in the soundness of that logic w.r.t. an optimizing compiler [1]. That logic is very highly expressive, with many features (concurrency, impredicativity, indirection) that are entirely unneeded by the Smallfoot soundness proof. But if static analyzers such as this one are to be connected to other automatic or semiautomatic program analyses and program-proof systems, it would be helpful to have them all proved sound w.r.t. the same logic—hence the desire for "expressiveness overkill" in the logic.

Furthermore, there are difficult results in the specification of optimizing compilers for thread-concurrent languages (such as C with Pthreads) or for programs that do shared-memory interaction with an operating system—and in the connection of these specifications to the Hoare logic [8]. But having all that infrastructure in place means that our soundness result is not just "if the shape checker says true then there's a Separation Logic proof;" it means that *and* "if there's a Separation Logic proof then the assembly code that results from the optimizing compilation in CompCert behaves according to the specification checked by the shape checker." Connecting all the components end-to-end gives a much more valuable result.

Acknowledgments. This research was supported in part by the Air Force Office of Scientific Research (grant FA9550-09-1-0138) and the National Science Foundation (grant CNS-0910448).

References

1. Andrew Appel. Verified software toolchain. In Gilles Barthe, editor, *ESOP'11: European Symposium on Programming*, volume 6602 of *LNCS*, pages 1–17. Springer, 2011.

2. Josh Berdine, Cristiano Calcagno, and Peter OHearn. Symbolic execution with separation logic. In *APLAS'05*, volume 3780 of *LNCS*, pages 52–68. Springer, 2005.
3. Josh Berdine, Cristiano Calcagno, and Peter OHearn. Smallfoot: Modular automatic assertion checking with separation logic. In F. de Boer *et al.*, editor, *Formal Methods for Components and Objects*, volume 4111 of *LNCS*, pages 115–137. Springer, 2006.
4. Josh Berdine, Byron Cook, and Samin Ishtiaq. Slayer: Memory safety for systems-level code. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, volume 6806 of *LNCS*, pages 178–183. Springer, 2011.
5. Richard Bornat, Cristiano Calcagno, and Hongseok Yang. Variables as resource in separation logic. *Electronic Notes in Theoretical Computer Science*, 155:247 – 276, 2006. Proc. 21st Annual Conf. on Mathematical Foundations of Programming Semantics (MFPS XXI).
6. David Cachera and David Pichardie. Comparing techniques for certified static analysis. In *Proc. 1st NASA Formal Methods Symposium (NFM'09)*, pages 111–115. NASA Ames Research Center, 2009.
7. Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *POPL '09: Proc. 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 289–300. ACM, 2009.
8. Robert Dockins and Andrew W. Appel. Behavioral refinement for unsafe languages. submitted for publication, 2011.
9. Robert Dockins, Aquinas Hobor, and Andrew W. Appel. A fresh look at separation algebras and share accounting. In *7th Asian Symposium on Programming Languages and Systems (APLAS 2009)*, pages 161–177, December 2009.
10. Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.
11. Juan Antonio Navarro Pérez and Andrey Rybalchenko. Separation logic + superposition calculus = heap theorem prover. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 556–566, New York, NY, USA, 2011. ACM.
12. Robert Nieuwenhuis and Albert Rubio. Paramodulation-based theorem proving. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning, volume I*, chapter 7, pages 371–443. Elsevier, 2001.
13. Gordon Stewart, Lennart Beringer, and Andrew W. Appel. Verified heap theorem prover by paramodulation. in preparation, 2011.
14. Thomas Tuerk. *A Separation Logic Framework for HOL*. PhD thesis, Univ. of Cambridge, June 2011.
15. Dinghao Wu, Andrew W. Appel, and Aaron Stump. Foundational proof checkers with small witnesses. In *5th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, August 2003.
16. Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter OHearn. Scalable shape analysis for systems code. In *CAV'08: Computer Aided Verification*, volume 5123 of *LNCS*, pages 385–398. Springer, 2008.