Mostly Sound Type System Improves a Foundational Program Verifier

Josiah Dodds and Andrew W. Appel

Princeton University

Abstract. We integrate a verified typechecker with a verified program logic for the C language, proved sound with respect to the operational semantics of the CompCert verified optimizing C compiler. The C language is known to not be type-safe but we show the value of a provably *mostly sound* type system: integrating the typechecker with the program logic makes the logic significantly more usable. The computational nature of our typechecker (within Coq) makes program proof much more efficient. We structure the system so that symbolic execution—even tactical (nonreflective) symbolic execution—can keep the type context and typechecking always in reified form, to avoid expensive re-reification.

1 Introduction

Hoare logics [15] and separation logics are valuable tools for program understanding. These logics can be straightforward to design when they target small, type-safe programming languages. Program logics for less cooperative programming languages often have complex inference rules that are difficult to apply, requiring extensive proofs even for simple operations.

Despite these difficulties we built a usable program logic for C, proved sound with respect to CompCert's operational semantics (a thorough description of this logic will soon be available [4]). Applying the Hoare logic directly to the C program instead of a lower-level language makes it easy for users to understand how the program relates to the proof. Other tools such as Frama-C [12] and VCC [11] utilize intermediate languages, and translate back to C source when user interaction is required. Our tool permits user interaction at the source level, the *same* source program the user wrote.

Some features of C are unfriendly to Hoare logic, in particular subexpressions with side effects, so we use a slightly different language. This factored language, called C light, is already one of the high-level intermediate languages of the CompCert compiler. Every C light program is a C program and every C program[1] can be translated to C light with only automatic local transformations. If the C program is already in the C light subset, the first phase of CompCert will leave it unchanged (except for parsing it from ASCII into abstract syntax trees).

Our program logic (a higher-order impredicative concurrent separation logic [2]) can be used in (at least) two ways: by applying it interactively in a proof

[1] We use the same specification of the C language as CompCert [16]

To appear in 3rd International Conference on Certified Programs and Proofs (CPP 2013), (Springer Lecture Notes in Computer Science), December 11, 2013.

assistant, or by using the program logic to prove the soundness of a fully automatic static analysis. We have previously demonstrated such a foundationally verified shape analysis [3]; in this paper we focus on interactive proof.

We address the problem of numerous and complex verification side conditions that arise when verifying C programs. In idealized presentations of Hoare logics, we write $P[e/x]$ meaning "assertion P with the value of expression e substituted for program variable x." We implicitly assume that e *has* a value, that is, will evaluate deterministically in the current dynamic context without getting stuck; and we implicitly assume that the value will match the type of variable x (pencil-and-paper presentations may even assume a unityped language). Proof rules for a real language (especially one not designed for Hoare logic) will need many side conditions to establish that these assumptions hold. These hypotheses and side conditions become tedious proof obligations for the user. In this paper we show how to use the type system of C to automatically discharge these hypotheses in the majority of cases. To make this process efficient, we design for the use of computational reflection (Section 6). In many cases we are able to make interactive proofs work in the same way as a pencil-and-paper proof would.

One might think this is obvious: *well-typed programs don't go wrong.* But that is only in a language with a sound type system, and C does not have a sound type system: it is *mostly sound.* In this paper we show that a *mostly sound* type system can still be very useful. In a mostly sound type system well-typed programs go wrong only in well defined cases that can be avoided by proving specific obligations.

Contributions. We formalize a mostly sound type system for C, we prove its mostly-soundness, we implement it computationally in Coq, we integrate it into a program logic for C, we prove the integration is entirely sound with respect to the operational semantics of CompCert C, and we demonstrate that our type system integrates into a tactical proof system (written in Ltac) that is convenient and efficient to apply to C programs using semiautomatic forward symbolic execution[2]. In our mostly sound type system, the typechecker does not just succeed or fail, it calculates an appropriate precondition assertion for the safe evaluation of an expression. In practice, this assertion is **True** for many expressions.

We also discuss other design decisions regarding the interface of a program logic to operational semantics of the C language.

2 Example

Consider a naive Hoare assignment rule for the C language.

$$\vdash \{P[e/x]\}\, x := e\, \{P\} \qquad \text{(naive-assignment)}$$

This rule is not sound with respect to the operational semantics of C. We need a proof that e evaluates to a value. It could, for example, be a division

[2] Our source code can be found at `http://vst.cs.princeton.edu/typecheck/`.

by zero, in which case the program would crash and Hoare triples would not hold. This is an example of the mostly-sound property of the C type system. The expression e might typecheck in the C compiler, but can still get stuck in the operational semantics ("crash" during expression evaluation). A better assignment rule requires e to evaluate:

$$\frac{\exists v. e \Downarrow v}{\vdash \{P[v/x]\}\, x := e\, \{P\}}\text{assignment-ex}$$

The proof of this rule is relatively easy, but the rule is inconvenient to apply because we must use the operational semantics to show that v exists. In fact, any time that we wish to talk about the value that results from the evaluation of an expression, we must add an existential quantifier to our assertion. Showing that an expression evaluates can require a number of additional proofs. If our expression is (y / z), we will need to show that our precondition implies: y and z are both initialized, z ≠ 0, and ¬(y = int_min ∧ z = -1). The latter case causes overflow, which is undefined in the C standard. These requirements will become apparent as we apply the semantic rules.

We can remove the existential variables and make the requirements for evaluation easier to discover by creating specialized rules:

$$\frac{\mathsf{initialized}(y) \qquad \mathsf{initialized}(z) \qquad z \neq 0 \qquad \neg(y = \mathsf{int\_min} \wedge z = -1)}{\vdash \{P[(y/z)/x]\}\ \ x := y/z\ \ \{P\}}\text{intdiv}$$

This moves the proof of expression evaluation into the rule's soundness proof where it only needs to be done once. Such an approach would lead to an overwhelming number of new rules—and it hardly allows for any nested expressions, requiring substantial program rewrites.

Instead, we build a static analysis to generate simple preconditions that will ensure expression evaluation. We define a function typecheck_expr (Section 7) to tell us when expressions evaluate. Now our assignment rules are,

$$\frac{}{\Delta \vdash \{\mathsf{typecheck\_expr}(e, \Delta)\ \wedge\ P[e/x]\}\, x := e\, \{P\}}\text{tc-assignment}$$

$$\frac{}{\Delta \vdash \{\mathsf{typecheck\_expr}(e, \Delta)\ \wedge\ P\}\, x := e\, \{\exists v. x = \mathsf{eval}(e[v/x]) \wedge P[v/x]\}}\text{tc-floyd-assignment}$$

The typecheck_expr is not a side condition, as it is not simply a proposition (Prop in Coq) but a separation-logic predicate quantified over an environment. When run on the expression (y/z) it computes to the assertion $z \neq 0 \wedge \neg(y \neq \mathsf{int\_min} \vee z \neq -1)$ where z and y are not the variables, but the values that result when z and y are evaluated in some environment. The assertions $\mathsf{initialized}(y)$ and $\mathsf{initialized}(z)$ may not be produced as proof obligations if the type-and-initialization context Δ assures that y and z are initialized (Section 5). The calculation of Δ is also part of our type system.

We use the Floyd-style forward assignment rule, instead of the Hoare-style weakest-precondition rule. This is not related to type-checking; separation logic with backward verification-condition generation gives us magic wands which are best avoided when possible [6].

3 Expression evaluation

CompCert defines expression evaluation by an inductive relation eval_expr; failure to evaluate is denoted by omitting tuples from the relation. This is a standard technique in operational semantics, but it is inconvenient in a program logic: assertions would need existential quantifiers, as in $\exists v.e \Downarrow v \wedge P(v)$, "there exists some value such that e evaluates to v and P holds on v." It is much cleaner to say $P(\text{eval\_expr}(e))$, or "P holds on the evaluation of e". We want eval_expr(e) to be a value, not a value-option, or else we (again) need existential quantifiers. Defining evaluation as a function also makes proofs more computational—more efficient to build and check.

We simplify eval_expr in our program logic—and make it computational—by leveraging the typechecker's guarantee that evaluation will not fail. Our total recursive function eval_expr (e: expr) (rho: environ): in environment ρ, expression e evaluates to the value (eval_expr e ρ). When CompCert.eval_expr fails, our own eval_expr (though it is a total function) can return an arbitrary value. We can do this because the function will be run on a program that typechecks—the failure is unreachable in practice. We then prove the relationship between the two definitions of evaluation on expressions that typecheck (we state the theorem in English and in Coq):

Theorem 1 *For all logical environments ρ that are well typed with respect to a type context Δ, if an expression e typechecks with respect to Δ, the CompCert evaluation relation relates e to the result of the computational expression evaluation of e in ρ.*

Lemma eval_expr_relate :
∀ Δ ρ e m ge ve te, typecheck_environ Δ ρ → mkEnviron ge ve te = ρ →
denote_tc_assert (typecheck_expr Δ e) ρ →
Clight.eval_expr ge ve te m e (eval_expr e ρ)

Expression evaluation requires an environment, but when writing assertions for a Hoare logic, we actually write assertions that are functions from environments to Prop. So if we wish to say "the expression e evaluates to 5", we write fun ρ ⇒ eq (eval_expr e ρ) 5. Because Coq does not match or rewrite under lambda (fun), assertions of this form hinder proof automation. Our solution is to follow Bengtson *et al.* [5] in *lifting* eq over ρ: 'eq (eval_expr e) '5. This produces an equivalent assertion, but one that we are able to rewrite and match against. The first backtick lifts eq from val→ val→ Prop to (environ→ val)→ (environ→ val)→ Prop, and the second backtick lifts 5 from val to a constant function in environ→ val.

4 C light

Our program logic is for C, but the C programming language has features that are unfriendly to Hoare logic: *side effects within subexpressions* make it impossible to simply talk about "the value of e" and *taking the address of a local variable*

means that one cannot reason straightforwardly about substituting for a program variable (as there might be aliasing).

The first passes of CompCert translate *CompCert C* (a refined and formalized version of C90 [16]) into *C light*. These passes remove side effects from expressions and distinguish *nonaddressable* local variables from *addressable* locals.[3] We recommend that the user do this in their C code, however, so that the C light translation will exactly match the original program.

C has pointers and permits pointer dereference in subexpressions: d = p→head+q→head. Traditional Hoare logic is not well suited for pointer-manipulating programs, so we use a separation logic, with assertions such as $(p\rightarrow \mathsf{head} \mapsto x) * (q\rightarrow \mathsf{head} \mapsto y)$. Separation logic does not permit pointer-dereference in subexpressions, so to reason about d = p→head+q→head the programmer should factor into: t = p→head; u = q→head; d=t+u; where dereferences occur only at top-level in assignment commands. Adding these restrictions to C light gives us *Verifiable C*, which is not a different semantics but a proper sublanguage, enforced by our typechecker.

A well typed C program might still go wrong. These are the cases where the typechecker must generate assertions. A few of these cases might be surprising, even to experienced C programmers. The following operations are undefined in the C standard, and *stuck* in CompCert C:

- shifting an integer value by more than the word size,
- dividing the minimum int by −1 (overflows),
- subtracting two pointers with different base addresses (i.e., from different malloc'ed blocks or from different addressable local variables),
- casting a float to an int when the float is out of integer range,
- dereferencing a null pointer, and
- using an uninitialized variable.

Some operations, like overflow on integer addition, are undefined in the C standard but defined in CompCert. The typechecker permits these cases.

5 Type Context

Expression evaluation requires an expression and an environment. An expression will evaluate to different values (or Vundef) depending on the environment. To guarantee that certain expressions will evaluate, we will need to control what values can appear in environments. We use a type context to describe environments where our expressions will evaluate to defined values.

Definition tycontext: Type :=
(PTree.t (type * bool) * (PTree.t type) * type * (PTree.t global_spec)).

[3] Xavier Leroy added the SimplLocals pass to CompCert 1.12 at our request, pulling nonaddressable locals out of memory in C light. Prior to 1.12, source-level reasoning about local variables (represented as memory blocks) was much more difficult.

PTree.t(τ) is CompCert's efficient computational mapping data-structure (from identifiers to τ) implemented and proved correct in Coq. The elements of the type context are

- a mapping from temp-var names to type and initialization information,
- a mapping from local variable names to types,
- a return type for the current function, and
- a mapping from global variable names to types (and Hoare specifications for global functions).

The first, second, and fourth items match exactly with the three parts of an environment (environ), which is made up of temporary variable mappings, local variable mappings, and global variable mappings.

A temporary variable is a (local) variable whose address is not taken anywhere in the procedure. Unlike local and global variables, temporaries do not alias—so we can statically determine when their values are modified. If the typechecker sees that a temporary variable is initialized, it knows that it will stay initialized. If the typechecker is unsure, it can emit an assertion guard for the user to prove initialization. Calculating initialization automatically is a significant convenience for the user; proofs in the previous generation of our program logic were littered with definedness assertions in invariants.

The initialization information is a Boolean that tells us if a temporary variable has certainly been initialized. The rules for this are simple, if a variable is assigned to, that variable will always be initialized in code executed after it. The initialization status on leaving an **if-then-else** is the GLB of the two branches. Loops have similar rules.

typecheck_environ checks an environ with respect to a tycontext. It does not generate assertions as typecheck_expr does, it simply returns a Boolean that if true claims all of the following:

- If the type context contains type information for a temporary variable the temp environment contains a value for that variable. If the variable is claimed to be initialized, that value must belong to the type claimed in the type environment.
- If the type context contains type information for a (addressable) local variable, the local variable environment contains a local variable of matching type.
- If the type context contains type information for a global variable, the global environment contains a global variable of matching type.
- If the type context contains type information for a global variable, either
  - the local variable environment does not have a value for that variable or
  - the type context has type information for that variable as a local variable.

The fourth point is required because local variables shadow global variables.

Initialization information is changed by statements. We only know a variable is initialized once we see that it is assigned to. Our typechecker only needs to operate at the level of expressions, so we can merge maintainence of the type

context into the definition of our logic rules. We will now give some of these rules and explain how they work to keep the type context correct.

We provide a function

Definition func_tycontext (func: function) (V: varspecs) (G: funspecs): tycontext

that automatically builds a correct type context (for the beginning of the function body) given the function, local, and global specifications. The resulting context contains every variable used in the function matched with its correct type. We have proved that the environment created by the operational semantics when entering a function body typechecks with respect to the context generated by this function. Once the environment is created, the Hoare rules use the function updatetycon to maintain the type context across statements.

$$\frac{\Delta \vdash \{P\}\, c\, \{Q\} \qquad \Delta' = \mathsf{updatetycon}(\Delta, \mathsf{c}) \qquad \Delta' \vdash \{Q\}\, d\, \{R\}}{\Delta \vdash \{P\}\, c; d\, \{R\}}\text{seq}$$

updatetycon tells us that variables are known to be initialized after they are assigned to. It also says that variables are initialized if they were initialized before the execution of any statement, and that a variable is initialized if we knew it was initialized at the end of both branches of a preceding **if** statement. When we say initialized, we mean *unambiguously* initialized, meaning that it will be initialized during all possible executions of the program.

The type context is deeply integrated with the type rules. We write our hoare judgment as $\Delta \vdash \{P\}\, c\, \{Q\}$. We added the type context Δ because instead of quantifying over all environments as a normal Hoare triple does, we quantify only over environments that are well typed with respect to Δ. This has a huge benefit to the users of the rules: they do not need to worry about the contents of Δ, and they do not need to show that the environment typechecks or mention Δ explicitly in preconditions. Our *rule of consequence* illustrates what we always know about Δ:

$$\frac{\mathsf{typecheck\_environ}(\rho, \Delta) \wedge P \vdash P' \qquad \Delta \vdash \{P'\}\, c\, \{R\}}{\Delta \vdash \{P\}\, c\, \{R\}}$$

The conjunct $\mathsf{typecheck\_environ}(\rho, \Delta)$ gives the user more information to work with in proving the goal. Without this, the user would need to explicitly strengthen assertions and loop invariants to keep track of the initialization status of variables and the types of values contained therein.

With func_tycontext and updatetycon the rules can guarantee that the type context is sound at all times. To keep the type context updated, the user must simply apply the normal Hoare rules, with our special Hoare rule for statement sequencing shown above.

6 Keeping it real

Proof by reflection is a three-step process. A program is *reified* (made real) by translating it from Prop to a data structure that can be reasoned about

computationally. Computation is then performed on that data structure and the result is *reflected* back into Prop where it can be used in a proof (see bottom of Fig. 1). Reification is costly, however, so our approach is different. We provide a brief example of standard reflection in order to discuss the differences.

We could use reflection, for example, to remove True and False from propositions containing conjunctions and disjunctions. Chlipala discusses a similar problem in more detail [10]. The first step is to define a syntax that represents the propositions of interest. Our tc_assert syntax has 14 cases—to cover issues described in Section 4—of which we show the first four, followed by a function to *reflect* this syntax into the logic of propositions:

```
Inductive tc_assert :=
| tc_FF | tc_TT: tc_assert
| tc_andp': tc_assert → tc_assert → tc_assert
| tc_nonzero: expr → tc_assert
| ... end.

Definition denote_tc_nonzero (v: val) :=
    match v with Vint i ⇒ if negb (Int.eq i Int.zero) then True else False
                 | _⇒ False end.
  ⋮
Fixpoint denote_tc_assert (a: tc_assert) : environ → Prop :=
  match a with
  | tc_FF ⇒ 'False | tc_TT ⇒ 'True
  | tc_andp' b c ⇒ 'and (denote_tc_assert b) (denote_tc_assert c)
  | tc_nonzero e ⇒ 'denote_tc_nonzero (eval_expr e)
  | ... end.
```

If we were doing standard reflection—which we are not—we would then write a *reification* tactic,

```
Ltac p_reify P :=
  match P with
    | True ⇒ tc_TT        | False ⇒ tc_FF
    | ?P1 ∧ ?P2 ⇒ let t1 := p_reify P1 in let t2 := p_reify P2 in constr:(tc_andp t1 t2) ...
```

Finally, we do write a simplification function that operates by recursion on tc_assert. Comparing the steps, we see that the reflection step, as well as any transformations on our reified data, will be computational. Reification, on the other hand, operates by matching proof terms. The computational steps are efficient because they operate in the same way as any functional program. Ltac is less efficient because it operates by matching on arbitrary proof terms.

To avoid the costly reification step, the typechecker generates *syntax* directly—so we can perform the computation on it immediately, without need for reification. This keeps interactive proofs fast. The typechecker keeps all of its components real, meaning there are no reification tactics associated with it.

We use this design throughout the typechecker. We keep data reified for as long as possible, reflecting it only when it is in a form that the user needs to solve directly. The difference between the two approaches can be seen in Fig. 1.

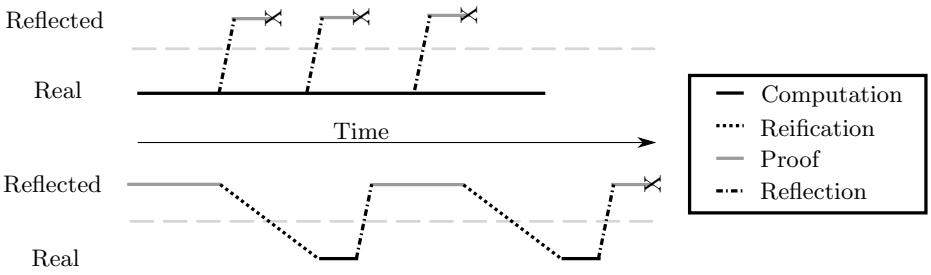

Fig. 1: Our approach (top) vs. standard reflection (bottom)

7 Typechecker

The typechecker produces assertions that, if satisfied, prove that an expression will always evaluate to a value.

In the C light abstract syntax produced by CompCert from C source code, every subexpression is syntactically annotated with a C-language type, accessed by (typeof e). Thus our typing judgment does not need to be of the form $\Delta \vdash e : \tau$, it can be $\Delta \vdash e$, meaning that e typechecks according to its own annotation.

We define a function to typecheck expressions with respect to a type context:

```
Fixpoint typecheck_expr (Δ : tycontext) (e: expr) : tc_assert :=
  let tcr := typecheck_expr Δ in  match e with
    | Econst_int _(Tint ___) ⇒ tc_TT
    | Eunop op a ty ⇒ tc_andp
          (tc_bool (isUnOpResultType op a ty) (op_result_type e)) (tcr a)
    | Ebinop op a1 a2 ty ⇒ tc_andp
          (tc_andp (isBinOpResultType op a1 a2 ty) (tcr a1)) (tcr a2)
 ... end.
```

This function traverses expressions emitting conditions that ensure that the expressions will evaluate to a value in a correctly typed environment. The typechecker is actually a mutually recursive function: one function typechecks rvalues and the other typechecks lvalues. For convenience, this paper only discuss rvalues. Although CompCert's operational semantics are written as an inductive Coq type, they also have parts that are computational. For example, when we need to typecheck operation expressions, we use functions from CompCert that *classify* them. The following function is used to typecheck binary operations:

```
Definition isBinOpResultType op a1 a2 ty : tc_assert :=
match op with
  | Oadd ⇒ match classify_add (typeof a1) (typeof a2) with
            | add_default ⇒ tc_FF
            | add_case_ii _ ⇒ tc_bool (is_int_type ty)
            | add_case_pi _ _⇒ tc_andp (tc_isptr a1) (tc_bool (is_pointer_type ty))
                  ... end
  ... end.
```

Classification functions determine which of the overloaded semantics of operators should be used. These semantics are determined based on the types of the operands. The C light operational semantics uses the constructors (add_case_ii, add_case_pi, (and so on)) to choose whether to apply integer-integer add (ii), pointer-integer add (pi), and so on. The typechecker uses the same constructors add_case_ii, add_case_pi, to choose type-checking guards, as shown above.

Despite the reuse of CompCert code on operations, the bulk of the typechecker's code checks binary operations. This is because of the operator overloading on almost every operator in C. The typechecker looks at eight types of operations (shifts, boolean operators, and comparisons can be grouped together as they have the exact same semantics with respect to the type returned). Each of these has approximately four behaviors in the semantics giving a total of around thirty cases that need to be handled individually for binary operations.

The code above is a good representation of how the typechecker is implemented. The first step is to match on the syntax. Next, if the expression is an operation, we use CompCert's classify function to decide which overloaded behavior to use. From there, we generate the appropriate assertion.

8 Soundness

The soundness statement for our typechecker is:

Theorem 1 *If the dynamic environment ρ is well typed with respect to the static type context Δ (Section 5), and the expression e typechecks with respect to Δ producing an assertion that in turn is satisfied in ρ, then the value we get from evaluating e in ρ (Section 3) will match the type that e is labeled with.*

```
typecheck_environ ρ Δ = true → denote_tc_assert (typecheck_expr Δ e) ρ →
typecheck_val (eval_expr e ρ) (typeof e) = true.
```

This guarantees that an expression will evaluate to the right kind of value: integer, or float, or pointer. As a corollary we guarantee the absence of Vundef, which has no type.

The proof proceeds by induction on the expression. One of the most difficult parts of the soundness proof is the proofs about binary operations. We need to prove that when a binary operation typechecks it evaluates to a value as a case for the main soundness proof. The proof is difficult because of the number of

cases. When all integers and floats of different sizes and signedness are taken into account, there are seventeen different CompCert types. This means that there are 289 combinations of two types. A proof needs to be completed for each combination of types for all seventeen C light operators, leading to a total of 4913 cases that need to be proved. Each proof requires a decent amount of work, so the amount of memory taken by the proof becomes a problem. We use lemmas to group some of the cases together to keep the proof time reasonable.

These cases are not all written by hand: we automate using Ltac. Still, the proofs are large, and Coq takes almost 4 minutes to process the file containing the binary operation proofs.

9 A Tactical Proof

In this section, we apply our C light program logic to verify a simple C program interactively in Coq. We will verify the C program:

int assigns (**int** a) { **int** b, c; c = a∗a; b = c/3; **return** b; }

We begin by passing our program through the CompCert clightgen tool to create a file that we can read into Coq. The next step is to specify our program. The specification for this program is:

$$\Delta \vdash \{\mathsf{Vint}(v) = \mathrm{eval}\,\mathsf{a}\,\rho\}\ \mathsf{assigns}\ldots\ \{\mathsf{retval}\,\rho = \mathsf{Vint}((v*v)/3)\}$$

Barring any unexpected evaluation errors, we expect this specification to hold. The specification states that in an arbitrary initial state, the program will either infinite loop or terminate in a state in which retval = (a∗a)/3. For this example we will focus on proving the specification of the function body:

```
Lemma body_test : semax_body Vprog Gtot f_assigns assign_spec.
Proof. start_function. name a _a. name b _b. name c _c.
       forward. forward. go_lower. normalize. solve_tc. forward. go_lower.
       (* ... prove that the function-body postcondition implies the
       function-specification postcondition ... *) Qed.
```

The function semax_body creates a triple for a function body given a list of global variable specifications (Vprog, the empty list), a list of global function specifications (Gtot, list of this function and main), the pointer to the function (f_assigns, pointer from program .v file), and a specification (assign_spec, the Coq version of the triple shown above). The tactic start_function unfolds semax_body and ensures that the precondition is in a usable form. The relation semax defines the triple we have seen throughout the paper.

The name tactic, and the name hypotheses it generates, relate variable names to value names. For example _a is the name of the variable a in the program. The tactic name a _a tells the tactics that values associated with evaluating _a should be called a.

We will examine the proof state at a few points to highlight the forward and go_lower tactics and show goals generated by the typechecker. We have replaced

C light AST with C-like syntax in the lines marked *(∗pseudocode ∗)*. Assertions are in a canonical form, separated into PROP (propositions that don't require state), LOCAL (assertions lifted over the local environment), and SEP (separation assertions over memory). Empty assertions for any of these mean True.

```
a : name _a
b : name _b
c : name _c
Δ := initialized _c (func_tycontext f_assigns Vprog Gtot) : tycontext
============================
 semax Δ (PROP ()
    LOCAL (‘eq (eval_id _c) (eval_expr(_a ∗ _a)); (‘eq (eval_id _a) v))  SEP ())
    (_b = _c / 3; return _b;) (∗ pseudocode standing for C-light AST ∗)
    (function_body_ret_assert tint (_a ∗ _a / 3) = retval)
```

Above is the state after we apply forward for the first time. This tactic performs forward symbolic execution using Coq tactic programs, as various authors have demonstrated [1,9,5,17]. In effect, forward applies the appropriate Hoare rule for the next command, using the sequence rule if necessary. The backtick (‘) is the "lifting" coercion of Bengtson *et al.* [5]. function_body_ret_assert tells us that our postcondition talks about the state when the program returns successfully. A program that does not return successfully will not satisfy this triple.

The forward tactic makes a decision when it sees an assignment. In general, it uses the Floyd assignment rule that existentially quantifies the "old" value of the variable being assigned to (in this case c). It needs to do this because otherwise we would lose information from our precondition by losing the old value of c. This means the postcondition would end up in the form "∃ old, ...". If the variable doesn't appear in the precondition, however, the existential can be removed because it will never be used. The forward tactic checks to see if it needs to record the old value of the variable or not. In this case, it sees that c is not in the precondition and does not record its old value.

In the proof so far (after symbolic execution of the command c=a∗a;) we have not yet seen a typechecking side condition—not because they were automatically solved, but because they were never generated in the first place. They were checked computationally, but no assertion about them is given. The condition that a be initialized immediately evaluates to True and is dispelled trivially.

Finally we notice that Δ has been updated with initialized _c. This was done by the sequence rule as discussed in Section 5.

Applying forward again gives the following separation-logic side condition:

```
a : name _a
b : name _b
c : name _c
Δ := initialized _b (initialized _c (func_tycontext f_assigns Vprog Gtot) : tycontext
============================
 PROP()  LOCAL(tc_environ Δ; ‘eq (eval_id _c) (eval_expr (_a ∗ _a)); (‘eq (eval_id _a) v))
 SEP(‘TT) ⊢ local (tc_expr Δ (_c / 3)) && local (tc_temp_id _b tint Δ)
```

This is an entailment, asking us to prove the right hand side given the left hand side. We need to show that the expression on the right hand side of the assignment typechecks, and that the id on the left side typechecks. We would expect to see that: c is initialized, $3 \neq 0$ and $\neg(c = \mathsf{min\_int} \wedge 3 = -1)$.

Why is it useful to have tc_environ Δ? This entailment is *lifted* (and quantified) over an abstract environ ρ; if we were to intro ρ and make it explicit, then we would have conditions about eval_id _c ρ, and so on. To prove these entailments, we need to know that (eval_id _a ρ) and (eval_id _c ρ) are defined and well-typed.

In a paper proof it is convenient to think of an integer *variable* _a as if it were the same as the *value* obtained when looking _a up in environment ρ—we write this (eval_id _a ρ). In general, we can not think this way about C programs because in an arbitrary environment, _a may be of the incorrect type or uninitialized. In an environment ρ that typechecks with respect to some context Δ, however, we can bring this way of thinking back to the user. Our automatic go_lower tactic, after introducing ρ, uses the name hints to replace every use of (eval_id _a ρ) with simply a, *and* it proves a hypothesis that the value a has the expected type. In the case of an int, it does one step more: knowing that the value (eval_id _a ρ) typechecks implies it must be Vint x for some x, so it introduces a as that value x. (Again, the name a is chosen from the hint, a: name _a.) Thus, the user can think about values, not about evaluation, just as in a paper proof. Our go_lower tactic, followed by normalize for simplification converts the entailment into

```
c : int
a : int
H0 : Vint c = Vint (Int.mul a a) (*simplified*)
============================
 denote_tc_nodivover (Vint c) (Vint (Int.repr 3))
```

All we are left with is the case that the division doesn't overflow. The other conditions (c is initialized, $3 \neq 0$) have computed to True and simplified away. We can no longer see the variables _c and _a.

Now we can apply some simple Boolean rewrite rules with solve_tc and solve the goal. Not all typechecker-introduced assertions will be so easy to solve, of course; in place of solve_tc the user might have to do some real work.

The rest of the proof advances through the return statement, then proves that the postcondition after the return matches the postcondition for the specification. In this case it is easy, just a few unfolds and rewrites.

10 Related Work

Frama-C is a framework for verifying C programs [12]. It presents a unified assertion language to enable static analysis cooperation. The assertion language allows users to specify only first-order properties about programs, and does not include separation logic. The Value analysis [8] uses abstract interpretation to determine possible values, giving errors for programs that might have runtime

errors. The WP plugin uses weakest precondition calculus to verify triples. WP is only correct when first running Value which may result in some verification conditions that can then be verified by WP along with the function specifications. Frama-C does not seem to have any soundness proof.

VCC is a verifier for concurrent C programs. It works by translating C programs to Boogie, which is a combination of an intermediate language, a VC generator, and an interface to pass VCs off to SMT solvers such as Z3. VCC adds verification conditions that ensure that expressions evaluate.

Greenaway *et al.* [14] show a verified conversion from C into a high-level specification that is easy for users to read. They do this by representing the high-level specification in a monadic language. They add guards during their translation out of C in order to ensure expression evaluation (this is done by Norrish's C parser [18]). Many of these guards will eventually be removed automatically. Their tool is proved correct with respect to the semantics of an intermediate language, not the semantics of C. The expression evaluation guards are there to ensure that expressions always evaluate in the translated program, because there is no concept of undefined operations in the intermediate language. Without formalized C semantics, however, the insertion of guards must be trusted to actually do this. This differs from our approach where the typechecker is proved sound with respect to a C operational semantics; so we have more certainty that we have found all the necessary side conditions. Another difference is that they produce all the additional guards and then solve most of them automatically, while we avoid creating most such assertions. Expression evaluation is not the main focus of Greenaway's work, however, and the ideas presented for simplifying C programs could be useful in conjunction with our work.

Bengtson *et al.* [5] provide a framework for verifying correctness of Java-like programs with a higher-order separation logic similar to the one we use. They use a number of Coq tactics to greatly simplify interactive proofs. Chlipala's Bedrock project [9] also aims to decrease the tedium of separation logic proofs in Coq, with a focus on tactical- and reflection-based automation of proofs about low level programs. Bengtson operates on a Java-like language and Chlipala uses a simple but expressive low-level continuation-based language. Earlier versions of our work (Appel [1]) used a number of tactics to automate proofs as well. In this system, the user was left with the burden of completing proofs of expression evaluation.

The proof rules we use in this paper are also used in the implementation of a verified symbolic execution called VeriSmall [3]. VeriSmall does efficient, completely automatic shape analysis.

Tuerk's HolFoot [19] is a tactical system in HOL for separation logic proofs in an imperative language. Tuerk uses an idealized language "designed to resemble C," so he did not have to address many of the issues that our typechecker resolves.

One of Tuerk's significant claims for HolFoot is that his tactics solve purely shape-analysis proofs without any user assistance, and as a program specification is strengthened from shape properties to correctness properties, the system smoothly "degrades" leaving more proof obligations for the interactive user. This

is a good thing. As we improve our typechecker to include more static analysis, we hope to achieve the same property, with the important improvement that the static analysis will run much faster (as it is fully reified), and only the user's proof goals will need the tactic system.

Our implementation of computational evaluation is similar to work on executable C semantics by Ellison and Rosu [13] or Campbell [7]. Their goals are different, however. Campbell, for example, used his implementation to find bugs in the specification of the CompCert semantics. We, on the other hand, are accepting the CompCert semantics as the specification of the language we are operating on. Ellison and Rosu have the goal of showing program correctness, which is a similar goal to ours. They show program correctness by using their semantics as a debugger or an engine for symbolic execution.

11 Conclusion

By integrating a typechecker with a program logic, we improve the usability of the logic. Our system maintains type and initialization information through a Hoare-logic proof in a continuously reified form, leading to efficiency and improved automation. Automatic maintenance of the well-typedness of the local-variable environment (tc_environ) makes it easy to discharge "trivial" (but otherwise annoying) subgoals. We have a proof of soundness of the whole system (w.r.t. the CompCert C operational semantics) even though C does not actually have a sound type system.

We have used the tool to prove full functional correctness of programs such as list sum, list reverse, imperative thread queue, and object-oriented message passing. We are currently working on safety and correctness of an implementation of the SHA-256 hash function.

Our style of integrating a static analysis with a program logic should not be limited to a typechecker. Many of the ideas presented in this paper could be used to integrate other static analyses with program logics. Symbolic execution, abstract interpretation, and more (or less) sound typecheckers could all be integrated in a similar fashion

References

1. Andrew W. Appel. Tactics for separation logic, 2006.
2. Andrew W. Appel. Verified Software Toolchain. In *European Symposium on Programming*, pages 1–17, 2011.
3. Andrew W. Appel. VeriSmall: Verified Smallfoot shape analysis. In *First International Conf. on Certified Programs and Proofs (CPP'11), LNCS 7086*, pages 231–246, 2011.
4. Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. *Program Logics for Certified Compilers*. Cambridge, (to appear) 2014.

5. Jesper Bengtson, Jonas Braband Jensen, and Lars Birkedal. Charge! A framework for higher-order separation logic in Coq. In *Third International Conference on Interactive Theorem Proving, LNCS 7406*, pages 315–331. Springer, August 2012.
6. Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. In *APLAS'05: Third Asian Symposium on Programming Languages and Systems (LNCS 3780)*, pages 52–68, 2005.
7. Brian Campbell. An executable semantics for compcert c. In Chris Hawblitzel and Dale Miller, editors, *Certified Programs and Proofs*, volume 7679 of *Lecture Notes in Computer Science*, pages 60–75. Springer Berlin Heidelberg, 2012.
8. Géraud Canet, Pascal Cuoq, and Benjamin Monate. A value analysis for C programs. In *Ninth Source Code Analysis and Manipulation, 2009*, pages 123–124. IEEE, 2009.
9. Adam Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *PLDI'11*, pages 234–245, 2011.
10. Adam Chlipala. *Certified Programming With Dependent Types*, chapter Reflection. MIT Press, 2013.
11. Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Micha Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. Vcc: A practical system for verifying concurrent C. In *Theorem Proving in Higher Order Logics*, volume 5674 of *LNCS*, pages 23–42. Springer Berlin Heidelberg, 2009.
12. Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C. In *Software Engineering and Formal Methods*, pages 233–247. Springer, 2012.
13. Chucky Ellison and Grigore Roşu. An executable formal semantics of C with applications. In *Proceedings of the 39th Symposium on Principles of Programming Languages (POPL'12)*, pages 533–544. ACM, 2012.
14. David Greenaway, June Andronick, and Gerwin Klein. Bridging the gap: Automatic verified abstraction of C. In *Third International Conference on Interactive Theorem Proving, LNCS 7406*, pages 99–115. Springer, August 2012.
15. C A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):578–580, October 1969.
16. Xavier Leroy. The CompCert verified compiler, software and commented proof. http://compcert.inria.fr, June 2013.
17. Andrew McCreight. Practical tactics for separation logic. In *TPHOL: International Conference on Theorem Proving in Higher Order Logics*, pages 343–358, 2009.
18. Michael Norrish. C-to-isabel parser. http://www.ssrg.nicta.com.au/software/TS/c-parser/, 2013.
19. Thomas Tuerk. A formalisation of Smallfoot in HOL. In *Theorem Proving in Higher Order Logics*, LNCS, pages 469–484. Springer, 2009.

This material is based on research sponsored by the Air Force Office of Scientific Research under agreement FA9550-09-1-0138 and by DARPA under agreement number FA8750-12-2-0293. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA of the U.S. Government.