Modular Development of Certified Program Verifiers with a Proof Assistant *

Adam Chlipala
University of California, Berkeley, Computer Science Division
adamc@cs.berkeley.edu

Abstract

I report on an experience using the Coq proof assistant to develop a program verification tool with a machine-checkable proof of full correctness. The verifier is able to prove memory safety of x86 machine code programs compiled from code that uses algebraic datatypes. The tool's soundness theorem is expressed in terms of the bit-level semantics of x86 programs, so its correctness depends on very few assumptions. I take advantage of Coq's support for programming with dependent types and modules in the structure of my development. The approach is based on developing a library of reusable functors for transforming a verifier at one level of abstraction into a verifier at a lower level. Using this library, it's possible to prototype a verifier based on a new type system with a minimal amount of work, while obtaining a very strong soundness theorem about the final product.

Categories and Subject Descriptors F.3.1 [*Specifying and Verifying and Reasoning about Programs*]: Mechanical verification; F.3.3 [*Studies of Program Constructs*]: Type structure; D.2.13 [*Reusable Software*]: Domain engineering

General Terms Languages, Verification

Keywords interactive proof assistants, programming with dependent types, proof-carrying code

1. Introduction

It is widely accepted that bugs in software are a very serious problem today, creating both high costs of software development and far too many exploitable security holes. The research community has developed a plethora of techniques for finding bugs in programs or even proving programs to be free of certain classes of bugs. In most cases, these bug-finders and verifiers are applied post-facto to programs developed using standard, informal techniques. However, there has long been support in the community for the idea of applying formal methods throughout the software lifecycle. In a sense, increasingly rich static type systems are such a class of solutions. It seems fair to classify them as formal specification and proof systems, but the prevalence of tools that make type systems easy to use prevents most programmers from thinking of them in such imposing terms. A general and interesting question is, how much more effective can we make the software development process by using even more expressive formal systems from the time the first line of code is written?

In this paper, I will present the results of a particular experiment along these lines. The interesting twist to the specific problem I tackle is that it adds an additional layer of reflection to the approach I just described: I have been working on proving the correctness of programs that prove the correctness of programs. A proof of this kind provides correctness proofs "for free" for all the inputs the verified verifier can handle.

In particular, I have developed a framework for coding *certified program verifiers* for x86 machine code programs. The end results are executable programs that take x86 binaries as input and return either "Yes, this program satisfies its specification" or "I'm not sure." By virtue of the way that these verifiers are constructed using the Coq proof assistant, it is guaranteed that they are sound with respect to the unabstracted bit-level semantics of x86 programs. Yet this guarantee does not make development impractical; by re-using components outfitted with rich semantic interfaces, it's possible to whip together a certified verifier based on, for example, a new type system in a few hundred lines of code and an afternoon's time.

This work is related to two main broad research agendas, which I will describe next: proof-carrying code and general software development techniques based on dependent types and interactive theorem proving.

1.1 Applications to Proof-Carrying Code

The idea of certified program verifiers has important practical ramifications for foundational proof-carrying code (FPCC) [App01]. Like traditional proof-carrying code (PCC), FPCC is primarily a technique for allowing software consumers to obtain strong formal guarantees about programs before running them. The author of a piece of software, who knows best why it satisfies some specification that users care about, is responsible for distributing with the executable program a formal, machine-checkable proof of its safety. He might construct this proof manually, but more likely he codes in a high-level language that enforces the specification at the source level through static checks, allowing a *certifying compiler* for that language to translate the proofs (explicit or implicit) that hold at the source level into proofs about the resulting binaries.

The original PCC systems were very specialized. A particular system would, for instance, only accept proofs based on a fixed type system. FPCC addresses the two main problems associated with this design.

First, traditional PCC involves trusting a set of relatively high-level axioms about the soundness of a type system. We would rather

* This research was supported in part by a National Defense Science and Engineering Graduate Fellowship and NSF Grants CCF-0524784, CCR-0234689, CNS-0509544, and CCR-0225610.

ICFP'06 September 16–21, 2006, Portland, Oregon, USA.

not have to place our faith in the soundness of so large a formal development, so FPCC reduces the set of axioms to deal only with the *concrete* semantics of the underlying machine model. If the soundness of a type system is critical to a proof, that soundness lemma must be proved from first principles.

The other problem is that a specialized PCC system is not very flexible. Typically, one of these systems can only check safety proofs for the outputs of a particular compiler for a particular source language. If you want to run programs produced with different compilers or that otherwise require fundamentally different proof strategies, then you will need to install one trusted proof checker or set of axioms for each source. This is far from desirable from a security standpoint, and FPCC fixes this problem by requiring all proofs to be in the same language and to use the same relatively small set of axioms. The axiomatization of machine semantics is precise enough that the more specific sets of axioms used in traditional PCC are usually derivable with enough work, if they were sound in the first place.

The germ of the project I'll describe comes from past work on improving the runtime efficiency of FPCC program checking [CCN06]. Perhaps the largest obstacle to practical use of FPCC stems from the delicate trade-offs between generality on one hand and space and time efficiency of proofs and proof checkers on the other. Program verifiers like the Java bytecode verifier have managed to creep into wide use almost unnoticed by laypeople, but naive FPCC proofs are much larger than the metadata included with Java class files and take much longer to check. It's unlikely that this increased burden would be acceptable to the average computer user.

Fundamentally, custom program verifiers with specialized algorithms and data structures have a leg up on very general proof-based verifiers. In our initial work on certified program verifiers, we proposed getting the best of both worlds by moving up a level of abstraction: allow developers to ship their software with specialized *proof-carrying verifiers*. These verifiers have the semantic functionality of traditional program verifiers and model-checkers, but they also come with machine-checkable proofs of soundness. Each such proof can be checked *once* when a certified verifier is installed. After the proof checks out, the verifier can be applied to any number of similar programs. These later verifications require no runtime generation or checking of uniform proof objects, which we found to be the major bottleneck in previous experience with FPCC. Our paper [CCN06] presents performance results showing an order of magnitude improvement over all published verification time figures for FPCC systems for Typed Assembly Language [MWCG99] programs, by using a certified verifier. The verifier had a complete soundness proof, so no formal guarantees were sacrificed to win this performance.

The main problem that we encountered was in the engineering issues of proof construction. We used a more or less traditional approach to program verification in proving the soundness of our verifiers, writing them in a standard programming language and extracting verification conditions that imply their soundness. Keeping the proof developments in sync with changes to verifier source code was quite a hassle. We also found that the structure of the verifier program and its proof were often very closely related, leading to what felt like duplicate work. I decided to try investigating what could be gained by writing verifiers from the start in a language expressive enough to encode verifier soundness in its type system.

1.2 Programming with Dependent Types and Proofs

Coq and related formal logic tools are based on Martin-Löf constructive type theory. They identify logical specifications with types and proofs with values of those types. Coq allows values traditionally thought of as "programs" and "proofs" to coexist in the same calculus; the former simply have the kinds of types we're used to seeing, while the latter have logical propositions as types. If we avoid dependent types, Coq essentially provides a pure and total subset of ML. Through selective use of dependent types and "logical" features, we can choose the precision of specifications that the type checker should ensure, working towards a program type that implies full correctness. Through its *program extraction* feature, Coq can build an ML version of a Coq term that has a type associated with "programs," which can then be compiled into an efficient executable version. Thus, one reasonable view of Coq is as a programming environment supporting very expressive dependent types.

Recent programming languages like Epigram [MM04], ATS [CX05], and RSP [WSW05] have drawn on the theory underlying more traditional approaches associated with theorem provers in providing support for practical programming with dependent types. Why create these new languages when tools like Coq already exist? The answer is that Coq is primarily designed for doing math, not writing software. It is missing many convenient features we expect in "real" programming languages, like non-terminating functions, imperative state, and exceptions. ATS and RSP allow the sound use of features like these in the presence of explicit proofs. It's also true that dealing with type equalities in Coq can be quite aggravating. Support for automatic and implicit proof and use of type equalities and other "obvious" lemmas is an important time-saving feature.

The hot subject of generalized algebraic datatypes (GADTs) [She04] is also closely related to these issues. GADTs are a particular restriction of the type systems supported by the tools I've mentioned above. The restriction is designed to make type inference more feasible than it has any hope of being in any of those tools, whose type systems are strong enough to express most of mathematics.

Despite the potential objections to use of Coq that I've listed, I hope to make a case here that it is a good choice for programming with rich specifications. The foundations of both Coq's implementation and its formal metatheory are very simple and elegant compared to approaches based on traditional programming. A small dependently-typed lambda calculus suffices for the effective encoding of most of math and, as I hope to justify, most of programming. As a mature tool for formal math, Coq has many features for organizing mathematical developments and automating proofs that don't have clear translations to environments with larger sets of orthogonal primitive features.

Program verifiers make a nice subject for a study of this kind. As I summarized in the last section, certification of verifiers has significant application to proof-carrying code and related areas. There are also established, rigorous standards of what the correctness of a verifier is. Finally, program analysis tools are frequently written in a purely functional style with no non-terminating functions.

Leroy's recent work on certifying a complete compiler written in Coq [Ler06] provides some strong evidence that Coq can be, at the least, an effective starting point in developing the ideal system for programming with specifications. That work mostly takes the traditional approach of implementing the compiler without dependent types and then proving it correct. In the work I will present here, I've tried to take as much advantage of dependent types as I can to simplify development.

1.3 Contributions

In the remainder of the paper, I will describe my approach to the modular development of certified program verifiers. The key novelty is the use of dependent types in the "programming" part of development and in conjunction with Coq's ML-style module system. The end result is a set of components with rich interfaces that can be composed to produce a wide range of verifiers with low cost relative to the strength of the formal guarantees that result.

I'll begin by giving some preliminary background on the FPCC problem setting and on dependent types, extraction, and modules in Coq. With these tools available, I describe the design and implementation of a library to ease the development of certified verifiers via functors with rich interfaces. Next, I describe a particular completed application of that library, a memory safety verifier for machine code programs that use algebraic datatypes. I conclude by comparing with related work and summarizing the take-away lessons from the experience.

2. Preliminaries

2.1 Types and Extraction in Coq

To introduce the basics of dependent types in Coq, I'll start with a definition for the Coq version of the polymorphic `option` type familiar to ML programmers (and Haskell programmers as `Maybe`):

```
Inductive option (T : Set) : Set :=
  | Some : T -> option T
  | None : option T.
```

This has more or less the same information content as the ML definition. The Coq version is a little more verbose, because here we use a general mechanism designed to handle much more complicated types. Since Coq unifies values, types, proofs, and propositions in a single syntactic class, `option` is expressed as a function from sets to sets, with `T` bound as the name of the function's argument. Also, the full type of each constructor is given explicitly, without the result type being implicit. This will be familiar to readers who have seen GADTs, as the same explicitness is necessary there. This is because the result type of a constructor can depend on the types of the arguments, in the case of GADTs; or even on the *values* of the arguments, in Coq.

Here we use an *inductive definition* of a family of `Set`s. The high-level intuition is that runnable programs with computational content belong to the *sort* `Set`, while mathematical proofs belong to `Prop`. The types of programs are introduced with `Inductive` definitions with `Set` specified immediately before the `:=`, while propositions (i.e., the types of proofs) are introduced with `Prop` in that position.

Now we can consider this slight modification of `option`'s definition:

```
Inductive poption (P : Prop) : Set :=
  | PSome : P -> poption P
  | PNone : poption P.
```

Here I've changed the *argument type* of the polymorphic `poption` type to `Prop`, but left the *result type* the same at `Set`. A `poption` is a package that might contain a proof of a particular proposition or might contain nothing at all. The interesting thing about it is that, while it may contain it proof, it itself exists *as a program*. A helpful way to think about `poption` is as the rich return type of a potentially incomplete decision procedure that either determines the truth of a proposition or gives up.

As a concrete example, consider this function that determines if its argument is even:

```
Definition isEven : forall (n : nat),
    poption (even n).
  refine (fix isEven (n : nat)
      : poption (even n) :=
    match n return (poption (even n)) with
      | O => PSome _ _
      | S (S n) =>
        match isEven n with
          | PSome pf => PSome _ _
          | PNone => PNone _
        end
      | _ => PNone _
    end); auto.
Qed.
```

I'm using a lot of Coq notation here, but only a few details are relevant. First, the type of `isEven` is given as a dependent function type, where Coq uses `forall` in place of the more usual Π. Second, we provide a *partial* implementation for the function. We don't want to fill in the proofs manually; as Coq is designed for formalizing math, we rightfully expect that it can do this dirty work for us.

By using a `Definition` command (terminated with a period) without providing an expansion for our new definition, we declare that we will construct this value with Coq's *interactive proof development mode*. In this mode, proof goals are iteratively refined into subgoals known to imply the original, until all subgoals can be eliminated in atomic proof steps. Individual refinements are expressed as *tactics*, small, untyped programs in the language that Coq provides for scripting proof strategies. Theorem proving with tactics isn't my focus in this work, so I will just describe the two simple tactics that I've used in the example.

At any stage in interactive proof development, the goal is expressed as a search for a term having a particular type. The `refine` tactic specifies a *partial* term; it contains underscores indicating holes to be filled in, and we believe that there is some substitution for these holes that leads to a term of the proper type. Some holes are filled in automatically using standard type inference techniques, while the rest are added as *new subgoals in the proof search*.

In the use of `refine` in the example, I suggested a recursive function definition, filling in all of the computational content of the function and leaving out the details of constructing proofs. The holes standing for proofs turn out to be the only ones that Coq doesn't fill in through unification, and I invoke the `auto` automation tactic to solve these goals through Prolog-style logic programming.

We can make the code nicer-looking through some auxiliary definitions and by extending Coq's parser, which is built on "camlp4," the Caml Pre-Processor and Pretty Printer:

```
Definition isEven : forall (n : nat), [[even n]].
  refine (fix isEven (n : nat) : [[even n]] :=
    match n return [[even n]] with
      | O => Yes
      | S (S n) =>
        pf <- isEven n;
        Yes
      | _ => No
    end); auto.
Qed.
```

I introduce the syntax `[[P]]` for `poption P`, along with `Yes` and `No` for the `PSome` and `PNone` forms from the earlier example version. There's also the `pf <- isEven n; Yes` code snippet, which treats `poption` as a *failure monad* in the style familiar from Haskell programming. The meaning of that code is that `isEven n` should be evaluated. If it returns `PNone`, then the overall expression also evaluates to `PNone`. If it returns `PSome`, then bind the associated proof to the variable `pf` in the body `Yes`. Here, it looks like the proof is not used in the body, but remember that `Yes` is syntactic sugar for a `PSome` with a hole for a proof. `refine` will ask us to construct this proof in an environment where `pf` is bound.

We construct terms like this to use in programs that we eventually hope to execute. With Coq, efficient compilation of programs is achieved through *extraction* to computationally equivalent OCaml code. With the right settings, our example extracts to:

```
let rec isEven (n : nat) : bool =
  match n with
    | O -> true
    | S (S n) -> isEven n
    | _ -> false
```

Notice that the proof components have disappeared. More generally, extraction *erases* all terms with sorts other than `Set`, only leaving us with the OCaml equivalents of Coq terms that we designated as "programs." Thanks to some subtle conditions on legal Coq terms, Coq can guarantee that the extraction of any Coq term in `Set` has the same computational semantics as the original.

Besides `poption`, there is another type of similar flavor that will show up often in what follows. This is the `soption` type, which is an optional package of a value and a proof about that value. It is defined as

```
Inductive soption (T : Set) (P : T -> Prop) : Set :=
  | SSome : forall (x : T), P x -> soption T P
  | SNone : soption T P.
```

`soption` is the type of a potentially incomplete procedure that searches for a value satisfying a particular predicate. For instance, a type inference procedure `infer` for some object language encoded in Coq might have the type `forall (e : exp), soption type (fun t : type => hasType e t)`. We could then use this function in failure monad style with expressions like `t : pfT <- infer e; ...`, which attempts to find a type for `e`. If no type is found, the expression evaluates to `SNone`; otherwise, in the body `t` is bound to the value found, and `pfT` is bound to a proof that `t` has the property we need. The importance difference of `soption` with respect to `poption` is that the value found by a function like `infer` is allowed to have computational content and is preserved by extraction, while the only computational content of a `poption` is a yes/no answer.

In the bulk of this paper, I'll use a more eye-friendly, non-ASCII notation for these types. I'll denote `poption P` as $[\![P]\!]$ and `soption (fun x : T => P)` as $\{\!\{x : T \mid P\}\!\}$. I'll also use the usual Π instead of Coq's `forall` to denote dependent function types.

2.2 Coq's Module System

Coq also features a natural extension of ML-style module systems to Coq's dependently-typed world, and I use the module system extensively to structure re-usable verification components. Here's a simple example of a pattern of module usage that appears often:

```
Module Type PARAM.
  Parameter abstractState : Set.
  ...
  Axiom soundness :
    (* Theorem statement in terms of abstractState *)
End PARAM.

Module Type VERIFIER.
  Parameter verify :
    forall (p : program), [[(* p satisfies spec *)]]
End VERIFIER.

Module Verifier (P : PARAM) : VERIFIER.
  (* Code and proofs for a verifier that uses the
   * abstraction from P *)
End Verifier.
```

Here `Verifier` is a functor for building a program verifier based on a user-supplied abstraction. An abstraction includes, among other things, a set for the domain of abstract states and a proof of some soundness theorem that is key to the soundness of the way that `Verifier` will use the abstraction.

2.3 Problem Formulation

The goal of this work is to support the verification of safety properties of executable x86 machine code programs. I've opted to simplify the problem by focusing on a single safety policy, where the safety policy simply forbids execution of a special "Error" instruction. As in model-checking, many interesting safety policies can then be encoded with assertion checks that execute "Error" on failure.

The first task is to define formally the semantics of machine code programs. The style is standard for FPCC [App01], but I summarize the formalization here to make it clear exactly what a successful verification guarantees.

Machine words	word	w	$::=$	$0 \mid 1 \mid \ldots \mid 2^{32}-1$
Registers	reg	r	$::=$	$\mathsf{EAX} \mid \mathsf{ESP} \mid \ldots$
Flags	flag	f	$::=$	$\mathsf{Z} \mid \ldots$
Register files	regFile	R	$=$	$\mathsf{reg} \to \mathsf{word}$
Flag files	flagFile	F	$=$	$\mathsf{flag} \to \mathsf{bool}$
Memories	memory	M	$=$	$\mathsf{word} \to \mathsf{byte}$
Machine states	state	S	$=$	$\mathsf{word} \times \mathsf{regFile} \times \mathsf{flagFile} \times \mathsf{memory}$
Instructions	instr	I	$::=$	$\mathsf{ERROR} \mid \mathsf{MOV}\ r,\ [r] \mid \mathsf{JCC}\ f,\ w \mid \ldots$
Step relation		$\mapsto$	$:$	$\mathsf{state} \rightharpoonup \mathsf{state}$

The main thing to notice is that the semantics follows precisely a conservative subset of the programmer-level idea of the "real" semantics of x86 machine code. I've chosen a subset of x86 instructions that is sufficient to allow many interesting programs and only included in the semantics those aspects of processor state needed to support those instructions.

The various elements of the formalization follow from the official specification of the x86 processor family, with the exception of the ERROR instruction added to model the safety policy. I'll briefly review the different syntactic classes and definitions before continuing.

A machine state consists of a word for the program counter, giving the address in memory of the next instruction to execute; a register file, giving the current word value of every general purpose register; a boolean valuation to each of the flags, which indicate conditions like equality and overflow relevant to the last arithmetic operation; and a memory, an array of exactly 2^{32} bytes indexed by words. The instructions are a subset of the real x86 instruction set, with the addition of the ERROR instruction.

A small-step transition relation $\mapsto$ describes the semantics of program execution. One transition involves reading the instruction from memory at the address given by the program counter and then executing it according to the x86 instruction set specification. Actually, $\mapsto$ is a partial function; it fails to make progress if the instruction loaded is ERROR. In this way, violations of the safety policy are encoded with the usual idiom of the transition relation "getting stuck." A "production quality" implementation would no doubt keep the real semantics separate from a library of safety policies, but the design decision I made simplifies my formalization, and the main interesting issues therein are the same between the two approaches.

We can define what it means for a machine state to be *safe* with this co-inductive inference rule:

$$\frac{S \mapsto S' \quad \mathsf{safe}(S')}{\mathsf{safe}(S)}$$

This is defined using Coq's facility for co-inductive judgments, which may have infinite derivations that are well-formed in a particular sense. Infinite derivations are important here for non-terminating programs.

The last ingredient is a means to connect a program to the first machine state encountered when it is run. Assume the existence of a type program and a function load : program → state. Concretely, program is a particular file format that GCC will output, and load expresses the algorithm for extracting the initial contents of memory from such a file, zeroing out registers and flags, and setting the program counter to the fixed address of the program start. To simplify reasoning while still remaining faithful to real semantics, I deal with programs that run "on a bare machine" with no operating system, virtual memory, etc.; and in fact the programs really do run as such in an emulator.

We've now established enough machinery to define formally the correctness condition of a certified verifier. A certified verifier is any value of the type:

$$\Pi(p : \mathsf{program}).[\![\mathsf{safe}(\mathsf{load}(p))]\!]$$

The type of the extracted function is program → bool. By the soundness of extraction, we know that the value of the function on an input p is a boolean whose truth implies the safety of the program. Thus, if p is unsafe, the function must return false, and we can take a return of true as conclusive evidence that p is safe. A trivial certified verifier implementation is one that always returns false, but this is an issue of completeness, not soundness, to be dealt with through testing. There are some possibilities for proving completeness results in Coq, but they invariably have the flavor of proving that a complicated implementation accepts every input accepted by a simpler reference implementation, begging the question of how we know that the reference implementation is complete enough. Even the simplest reference implementation of a non-trivial static analysis technique will be quite large, increasing the trusted code base by orders of magnitude over what is required for FPCC, so it hasn't seemed worthwhile to attempt anything of this sort.

3. Components for Writing Certified Verifiers

The final goal of the case study I'm presenting here was to produce a certified x86 machine code memory safety verifier that supports general product, sum, and recursive types, which I'll call MemoryTypes. It would have been possible to write this verifier monolithically, but I thought it would be more interesting and useful to do it in stages, writing re-usable components with rich interfaces to handle different parts of verification and allow later components to reason at increasingly high levels of abstraction.

The component structure that I present here is born of necessity; a layered decomposition of verifier structure or something like it is critical to making the overall task feasible. As traditional software built from many simple pieces can become unmanageably complex, the problem is only exacerbated when formal correctness proofs are required, since now even the "simple" pieces can involve non-trivial proofs. The component structure I've settled on has been designed not just to support effective programming, but also effective proof construction, by minimizing the need for repeated work. The issues and complexities specific to my domain of machine code verifiers are probably not clear to readers who don't have experience in that field, but I hope that the following walk-through of the steps in my solution can shed some light on them. The important question at each stage of this abstraction hierarchy is "How hard would it be to develop and maintain a new verifier (with a soundness proof) handling all of the hidden lower-level details?". It's also true that I'll be drilling down to a significant level of detail in this section. I invite the overwhelmed reader to look ahead to the light at the end of the tunnel in Section 4, where I show how all of this machinery pays off in making it very easy to build a new certified verifier.

Figure 1 presents the particular component structure that I settled on. An arrow from one component to another indicates that the target component of the arrow builds on the source component. Ovals represent logical theories that are used in the correctness conditions of other modules. Boxes stand for components that contribute code to the extracted version of a verifier; i.e., they contain implementations of verifiers at particular levels of abstraction, along with the associated correctness proofs. Solid boxes are best viewed as library components, while transparent boxes represent certified verifiers, the final products. I include a number of verifier boxes with dashed borders. These stand for hypothetical verifiers that I haven't implemented but that I believe would best be constructed starting from the components that connect to them in the diagram.

I will describe each library module in detail in the following subsections, but I'll start by providing an overview of the big picture.

- The only module that belongs to the trusted code base is the **x86 Semantics**, the basic idea of which I presented in Section 2.3.
- **ModelCheck** provides the fundamental method of proving theorems about infinite state systems through exhaustive exploration of an appropriate abstract state space; or, since x86 states are finite in reality and in my formalization, proving theorems about intractably large state spaces through exhaustive exploration of smaller abstract state spaces.
- The CISC x86 instruction set involves lots of complications that one would rather avoid as much as possible, so I do most verification on a tiny RISC instruction set to which I reduce x86 programs. **SAL semantics** defines the behavior of this Simplified Assembly Language.
- **Reduction** enables multiple steps of abstraction: model checking an abstraction of an abstraction of a system suffices to verify that system. In the chain of component uses for MemoryTypes, Reduction is used to do model checking on the SAL version of an x86 program. One way of viewing traditional PCC approaches is that they apply proof-checking on the result of a reduction to whatever internal format they use to represent programs.
- **FixedCode** deals with a basic simplification used by most program verifiers, which is that a fixed region of memory is designated as code memory, and that memory cannot be modified in any run of the program. General FPCC frameworks in theory support verification of self-modifying programs, but we usually want to work at a higher level of abstraction. FixedCode's level of abstraction would be appropriate for an adaptation to machine code level of traditional verification in the style of Extended Static Checking [DLNS98].
- **TypeSystem** provides support for model checking where the primary component of an abstract state is an assignment of a type to every general purpose machine register. This would be a good starting point for traditional Typed Assembly Language [MWCG99, MCGW03], which handles stack and calling conventions with its own kind of stack types...
- ...but for most verifiers, **StackTypes** would be the module to use next. It takes as input a type system ignorant of stack and calling conventions and produces a type system that understands them.

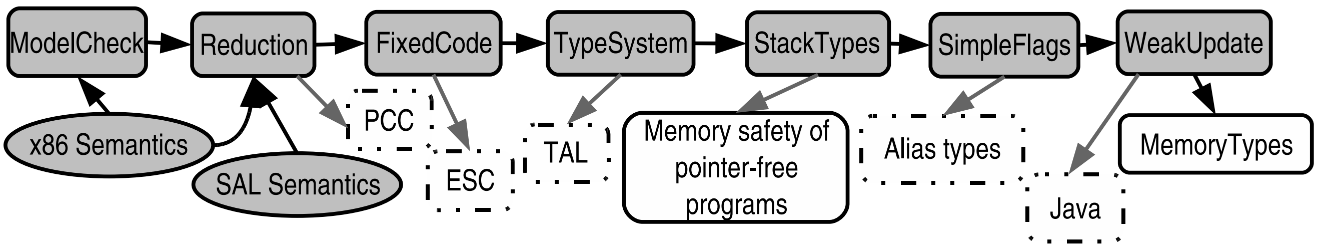

Figure 1. A component structure for certified verifiers

An application of StackTypes to a trivial type system gives us a verifier capable of checking memory safety of simple C programs that don't use pointers.

- **FlagTypes** handles tracking of condition flag values relevant to conditional jumps. This is critical for verifying programs that use pointers that might be null, general sum types, or any of a large variety of type system features. FlagTypes would be a reasonable starting point for a verifier based on alias types or some other way of supporting manual memory management...
- ...but with automatic memory management, **WeakUpdate** provides a much more convenient starting point. WeakUpdate is used with type systems that have a notion of a partial map from memory addresses to types, where this map can only be extended, never modified, during a program execution. Though addresses can usually change types when storage is reclaimed, this is handled by, e.g., a garbage collector that is verified using different methods. WeakUpdate would also provide a good foundation for machine code-level verification of programs compiled from Java source code.

In general, each of these arrows between rounded boxes in Figure 1 indicates a functor translating a verifier at the target's level of abstraction to a verifier at the source's level. These functors are used as in the example in Section 2.2. For instance, for the arrow between TypeSystem and StackTypes, we have the form of the earlier example with `PARAM` changed to `STACK_TYPE_SYSTEM`, the output signature of the functor changed to `TYPE_SYSTEM`, and the functor's innards assembling a richer type system by extending that presented by its input module.

I will now describe the most important aspects of the interfaces and implementations of each of these reusable library components. For reasons of space, I avoid describing how the actual proofs are constructed, focusing instead on component interfaces and a bird's-eye view of an overall structure that I've found to work in practice. Nonetheless, there are many important engineering issues in proof construction. The main thing to keep in mind through the following sections is that every piece *is* supported by Coq proofs of the relevant properties, and that I was able to construct these proofs using the techniques sketched in Section 1.2.

3.1 ModelCheck

An input to ModelCheck specifies a particular machine semantics Mac, implemented as a module ascribing to a particular common `MACHINE` signature. An abstraction for this machine is defined by providing the elements in Figure 2.

Before describing their meanings, I note that this formalization of model-checking is specific to "first-order" uses of code pointers. Each point in the abstract state space can have any number of known code pointers that it is allowed to jump to, but the descriptions of these code pointers can't themselves refer to other code pointers. The formulation I give is expressive enough to handle, for instance, standard function call and exception handling conventions. Naturally, this design decision precludes the easy handling of functional languages, but one would simply write another component to serve as a starting point there; and there are plenty of interesting issues in this restricted setting, related to data structures and other program features.

$$
\begin{aligned}
absState &: \mathsf{Set}\\[1em]
context &: \mathsf{Set}\\
\vdash_S &: context \rightarrow Mac.state \rightarrow absState \rightarrow \mathsf{Prop}\\[1em]
init &: \{states : \mathsf{list}\ (absState \times \mathsf{list}\ absState)\\
&\qquad \mid \exists \Gamma : context, \exists \alpha : absState,\\
&\qquad\quad (\alpha, \mathsf{nil}) \in states \land \Gamma \vdash_S Mac.start : \alpha\}\\[1em]
step &: \Pi(hyps : \mathsf{list}\ absState)(\alpha : absState).\\
&\quad \{succs : \mathsf{list}\ absState\\
&\qquad \mid \forall (s : Mac.state)(\Gamma : context),\\
&\qquad \Gamma \vdash_S s : \alpha \Rightarrow \exists s' : Mac.state, s \mapsto_{Mac} s'\\
&\qquad \land \exists \alpha' : absState, ((\alpha' \in (hyps \cup succs)\\
&\qquad\quad \land \Gamma \vdash_S s' : \alpha')\\
&\qquad \lor (\exists hyps' : \mathsf{list}\ absState,\\
&\qquad\quad (\alpha', hyps') \in \pi_1(init)\\
&\qquad\quad \land \exists \Gamma' : context, \Gamma' \vdash_S s' : \alpha'\\
&\qquad\qquad \land \forall h' \in hyps', \exists h \in (hyps \cup succs),\\
&\qquad\qquad\quad \forall s'' : Mac.state,\\
&\qquad\qquad\qquad \Gamma' \vdash_S s'' : h' \Rightarrow \Gamma \vdash_S s'' : h))\}
\end{aligned}
$$

Figure 2. Elements of an abstraction for ModelCheck

Now I will give the high level picture of what an abstraction is and what properties it must satisfy. The fundamental piece of an abstraction is its set $absState$ of abstract states. These will be the constituents of the state spaces explored at verification time.

As per usual in abstraction-based model checking, we need to provide a relation characterizing compatibility of concrete and abstract states. $\vdash_S$ is a ternary relation filling this function. It relies on an extra, perhaps unexpected, component, a set $context$. The basic idea behind the separation of abstract states and contexts is that abstract states will be manipulated in the extracted OCaml version of a verifier, while contexts will be used only in the proof of correctness and erased during extraction. A canonical example of a context is a valuation to free type variables used in an abstract state. Contexts provide a sort of polymorphism that lets us check infinitely many different abstract states by checking a finite set of representatives. For instance, we check a finite set of abstract states containing type variables in place of checking the infinite set of all of their substitution instances. I use the infix notation $\Gamma \vdash_S s : \alpha$ to denote that, in context Γ, concrete state s and abstract state α are compatible; i.e., s belongs to α's concretization.

Now we need a way of computing an abstract state space that conservatively approximates the concrete state space. The values *init* and *step* are used to do this, with *init* providing the roots of the state space and *step* describing how to expand it by following the edges out of a single node.

Each element of the state space consists of one $absState$ describing the current state and zero or more *hypotheses* (represented with a $\mathsf{list}\ absState$) describing other abstract states known to be safe. The canonical example of a hypothesis is a function call's return pointer. If verification inside a function ever reaches an abstract state compatible with the return pointer's hypothesis, then there is no need to explore that branch of the state space further.

init provides a set (actually a list) of state descriptions of this type, along with a guarantee that some abstract state compatible with the concrete initial state is included. I overload $\in$ and other set notations to work for lists, where a list is interpreted as the set of its elements. The condition for *init* requires that some abstract state with no hypotheses (i.e., that makes no special assumptions) is found among the initial states.

step is the complicated part of the abstraction. Computationally, it's simple: given a point in the abstract state space, return a list of the states reachable from there in one step. The specification packaged with this function is where things get interesting.

First, note that while *init*'s type is a standard set comprehension, *step* has an `soption` type that allows it the option of failing for any input. Naturally, this is important, since otherwise the implementation of ModelCheck would somehow need to produce a model checker that is able to prove any program safe!

When *step* succeeds for hypotheses *hyps* and abstract state α, it returns a set of successor states that satisfies a particular correctness condition. First, it must be the case that any concrete state related to α makes at least one step of safe *progress*. Next, there is a *preservation* condition, broken into two cases.

The first kind of preservation is the simple one, corresponding to most instructions in a program. We need to be sure that we expand our state space to include a point for every concrete successor s' of every concrete state s related to α. The first disjunct of the preservation condition ensures this by requiring that s' is related (in the current context) to one of the abstract successor states that we are queueing to visit, or to one of the hypotheses. The first of these cases corresponds to a normal instruction in straight-line code. The second case would be used, for example, at a function return, where the abstract state has finally come to match the return pointer hypothesis.

The second kind of preservation is associated with direct jumps and function calls. It requires that we look into the roots of the state space and find one $(\alpha', hyps')$ that is compatible with s'. While the previous preservation case required that the compatibility hold in the current context Γ, in this case we can choose an arbitrary new context Γ'. Not only do we need to guarantee $\Gamma' \vdash_S s' : \alpha'$, but we also need to be sure that every hypothesis in $hyps'$ describes a set of states that are all safe. That's the purpose of the final condition, which says that every hypothesis hyp' in $hyps'$ has a counterpart hyp among the current hypotheses and successor states, such that any concrete state described by hyp' in Γ' is also described by hyp in Γ. This implication at first seems to be reversed from the natural order, but it makes sense in the light of standard function subtyping rules when we think of hypotheses as distinguished function arguments.

For a concrete example of this second preservation case, think of a function call in a C program. *step* finds a state space root for the function that we're calling. The function has associated with it a single hypothesis, for its return pointer. *step* handles this call by returning in *succs* a single new state to be visited, corresponding to the expected return state of the call. The proof for this case uses the second kind of preservation, where it shows first, that the immediate next state is compatible with the entry state for the callee; and second, that any state described by the callee's return pointer hypothesis also describes the local return state that was just queued. A different context is used in the callee than in the caller, reflecting a view shift into a new stack frame.

This example shows that *init* can't just be a complete description of the initial concrete state, but must rather contain enough states that every point in the program's execution reaches one in finitely many steps. In this sense, *init* is like a pre-computed fixed point of an abstract interpretation. I could have required that *init* contain enough elements to describe *every* reachable concrete state, but that would just contribute to verification-time inefficiency, and we really only need to fix enough abstract states to cut every cycle in the abstract state space. In the implementation, *init* will be computed mostly by ML code. This code gets the information by reading annotations out of the binary being analyzed. It could just as easily use abstract interpretation to infer the information from a less complete set of annotations. Notice that an "error" in constructing the fixed point can only effect completeness, not soundness, so it's OK to implement this part outside of Coq.

With these components provided to it, ModelCheck produces a standard model checker that uses the requested abstraction. This model checker performs a depth-first search through the state space, where the search terminates in every branch of the tree where preservation is shown through the second case above, corresponding to a jump or call to one of the root states. The computational content of *init* and *step* determines the shape of the state space to explore.

3.2 Reduction

The literal x86 machine language is not ideal for verification purposes. Single instructions represent what are conceptually several basic operations, and the same basic operations show up in the workings of many instructions. As a result, a verifier that must handle every instruction will find itself doing duplicate work. In my implementation, I handle this problem once and for all by way of a component to model check a program in one instruction set by reducing it to a simpler instruction set.

The particular simplified language that I use is a RISC-style instruction set called SAL (Simplified Assembly Language), after a family of such languages used in traditional proof-carrying code work [Nec97]. The main simplifications are the use of arbitrary arithmetic expressions, instead of separate instructions for loading a constant into a register, performing an arithmetic operation, etc.; and a new invariant that each instruction has a single effect on machine state. This latter property is accomplished by breaking instructions into multiple pieces responsible for the different effects.

Here's a brief summary of the language grammar:

$$
\begin{array}{rrcl}
\textit{SAL registers} & r_s & ::= & r \mid \mathsf{TMP}_i \\
\textit{Binary operators} & \circ & & \\
\textit{SAL expressions} & e & ::= & w \mid r_s \mid e \circ e \\
\textit{SAL instructions} & I_s & ::= & \mathsf{ERROR} \mid \mathsf{SET}\ r_s,\ e \\
& & & \mid\ \mathsf{LOAD}\ r_s,\ [e] \mid \mathsf{STORE}\ [e],\ e \mid \ldots
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{prog} & : & \mathsf{word} \to \mathsf{byte} \\
\mathit{code} & : & \mathsf{memoryRegion} \\
\mathit{absPc} & : & \Pi(\alpha : \mathit{absState}).\{\!\!\{ pc : \mathsf{word} \\
& & \quad \mid \ \forall \Gamma, s. \Gamma \vdash_S s : \alpha \Rightarrow s.pc = pc \}\!\!\} \\
\\
\mathit{instrOk} & = & \lambda s. \lambda ins. \\
& & \begin{cases} |dst|_s \notin \mathit{code}, & ins = \mathsf{STORE}\ [dst],\ src \\ \qquad \mathsf{True}, & \text{otherwise} \end{cases} \\
\\
\mathit{step} & : & \Pi(\mathit{hyps} : \mathsf{list}\ \mathit{absState})(\alpha : \mathit{absState}). \\
& & \quad \{\!\!\{ \mathit{succs} : \mathsf{list}\ \mathit{absState} \\
& & \quad \mid \forall (s : \mathit{Mac.state})(\Gamma : \mathit{context}), \\
& & \quad \Gamma \vdash_S s : \alpha \Rightarrow \exists s' : \mathit{Mac.state}, s \mapsto_{\mathit{Mac}} s' \\
& & \quad \wedge \underline{\mathit{instrOk}(st, ins)} \wedge \ldots \}\!\!\}
\end{array}
$$

Figure 3. New elements of a FixedCode abstraction

I add a finite set of extra temporary registers, needed when a single instruction is broken up into its constituents. For example, the x86 instruction PUSH [EAX], which pushes onto the stack the value pointed to by register EAX, is compiled into LOAD TMP_1, [EAX]; STORE [ESP$-$4], TMP_1; SET ESP, ESP$-$4.

The details of SAL and Reduction are not especially enlightening, so I omit them here. The main technical component is the expected compatibility relation between states of the two kinds of programs, along with a compilation function and a proof that it respects compatibility.

3.3 FixedCode

The most basic knowledge a model checker needs is how to determine which instructions are executed when. The full semantics of SAL programs allows writing to arbitrary parts of memory, including those thought of as housing the program. We usually don't want to allow for this possibility and would rather simplify the verification framework. The FixedCode module is used to build verifiers based on this assumption. It is a functor that takes as input an abstraction that assumes a fixed code segment and returns an abstraction that is sound for the true semantics.

Figure 3 shows the additions and modifications to FixedCode's signature for an abstraction over what ModelCheck requires. The first addition is a memory value *prog* that contains in some contiguous region of its address space the encoding of the fixed program. The memory region *code* tells us which address space range this is.

Next, we have a function $absPc$ for determining the program counter for some subset of the abstract states. The model checker that FixedCode outputs will take responsibility for determining which instruction is next to execute in any state for which $absPc$ returns SSome of a program counter. Other states may only be used as hypotheses and never appear directly in the abstract state space. For instance, we don't know the precise program counter of a hypothesis describing a return pointer, but this doesn't matter, since we are sure to visit all of the concrete program locations it could stand for.

The final change to the signature of an abstraction is that, of course, we must now require that the code is never overwritten. If it were, then we would no longer know at verification time which instruction was being executed when, since the verifier will simply look instructions up in *prog* for this purpose. The progress condition of *step* is augmented to require that the destination of any STORE instruction is outside of the code region, using an auxiliary function $instrOk$. I use the notation $|e|_s$ to stand for the result of evaluating expression e in machine state s.

$$
\begin{array}{rcl}
\mathit{ty} & : & \mathsf{Set} \\
\vdash_T & : & \mathit{context} \to \mathsf{word} \to \mathit{ty} \to \mathsf{Prop} \\
\leq_T & : & \Pi(\tau_1 : \mathit{ty})(\tau_2 : \mathit{ty}).[\![\forall \Gamma, w, \\
& & \qquad \Gamma \vdash_T w : \tau_1 \Rightarrow \Gamma \vdash_T w : \tau_2]\!] \\
\\
\mathit{typeof} & : & \Pi(\alpha : \mathit{absState})(\vec{r} : \mathsf{reg} \to \mathit{ty})(e : \mathsf{exp}). \\
& & \quad \{\!\!\{ \tau : \mathit{ty} \mid \forall \Gamma, s. \Gamma \vdash_S s : \alpha \\
& & \qquad \Rightarrow (\forall r, \Gamma \vdash_T s.\mathsf{regs}(r) : \vec{r}(r)) \\
& & \qquad \Rightarrow \Gamma \vdash_T |e|_s : \tau \}\!\!\} \\
\\
\mathit{viewShift} & : & \Pi(\alpha : \mathit{absState})(\vec{r} : \mathsf{reg} \to \mathit{ty})(i : \mathsf{instr}). \\
& & \quad \{\!\!\{ (\alpha', \vec{r'}) : \mathit{absState} \times (\mathsf{reg} \to \mathit{ty}) \\
& & \quad \mid \forall \Gamma, s. \Gamma \vdash_S s : \alpha \\
& & \qquad \Rightarrow (\forall r, \Gamma \vdash_T s.\mathsf{regs}(r) : \vec{r}(r)) \\
& & \qquad \Rightarrow \exists \Gamma', \Gamma' \vdash_S s : \alpha' \\
& & \qquad \wedge (\forall r, \Gamma' \vdash_T s.\mathsf{regs}(r) : \vec{r'}(r)) \}\!\!\}
\end{array}
$$

Figure 4. New elements of a TypeSystem abstraction

3.4 TypeSystem

The next stage in the pipeline is the first where a significant decision is made on structuring verifiers. The TypeSystem component provides support for a standard approach to structuring abstract state descriptions: considering the value of each register separately by describing it with a type. Figure 4 shows the key new components of a type-based abstraction. I'm omitting many of the details, but the pieces I chose to include illustrate the key points.

The basic idea is that we have an abstraction as before that can assume that, in addition to the custom abstract state that it maintains itself, a type assignment to each register is available at each step. The abstraction provides the set ty of types, along with a typing relation $\vdash_T$ to define their meanings, plus a subtyping procedure $\leq_T$. It's worth noting that there is no need to go into further detail on exactly how to allow typing relations to be defined. Coq's very expressive logic is designed for just such tasks, and natural-deduction style definitions of type systems via inference rules are accommodated naturally by the same mechanism for inductive type definitions that I demonstrated in Section 1.2, where the defined relation is placed in sort Prop.

Naturally TypeSystem will need a way to determine the types of expressions if it is to track the register information that an abstraction assumes is available. The provided *typeof* function explains how to do this. Given an abstract state α, a register type assignment $\vec{r}$, and an expression e, *typeof* must return a type that describes the value of the expression in any compatible context Γ and concrete state s. It only needs to work correctly under the assumption that, in Γ, α accurately describes s and $\vec{r}$ accurately describes all of s's register values.

viewShift provides an important piece of logic that might not be obvious by analogy from type systems for higher-level languages. At certain points in its execution (and so in model checking), a program "crosses an abstraction boundary" which takes a different view of the types of values. A canonical example is a function call.

$$
\begin{array}{rcl}
\mathit{stack} & : & \mathsf{memoryRegion} \\
\mathit{stackCodeDisjoint} & : & \mathsf{disjoint}(\mathit{stack}, \mathit{code}) \\
\\
\mathit{checkStore} & : & \Pi(\alpha : \mathit{absState})(\vec{r} : \mathsf{reg} \to \mathit{ty})(e : \mathsf{exp}) \\
& & \quad [\![\forall \Gamma, s.\Gamma \vdash_S s : \alpha \\
& & \qquad \Rightarrow (\forall r, \Gamma \vdash_T s.\mathsf{regs}(r) : \vec{r}(r)) \\
& & \qquad \Rightarrow |e|_s \notin \mathit{stack}]\!]
\end{array}
$$

Figure 5. New elements of a StackTypes abstraction

$$
\begin{array}{rcl}
\mathit{considerTest} & : & \Pi(\alpha : \mathit{absState})(\vec{r} : \mathsf{reg} \to \mathit{ty})(co : \mathsf{cond}) \\
& & \quad (\circ : \mathsf{binop})(e1\ e2 : \mathsf{exp})(b : \mathsf{bool}). \\
& & \quad \{\!\{(\alpha', \vec{r'}) : \mathit{absState} \times (\mathsf{reg} \to \mathit{ty}) \\
& & \qquad \mid \forall \Gamma, s.\Gamma \vdash_S s : \alpha \\
& & \qquad\quad \Rightarrow (\forall r, \Gamma \vdash_T s.\mathsf{regs}(r) : \vec{r}(r)) \\
& & \qquad\quad \Rightarrow |e_1 \circ e_2|_s^{co} = b \\
& & \qquad\quad \Rightarrow \exists \Gamma', \Gamma' \vdash_S s : \alpha' \\
& & \qquad\qquad \wedge\ (\forall r, \Gamma' \vdash_T s.\mathsf{regs}(r) : \vec{r'}(r))\}\!\}
\end{array}
$$

Figure 6. New elements of a SimpleFlags abstraction

The stack pointer register may switch from type "pointer to the fifth stack slot in my frame" to "pointer to the first stack slot in my frame." In the presence of type polymorphism via type variables, a register's type may change from "pointer to integer" to "pointer to β," where β is a type variable instantiated to "integer" for the call. There are many ways of structuring modularity in programs, so it's important that the requirements on $\mathit{viewShift}$ be very flexible. Its type in Figure 4 expresses that it may provide any new abstract state α' and register type assignment $\vec{r'}$ for which there exists some context Γ' in which α' and $\vec{r'}$ are correct whenever α and $\vec{r}$ were correct in the original Γ.

It's worth recalling the context in which TypeSystem is being used, which is to support construction of Coq terms to be extracted to OCaml code. $\vdash_T$ exists only in the Prop world, and so it will not survive the extraction process; it is only important in the proof of correctness of the resulting verifier. The types in ty are manipulated explicitly at verification time, so those survive extraction intact. $\leq_T$, typeof, and $\mathit{viewShift}$ have both computational and logical content. For instance, the extracted version of $\leq_T$ is a potentially incomplete decision procedure with boolean answers. The extracted OCaml version of the verifier ends up looking like a standard type checker. You can think of the Coq implementation as combining a type checker and a proof of soundness for the type system it uses. There is considerable practical benefit from developing both pieces in parallel through the use of dependent types.

3.5 StackTypes

There are a wide variety of interesting type systems worth exploring for verifying different kinds of programs. At least when using the standard x86 calling conventions, every one of these type systems needs to worry about keeping track of the types of stack slots, which registers point to which places in the stack, proper handling of callee-save registers, and other such annoyances. StackTypes handles all of these details by providing a functor from a stack-ignorant TypeSystem abstraction to a TypeSystem abstraction aware of stack and calling conventions. The input abstraction can focus on the interesting aspects of the new types that it introduces rather than getting bogged down in the details of stack and calling conventions.

To make this feasible, the input abstraction only needs to provide a few new elements, as shown in Figure 5. First, a region of memory is designated to contain the runtime stack. It is accompanied with a proof that it has no overlap with the region where the program is stored. The remaining ingredient is a way of making sure that the custom verification code of the abstraction will never allow the stack to be overwritten. The $\mathit{checkStore}$ function is used for this purpose, being called on an expression that is the target of a STORE instruction to make sure that it won't evaluate to an address in the stack region. In my current implementation, this involves "exposing" the underlying stack implementation, though the client of StackTypes can avoid worrying too much about these details through the use of a library of helper functions.

With these ingredients, StackTypes builds a verifier that adds a few new types to the input abstraction's set ty. First, there are types Stack_i, indicating the ith stack slot from the beginning of the stack frame. Types for the stack slots are tracked in another part of abstract states. With this additional information, it's possible to determine the type of the value lying at a certain offset from the address stored in a register of Stack_i type. There is also a type Saved_r for each callee-save register r, denoting the initial value of r on entry to the current function call. These values will probably be saved in stack slots, and we will require that each callee-save register again has its associated Saved type when we return from the function. We know that we've reached this point when we do an indirect jump to a value of type Retptr, where the saved return pointer on the stack is given this type at the entry point to the function.

3.6 SimpleFlags

In x86 machine language, there are no instructions that implement conditional test and jump atomically. Instead, all arithmetic operations set a group of flag registers, such as Z, to indicate a result of zero; or C, to indicate that a carry occurred. Each condition, formed from a flag and a boolean value, has a corresponding conditional jump instruction that jumps to a fixed code location iff that condition is true relative to the current flag settings. Thus, to properly determine what consequences follow from the fact that a conditional jump goes a certain way, it's necessary to track the relationship of the flags to the other aspects of machine states. Understanding these jumps is critical for such purposes as tracking pointer nullness and array bounds checks.

SimpleFlags is a functor that does the hard part of this tracking for an arbitrary abstraction, feeding its results back to the abstraction through a function whose signature is given in Figure 6. The type of $\mathit{considerTest}$ looks similar to the type of $\mathit{viewShift}$ from TypeSystem. Its purpose is to update an abstract state to reflect the information that a particular condition is true. The arguments α and $\vec{r}$ are as for $\mathit{viewShift}$. co names one of the finite set of conditions that can be tested with conditional jumps. $\circ$, $e1$, and $e2$ describe the arithmetic operation that was responsible for the current status of co. Finally, b gives the boolean value of co for this operation, determined from the result of a conditional jump. The notation $|e_1 \circ e_2|_s^{co}$ denotes the value of co resulting from evaluating the arithmetic operation $e_1 \circ e_2$ in state s.

Behind the scenes, SimpleFlags works by maintaining in each abstract state a partial map from flags to arithmetic expressions. The presence of a mapping from flag f to $e_1 \circ e_2$ means that it is known for sure that the value of f comes from $e_1 \circ e_2$, as it would be evaluated in the current state. SimpleFlags must be careful to invalidate a mapping conservatively each time a register that appears in it is modified. At each conditional jump, SimpleFlags

$$
\begin{array}{rcl}
ty & : & \mathsf{Set} \\
context & = & \mathsf{word} \to \mathsf{option}\ ty \\
\vdash_T & : & context \to \mathsf{word} \to ty \to \mathsf{Prop} \\
\leq_T & : & \Pi(\tau_1 : ty)(\tau_2 : ty).[\![\forall \Gamma, w, \\
& & \quad \Gamma \vdash_T w : \tau_1 \Rightarrow \Gamma \vdash_T w : \tau_2]\!] \\
\\
typeofConst & : & \Pi(w : \mathsf{word}).\{\tau : ty \mid \forall \Gamma . \Gamma \vdash_T w : \tau\} \\
\\
typeofArith & : & \Pi(\circ : \mathsf{binop})(\tau_1\ \tau_2 : ty).\{\tau : ty \\
& & \quad \mid \forall \Gamma, w_1, w_2 . \Gamma \vdash_T w_1 : \tau_1 \\
& & \qquad \Rightarrow\ \Gamma \vdash_T w_2 : \tau_2 \\
& & \qquad \Rightarrow\ \Gamma \vdash_T w_1 \circ w_2 : \tau\} \\
\\
typeofCell & : & \Pi(\tau : ty).\{\!|\tau' : ty \mid\ \forall \Gamma, w. \\
& & \quad \Gamma \vdash_T w : \tau \\
& & \quad \Rightarrow\ \Gamma(w) = \mathsf{Some}\ \tau'|\!\} \\
\\
considerNeq & : & \Pi(\tau : ty)(w : \mathsf{word}).\{\tau' : ty \mid\ \forall \Gamma, w'. \\
& & \quad \Gamma \vdash_T w' : \tau \\
& & \quad \Rightarrow\ w' \neq w \Rightarrow \Gamma \vdash_T w' : \tau'\}
\end{array}
$$

Figure 7. Elements of a WeakUpdate type system

checks to see if the relevant condition's value is known based on the flag map. If so, it calls *considerTest* to form each of the two abstract successor states, corresponding to the truth and falsehood of the condition.

3.7 WeakUpdate

We've now built up enough machinery to get down to the interesting part of a type-based verifier, designing the type system. WeakUpdate provides a functor for building verifiers from type systems of a particular common kind. These are type systems that are based on *weak update* of memory locations, where each accessible memory cell has an associated type that doesn't change during the course of a program run. A cell may only be overwritten with a value of its assigned type. Of course, with realistic language implementations, storage will be reused, perhaps after being reclaimed by a garbage collector. Though handling storage reclamation is beyond the scope of this work, I believe that the proper approach is to verify each program with respect to an abstract semantics where storage is never reclaimed, separately verify a garbage collector in terms of the true semantics, and combine the results via a suitable composition theorem.

Figure 7 shows the signature of a type system for WeakUpdate. In contrast to the signatures given for the previous components, this signature does not extend its predecessors. With one small exception that I will describe below, the elements listed in Figure 7 are all that a type system designer needs to provide to produce a working verifier with a proof of soundness. It's also true that, while I've simplified the presentation of the signatures in previous subsections, this signature is a literal transcription of most of the requirements imposed by the real implementation.

Like for the TypeSystem module, a WeakUpdate type system is based around a set ty of types, with a typing relation $\vdash_T$ and a subtyping procedure $\leq_T$. An important difference is that here we hard-code contexts to be partial maps from memory addresses to types.

A few simple procedures suffice to plug into a generic type-checker for machine code. *typeofConst* gives a type for every constant machine word value; *typeofArith* gives a formula for calculating the type of an arithmetic operation in terms of the types of its operands; and *typeofCell* provides a function from a pointer type to the type of any values that it may point to, returning SNone for non-pointer types.

The final element is a way of taking advantage of the knowledge of conditional jump results, based behind the scenes on SimpleFlags. When the result of a conditional jump implies that some value of type τ is definitely not equal to a word w, *considerNeq* is called with τ and w to update the type of that value to reflect this. A canonical example of use of *considerNeq* is with a nullness check on a pointer, to upgrade its type from "pointer" to "non-null pointer."

The two significant omissions from Figure 7 are functions very similar to *considerNeq*. They consider the cases where not a value itself but *the value it points to in memory* is determined to be equal to or not equal to a constant. A canonical example of usage of these functions is in compilation of case analysis over algebraic datatypes.

The proper use by WeakUpdate of these three functions requires some quite non-trivial bookkeeping. WeakUpdate performs a very simple kind of online points-to analysis to keep up-to-date on which values particular tests provide information on. The most complicated relationship tracked by the current implementation is one such as: TMP_1 holds the result of dereferencing EAX, which holds a value read from stack slot 6. If stack slot 6 is associated with a local variable of a sum type, then a comparison of TMP_1 with some potential sum tag should be used to update the types of both EAX and stack slot 6 to rule out some branches of the sum. As for SimpleFlags, WeakUpdate must be careful to erase a saved relationship when it can't be sure that a modification to a register or to memory preserves it.

Happily, these complications need not concern a client of WeakUpdate. In the next section, I illustrate this with a simple use of it to construct a type system handling some standard types for describing linked, heap-allocated structures.

4. Case Study: A Complete Verifier for Algebraic Datatypes

Figure 8 shows excerpts from the Coq implementation of the MemoryTypes verifier, based on the library components from the last section. Due to space limitations, I only show a few interesting snippets. I've also for clarity made some simplifications from the real Coq code, especially regarding dependent pattern matching.

You can see that the set `ty` of types includes the elements you would expect; namely, constructors for building product, sum, and recursive types in the usual ways. There are also `Constant` types for sum tags of known values and `Var` types to represent the bound variables of recursive types.

The typing relation `hasTy` is defined in terms of its inference rules in the standard way. You can see that the same inductive definition mechanism that is used for standard algebraic datatypes works just as naturally for defining judgments. We have that any word has the corresponding constant type; any word has the empty product type; a word has a non-empty product type if it points to a value with the first type in the product and the following word in memory agrees with the remainder of the product; a word has a sum type if it has type $\mathtt{Constant}(i) \times t$ where t corresponds to the ith element of the sum; and a word has a recursive type if it has the type obtained by unrolling the recursion one level.

```
Definition var := nat.

Inductive ty : Set :=
  | Constant : int32 -> ty
  | Product : product -> ty
  | Sum : ty -> ty -> ty
  | Var : var -> ty
  | Recursive : var -> ty -> ty

with product : Set :=
  | PNil : product
  | PCons : ty -> product -> product.

Inductive hasTy : context -> int32 -> ty -> Prop :=
  | HT_Constant : forall ctx v,
    hasTy ctx v (Constant v)
  | HT_Unit : forall ctx v,
    hasTy ctx v (Product PNil)
  | HT_Product : forall ctx v t ts,
    ctx v = Some t
    -> hasTy ctx (v + 4) (Product ts)
    -> hasTy ctx v (Product (PCons t ts))
  | HT_Suml : forall ctx v t1 t2,
    hasTy ctx v
    (Product (PCons (Constant 0)
      (PCons t1 PNil)))
    -> hasTy ctx v (Sum t1 t2)
  | HT_Sumr : forall ctx v t1 t2,
    hasTy ctx v
    (Product (PCons (Constant 1)
      (PCons t2 PNil)))
    -> hasTy ctx v (Sum t1 t2)
  | HT_Recursive : forall ctx x t v,
    hasTy ctx v (subst x (Recursive x t) t)
    -> hasTy ctx v (Recursive x t).

Definition subTy : forall (t1 t2 : ty),
  poption (forall ctx v,
    hasTy ctx v t1 -> hasTy ctx v t2).
  refine (fix subTy (t1 t2 : ty) {struct t2}
    : poption (forall ctx v,
      hasTy ctx v t1 -> hasTy ctx v t2) :=
    match (t1, t2) with
      | (Constant n1, Constant n2) =>
        pfEq <- int32_eq n1 n2;
        Yes
      | (Product (PCons (Constant n) (PCons t PNil)),
          Sum t1 t2) =>
        if int32_eq n 0 && ty_eq t t1 then Yes
        else if int32_eq n 1 && ty_eq t t2 then Yes
        else No
      | (Recursive x body, t2) =>
        pfSub <- subTy
          (subst x (Recursive x body) body) t2;
        Yes
      | ...
    end); ....
Qed.
```

Figure 8. Excerpts from the implementation of MemoryTypes

Pieces of the definition of the subtyping procedure `subTy` are also shown. It is defined as in the example from Section 1.2: the desired type of the function is asserted as a proof search goal, and we suggest a term with holes that we believe can be filled to yield a term of that type. A constant type is a subtype of another if they have the same constant, which I check with the dependently-typed `int32_eq` function, expressing the check and its use in the failure monad style. The next case demonstrates another way of using dependently-typed decision procedures, based on the `if` notation, which is overloaded to work over any type with two constructors. By similarly overloading the boolean `&&` operator, we can check subtyping of a product type with a sum type by looking for either of the two valid cases, returning `Yes` only if one of them holds. Remember that `Yes` and `No` are syntactic abbreviations for terms with proof holes in them, so that each of these `Yes`'s can have a different proof, constructed in a different context dependent on the `if` branch directions in its scope.

Another case recurses when no previous case has matched and we have a recursive type on the left of the subtyping check. `subTy` is called with the first argument unrolled. In the real implementation, I place a fixed bound of one unrolling on both the left and right sides of the test. This is necessary to ensure termination, and Coq even enforces the totality of every recursive function through primitive recursion in one of its arguments, so the definition exactly as shown isn't well-formed. It's worth noting that this bounded unrolling heuristic works fine at the machine code level, where every instruction performs a single simple operation.

Omitted from the diagram are the `typeof*` functions and the `consider*` functions, which are used to update sum types based on conditional jump results. All of these work as you would expect, with nothing especially enlightening about their implementations. The other big omission is the specification of proof scripts, or sequences of tactics, that are required to describe strategies for proof construction. In many cases, these proof scripts are atomic calls to automating tactics, but in some cases they are longer than would be desired. Improving that aspect of verifier construction is a fruitful area for future work.

Nonetheless, the entire MemoryTypes implementation is only about 600 lines long. I was able to develop it in less than a day of work. Thanks to the common library infrastructure, the reward for this modest effort is a verifier with a rigorous soundness theorem with respect to the real bit-level semantics of the target machine.

Most of the resulting implementation is Coq code which is extracted to ML. For simplicity, I chose to implement in OCaml some pieces that must inevitably belong to the trusted code base, like decoding of instructions. There is also some OCaml code that has no effect on soundness; for instance, to read metadata from a binary and pass it to the extracted verifier as suggested preconditions for the basic blocks. Bugs in this metadata parsing can hurt completeness, but they can never lead to incorrect acceptance of an unsafe program. It would even be possible to replace this code with a complicated abstract interpreter that infers much of what is currently attached explicitly, and the results could be fed to the unchanged extracted verifier with the same soundness guarantees.

In comparison with the library code, the MemoryTypes implementation is small and manageable. My implementation has about 7000 lines of Coq code implementing the library components, along with about 10,000 lines in a generic utility library, 1000 to formalize bitvectors and fixed-precision arithmetic, and 1000 to formalize a subset of x86 machine code. The final verifier can be checked for soundness by running the Coq `Check` command on its entry point function and verifying that the type that is printed matches the $\Pi(p : \mathsf{program}).[\![\mathsf{safe}(\mathsf{load}(p))]\!]$ type I gave in Section 2.3. The "backwards slice" of definitions that this type depends on constitutes the trusted part of the development, and it contains only small parts of the last three library pieces I mentioned above.

You can find the complete source code and documentation at `http://proofos.sourceforge.net/`.

5. Related Work

The verifiers produced in this project are used in the setting of proof-carrying code. Relative to our past work [CCN06] on certified verifiers, my new contribution here is first, to suggest developing verifiers with Coq in the first place, instead of extracting verification conditions about more traditional programs; and second, to report on experience in the effective construction of such verifiers through the use of dependent types and re-usable components. Several projects [App01, HST+02, Cra03] consider in a PCC setting proofs about machine code from first principles, but they focus on proof theoretical issues rather than the pragmatics of constructing proofs and verifying programs under realistic time constraints. Our certified verifiers approach allows the construction of verifiers with strong guarantees that nonetheless perform well enough for real deployment. Wu et al. [WAS03] tackle the same problem based on logic programming, but they provide neither evidence of acceptable scalability of the results nor guidance on the effective engineering of verifiers as logic programs.

Past projects have considered using proof assistants to develop executable abstract interpreters [CJPR04] and Java bytecode verifiers [KN01, Ber01]. My work differs in dealing with machine code, which justifies the kind of layered component approach that I've described, and my work focuses more on accommodating a wide variety of verification approaches without requiring the development of too much code irrelevant to the main new idea of a technique. My work has much in common with the CompCert project [Ler06], which works towards a fully certified C compiler developed in Coq. The main differences are my use of dependent types to structure the "program" part of a development and my emphasis on reusable library components.

I have already mentioned the Epigram [MM04], ATS [CX05], and RSP [WSW05] languages that attempt to inject elements of the approach behind Coq program extraction into a more practical programming setting. I believe I have taken good advantage of many of Coq's mature features for proof organization and automation in ways that would have been significantly harder with these newer languages, which focus more on traditional programming features and their integration with novel proof manipulations. It's also true that much of the specifics of my approach to designing and implementing certified verifiers is just as interesting transposed to the contexts of those languages, and the ideas are of independent interest to the PCC community.

6. Conclusion

There has been much interest lately in enriching the expressiveness of static type systems to capture higher-level properties. Based on the results I've reported here, I hope I've provided some evidence that technology that has been found in computer proof assistants for some time is actually already sufficient to support this kind of programming for non-toy problems. While recent proposals in this space focus on integrating proofs and dependent types with imperativity and other "impure" language features, I was able to construct a significant and reasonably efficient certified program verification tool without using such features. In other words, the advantages of pure functional programming are only amplified when applied in a setting based on rigorous logical proofs, and the strengths of functional programming and type theory are sufficient to support the construction of a program with a formal proof of a very detailed full correctness property. More and more convergence between programming and proving tools seems inevitable in the near future, and I think that working out the details of this convergence is a research direction with the potential for serious and lasting impact.

Acknowledgments

Thanks to Bor-Yuh Evan Chang, Tachio Terauchi, and the anonymous referees for helpful feedback on earlier versions of this paper.

References

[App01] Andrew W. Appel. Foundational proof-carrying code. In *LICS*, pages 247–258, June 2001.

[Ber01] Yves Bertot. Formalizing a JVML verifier for initialization in a theorem prover. In *CAV*, pages 14–24, London, UK, 2001. Springer-Verlag.

[CCN06] Bor-Yuh Evan Chang, Adam Chlipala, and George C. Necula. A framework for certified program analysis and its applications to mobile-code safety. In *VMCAI*, January 2006.

[CJPR04] David Cachera, Thomas Jensen, David Pichardie, and Vlad Rusu. Extracting a data flow analyser in constructive logic. In *Proceedings of The European Symposium on Programming*. Springer-Verlag, 2004.

[Cra03] Karl Crary. Toward a foundational typed assembly language. In *POPL*, volume 38(1) of *ACM SIGPLAN Notices*, pages 198–212, January 15–17 2003.

[CX05] Chiyan Chen and Hongwei Xi. Combining programming with theorem proving. In *ICFP*, pages 66–77, 2005.

[DLNS98] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. SRC Research Report 159, Compaq Systems Research Center, 130 Lytton Ave., Palo Alto, December 1998.

[HST+02] Nadeem A. Hamid, Zhong Shao, Valery Trifonov, Stefan Monnier, and Zhaozhong Ni. A syntactic approach to foundational proof-carrying code. In *LICS*, pages 89–100, Copenhagen, Denmark, July 2002.

[KN01] Gerwin Klein and Tobias Nipkow. Verified lightweight bytecode verification. *Concurrency – practice and experience*, 13(1), 2001.

[Ler06] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL*, pages 42–54, 2006.

[MCGW03] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. *J. Funct. Program.*, 13(5):957–959, 2003.

[MM04] Conor McBride and James McKinna. The view from the left. *J. Funct. Program.*, 14(1):69–111, 2004.

[MWCG99] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. *TOPLAS*, 21(3):527–568, May 1999.

[Nec97] George C. Necula. Proof-carrying code. In *POPL*, pages 106–119. ACM, January 1997.

[She04] Tim Sheard. Languages of the future. In *OOPSLA*, pages 116–119, 2004.

[WAS03] Dinghao Wu, Andrew W. Appel, and Aaron Stump. Foundational proof checkers with small witnesses. In *PPDP*, pages 264–274, August 2003.

[WSW05] Edwin Westbrook, Aaron Stump, and Ian Wehrman. A language-based approach to functionally correct imperative programming. In *ICFP*, pages 268–279, 2005.