JFP **18** (5 & 6): 599–647, 2008.
doi:10.1017/S0956796808006904 First published online 15 August 2008 Printed in the United Kingdom

Modular development of certified program verifiers with a proof assistant[1, 2]

ADAM CHLIPALA

University of California, Berkeley, CA, USA
(*e-mail:* adamc@cs.berkeley.edu)

Abstract

We report on an experience using the Coq proof assistant to develop a program verification tool with a machine-checked 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. We take advantage of Coq's support for programming with dependent types and modules in the structure of the 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 is 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.

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 life-cycle. 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, we will present the results of a particular experiment along these lines. The interesting twist to the specific problem we tackle is that it adds an

[1] This is a revised and extended version of the paper that appeared in the 11th ACM SIGPLAN International Conference on Functional Programming (ICFP'06) (Chlipala 2006).

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

additional layer of reflection to the approach we just described: we have been working on proving the correctness of **programs tools** 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, we 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 is 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 we 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) (Appel 2001). Like traditional proof-carrying code (PCC) (Necula 1997), 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* (Necula & Lee 1998) 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 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. To run programs produced with different compilers or that otherwise require fundamentally different proof strategies, it is necessary 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 we will describe comes from past work on improving the runtime efficiency of FPCC program checking (Chang *et al.* 2006). 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 widespread 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 is 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 (Chang *et al.* 2006) presented performance results showing an order of magnitude improvement over all published verification time figures for FPCC systems for Typed Assembly Language (Morrisett *et al.* 1999a) 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 (Dijkstra 1976) 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. We decided to investigate 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 (Bertot & Castéran 2004) and related formal logic tools are based on Martin-Löf constructive type theory (Martin-Löf 1996). 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 are 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 (McBride & McKinna 2004), ATS (Chen & Xi 2005), and RSP (Westbrook *et al.* 2005) 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 for 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. Similarly, 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) (Sheard 2004) is also closely related to these issues. GADTs are a particular restriction of the type systems supported by the tools we have mentioned above. The restriction is designed to make type inference more feasible. As the more general type systems used by tools like Coq are strong enough to express most of mathematics, there is little hope for general inference without sacrificing some expressivity.

Despite these potential objections to the use of Coq, we 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 we 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 do not 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 we 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. While Coq forces all recursive function definitions to follow very regular recursion schemes, almost all of the kinds of functions we associate with program analysis tend to be written via simple recursive case analysis in the first place. A common exception is a main loop that explores a state space or calculates a fixed point, but it is easy to express these as recursive in a new "time-out" parameter, which counts how many

more steps we are willing to allow. In practice, we can start these processes with very large time-outs and notice no difference from unbounded implementations, and the bothersome reasoning about time-outs is confined to the final steps of correctness proofs.

Leroy's recent work on certifying a complete compiler written in Coq (Leroy 2006b) 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 we will present here, we have tried to take as much advantage of dependent types as we can to simplify development.

1.3 Contributions

In the remainder of the paper, we will describe our 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.

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

1.4 Differences from the ICFP version

This article is an extended version of a paper presented at ICFP 2006 (Chlipala 2006). The main changes have been:

- The refactoring of Section 2.2 to better explain the advantages of the dependently-typed programming style used in this project
- The addition of Section 2.4, which gives some detail on the concrete program annotation format used by the case study verifier
- An alternate lay-out for Sections 3 and 4, interspersing code with prose discussion instead of keeping the code separate in floating figures, including breaking some monolithic code excerpts down further using auxiliary definitions
- Walking through a running example throughout Section 3
- In Sections 5.1 and 5.2, discussing briefly two libraries/tools of independent interest developed as part of this project
- Many improvements to the clarity of the prose, thanks to suggestions by the referees.

2 Preliminaries

2.1 Types and extraction in Coq

To introduce the basics of dependent types in Coq, we will 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 left implicit. This explicitness will be familiar to readers who have seen GADTs, as the same 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.

In the code fragment, 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 we have 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 a 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. Such types will show up often in describing the building blocks of a system that tackles an undecidable problem like program verification.

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.
```

We are using a lot of Coq notation here, but only a few details are relevant. First, in the first line, the type of `isEven` is given as a dependent function type, where Coq uses `forall` in place of the more usual Π. A type `forall (x : T1), T2` describes functions taking arguments of type `T1` and returning results of type `T2`, where the variable `x` is bound in `T2`, providing the opportunity for the function's result type to depend on the *value* of the argument.

Second, we provide a *partial* implementation for the function. We would rather not 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 the command form `Definition x : t. (* code *) Qed.` instead of `Definition x : t := e.`, 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 (Delahaye 2000). Theorem proving with tactics is not our focus in this work, so we will just describe the two simple tactics `refine` and `auto` that we have 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, we 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 does not fill in through unification, and we 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.
```

We introduce the syntax `[[P]]` for `poption P`, along with `Yes` and `No` for the `PSome _ _` and `PNone _` forms from the earlier example version. There is also the `pf <- isEven n; Yes` code snippet, which treats `poption` as a *failure monad* in the style familiar from Haskell programming (Wadler 1992). 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. In general, extraction *erases* all terms with sort `Prop`, leaving us with only 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. The `soption` type, which is an optional package of a value and a proof about that value, is defined as

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

The type `soption` is the return 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 important 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, we will use a more eye-friendly, non-ASCII notation for these types. We will denote `poption P` as $[\![P]\!]$ and `soption (fun x : T => P)` as $\{\!\{x : T \mid P\}\!\}$. We will also use the usual Π instead of Coq's `forall` to denote dependent function types.

2.2 Modules and correctness-by-construction

The development strategy that we just outlined is certainly not the fastest route to certified functions. More traditionally, the programmer writes a function in a standard programming language with a relatively weak type system. He next uses a *verification condition generator* to produce a logical formula whose truth implies that the function satisfies its specification. Finally, he proves this verification condition, maybe even using an automated first-order theorem prover.

Such techniques tend to be geared towards views of software as static artifacts. While the tools driving traditional verifiers do often exploit modularity, support for this tends to be viewed as an optimization. With an appropriately efficient theorem proving black box, we would be just as happy to verify whole programs from scratch after each change to their source code. Systems like those in the ESC family (Detlefs *et al.* 1998) exemplify this sort of approach. They tend to be focused more on the "prove shallow properties of large programs" end of the verification spectrum, rather than the "prove deep properties of modestly-sized programs" tasks that concern us here.

There does not seem to be much hope of automating the tricky parts of a proof of verifier soundness. This means that a human must take an active role as proof architect. Just as in software engineering, the inevitable consequence is that abstraction and modularity techniques are critical in enabling this human architect to do his job. What we really need is a convenient way of *composing proofs about software components in parallel to the components themselves*. We turn now to some examples demonstrating the value of this idea, which takes a variety of forms in idiomatic Coq code.

2.2.1 Dependent types and composition

First, a trivial example based on the isEven function from the last subsection. Imagine that we had elected to code isEven in a more traditional way, with no dependent types, leading to

$$\mathsf{isEven} \quad : \quad \mathsf{nat} \rightarrow \mathsf{bool}$$

Now we want to build a new function to check that both components of a pair of natural numbers are even.

$$\mathsf{pairEven}(n_1, n_2) \quad = \quad \mathsf{isEven}(n_1)\ \&\&\ \mathsf{isEven}(n_2)$$

Next, we write down the obvious specification for pairEven:

$$\forall n_1, n_2 \in \mathbb{N}, \mathsf{pairEven}(n_1, n_2) = \mathsf{true} \Rightarrow (\exists k_1 \in \mathbb{N}, n_1 = 2k_1) \wedge (\exists k_2 \in \mathbb{N}, n_2 = 2k_2)$$

How should we go about proving this theorem? We can prove a similar correctness theorem for isEven and use it to derive our goal straightforwardly. This verification style seems natural, but in practice it has a serious flaw. What happens as a certified program built in this way from components evolves? We have many *ad-hoc, implicit connections* between parts of programs and parts of proofs. For instance, the inductive argument used to prove the correctness of a recursive function will usually mirror the recursive structure of that same function. When we modify the function's source code, we need to hunt down the affected proof pieces and rewrite them, in a quite unprincipled way. In traditional software, we see such implicit dependencies as evidence of poor use of abstraction and modularity. There is no reason to be more forgiving when we move to certified programming.

To see a better approach, we return to the original version of isEven, having type

$$\mathsf{isEven} \quad : \quad \Pi(n : \mathsf{nat}).[\![\exists k \in \mathbb{N}, n = 2k]\!]$$

Now we can write pairEven like this, using the monadic notation introduced in the last subsection:

$$\begin{array}{rcl} & & pf_1 \leftarrow \mathsf{isEven}(n_1); \\ \mathsf{pairEven}(n_1, n_2) & = & pf_2 \leftarrow \mathsf{isEven}(n_2); \\ & & \mathsf{Yes} \end{array}$$

The final "Yes" adds a logical subgoal to be proved using tactics.

Why is this version superior? First, we have made explicit the connection between each function and its correctness proof. This helps us in the same way that grouping related program pieces into modules or classes helps. It establishes a machine-checked convention that can help keep a lone developer honest and facilitate collaboration between multiple developers with a reduced need for informal documentation.

Second, the explicit program-proof connection makes it easier to express compiler/prover error messages with the traditional idiom of type errors. Proof-related error messages can signal not only a point within a proof with ad-hoc structure, but also the particular line of code in the program where that proof was built. When a program modification leads to incompatibilities in transitive dependencies, this kind of error message can be much more effective in determining what needs changing.

Another very compelling advantage is not illustrated directly by our example. To build a new function by composing old functions, we had to do some new tactic-based proving. However, in many cases, dependent types let us write new certified code without ever writing anything we would call a "proof." For instance, say we generalized the basic idea of pairEven with a library function:

$$\begin{array}{rcl} \mathsf{pairCheck} & : & \Pi(\tau : \mathsf{Set})(P : \tau \rightarrow \mathsf{Prop}). \\ & & (\Pi(x : \tau).\ [\![P(x)]\!]) \\ & & \rightarrow \Pi(y : \tau \times \tau).\ [\![P(\pi_1 y) \wedge P(\pi_2 y)]\!] \end{array}$$

We define pairCheck as a polymorphic function for lifting a test on values of type τ to a test on values of type $\tau \times \tau$, where it should be verified that the original property holds for both components of the pair. Now we can write a complete implementation of pairEven with

$$\mathsf{pairEven} \quad = \quad \mathsf{pairCheck}\ \mathbb{N}\ (\lambda n, \exists k \in \mathbb{N}, n = 2k)\ \mathsf{isEven}$$

We get the same strong type that we came up with originally, without doing any more work than we would in traditional programming. (Here we give the evenness predicate as an explicit argument that the programmer must type, but Coq can actually infer such arguments automatically, in this case from the type of isEven.)

2.2.2 Modules and proofs

The quest for "correctness theorems for free" has driven the design of much of the certified software architecture that we will present in this paper. In practice, most of the interesting cases do require some new proof inputs. It is beneficial to seek out reusable idioms for soliciting these proofs. Analogous problems in traditional programming have been solved quite elegantly by ML-style module systems (MacQueen 1984). Coq features a natural extension of such systems to its dependently-typed setting, and we make essential use of modules for structuring certified components.

As an example, we can revisit the standard example of functorized containers, which past work has treated in the context of certified Coq programming (Filliatre & Letouzey 2004). A container family implemented with balanced trees requires a comparison operation over the type of keys that it stores. In ML, additional requirements on the comparator are stated in documentation but not checked. One reasonable requirement is that the comparator implement a total order. In Coq, we can make this requirement explicit and enable machine checking of it with this signature:

$$\begin{array}{lll}
\mathsf{ORDERED} = \textbf{sig} & & \\
\qquad T & : & \mathsf{Set} \\
\qquad leq & : & T \to T \to \mathsf{bool} \\
\qquad leq_antisym & : & \forall x, y : T, leq\ x\ y = \mathsf{true} \Rightarrow leq\ y\ x = \mathsf{true} \\
 & & \Rightarrow x = y \\
\qquad leq_trans & : & \forall x, y, z : T, leq\ x\ y = \mathsf{true} \Rightarrow leq\ y\ z = \mathsf{true} \\
 & & \Rightarrow leq\ x\ z = \mathsf{true} \\
\qquad leq_total & : & \forall x, y : T, leq\ x\ y = \mathsf{true} \vee leq\ y\ x = \mathsf{true} \\
\textbf{end} & &
\end{array}$$

The signature resembles one from an algebra textbook. Here we avoid using dependent types proper and rely instead on a more traditional axiom-based presentation. An equivalent formulation would leave out the three axioms and declare *leq* to have type $\{f : T \to T \to \mathsf{bool} \mid (*\ \text{the axioms hold for every input pair}\ *)\}$. This kind of type formed from a base type and a predicate over its values is called a *subset type*. The choice of which style to use depends on the circumstances. When we care

about multiple distinct properties of a single operation, the axiomatization approach is often more convenient. It is worth noting here that we can always transform an axiom-based implementation into one using only dependent types and no modules. In contrast to the situation for ML, the Coq module system adds no expressive power, but only convenience. Dependent record types subsume its features, though they can be significantly more clumsy to use.

We can define a signature of applicative sets, with some representative operations and one representative axiom:

$$
\begin{array}{lll}
\mathsf{SET} = \textbf{sig} & & \\
\quad T & : & \mathsf{Set} \\
\quad set & : & \mathsf{Set} \\
\quad member & : & set \to T \to \mathsf{bool} \\
\quad empty & : & set \\
\quad add & : & set \to T \to set \\
\quad add\_ok & : & \forall (S : set)(x : T), member\ (add\ S\ x)\ x = \mathsf{true} \\
\textbf{end} & &
\end{array}
$$

We can write a Coq functor that transforms ORDEREDs into SETs:

```
Module MakeSet(O : ORDERED) : SET with Definition T := O.T
  := struct (*...*) end.
```

The body of the functor will define the types and operations just as they would be defined in ML. The axiom *add_ok* would probably be proved using tactics, drawing on the axioms from the parameter O as lemmas.

Our main use of functors in this work has been for lowering a program verifier's level of abstraction. That is, such a functor takes as input a verifier and returns a new verifier at a lower level of abstraction, filling in the details that separate the two levels. The composition of these functors will give a method of transforming a high-level type-system description into a machine code analyzer. Naturally, we will require axioms establishing properties like type system soundness. Section 3 goes into detail on the component architecture that we use.

2.3 Problem formulation

The goal of this work is to support the verification of safety properties of executable x86 machine code programs. We have 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 (Appel 2001), but we summarize the formalization here to make it clear exactly what a successful verification guarantees.

$$
\begin{array}{rrlll}
\textit{Machine words} & \mathsf{word} & w & ::= & 0 \mid 1 \mid \ldots \mid 2^{32} - 1 \\
\textit{Registers} & \mathsf{reg} & r & ::= & \mathsf{EAX} \mid \mathsf{ESP} \mid \ldots \\
\textit{Flags} & \mathsf{flag} & f & ::= & \mathsf{Z} \mid \ldots
\end{array}
$$

Register files	regFile	R	$=$	reg $\rightarrow$ word
Flag files	flagFile	F	$=$	flag $\rightarrow$ bool
Memories	memory	M	$=$	word $\rightarrow$ byte
Machine states	state	S	$=$	word $\times$ regFile $\times$ flagFile $\times$ memory
Instructions	instr	I	::=	ERROR | MOV r, $[r]$ | JCC f, w | ...
Step relation		$\mapsto$	:	state $\rightharpoonup$ 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. We have 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 (Intel 2006), with the exception of the ERROR instruction added to model the safety policy. We 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, when we treat only non-terminating programs, 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 we made simplifies the 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 (Giménez 1995), which may have infinite derivations that are well-formed in a particular sense. Infinite derivations are important here for non-terminating programs. Similar techniques

have been used in recent related work, including Leroy's co-inductive big-step operational semantics (Leroy 2006a).

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, we 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 have 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. Alternatively, one could craft a specialized declarative proof or type system, as is common in Typed Assembly Language, and prove that the verifier enforces exactly its rules, though we have not done this to date. This is not so urgent a requirement as it may sound, as a formally-defined decision procedure is already quite close to a declarative proof system.

2.4 An example input

The techniques we describe in this paper are meant to be used with certifying compilers. However, since we chose to analyze machine code that follows a relatively simple type discipline not supported by any existing certifying compiler, it seemed more expedient to simulate the operation of a hypothetical compiler in producing test cases. To provide an idea of the real input format that these verifiers will be dealing with, we have included in Figure 1 a concrete listing of one of our test cases. This is meant to be used with one of the verifiers produced using the component architecture that we will describe in the next section: a verifier based on a simple type system of integers and lists.

This C source file, `length.c`, uses GCC-specific code to annotate the eventual machine code with metadata. This metadata can be thought of as consisting of a precondition for each basic block of the program. The header files shown in Figure 2 (generic metadata support) and Figure 3 (type system-specific support) define macros that emit inline assembly to save binary-encoded metadata in a special section of program binaries.

```
#include "intlist.h"

BEGIN_FUNC(70);
STACK_TYPE(4, CUSTOM(LIST(INT)));
END_FUNC();
int int_list_len(list *ls) {
  int len = 0;

  BEGIN_CUTPOINT(50);
  REG_TYPE(ESP, STACK(16));
  REG_TYPE(EBP, STACK(12));
  STACK_TYPE(4, CUSTOM(LIST(INT)));
  STACK_TYPE(8, RETPTR);
  STACK_TYPE(12, OLDEBP);
  END_CUTPOINT();
  for (;;) {
    if (ls) {
      ++len;
      ls = ls->next;
    } else {
      BEGIN_CUTPOINT(50);
      REG_TYPE(ESP, STACK(16));
      REG_TYPE(EBP, STACK(12));
      STACK_TYPE(8, RETPTR);
      STACK_TYPE(12, OLDEBP);
      END_CUTPOINT();
      return len;
    }
  }
}
```

Fig. 1. `length.c`.

The arguments of the `BEGIN_FUNC` and `BEGIN_CUTPOINT` macros are stack size bounds, used to support usage of a stack discipline without dynamic overflow checks, which can be trickier to reason about than fixed stack bounds. `REG_TYPE` annotations assign types to registers, and `STACK_TYPE` annotations assign types to stack slots numbered by offset from the stack pointer in effect at entry to the current function. The various uses of literal stack sizes and offsets are an example of mixing the C level of abstraction with the machine code level of abstraction. The example code listing should be thought of as a piece of machine code that we write using C as a macro assembler to automate the parts of low-level layout that it can easily be coerced to perform, without hesitating to calculate some offsets ourselves where necessary, based on knowledge of how GCC operates.

After sketching this hairy, ad-hoc scheme, it is worth pointing out that there are no formal proofs about it. Rather, an uncertified piece of OCaml code is responsible for reading this data and putting it into a nice, purely functional format at verification time. The certified Coq code then picks up this description and uses it as a "hint" in verification. A similar scheme could be used for other sorts of verifiers, including

```
#define EAX "0"
#define ECX "1"
#define EDX "2"
#define EBX "3"
#define ESP "4"
#define EBP "5"
#define ESI "6"
#define EDI "7"

#define TOP "0"
#define CONST(n) "1, " #n
#define STACK(n) "2, " #n
#define OLDEBP "3"
#define RETPTR "4"
#define CUSTOM(n) "5, " n

#define BEGIN_FUNC(ss) __asm__("0:;.section .cutpoints; .int 1, 0b, " #ss)
#define END_FUNC() __asm__(".int 0; .section .text")

#define BEGIN_CUTPOINT(ss) __asm__("0:;.section .cutpoints;.int 2, 0b," #ss)
#define END_CUTPOINT() __asm__(".int 0; .section .text")

#define REG_TYPE(r, t) __asm__(".int 1, " r ", " t)
#define STACK_TYPE(r, t) __asm__(".int 2, " #r ", " t)
```

Fig. 2. `firstorder.h`.

```
#include "firstorder.h"

#define INT "0"
#define LIST(t) "1, " t
#define NELIST(t) "2, " t

typedef struct list {
  void *data;
  struct list *next;
} list;
```

Fig. 3. `intlist.h`.

more ambitious approaches that require the inclusion of arbitrary logical formulas or even proof snippets with program binaries, although the particular component architecture that we will present next is not very well-suited to many such styles of verification.

3 Components for writing certified verifiers

The final goal of the case study we are presenting here was to produce a certified x86 machine code memory safety verifier that supports general product, sum, and recursive types, which we will call MemoryTypes. It would have been possible to

write this verifier monolithically, but we thought it would be more interesting and useful to do it in stages, writing reusable components with rich interfaces to handle different parts of verification and allow later components to reason at increasingly high levels of abstraction.

A word about generality. The rest of this paper can be thought of as presenting two distinct contributions. One is the particular architecture that is the subject of this section. We present a particular set of components that can be put to good use crafting verifiers for a particular class of programs.

That class is far from universal, however. The architecture we will describe cannot be used in the verification of programs with first-class code pointers, to name just one major weakness. Rather, this architecture is intended as a case study in support of the second contribution, which is a methodology for constructing certified program analysis tools.

We believe that the techniques we present, based on dependent types and abstraction-spanning functors, could be put to good use in designing a system to make it easier to implement, say, traditional typed assembly languages (Morrisett *et al.* 1999a). We have no empirical evidence to present here for that claim, but, in any case, the results here are interesting in their own right. The world of certified verifiers is a new one, and, as is usually the case in domains like this that depend on formal proofs, it is much easier to draw conclusions about what works and what does not by starting with relatively simple examples. So, in that light, we invite the reader to consider the final verifier produced in this work as a case study within a case study.

Introduction to the architecture. The component structure that we 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 nontrivial proofs. The component structure we have settled on has been designed not just to support effective programming, but also to support effective proof construction, by minimizing the need for repeated work. The issues and complexities specific to our domain of machine code verifiers are probably not clear to readers who do not have experience in that field, but we hope that the following walk-through of the steps in our 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?". Some designs that would work fine in traditional programming may force the proving of overly complicated soundness theorems for single modules, motivating better use of abstraction and modularity.

It is also true that we will be drilling down to a significant level of detail in this section. We invite the overwhelmed reader to look ahead to the light at the end of the tunnel in Section 4, where we show how all of this machinery pays off in making it very easy to build a new certified verifier.

Figure 4 presents the particular component structure that we 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, such as the x86 semantics. Rounded 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 (like the WeakUpdate component) are best viewed as library components, while transparent boxes represent certified verifiers, the final products. We include a number of verifier boxes with dashed borders. These stand for hypothetical verifiers that we have not implemented but that we believe would best be constructed starting from the components that connect to them in the diagram.

The basic paradigm here is that we use *functors* to transform *abstractions* to "lower levels;" that is, we enrich abstractions with explicit handling of details that they had previously considered to be tracked by "their environments." At the end of this process, an abstraction that makes the simplifying assumptions typical of high-level type systems will have "learned" to track all of the pertinent minutiae of x86 machine code. For our purposes here, an "abstraction" can be thought of as a perspective on how to "think about" the execution of programs, assuming that certain kinds of (abstraction-specific) information are available from an oracle. The concept of abstraction will remain informal, as different verification strategies suggest different categories of abstraction. The unifying similarity across all cases will be that each abstraction provides a language for describing program states and an account of how these states are affected by different concrete operations.

We will describe each library module in detail in the following subsections, but we will 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 we 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 our formalization, proving theorems about intractably large state spaces through exhaustive exploration of smaller abstract state spaces.
- The x86 instruction set is in the CISC family and thus involves lots of complications that one would rather avoid as much as possible, so we do most verification on a tiny RISC instruction set to which we 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 component chain used to build 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 code 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 where we think of program code as fixed and independent of machine state. 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 (Detlefs *et al.* 1998).

- **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 (Morrisett *et al.* 1999a; Morrisett *et al.* 2003), which handles stack and calling conventions with its own kind of stack types.
- However, for most verifiers, **StackTypes** would be a more convenient starting point than **TypeSystem**. It takes as input a type system ignorant of stack and calling conventions and produces a type system that understands them. An application of StackTypes to an "`int`-only" type system gives us a verifier capable of checking memory safety of simple pointer-free C programs.
- **SimpleFlags** 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. SimpleFlags would be a reasonable starting point for a verifier based on alias types (Smith *et al.* 2000) or some other way of supporting manual memory management.
- However, in the case of 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 4 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 `ORDERED` changed to `STACK_TYPE_SYSTEM`, the output signature of the functor changed from `SET` to `TYPE_SYSTEM`, and the functor's innards assembling a richer type system by extending that presented by its input module.

We will now describe the most important aspects of the interfaces and implementations of each of these reusable library components. We avoid describing how the actual proofs are constructed, focusing instead on component interfaces and a bird's-eye view of an overall structure that we have found to work in practice. Nonetheless, there are many important engineering issues in proof construction, and doing the subject justice would require another article of its own. The main thing to keep in mind through the following sections is that every piece *is* supported by Coq

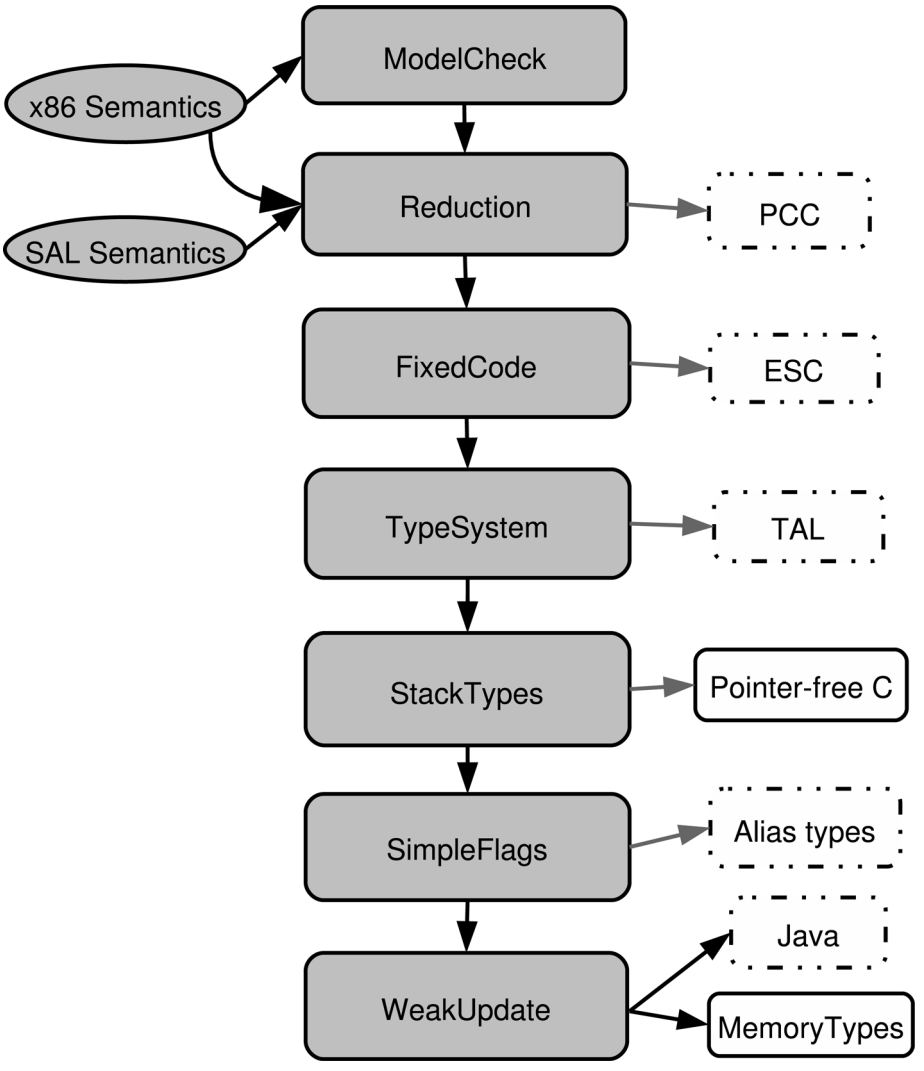

Fig. 4. A component structure for certified verifiers.

proofs of the relevant properties, and that we were able to construct these proofs using the techniques sketched in Section 1.2.

The reader can follow along with a more detailed version of this discussion at*

`http://proofos.sourceforge.net/doc/`

The section of that page headed "The AbsInt Coq library" links to automatically-generated HTML pages giving the interfaces of all of the Coq module implementations. In this article, we present simplified versions of the modules that may have some incompatibilities with their "real" counterparts, but the online documentation should make the broad correspondences clear.

* Also available at `http://journals.cambridge.org/issue_Journaloffunctionalprogramming/Vol18No5-6`.

```
Module Type MC_ABSTRACTION.
    Module      Mac     :   MACHINE.

  Parameter   absState    :   Set.
  Parameter   context     :   Set.
  Parameter   ⊢S          :   context → Mac.state → absState → Prop.

   Definition  mcstate    :=  absState × list absState.

  Parameter     init      :   {states : list (absState × list absState)
                               | ∃Γ : context, ∃α : absState,
                                 (α, nil) ∈ states ∧ Γ ⊢S Mac.start : α}.

                           (* Auxiliary definitions *)

  Parameter     step      :   Π(hyps : list absState)(α : absState).
                               {{succs : list absState | progress(α)
                                 ∧ preservation(α, hyps, succs)}}.
End MC_ABSTRACTION.

Module Type VERIFIER.
    Module    Mac     :   MACHINE.

  Parameter   check     :   Π(p : Mac.program).[[Mac.safe(Mac.load(p))]].
End VERIFIER.

Module ModelCheck (A : MC_ABSTRACTION)
       : VERIFIER with Module Mac := A.Mac.
  (* Definitions *)
End ModelCheck.
```

Fig. 5. Input and output signatures of ModelCheck.

3.1 ModelCheck

Figure 5 shows the input and output signatures of the first component in the pipeline, ModelCheck. Since such figures can be hard to digest, in the rest of this section we have opted to follow a different convention. We will not discuss the output signatures of functors, as they just match up with the inputs of earlier pipeline stages, in general. We will present input signatures member-by-member, interleaved with explanatory text in the main body of this section. Moving through the subsections describing the different functors, the input signatures will build on their predecessors. Thus, we will only describe *new* members; old members that do not draw explicit mention should be assumed to stay the same as in the previous cases. When an old member is mentioned again later, the new definition should be taken to replace the original.

The definitions of the fixed component input signatures will alternate with illustrative examples. To help distinguish which symbols are which, names of signature pieces ("formal parameters") will be written in *italics*, while names of pieces of examples ("actual parameters") will be written in a serif font. Following

the same "interface versus implementation" convention, standard abbreviations that are included in signatures will also have serif names.

The astute reader will be able to notice cases where, following this convention literally, some of the signatures that would be assigned to the different components will be incomplete or inconsistent. We have chosen what to include with pedagogic effectiveness in mind. A pleasant consequence of describing a completely formal piece of theory is that we need not worry about this sort of omission as much as usual, since the formalization is available in full on the Web. (See Section 5.) We hope that the result in these pages manages to express the key ideas of the components while not taxing the reader's patience.

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 with an implementation of a signature that we will describe piece by piece.

Before going into these details, we 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 cannot themselves refer to other code pointers. The formulation we 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.

Now we 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.

$$absState \quad : \quad \mathsf{Set}$$

Next, we have an abstraction relation:

$$\begin{aligned} context \quad &: \quad \mathsf{Set} \\ \vdash_S \quad &: \quad context \to Mac.state \to absState \to \mathsf{Prop} \end{aligned}$$

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 serving in this role. It is very closely related to the abstraction relations of abstract interpretation (Cousot & Cousot 1977). However, here 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. Though we define contexts to have computational content, it will turn out that, if we use a naive extraction that preserves all definitions in sort Set, a dependency analysis starting from the resulting verifier's entry point will show that contexts are not used. Coq's `Recursive`

`Extraction` command achieves the same effect by outputting definitions lazily, as needed to export one root definition.

A canonical example of a context is a valuation to free type variables used in an abstract state:

$$\begin{array}{rrrcl} \text{Type variables} & \mathsf{tyvar} & \beta & & \\ \text{Types} & \mathsf{type} & \tau & ::= & \beta \mid \ldots \\ & & \mathit{absState} & = & \mathsf{reg} \to \mathsf{type} \\ & & \mathit{context} & = & \mathsf{tyvar} \to \mathsf{type} \end{array}$$

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. We 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. To describe ModelCheck and the functors that follow it, we will use a simple running example, based on elements of verifying simple compiled C programs. Some features that we present monolithically here will actually be added modularly by earlier functors, and we omit some details here necessary for real verification.

$$\begin{array}{rrcl} \mathsf{type} & \tau & ::= & \mathsf{int} \mid \tau\ \mathsf{ptr} \mid \mathsf{retptr} \\ & \mathit{absState} & = & (\mathsf{word} \cup \{\mathsf{retptr}\}) \times (\mathsf{reg} \to \mathsf{type}) \\ & \mathit{context} & = & \mathsf{word} \end{array}$$

As in the real final verifier of our case study, this is a type-based checker. Our types include untagged integers, pointers, and a distinguished type for a saved function return pointer.

Abstract states have two components. First, we have a representation of the program counter. This will either be a word, giving the known program counter value; or a distinguished `retptr` value, for a state immediately after jumping to the saved return pointer. The second abstract state component is the standard map from registers to types.

Our contexts are very simple here. There is only one piece of information needed in reasoning about the verifier's correctness but not in its algorithmic operation: the concrete value of the return pointer. We check functions with the return pointer treated symbolically, but our soundness proof refers to the context to show that the verification applies to any concrete function call that might be made.

In general, each element of a ModelCheck state space consists of one *absState* describing the current state and zero or more *hypotheses* (represented with a list *absState*) describing other abstract states known to be safe.

$$\mathsf{mcstate} = \mathit{absState} \times \mathsf{list}\ \mathit{absState}$$

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

0 :	MOV EAX, p	$((0, []), \emptyset)$
4 :	CALL 12	$((4, [\mathsf{EAX} \mapsto \mathsf{int\ ptr\ ptr}]), \emptyset)$
8 :	JMP 8	$((8, [\mathsf{EAX} \mapsto \mathsf{int\ ptr\ ptr}]), \emptyset)$
12 :	MOV [EAX], EAX	$((12, [\mathsf{EAX} \mapsto \mathsf{int\ ptr\ ptr}, \mathsf{EBX} \mapsto \mathsf{retptr}]), \{(\mathsf{retptr}, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}])\})$
16 :	JMP EBX	$((16, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}, \mathsf{EBX} \mapsto \mathsf{retptr}]), \{(\mathsf{retptr}, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}])\})$
20 :	MOV [EAX], EAX	$((20, [\mathsf{EAX} \mapsto \mathsf{int\ ptr\ ptr}, \mathsf{EBX} \mapsto \mathsf{retptr}]), \{(\mathsf{retptr}, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}])\})$
24 :	JMP 28	$((24, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}, \mathsf{EBX} \mapsto \mathsf{retptr}]), \{(\mathsf{retptr}, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}])\})$
	...	
28 :	...	$((28, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}, \mathsf{EBX} \mapsto \mathsf{retptr}]), \{(\mathsf{retptr}, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}])\})$
	...	
45 :	JMP 28	$((45, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}, \mathsf{EBX} \mapsto \mathsf{retptr}]), \{(\mathsf{retptr}, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}])\})$

Fig. 6. Example program for Section 3.1.

return pointer's hypothesis, then there is no need to explore that branch of the state space further.

Here is an example element of a ModelCheck state space. We overload set notations to work for lists, where a list is interpreted as the set of its elements. We also write $[r_1 \mapsto \tau_1, \ldots, r_n \mapsto \tau_n]$ to denote the function mapping each register r_i to type τ_i and all other registers to int.

$$((42, [\mathsf{EAX} \mapsto \mathsf{int\ ptr\ ptr}, \mathsf{EBX} \mapsto \mathsf{retptr}]), \{(\mathsf{retptr}, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}])\}) : \mathsf{mcstate}$$

This tuple is a standard state for the inside of a function body. The program counter is known to be 42, register EAX points to a pointer to an integer, and register EBX holds the saved return pointer. Our single hypothesis reflects the conditions under which it is safe to return from the function: the program counter must be restored to the saved value, and EAX must point to an integer.

Our next ingredients are the algorithmic pieces of a verifier. These are *init*, which calculates the roots of the state space; and *step*, which describes how to expand the visited state space by following the edges out of a single node. Starting with *init*:

$$\begin{aligned} \mathit{init} \quad : \quad & \{\mathit{states} : \mathsf{list}\ (\mathit{absState} \times \mathsf{list}\ \mathit{absState}) \\ & \mid \exists \Gamma : \mathit{context}, \exists \alpha : \mathit{absState}, \\ & \quad (\alpha, \mathsf{nil}) \in \mathit{states} \wedge \Gamma \vdash_S \mathit{Mac.start} : \alpha\} \end{aligned}$$

The value *init* of subset type provides a set (actually a list) of state descriptions, along with a guarantee that some abstract state compatible with the concrete initial state is included. 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.

An example should make the intended usage of *init* clearer. Consider the program (annotated with abstract states) in Figure 6. Let us say that the concrete initial state has the program counter, registers, and memory cells all initialized to 0. The program we want to verify has, in addition to start-up code at PC 0, two functions. The start-up code calls the first function and then enters an infinite loop.

Each function takes an int ptr ptr as an argument and returns an int ptr. Their entry points are PC's 12 and 20. The first function has just a single basic block, and

the second function has an additional basic block starting at PC 28 and containing a loop. For the purposes of this example, we will use a calling convention where the first argument to a function is passed in register EAX, and the return value is stored in the same register. We also use register EBX to store the saved return pointer. This determines the states on entry to the functions, and let us consider that the loop entry point (PC 28) assumes that EAX has been modified to contain an int ptr.

Here is a good set of initial states to give for the computational piece of *init*. There is one entry for each underlined program counter in Figure 6.

$$\{((0, []), \emptyset),$$
$$((8, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}\ \mathsf{ptr}]), \emptyset),$$
$$((12, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}\ \mathsf{ptr}, \mathsf{EBX} \mapsto \mathsf{retptr}]), \{(\mathsf{retptr}, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}])\})$$
$$((20, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}\ \mathsf{ptr}, \mathsf{EBX} \mapsto \mathsf{retptr}]), \{(\mathsf{retptr}, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}])\})$$
$$((28, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}, \mathsf{EBX} \mapsto \mathsf{retptr}]), \{(\mathsf{retptr}, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}])\})\}$$

As the discussion to follow shortly will show, *init* cannot simply return 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 precomputed fixed point of an abstract interpretation. We 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 is OK to implement this part outside of Coq.

We now turn to the *step* function. It has a complicated type with several pieces, so we will present it in stages, using a series of predicates to be defined one at a time.

$$\begin{aligned} \mathit{step} \quad : \quad & \Pi(\mathit{hyps} : \mathsf{list}\ \mathit{absState})(\alpha : \mathit{absState}). \\ & \{\!\{\mathit{succs} : \mathsf{list}\ \mathit{absState} \mid \mathsf{progress}(\alpha) \wedge \mathsf{preservation}(\alpha, \mathit{hyps}, \mathit{succs})\}\!\} \end{aligned}$$

The overall form of *step* is simple enough. It is a dependent function taking as inputs the current hypothesis set and the current abstract state. Note that while *init*'s type is a standard subset type, *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, it returns the set of new states that should be explored before declaring the program safe. The subset type imposes some standard requirements on such a set of states. We have a *progress* condition, expressing that the program is able to make at least one execution step; and a *preservation* condition, expressing that every possible concrete state transition matches up with an abstract transition to a member of *succs*.

The progress condition is the simpler of the two:

$$\begin{aligned}\mathsf{progress}(\alpha) \quad = \quad & \forall (s : Mac.state)(\Gamma : context), \\ & \Gamma \vdash_S s : \alpha \Rightarrow \exists s' : Mac.state, s \mapsto_{Mac} s'\end{aligned}$$

For every concrete state and context compatible with the current abstract state, execution must make at least one more step of execution. In other words, the program counter must not be pointing to an ERROR instruction.

The preservation condition is broken into two cases, roughly corresponding to regular and control-flow instructions.

$$\begin{aligned}\mathsf{preservation}(\alpha, hyps, succs) \quad = \quad & \forall (s, s' : Mac.state)(\Gamma : context), \\ & s \mapsto_{Mac} s' \wedge \Gamma \vdash_S s : \alpha \\ & \Rightarrow \exists \Gamma' : context,\ \mathsf{regular}(hyps, succs, s', \Gamma') \\ & \vee \mathsf{controlFlow}(hyps, succs, s', \Gamma, \Gamma')\end{aligned}$$

In the definition of preservation, we consider any concrete state transition with a source state compatible with α in some context Γ. There must exist a new context Γ' such that one of the two subconditions regular and controlFlow applies. The first kind of preservation is the simple one, corresponding to most instructions in a program:

$$\begin{aligned}\mathsf{regular}(hyps, succs, s', \Gamma') \quad = \quad & \exists \alpha' : absState, \alpha' \in (hyps \cup succs) \\ & \wedge \Gamma' \vdash_S s' : \alpha'\end{aligned}$$

When regular holds, we have a new abstract state α' that is either a hypothesis or one of the new states queued for exploration. In either case, the fact that α' describes s' in Γ' shows us that the work we have already done or queued to do will cover this concrete transition.

The definitions so far are sufficient to demonstrate the handling of two representative cases for our running example. Consider first the entry point:

$$((12, [\mathsf{EAX} \mapsto \mathsf{int\ ptr\ ptr}, \mathsf{EBX} \mapsto \mathsf{retptr}]), \{(\mathsf{retptr}, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}])\})$$

for the example's first function. The first instruction is MOV [EAX], EAX, which dereferences the pointer stored in register EAX and writes the result back into EAX. Passed the given hypotheses and state, *step* could return

$$\{(16, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}, \mathsf{EBX} \mapsto \mathsf{retptr}])\}$$

We establish the progress condition by looking up address 12 in the program code and verifying that it contains the encoding of a non-ERROR instruction. We establish preservation using the regular case, setting α' to be the sole member of our successor state set and keeping the original context Γ as Γ'. Proving $\Gamma' \vdash_S s' : \alpha'$ involves proving a typing condition for each register. We handle it trivially for the registers that have not changed, and the case for EAX is proven directly from the semantics of the MOV instruction and ptr types.

A more interesting case is a return from the function. Observe that the last instruction that we considered put the program into a state where it is ready for

a return, and the next instruction JMP EBX does exactly that. The abstract state beforehand is

$$((16, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}, \mathsf{EBX} \mapsto \mathsf{retptr}]), \{(\mathsf{retptr}, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}])\})$$

The function *step* can return the empty set, since we will assume that all possible return states are queued for visitation at the times their calls are made. We establish progress in the same way as before. To show preservation, we exhibit this state as α':

$$(\mathsf{retptr}, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}])$$

The only changes from the starting state are that the program counter has changed to retptr and we have forgotten what we knew about the type of EBX. We exercise the first possibility in the condition $\alpha' \in (hyps \cup succs)$, as α' is exactly our return pointer hypothesis. Assuming reasonable definitions for the underlying typing judgment, it is easy to establish $\Gamma' \vdash_S s' : \alpha'$ when we set $\Gamma' = \Gamma$. A primary element of this reasonableness is that retptr should be interpreted as the singleton type of words equal to the return pointer saved in the context Γ.

The second kind of preservation is associated with direct jumps and function calls

$$\begin{aligned}\mathsf{controlFlow}(hyps, succs, s', \Gamma, \Gamma') \quad = \quad & \exists (hyps' : \mathsf{list}\ absState)(\alpha' : absState), \\ & (\alpha', hyps') \in \pi_1 init \\ & \wedge\ \Gamma' \vdash_S s' : \alpha' \\ & \quad \wedge\ (\Gamma', hyps') \leqslant_S^{succs} (\Gamma, hyps)\end{aligned}$$

It requires that we look into the roots of the state space and find one $(\alpha', hyps')$ that is compatible with s'. The operator π_1 expresses projecting out the computational part of a subset type, in this case giving us the list of roots without the proof concerning them. Again we may provide a new context Γ'. However, this time 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 is 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. The notation $\leqslant_S$ expresses this requirement:

$$\begin{aligned}(\Gamma', hyps') \leqslant_S^{succs} (\Gamma, hyps) \quad = \quad & \forall h' \in hyps', \exists h \in (hyps \cup succs), \\ & \forall s, \Gamma' \vdash_S s : h' \Rightarrow \Gamma \vdash_S s : h\end{aligned}$$

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.

Another two example cases help illustrate the use of controlFlow. We look first at a direct jump within a function. Let us start at PC 24, a point in the second function just after dereferencing the function's argument into EAX. We are in this abstract state:

$$((24, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}, \mathsf{EBX} \mapsto \mathsf{retptr}]), \{(\mathsf{retptr}, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}])\})$$

We encounter a direct jump to program counter 28. Now we use the following singleton set of successor states and draw α' from it:

$$\{(28, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}, \mathsf{EBX} \mapsto \mathsf{retptr}])\}$$

We can verify that the pairing of this state with the current return hypothesis is indeed found among the elements of *init*. The compatibility conditions from controlFlow are verified trivially, since the starting abstract state differs from the result only in a change of program counter.

Finally, let us look at the entry point to the example program, which initializes EAX to a valid pointer of proper type, calls the function, and then loops forever at its return site. Before the call, the abstract state would look something like

$$((4, [\mathsf{EAX} \mapsto \mathsf{int\ ptr\ ptr}]), \emptyset)$$

The $\emptyset$ shows that we have no hypotheses yet. Upon using a fictitious function call instruction that sets the program counter and EBX simultaneously, we would want to present a list of successor states like this one:

$$\{((8, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}]), \emptyset)\}$$

The single successor is for the return point. It is our responsibility as the caller to queue it for visitation, so that the function may assume it safe to return to. To prove preservation, we present an $(\alpha', hyps')$ that has gone through a point-of-view shift to the function's perspective:

$$((((12, [\mathsf{EAX} \mapsto \mathsf{int\ ptr\ ptr}, \mathsf{EBX} \mapsto \mathsf{retptr}]), \{(\mathsf{retptr}, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}])\})$$

It is easy to show $\Gamma' \vdash_S s' : \alpha'$ with Γ' set to 8, the address of the return site. The hypothesis safety condition also comes easily, establishing the safety of the single hypothesis by pointing out that an equivalent to it has been queued for visitation.

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 controlFlow case, 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.

This description reveals that ModelCheck does not *quite* produce a standard model checker. Rather, it produces a standard *validator* for the results of a traditional fixed point calculation. The calculation must have been performed ahead of time by a certifying compiler or some other noncertified (and untrusted) algorithm that relays its answer to the certified piece of the code through *init*, as described in Section 2.4.

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 our implementation, we 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 we 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 (Necula 1997). 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.

Here is a brief summary of the language grammar:

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

We add a finite set of extra temporary registers, to be used when a single complicated x86 instruction is translated into an equivalent RISC sequence. 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.

It is worth pointing out that, while the temporary registers introduced here bear a superficial resemblance to the infinite supplies of temporaries typically associated with compiler intermediate languages, they are purely a semantic trick and need no accounting for how they may be supported "on real hardware." They only appear in the language to allow us to decompose x86 instructions as in the example above, since there are not, in general, any spare "real" registers to take the place of, e.g., TMP_1 in the example.

The details of SAL and Reduction are not especially enlightening, so we 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 do not 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.

We modify the verifier signature used by ModelCheck. (Recall the convention introduced in Section 3.1 that definitions that we present in the context of a

component should be treated as emending the input signature of the previous component.) 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. Here we use a type memoryRegion that describes a contiguous span of a 32-bit address space.

$$\begin{array}{lcl} prog & : & \mathsf{word} \rightarrow \mathsf{byte} \\ code & : & \mathsf{memoryRegion} \end{array}$$

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 do not know the precise program counter of a hypothesis describing a return pointer, but this does not matter, since we are sure to visit all of the concrete program locations it could stand for.

$$\begin{array}{lcl} absPc & : & \Pi(\alpha : absState)\,.\,\{\!\{pc : \mathsf{word} \\ & & \quad \mid \forall \Gamma, s, \Gamma \vdash_S s : \alpha \Rightarrow s.pc = pc\}\!\} \end{array}$$

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.

$$\begin{array}{lcl} \mathsf{progress}(\alpha) & = & \forall (s : Mac.state)(\Gamma : context), \\ & & \Gamma \vdash_S s : \alpha \Rightarrow \exists s' : Mac.state, s \mapsto_{Mac} s' \\ & & \wedge\ \underline{\mathsf{instrOk}(s)} \end{array}$$

To define instrOk, we use the notation $|e|_s$ to stand for the result of evaluating expression e in machine state s, and the operation lookupInstr(s) decodes the instruction pointed to by s's program counter in s's memory.

$$\mathsf{instrOk}(s) = \begin{cases} |dst|_s \notin code, & \mathsf{lookupInstr}(s) = \mathsf{STORE}\ [dst],\ src \\ \mathsf{True}, & \text{otherwise} \end{cases}$$

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. This rules out, for instance, direct verification based on the relational domains common in abstract interpretation.

We give an overview here of the signature required of type systems. We omit many of the details, but the pieces we have chosen 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 $\leqslant_T$.

$$\begin{array}{rcl} ty & : & \mathsf{Set} \\ \vdash_T & : & context \to \mathsf{word} \to ty \to \mathsf{Prop} \\ \leqslant_T & : & \Pi(\tau_1 : ty)(\tau_2 : ty).\ [\![\forall \Gamma, w, \\ & & \quad \Gamma \vdash_T w : \tau_1 \Rightarrow \Gamma \vdash_T w : \tau_2]\!] \end{array}$$

It is 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 we demonstrated in Section 1.2, where the defined relation is placed in sort Prop.

We can see how our running example fits this signature, setting $ty = \mathsf{type}$. We first expand our notion of contexts to include a map from memory locations to types, in addition to the return pointer it stored before:

$$context \quad = \quad \mathsf{word} \times (\mathsf{word} \to \mathsf{type})$$

Now a simple set of typing rules suffices

$$\frac{}{\Gamma \vdash_T w : \mathsf{int}} \quad \frac{\Gamma(w) = \tau}{\Gamma \vdash_T w : \tau\ \mathsf{ptr}} \quad \frac{}{\Gamma \vdash_T \Gamma.retptr : \mathsf{retptr}}$$

The notation $\Gamma(w)$ denotes looking up the type of a memory location in the type map part of Γ, and $\Gamma.retptr$ denotes projecting out Γ's return pointer. We define the state abstraction relation $\vdash_S$ to enforce that every memory location really does have the type assigned to it. Here we are following the syntactic approach to FPCC (Hamid *et al.* 2003) by using these type maps in place of potentially-complicated recursive conditions in typing rules.

We can also define a simple subtyping relation:

$$\frac{}{\tau \leqslant_T \mathsf{int}} \quad \frac{}{\tau \leqslant_T \tau}$$

This is "semantic" subtyping in the sense that we are identifying a (possibly strict) subset of the pairs of types that can be proved compatible in terms of $\vdash_T$, rather than defining a new syntactic notion.

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

typeof function that a client abstraction must provide explains how to do this.

$$\begin{aligned} \mathit{typeof} \quad : \quad & \Pi(\alpha : \mathit{absState})(\vec{r} : \mathsf{reg} \to \mathit{ty})(e : \mathsf{exp}). \\ & \{\tau : \mathit{ty} \mid \forall \Gamma, s, \\ & \quad \Gamma \vdash_S s : \alpha \\ & \quad \Rightarrow (\forall r, \Gamma \vdash_T s.\mathsf{regs}(r) : \vec{r}(r)) \\ & \quad \Rightarrow \Gamma \vdash_T |e|_s : \tau\} \end{aligned}$$

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.

The *typeof* function is quite uninteresting for our running example. Omitting proof components, we have

$$\mathit{typeof}(\alpha, \vec{r}, e) = \begin{cases} \vec{r}(r), & e = \text{register } r \\ \mathsf{int}, & \text{otherwise} \end{cases}$$

As SAL expressions are side effect-free, we need a separate *typeofLoad* function for a memory dereference.

$$\begin{aligned} \mathit{typeofLoad} \quad : \quad & \Pi(\alpha : \mathit{absState})(\vec{r} : \mathsf{reg} \to \mathit{ty})(e : \mathsf{exp}). \\ & \{\!\{\tau : \mathit{ty} \mid \forall \Gamma, s, \\ & \quad \Gamma \vdash_S s : \alpha \\ & \quad \Rightarrow (\forall r, \Gamma \vdash_T s.\mathsf{regs}(r) : \vec{r}(r)) \\ & \quad \Rightarrow \Gamma \vdash_T s(|e|_s) : \tau\}\!\} \end{aligned}$$

Note that *typeofLoad*'s range is a partial subset type, since not all types are valid for reading. For our running example, we have the following, using $\perp$ to denote `SNone`, the failure case:

$$\mathit{typeofLoad}(\alpha, \vec{r}, e) = \begin{cases} \tau, & \mathit{typeof}(\alpha, \vec{r}, e) = \tau\ \mathsf{ptr} \\ \perp, & \text{otherwise} \end{cases}$$

The full TypeSystem signature also includes an analogue for writes, enforcing that a "writable pointer" cannot point into program memory and other similar conditions.

A value *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. 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 is

important that the requirements on *viewShift* be very flexible. Its type is

$$\begin{aligned} \mathit{viewShift} \quad : \quad & \Pi(\alpha : \mathit{absState})(\vec{r} : \mathsf{reg} \to \mathit{ty})(i : \mathsf{instr}). \\ & \{\!\{(\alpha', \vec{r'}) : \mathit{absState} \times (\mathsf{reg} \to \mathit{ty}) \\ & \quad \mid \forall \Gamma, s, \\ & \qquad \Gamma \vdash_S s : \alpha \\ & \qquad \Rightarrow (\forall r, \Gamma \vdash_T s.\mathsf{regs}(r) : \vec{r}(r)) \\ & \qquad \Rightarrow \exists \Gamma', \Gamma' \vdash_S s : \alpha' \\ & \qquad\quad \wedge (\forall r, \Gamma' \vdash_T s.\mathsf{regs}(r) : \vec{r'}(r))\}\!\} \end{aligned}$$

The type expresses that *viewShift* 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. Another way of thinking of *viewShift* is that it is a hook into the generic verifier that makes it possible to "cast" abstract states whenever a proof can be provided that the cast's target state is "no narrower than" the source state.

To illustrate a simple use of *viewShift*, we extend our running example type system temporarily with parametric polymorphism:

$$\begin{array}{rrcl} \mathsf{typevar} & \beta & & \\ \mathsf{type} & \tau & ::= & \mathsf{int} \mid \tau\ \mathsf{ptr} \mid \mathsf{retptr} \mid \beta \\ & \mathit{context} & = & \mathsf{word} \times (\mathsf{word} \to \mathsf{type}) \times (\mathsf{typevar} \to \mathsf{type}) \end{array}$$

We might specify the entry point to a polymorphic identity function like this, where return values are passed through register EAX:

$$((39, [\mathsf{EAX} \mapsto \beta, \mathsf{EBX} \mapsto \mathsf{retptr}]), \{(\mathsf{retptr}, [\mathsf{EAX} \mapsto \beta])\})$$

If the concrete return pointer for a particular call is 92, then the state before calling the function for type int might be

$$((88, [\mathsf{EAX} \mapsto \mathsf{int}]), \{(\mathsf{retptr}, [])\})$$

Incorporating the effects of the call instruction, *viewShift* returns the abstract state for the function's entry point. Starting from a context (pc, w, v), we construct a new context $(92, w, [\beta \mapsto \mathsf{int}])$ to justify the equivalence of the old and new abstract states.

It is worth recalling the setting 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. $\leqslant_T$, *typeof*, and *viewShift* have both computational and logical content. For instance, the extracted version of $\leqslant_T$ is a potentially incomplete decision procedure with boolean answers. The extracted OCaml version of the verifier ends up looking like a standard type checker. One 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. 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.

$$\begin{aligned} \mathit{stack} &: \mathsf{memoryRegion} \\ \mathit{stackCodeDisjoint} &: \mathsf{disjoint}(\mathit{stack}, \mathit{code}) \end{aligned}$$

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 *checkStore* function is used for this purpose. The verifier StackTypes builds calls *checkStore* on an expression that is the target of a STORE instruction to make sure that it will not evaluate to an address in the stack region.

$$\begin{aligned} \mathit{checkStore} : \Pi(\alpha : \mathit{absState})(\vec{r} : \mathsf{reg} \to \mathit{ty})(e : \mathsf{exp}). \\ [\![\forall \Gamma, s, \Gamma \vdash_S s : \alpha \\ \Rightarrow (\forall r, \Gamma \vdash_T s.\mathsf{regs}(r) : \vec{r}(r)) \\ \Rightarrow |e|_s \notin \mathit{stack}]\!] \end{aligned}$$

In our current implementation, the type of *checkStore* "exposes" 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 is 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 have 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. While we have included an ad-hoc retptr type in our example so far,

from here on we will let StackTypes add it uniformly to whatever type system we are considering.

Perhaps surprisingly, StackTypes was the most involved to construct out of the components that we have listed. The most complicated implementation work dealt with the view shifts that occur during function calls and returns. Precise, bit-level proofs about calling conventions end up much more complex than they seem from an informal understanding. There is definitely room here for better automatic decision procedures for the theory of fixed-precision integers, as those sorts of proofs made up a good portion of the work.

We can now look at our running example, minus the ad-hoc retptr type, run through the StackTypes functor. While we have been using a simple fictitious calling convention to this point, we are ready now for a more realistic example. Here is a reasonable state description to apply right after the preamble at the start of a function from int to int ptr. The general organization is reminiscent of stack-based typed assembly language (Morrisett *et al.* 2003).

$$((81, [\mathsf{ESP} \mapsto \mathit{Stack}_8], [0 \mapsto \mathsf{int}, 4 \mapsto \mathit{Retptr}, 8 \mapsto \mathit{Saved}_{\mathsf{EBX}}])$$
$$\{(\mathit{Retptr}, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}, \mathsf{EBX} \mapsto \mathit{Saved}_{\mathsf{EBX}}], [\,])\})$$

We follow the x86 C calling convention, though we omit some of the details here for clarity. On the stack, we have (in order) the function's argument, the saved return pointer, and the saved value of register EBX (which is designated as callee-save). The stack pointer register ESP points to the end of these values. The return hypothesis tells us that, before returning, the function must have stored the int ptr return value in EAX and restored EBX to its original value.

To allow us to define *checkStore*, we need to modify our original typing rules to preclude ptr-type values that point into the stack:

$$\frac{\Gamma(w) = \tau \quad w \notin \mathit{stack}}{\Gamma \vdash_T w : \tau\ \mathsf{ptr}}$$

We as clients of StackTypes do not need to define typing rules for the stack-related type constructors, because StackTypes adds those to our typing judgment parametrically. The domain of contexts is expanded similarly to include the concrete starting address of the current stack frame and a map from callee-save registers to their initial values. The custom abstract states are extended to record stack frame length information.

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 is 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 this function:

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

The type of *considerTest* looks similar to the type of *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 *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 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.

We can demonstrate the basic operation with another extension of our running example. We consider the simplified setting where the only flags are Z (zero) and C (carry). For clarity, we ignore the StackTypes functionality added in the last subsection and focus instead on registers alone.

We extend our type system with nullness information on pointers:

$$\mathsf{type} \quad \tau \quad ::= \quad \mathsf{int} \mid \tau\ \mathsf{ptr} \mid \underline{\tau\ \mathsf{ptr}^?} \mid \mathsf{retptr}$$

$\tau\ \mathsf{ptr}^?$ is the new type standing for a possibly-null pointer to τ.

Now consider the following program state.

$$((81, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}^?], []), \{(\mathit{Retptr}, [], [])\})$$

We omit stack slot type information and include flag information in its place. Here, EAX is a possibly-null pointer to int, and nothing is known about the values of the flags.

If the next instruction is CMP 0, EAX, which compares the value in EAX against the constant 0 to set the flags, we would transition to a state like

$$((85, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}^?], [Z \mapsto \mathsf{EAX} - 0, C \mapsto \mathsf{EAX} - 0]), \{(Retptr, [], [])\})$$

The state describes each flag as arising from the expression EAX − 0 because comparison on x86 processors is modeled as performing a subtraction only for its side effects of setting flags. If after this we have a JCC Z, 123 instruction, two successor states are produced. When the Z flag is true, we get the successor

$$((123, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}^?], [Z \mapsto \mathsf{EAX} - 0, C \mapsto \mathsf{EAX} - 0]), \{(Retptr, [], [])\})$$

When Z is false, we get

$$((89, [\mathsf{EAX} \mapsto \mathsf{int\ ptr}], [Z \mapsto \mathsf{EAX} - 0, C \mapsto \mathsf{EAX} - 0]), \{(Retptr, [], [])\})$$

While we describe the choice here as though the program verifier determines which branch is taken, it actually conservatively queues both successors for exploration.

How were these successors generated? The *considerTest* function provided by our running example must recognize the special case where $co = (\mathsf{Z}, \mathsf{true})$, $\circ = -$, e_1 is some register r, $e_2 = 0$, $b = \mathsf{false}$, and $\vec{r}(r) = \tau\ \mathsf{ptr}^?$ for some τ. In other words, we have verified that some register of type $\tau\ \mathsf{ptr}^?$ is nonzero. In such a case, we promote r to type $\tau\ \mathsf{ptr}$ in the returned $\vec{r'}$.

If the instruction at location 123 is MOV EAX, 3, which stores the constant 3 in register EAX, then the state that will result is

$$((123, [\mathsf{EAX} \mapsto \mathsf{int}], []), \{(Retptr, [], [])\})$$

The custom code for our example is not involved in this transition. Instead, SimpleFlags sees that the value of a register mentioned in all of the known flag states has changed, and so it erases all of that flag information.

After reading these last two subsections, the reader may be wondering why StackTypes and SimpleFlags received this relative order. Indeed, neither depends on the other, and this is the one case in the component pipeline where a reordering would have been acceptable.

3.7 WeakUpdate

We have 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 does not 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, we 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.

We will now present the signature of a WeakUpdate type system piece by piece. In contrast to the signatures given for the previous components, this signature does not extend its predecessors. With one small exception that we will describe below, the elements that we will list are all that a type system designer needs to provide to produce a working verifier with a proof of soundness. It is also true that, while we have 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 $\leqslant_T$.

$$\begin{array}{rcl} ty & : & \mathsf{Set} \\ context & = & \mathsf{word} \to \mathsf{option}\ ty \\ \vdash_T & : & context \to \mathsf{word} \to ty \to \mathsf{Prop} \\ \leqslant_T & : & \Pi(\tau_1 : ty)(\tau_2 : ty).\ [\![\forall \Gamma, w, \\ & & \quad \Gamma \vdash_T w : \tau_1 \Rightarrow \Gamma \vdash_T w : \tau_2]\!] \end{array}$$

An important difference is that here we hard-code contexts to be partial maps from memory addresses to types. Other standard context elements from previous examples, such as the saved return pointer, will now be handled internally by the WeakUpdate functor. As WeakUpdate is only intended as one simple example of a terminal verification component, it does not support polymorphism in the style found in some earlier examples.

A few simple procedures suffice to plug into a generic type-checker for machine code:

$$typeofConst \quad : \quad \Pi(w : \mathsf{word}).\{\tau : ty \mid \forall \Gamma, \Gamma \vdash_T w : \tau\}$$

$$\begin{array}{rcl} typeofArith & : & \Pi(\circ : \mathsf{binop})(\tau_1\ \tau_2 : ty).\{\tau : ty \\ & & \quad \mid \forall \Gamma, w_1, w_2, \Gamma \vdash_T w_1 : \tau_1 \\ & & \qquad \Rightarrow \Gamma \vdash_T w_2 : \tau_2 \\ & & \qquad \Rightarrow \Gamma \vdash_T w_1 \circ w_2 : \tau\} \end{array}$$

$$\begin{array}{rcl} typeofCell & : & \Pi(\tau : ty).\{\!\{\tau' : ty \mid \forall \Gamma, w, \\ & & \quad \Gamma \vdash_T w : \tau \\ & & \quad \Rightarrow \Gamma(w) = \mathsf{Some}\ \tau'\}\!\} \end{array}$$

The *typeofConst* function 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 nonpointer types.

The final element is a way of taking advantage of knowledge of conditional jump results, based behind the scenes on SimpleFlags:

$$\begin{aligned} \mathit{considerNeq} \quad : \quad & \Pi(\tau : \mathit{ty})(w : \mathsf{word}).\{\tau' : \mathit{ty} \mid \forall \Gamma, w', \\ & \Gamma \vdash_T w' : \tau \\ & \Rightarrow\ w' \neq w \Rightarrow \Gamma \vdash_T w' : \tau'\} \end{aligned}$$

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 this signature description 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 nontrivial 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 cannot 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, we illustrate this with a simple use of it to construct a type system handling some standard types for describing linked, heap-allocated structures.

First, we will give a quick overview of how our running example can be expressed succinctly as a WeakUpdate type system, ignoring proof components as in previous sections. We add a singleton integer type to our type system, for reasons that should become clear shortly.

$$\mathsf{type} \quad \tau \quad ::= \quad \mathsf{int} \mid \tau\ \mathsf{ptr} \mid \tau\ \mathsf{ptr}^? \mid \underline{\mathsf{S}(n)}$$

We use these typing rules:

$$\frac{}{\Gamma \vdash_T w : \mathsf{int}} \qquad \frac{}{\Gamma \vdash_T w : \mathsf{S}(w)}$$

$$\frac{\Gamma(w) = \tau \quad w \notin \mathit{code} \cup \mathit{stack}}{\Gamma \vdash_T w : \tau\ \mathsf{ptr}} \qquad \frac{}{\Gamma \vdash_T 0 : \tau\ \mathsf{ptr}^?} \qquad \frac{\Gamma(w) = \tau \quad w \notin \mathit{code} \cup \mathit{stack}}{\Gamma \vdash_T w : \tau\ \mathsf{ptr}^?}$$

We can use this simple subtyping relation:

$$\frac{}{\tau \leqslant_T \mathsf{int}} \qquad \frac{}{\tau \leqslant_T \tau} \qquad \frac{}{\mathsf{S}(0) \leqslant_T \tau\ \mathsf{ptr}^?} \qquad \frac{}{\tau\ \mathsf{ptr} \leqslant_T \tau\ \mathsf{ptr}^?}$$

The pieces we need to plug into the extensible type-checker are

$$\begin{aligned} \mathit{typeofConst}(w) &= \mathsf{S}(w) \\ \mathit{typeofArith}(\circ, \tau_1, \tau_2) &= \mathsf{int} \\ \mathit{typeofCell}(\tau) &= \begin{cases} \tau', & \tau = \tau'\ \mathsf{ptr} \\ \bot, & \text{otherwise} \end{cases} \\ \mathit{considerNeq}(\tau, w) &= \begin{cases} \tau'\ \mathsf{ptr}, & \tau = \tau'\ \mathsf{ptr}^? \text{ and } w = 0 \\ \tau, & \text{otherwise} \end{cases} \end{aligned}$$

...and that is about the entirety of a formal description of this verifier. We have omitted proofs in this summary, but an actual implementation of this verifier (included as an example with the software distribution) is only about 150 lines of Coq code, most of it concerned with filling in trivial proofs for default implementations of the various operations. The end result is a certified verifier that runs on real x86 binaries.

3.8 Architecture summary

We review the overall pipeline by summarizing it again, this time in top-down order:

1. **WeakUpdate**: High-level type system description
2. **SimpleFlags**: Add instrumentation tracking correlations between conditional flags and register values.
3. **StackTypes**: Add standard types modeling stack and calling conventions.
4. **TypeSystem**: Add tracking of register types by plugging user-specified handlers into a generic type checker.
5. **FixedCode**: Add enforcement of immutability of the program code area in memory.
6. **Reduction**: Convert from a verifier for a simple RISC language to a verifier for x86 assembly language.
7. **ModelCheck**: Implement the actual state-space traversal.

The input to any stage involves both algorithmic pieces and proof pieces. A client of the library need not concern himself with the interfaces of components that come later in this list than the one he chooses to use as a starting point. Each component assembles new executable verifier code and new soundness proofs from those passed to it as input, with the composition of the components in any suffix of this list spanning the entire corresponding abstraction gap.

4 Case study: A verifier for algebraic datatypes

In this section, we will present some highlights of the Coq implementation of the MemoryTypes verifier, based on the library components from the last section. Recall that this verifier is designed to handle programs that use algebraic datatypes. For clarity, we have made some simplifications from the real Coq code, especially regarding dependent pattern matching.

MemoryTypes is implemented simply using WeakUpdate, the last component described in the last section. All we need to do is provide an implementation of

WeakUpdate's input signature, which we described in Section 3.7. The first step is to define the language of types, which includes the standard low-level types used to implement algebraic datatypes.

```
Definition var := nat,

Inductive ty : Set :=
  | Constant : int32 -> ty
  | Product : list ty -> ty
  | Sum : ty -> ty -> ty
  | Var : var -> ty
  | Recursive : var -> ty -> ty.
```

The set `ty` of types includes the usual elements; 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.

Next we must define the typing relation:

```
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 nil)
  | HT_Product : forall ctx v t ts,
    ctx v = Some t
    -> hasTy ctx (v + 4) (Product ts)
    -> hasTy ctx v (Product (t :: ts))
  | HT_Suml : forall ctx v t1 t2,
    hasTy ctx v (Product (Constant 0 :: t1 :: nil))
    -> hasTy ctx v (Sum t1 t2)
  | HT_Sumr : forall ctx v t1 t2,
    hasTy ctx v (Product (Constant 1 :: t2 :: nil))
    -> 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).
```

The typing relation `hasTy` is defined in terms of its inference rules in the standard way. 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 nonempty 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 `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.

Next we define the subtyping procedure:

```
Definition subTy : forall (t1 t2 : ty),
  [[forall ctx v, hasTy ctx v t1 -> hasTy ctx v t2]].
  intros t1 t2.
  refine (subTy' t1 t2
    || subTy' (tryUnrollingOnce t1) t2
    || subTy' t1 (tryUnrollingOnce t2));....
Qed.
```

The top-level procedure uses a subroutine `subTy'` to do most of the work. `subTy'` has no special handling of recursive types. Instead, `subTy` tries a heuristic set of possible unrollings of recursive types, calling `subTy'` on each result and concluding that the subtyping relation holds if any of these calls succeeds. It is the responsibility of a certifying compiler targeting this verifier to emit enough typing annotations that no more sophisticated subtyping relation is required, effectively splitting a multi-unrolling check into several simpler checks.

The definition of `subTy'` proceeds in the standard way, defining a recursive function with holes left to be filled in with tactic-based proof search:

```
Definition subTy' : forall (t1 t2 : ty),
  [[forall ctx v, hasTy ctx v t1 -> hasTy ctx v t2]].
  refine (fix subTy' (t1 t2 : ty) {struct t2}
    : [[forall ctx v,
      hasTy ctx v t1 -> hasTy ctx v t2]] :=
    match (t1, t2) with
      |...
    end);....
Qed.
```

The pattern matching cases have the expected implementations. For one example, consider the case for subtyping between constant types:

```
      | (Constant n1, Constant n2) =>
        pfEq <- int32_eq n1 n2;
        Yes
```

Constant types are only compatible if their constants are equal. We use a richly-typed integer comparison procedure `int32_eq` with our monadic notation. The proof `pfEq` that results from a successful equality test will be used to discharge the proof obligation arising for the correctness of this case.

Another example case is that for comparing a product type of the proper form to a sum type:

```
      | (Product (Constant n :: t :: nil), Sum t1 t2) =>
        (int32_eq n 0 && ty_eq t t1)
        || (int32_eq n 1 && ty_eq t t2)
```

A product type is only compatible with a sum type if the product starts with a constant tag identifying a branch of that sum and the next field of the product

draws its type from the same sum branch. Here, the standard boolean operators are custom "macro" syntax defined to do proper threading of known facts through the expression to the points triggering proof obligations.

Omitted from this discussion 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 expected, 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. We were 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. Thanks to the support described below in Section 5.2, the verifier uses the native types of the host processor in place of their first-principles recursively-defined counterparts in Coq. Thus, the run-time behavior and performance of the result are comparable to those of idiomatic OCaml programs, though the idioms used by the generated OCaml code can be bizarre in places and involve unchecked casts. We do not have meaningful performance benchmark results to share, since coming up with example inputs was time consuming without the help of a certifying compiler. However, the case study verifier works in roughly the same way as the Coq-based verifier whose performance we measured in past work (Chang *et al.* 2006), with the additional benefit of using native processor types where appropriate, so it seems reasonable to assume that a production-quality version could hold its own against uncertified analysis tools that use similar algorithms.

5 Implementation

The source code and documentation for the system described in this paper can be obtained from*

`http://proofos.sourceforge.net/`

The implementation is broken up into a number of separately-usable components: a library of Coq and OCaml code dealing with semantics and parsing of machine code in general and x86 machine code in particular; a Coq extension in support of extracting programs that use native integer types; and the certified verifiers library proper, including trusted code formalizing the problem setting and the collection of untrusted verification components highlighted in Section 3.

5.1 The Asm library

When we began this project, we set out to survey the different choices of pre-packaged libraries formalizing useful subsets of x86 assembly language. We were

* Also available at `http://journals.cambridge.org/issue_Journaloffunctionalprogramming/Vol18No5-6`.

quite surprised to find that, not only was there no such package available for our chosen proof assistant, but the situation seemed to be the same for all other choices of tools, as well! Since this kind of formalization is key to the project, we created our own, designing it as a library useful in its own right, called Asm.

Some of our choices veer away from formality and small trusted code bases in favor of practicality. The library deals with a small subset of x86 machine language that covers the instructions generated by GCC for the examples that we have tried. It defines in both Coq and OCaml a language of abstract syntax trees for machine code programs. There is a Coq operational semantics for these ASTs, along with an OCaml parser from real x86 binaries to ASTs. The extracted Coq verifiers use the OCaml parser as a "foreign function," sacrificing some "free" formal guarantees. For instance, the parser uses imperative code for file IO to initialize a global variable with program data. Coq's formal semantics contains no account of such phenomena, but we believe that no unsoundness is introduced because all OCaml "foreign functions" are observationally pure over single executions. In any case, some worthwhile future work would be to move more of the process into genuine Coq code.

The Asm library contains a few other related pieces that we were not able to find elsewhere, including formalizations of different fixed-width bitvector arithmetic operations and their properties. Altogether, the Coq code size tallies for Asm run to about 10,000 lines in a generic utility library, 1,000 to formalize bitvectors and fixed-precision arithmetic, and 1,000 to formalize a subset of x86 machine code.

5.2 Proof Accelerator

Different data structures make sense for rigorous formalization than for efficient execution. For instance, Coq's standard library defines natural numbers with the standard Peano-style recursive type, effectively representing them in unary. This provides a ready and effective induction principle, and issues of representation efficiency do not come into play when we are proving generic properties of the naturals and not examining particular individuals. However, when we extract to OCaml programs that use natural numbers and run them on particular inputs, we find ourselves in the unusual situation that basic arithmetic operations on natural numbers are a primary performance bottleneck. We want to *reason about* mathematical natural numbers but *represent them at run-time* in a way that takes advantage of built-in processor capabilities. Similar concerns apply for the fixed-width bitvector types that abound in any project working with machine language, though there we only hope for constant-factor performance improvements by using native types in our verifiers.

We have developed a Coq extension that we call "Proof Accelerator" which achieves this by modifying the program extraction process in a very simple way. Custom type mappings are already supported by Coq, so we request that `nat` be mapped to the OCaml infinite-precision integer type (since the potential for silent overflow would invalidate formal proofs), the Coq type of 32-bit words to the native 32-bit word type, etc. Using the same mechanism, we can identify some common

functions (e.g., addition) that should be extracted to use their standard OCaml implementations.

This leaves one more important class of modifications that Coq does not support out of the box. In particular, we need to rewrite uses of *pattern matching*. We have replaced inductive types with what are effectively abstract types, like native integer types. Proof Accelerator rewrites pattern matches using `when` guards and local bindings. The results should have the same semantics as the originals.

The whole approach and implementation are quite informal. The practical effect of this implementation choice is that all of the Proof Accelerator must be counted among the trusted code base, along with the Coq proof checker and extractor and the OCaml compiler. An interesting future project would be to look at a more general and rigorous system of proof-preserving transformations that replace one implementation of an abstract data type with another.

5.3 Certified Verifiers library

We have already gone into detail in the previous sections on the content of the certified verifiers library. The different components combine to take up about 7,000 lines of Coq code, highlighting the effectiveness of reuse, as only 700 lines were needed for the last section's case study, with about 150 of them either reasonable candidates for relocation to the general utility library or unavoidable boilerplate that would not grow with verifier complexity.

Most of the resulting implementation is Coq code which is extracted to ML. For simplicity, we chose to implement in OCaml some pieces that must inevitably belong to the trusted code base, along the lines of the OCaml instruction decoding in the Asm library. 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. (This, of course, is modulo the lack of formality that we accept when interfacing Coq and OCaml code.)

We can summarize the big picture of what these implementation pieces give us. A 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 we 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 Asm library and brief definitions of abstract machines and safety from the certified verifiers library.

6 Related work

The verifiers produced in this project are used in the setting of proof-carrying code. Relative to our past work (Chang *et al.* 2006) on certified verifiers, our new

contribution here is first, to suggest developing verifiers with Coq in the first place, instead of extracting verification conditions about the more traditional programs; and second, to report on experience in the effective construction of such verifiers through the use of dependent types and reusable components. Several projects (Appel 2001; Hamid *et al.* 2003; Crary 2003) 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.* (2003) 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. We showed in our initial work on certified verifiers (Chang *et al.* 2006) that we can achieve verification times an order of magnitude better than both those of Wu *et al.* and those from our past work on the Open Verifier (Chang *et al.* 2005). At the same time, the prototype verifier that we used for these measurements stayed within a factor of 2 of the running time of the traditional, uncertified Typed Assembly Language checker (Morrisett *et al.* 1999b).

The architecture that we have presented is only a first step towards a general and practical system. Many Foundational PCC projects have considered in a proof theoretical setting ideas that could profitably be used in concert with certified verifiers. For instance, the component library that we have described only supports whole-program verification. In contrast, the progression of verification frameworks that culminates in OCAP (Feng *et al.* 2007) provides a rich setting for modular verification of systems whose pieces are certified using different program logics. It seems natural to consider the adaptation of this idea to cooperative verification using different certified verifiers. The technology of certified verifiers also has a good way to go to "catch up" with the FPCC world by common metrics like sophistication of language features verified and minimality of the trusted code base.

Past projects have considered using proof assistants to develop executable abstract interpreters (Cachera *et al.* 2005; Besson *et al.* 2006) and Java bytecode verifiers (Klein & Nipkow 2001; Bertot 2001). Our work differs in dealing with machine code, which justifies the kind of layered component approach that we have described, and our 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. Our work has much in common with the CompCert project (Leroy 2006b), which works towards a fully certified C compiler developed in Coq. The main differences are our use of dependent types to structure the "program" part of a development and our emphasis on reusable library components.

The Rhodium project (Lerner *et al.* 2005) also deals with the construction of certified program analyses. By requiring that analyses be stated in a very limited Prolog-like language, Rhodium makes it possible to use an automated first-order theorem prover to discharge all proof obligations. This approach is very effective for the traditional compiler optimizations that the authors target, but it does not

seem to scale to the kinds of analyses associated with FPCC. For instance, proofs about type systems like the one demonstrated here are subtle enough that human control of the proving process seems necessary, justifying our choice of a much richer analysis development environment.

We have already mentioned the Epigram (McBride & McKinna 2004), ATS (Chen & Xi 2005), and RSP (Westbrook *et al.* 2005) languages that attempt to inject elements of the approach behind Coq program extraction into a more practical programming setting. We believe that we 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 is also true that much of the specifics of our 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.

7 Conclusion

There has been much interest lately in enriching the expressiveness of static type systems to capture higher-level properties. Based on the results we have reported here, we hope we have 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 nontoy problems. While recent proposals in this space focus on integrating proofs and dependent types with imperativity and other "impure" language features, we were 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 we 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

Appel, Andrew W. (2001) Foundational proof-carrying code. In *LICS '01: Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*. Boston, MA, USA, pp. 247–256.

Bertot, Yves. (2001) Formalizing a JVML verifier for initialization in a theorem prover. In *CAV '01: Proceedings of the 13th International Conference on Computer Aided Verification*. Paris, France, pp. 14–24.

Bertot, Yves & Castéran, Pierre. (2004) *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, Berlin.

Besson, Frédéric, Jensen, Thomas & Pichardie, David. (2006) Proof-carrying code from certified abstract interpretation and fixpoint compression. *Theor. Comput. Sci.* **364**(3), 273–291.

Chang, Bor-Yuh Evan, Chlipala, Adam, Necula, George C. & Schneck, Robert R. (January 2005) The open verifier framework for foundational verifiers. In *TLDI'05: Proceedings of the 2nd ACM Workshop on Types in Language Design and Implementation*, Long Beach, CA, USA.

Cachera, David, Jensen, Thomas, Pichardie, David & Rusu, Vlad. (2005) Extracting a data flow analyser in constructive logic. *Theor. Comput. Sci.*, **342**(1), 56–78.

Chang, Bor-Yuh Evan, Chlipala, Adam & Necula, George C. (2006) A framework for certified program analysis and its applications to mobile-code safety. In *VMCAI '06: Proceedings of the Seventh International Conference on Verification, Model Checking, and Abstract Interpretation*. Charleston, South Carolina, USA, pp. 174–189.

Chen, Chiyan & Xi, Hongwei. (2005) Combining programming with theorem proving. In *ICFP '05: Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*. Tallinn, Estonia, pp. 66–77.

Chlipala, Adam. (2006) Modular development of certified program verifiers with a proof assistant. In *ICFP '06: Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*. Portland, OR, USA, pp. 160–171.

Cousot, Patrick & Cousot, Radhia. (1977) Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'77: Proceedings of the 4th ACM Symposium on Principles of Programming Languages*. Los Angeles, CA, USA, pp. 234–252.

Crary, Karl. (2003) Toward a foundational typed assembly language. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New Orleans, LA, USA, pp. 198–212.

Delahaye, David. (2000) A tactic language for the system Coq. In *LPAR '00: Proceedings of the 7th International Conference on Logic for Programming and Automated Reasoning*. Reunion Island, France, pp. 85–95.

Detlefs, David L., Leino, K. Rustan M., Nelson, Greg & Saxe, James B. (1998) Extended static checking. In *SRC Research Report 159*. Compaq Systems Research Center, Palo Alto.

Dijkstra, Edsger W. (1976) *A Discipline of Programming*. Englewood Cliffs, NJ: Prentice Hall PTR.

Feng, Xinyu, Ni, Zhaozhong, Shao, Zhong & Guo, Yu. (January 2007) An open framework for foundational proof-carrying code. In *TLDI'07: Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Language Design and Implementation*. Nice, France, pp. 67–78.

Filliatre, Jean-Christophe & Letouzey, Pierre. (2004) Functors for proofs and programs. In *ESOP '04: Proceedings of the 13th European Symposium on Programming*. Barcelona, Spain, pp. 370–384.

Giménez, Eduardo. (1995) Codifying guarded definitions with recursive schemes. In *TYPES '94: Selected Papers from the International Workshop on Types for Proofs and Programs*, Båstad, Sweden, pp. 39–59.

Hamid, Nadeem A., Shao, Zhong, Trifonov, Valery, Monnier, Stefan & Ni, Zhaozhong. (2003) A syntactic approach to foundational proof-carrying code. *J. Automated Reasoning* **31**(3–4), 191–229.

Intel. (2006) *IA-32* Intel Architecture Software Developer's Manual, Vol. 2: Instruction set reference.

Klein, Gerwin & Nipkow, Tobias. (2001) Verified lightweight bytecode verification. *Concurrency – Practice and Experience* **13**(1), 1133–1151.

Lerner, Sorin, Millstein, Todd, Rice, Erika & Chambers, Craig. (2005) Automated soundness proofs for dataflow analyses and transformations via local rules. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Long Beach, CA, USA, pp. 364–377.

Leroy, Xavier. (2006a) Coinductive big-step operational semantics. In *ESOP '06: Proceedings of the 15th European Symposium on Programming*. Vienna, Austria, pp. 54–68.

Leroy, Xavier. (2006b) Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Charleston, SC, USA, pp. 42–54.

MacQueen, David. (1984) Modules for standard ML. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*. Austin, TX, USA, pp. 198–207.

Martin-Löf, Per. (1996) On the meanings of the logical constants and the justifications of the logical laws. *Nordic J. Philos. Logic* **1**(1), 11–60.

McBride, Conor & McKinna, James. (2004) The view from the left. *J. Funct. Programming* **14**(1), 69–111.

Morrisett, Greg, Crary, Karl, Glew, Neal, Grossman, Dan, Samuels, Richard, Smith, Frederick, Walker, David, Weirich, Stephanie & Zdancewic, Steve. (1999b) TALx86: A realistic typed assembly language. In *Proceedings of the 1999 ACM SIGPLAN Workshop on Compiler Support for System Software*. Atlanta, GA, USA, pp. 25–35.

Morrisett, Greg, Crary, Karl, Glew, Neal & Walker, David. (2003) Stack-based typed assembly language. *J. Funct. Programming* **13**(5), 957–959.

Morrisett, Greg, Walker, David, Crary, Karl & Glew, Neal. (1999a) From system F to typed assembly language. *ACM Trans. Programming Languages Syst.* **21**(3), 527–568.

Necula, George C. (1997) Proof-carrying code. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Paris, France, pp. 106–119.

Necula, George C. & Lee, Peter. (1998) The design and implementation of a certifying compiler. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*. Montreal, Quebec, Canada, pp. 333–344.

Sheard, Tim. (2004) Languages of the future. In *OOPSLA '04: Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*. Vancouver, BC, Canada, pp. 116–119.

Smith, Frederick, Walker, David & Morrisett, J. Gregory. (2000) Alias types. In *ESOP '00: Proceedings of the 9th European Symposium on Programming*. Berlin, Germany, pp. 366–381.

Wadler, Philip. (1992) The essence of functional programming. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Albuquerque, NM, USA, pp. 1–14.

Westbrook, Edwin, Stump, Aaron & Wehrman, Ian. (2005) A language-based approach to functionally correct imperative programming. In *ICFP '05: Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*. Tallinn, Estonia, pp. 268–279.

Wu, Dinghao, Appel, Andrew W. & Stump, Aaron. (2003) Foundational proof checkers with small witnesses. In *PPDP '03: Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*. Uppsala, Sweden, pp. 264–274.