Foundational Certified Code in a Metalogical Framework

KARL CRARY and SUSMIT SARKAR
Carnegie Mellon University

Foundational certified code systems seek to prove untrusted programs to be safe relative to safety policies given in terms of actual machine architectures, thereby improving the systems' flexibility and extensibility. Previous efforts have employed a structure wherein the proofs are expressed in the same logic used to express the safety policy. We propose an alternative structure wherein safety proofs are expressed in the Twelf metalogic, thereby eliminating from those proofs an extra layer of encoding needed in the previous accounts. Using this metalogical approach, we have constructed a complete, foundational account of safety for a fully expressive typed assembly language.

Categories and Subject Descriptors: F.3.1 [**Specifying and Verifying programs**]: Mechanical Verification; F.4.1 [**Mathematical Logic**]: Logic programming

General Terms: Languages, Verification

Additional Key Words and Phrases: foundational certified code, meta–logic, logic programming

1. INTRODUCTION

The proliferation of low-cost computing hardware and the ubiquity of the Internet has created a situation where a huge amount of computing power is both idle and—in principle—accessible to developers. The goal of exploiting these idle computational resources has existed for years, and, beginning with SETI@Home [2000] in 1997, a handful of projects have successfully made profitable use of idle computation on the Internet. More recently, this paradigm, now called *grid computing,* has elicited serious interest among academics [Buyya and Baker 2000; Lee 2001; Parashar 2002] and in industry as a general means of conducting low-cost supercomputing.

Despite the increasing interest in grid computing, a remaining obstacle to its growth is the (understandable) reluctance of computer owners to download and execute software from developers they do not know or trust, and may not have even heard of. This has limited the practical use of grid computing to the small number of potential users that have been able to obtain the trust of thousands of computer owners they do not know.

The ConCert project at CMU [Chang et al. 2002] is seeking to overcome this obstacle by developing a system for trustless dissemination of software. In the ConCert framework, a machine owner installs a "steward" program that ensures the safety of any downloaded software. When a new grid application is obtained for execution (other parts of the ConCert framework determine when and how this takes place), that application is expressed in the form of *certified code,* in which

This material is based on work supported in part by NSF grants CCR-9984812 and CCR-0121633. Any opinions, findings, and conclusions or recommendations in this publication are those of the authors and do not reflect the views of this agency.

the executable code is accompanied by a certificate proving that the code is safe. The steward then verifies that the certificate is valid before permitting the code to be executed. The mechanism of certified code moves the burden of determining whether the code is safe from the code consumer to the code supplier.

An important aspect of a certified code system is the makeup of the system's *trusted elements,* those elements that must be correct for the system to work properly, such as a verifier (*e.g.,* a type checker) or runtime library. One form of trusted element common to the early certified code systems [Lindholm and Yellin 1996; Necula and Lee 1996; Morrisett et al. 1999] was a trusted type system. These systems relied on a type system (or similar artifact) to ensure safety, and code recipients were required to trust that the type system sufficed for that purpose. Although in some cases the type system was backed up by a published proof of safety [Necula 1997; Morrisett et al. 1999; Morrisett et al. 2002; Necula 1998], each such proof was carried out at an abstract level some distance from any real machine architecture.

More recently, there has been interest in the development of certified code systems that do not include a type system among the systems' trusted elements,[1] such as the accounts of Appel and Felty [2000], and Hamid *et al.* [2002]. Such systems, dubbed "foundational," constitute an improvement because they remove a substantial trusted element.

Removing trusted elements is desirable for two reasons: First, a trusted element could be faulty, so eliminating it improves security. This is particularly compelling for software elements; for example, Bernard and Lee [2002] eliminate the verification condition generator from the trusted elements of the SpecialJ PCC system [Colby et al. 2000], and Appel *et al.* [2002] seek to minimize the number of lines of code in an LF proof checker [Harper et al. 1993]. This is valuable, but even complex elements can become trustworthy if given time to mature. Moreover, there will always (at least for the foreseeable future) remain substantial artifacts among the trusted elements. Second, and more important, a trusted element is one that every participant is stuck with, so eliminating it improves flexibility and extensibility.

The second issue is particularly relevant for trusted type systems. All type systems in use for certified code impose limitations on the programs they will pass. Our aim is to enable the establishment of a decentralized grid computing fabric available for a wide array of participants. For this purpose, it is implausible to imagine that a single type system could ever suffice for all potential grid applications. Therefore, it is necessary that the basic steward can be extended with new type systems from untrusted sources, and this is not possible when a single type system is hardwired into the steward. Consequently, a foundational certified code system is essential to our purposes.

This paper presents the foundational certified code system we have developed as part of ConCert. At a high-level, our system shares the same structures as other foundational efforts: The trusted elements include a proof checker and a safety policy that incorporates a formalization of the machine semantics. The untrusted elements (*i.e.,* the argument that a particular program is safe) include a

[1]In such systems, the safety policy is not defined as a type system, although a type system may be used as a device in a *proof* of adherence to the policy.

type system that accepts the given program, and a proof that all programs passing the type system satisfy the safety policy. In each case, the main development is in the untrusted elements, particularly the safety proof.

A closer analysis reveals important methodological differences between the foundational efforts. Appel and Felty [2000] is essentially denotational, with a semantic model of types given not in domain or category theory, but rather in terms of concrete machine instructions and the safety policy. In contrast, Hamid, *et al.* [2002] is operational, with safety given by conventional type preservation and progress lemmas [Wright and Felleisen 1994; Harper 1994].

However, both systems are similar in that (1) the safety policy and the entire compliance proof are given in a single logic (for Appel and Felty, higher-order logic encoded in LF [Harper et al. 1993], and for Hamid *et al.*, the calculus of inductive constructions [Paulin-Mohring 1993]), and in that (2) the compliance proof centrally involves at least one intermediate formal system (such as a type system) that is shown to imply the safety policy. In each case, since the entire proof is conducted in a single logic, the intermediate formal system must be *encoded* in that logic, and any reasoning about the formal system deals with encodings of the intermediate derivations, not the intermediate derivations directly.

1.1 Our approach.

Our system takes a different logical perspective, one based on *metalogic:*

(1) For our safety policy, which we call TSP, we define a logic that expresses the operational semantics of the architecture, but is limited to safe operations only. (Consequently, a program performing an unsafe operation would become stuck.) This is the main body of the safety policy. Our work is specialized to the Intel IA-32 architecture[2] [Intel Corporation 2001], but it should be adaptable to other architectures.

(2) The code supplier is invited to provide a second logic that defines an untrusted *safety checker* when interpreted as a logic program (Section 2.2). The safety checker could identify a single program, but in practice is more likely to implement a type system accepting many programs.

(3) The final component of the safety policy is the statement of a metatheorem: for any program P that satisfies the supplier's safety checker, if a machine loaded with P eventually runs to a state S, then S transitions to some state S'. It follows from the metatheorem that P cannot become stuck, and thus—by construction of the operational semantics—it follows further that that P never performs an unsafe operation.

(4) The code supplier then has the responsibility to fill in the proof of the safety metatheorem, which is then itself verified by a meta-proof checker.

(5) Finally, the code consumer executes the safety checker, giving it as arguments (a) the program in question, and (b) some "advice" that is packaged with the program. The program is deemed to pass if the safety checker finishes without an error. In that event, it follows from the metatheorem that the given program is safe.

[2]Popularly known as the "x86" architecture.

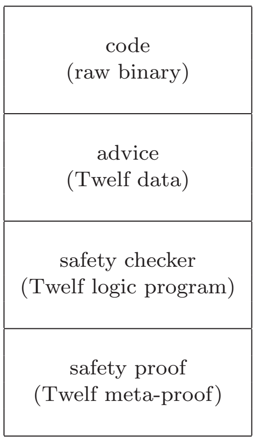

Fig. 1. A Certified Binary

Figure 1 illustrates schematically the components of a certified binary in our system. The top three components are passed to a logic program interpreter, and the bottom two are passed to a meta-proof checker. If both succeed, the program is accepted and the code may be safely executed. As a practical matter, the same checker will often be used for many different programs, so it would be profitable in practice to check the meta-proof once and then record that the safety checker is known to be sound.

(Note that there are two "checkers" involved in the system: One is the *trusted* meta-proof checker that is used to verify metatheorems, and the other is the *untrusted* safety checker that comes from the code supplier.)

At a high-level, this structure is unsurprising; our safety policy is precisely the now-standard statement of type safety, given at the architecture level. The difference from other foundational efforts (at this high level) is that the code supplier's proof is provided in a metalogic rather than in the same logic used to define the safety policy.

The importance of this difference is a pragmatic one. Using the Twelf metalogical framework [Pfenning and Schürmann 1999; 2002], one may conveniently work directly with derivations in a logic, avoiding the extra layer of encoding needed in previous accounts of foundational certified code. As a result, we were able to develop our system, including the foundational safety proof for an expressive typed assembly language called TALT ("TAL Two") [Crary 2003], in less than two man-years, a fraction of the time invested in other foundational efforts.

Of course, it may be argued that our system, like previous systems, is still built around a single logic, in our case the Twelf metalogic. Conversely, it may be argued that the previous systems are also built on metalogics, in that their central logics are used as metalogics (*i.e.,* the are used to encode formal systems). Nevertheless, our system is distinguished in that the Twelf metalogic is *designed* for the very purpose of encoding and manipulating other logics. It is this design, wherein logical derivations may be manipulated conveniently and directly, that gives our approach

$$n \ ::= \ 0 \mid s(n)$$

$$\frac{}{0 + n = n} \qquad \frac{n_1 + n_2 = n_3}{s(n_1) + n_2 = s(n_3)}$$

Fig. 2. Natural Numbers

its pragmatic advantage.

The remainder of the paper is structured as follows: In Section 2 we discuss how logics, metatheorems, and meta-proofs are expressed in Twelf. In Section 3 we present TSP, our system's architecture semantics and safety policy, the elements of a certified code system imposed by the code consumer. In Section 4 we discuss a safety checker based on the typed assembly language TALT [Crary 2003], and in Section 5 we outline our meta-proof that our TALT-based checker satisfies TSP. Concluding remarks appear in Section 6.

2. THE TWELF METALOGIC

We begin with a brief tutorial on the use of Twelf to express logics and meta-reasoning. The ideas underlying this methodology originated with Pfenning [Pfenning 1991] and were developed further in a variety of papers on Twelf and its predecessor Elf [Pfenning and Rohwedder 1992; Rohwedder and Pfenning 1996; Pientka and Pfenning 2000; Schürmann 2000; Pfenning and Schürmann 2002]. We assume familiarity with logic programming, and with the methodology of encoding logics in LF [Harper et al. 1993], wherein syntactic classes and judgements become types, and syntactic objects and derivations becomes terms.

2.1 A Simple Logic

As a running example, we will use a very simple logic of natural numbers. The sole syntax in our logic is that of natural numbers (expressed in unary), and the sole judgement states that the sum of two numbers is a particular third number. This logic is given in standard mathematical notation in Figure 2.

The syntax for the logic is expressed by the following Twelf definitions:

```
nat : type.
0 : nat.
s : nat -> nat.
```

The sum judgement is expressed by the Twelf definition:

```
sum : nat -> nat -> nat -> type.
```

Finally, the two inference rules for sum are expressed by:

```
sum_z : sum 0 N N.

sum_s : sum (s N1) N2 (s N3)
         <- sum N1 N2 N3.
```

Given these six declarations defining the logic, we can construct logical derivations by composing the constants that represent inference rules. For example, the judgement $1 + 2 = 3$ is written as the type:

```
sum (s 0) (s (s 0)) (s (s (s 0)))
```

and a derivation of that judgement (*i.e.,* a term of that type) is:

```
sum_s sum_z
```

2.1.0.1 *Implicit arguments.* Note Twelf's use of implicit arguments to express the rules `sum_z` and `sum_s` in a convenient form. In explicit form, the implicit argument `N` to `sum_z` (for example) would be explicit, resulting in the type:

```
{N:nat} sum 0 N N
```

(where `{N:nat}` is the Twelf rendering of the universal quantifier Π`N:nat`), and every use of `sum_z` would need explicitly to provide the argument `N`. In the form given, `N` is an implicit argument, and Twelf uses type reconstruction and unification to determine `N`'s value in every place where `sum_z` is used.

In explicit form, the above derivation of $1+2 = 3$ would be written less compactly as:

```
sum_s 0 (s (s 0)) (s (s 0)) (sum_z (s 0))
```

2.2 The Operational Interpretation

A Twelf signature is a collection of Twelf declarations. A Twelf signature can be interpreted in two ways. The interpretation we took in the previous section is that a signature defines one or more logics. Alternatively, we can adopt an operational interpretation in which a signature is viewed as a logic program.

In the operational interpretation, the user presents the system with a query in the form of a judgement (*i.e.,* a Twelf type) containing existentially quantified variables. Twelf then conducts depth-first proof search, attempting to find values for those variables such that the judgement is derivable (*i.e.,* the type is inhabited).

For example, given the query $1 + 2 = n$, Twelf replies that n can be 3:

```
?- sum (s z) (s (s z)) N.
Solving...
N = s (s (s z)).
```

In this case, of course, no further solutions are found.

As an aside, note that Twelf's interpretation of free variables in queries is different from those in declarations: in the query above, `N` is taken to be existentially quantified, but in the declaration of `sum_z`, `N` is taken to be an implicit argument, which is therefore universally quantified.

It may be convenient to view a Twelf logic program as a typed Prolog program. Provided one sets aside Twelf's higher-order features,[3] we may obtain a Prolog

[3]Unlike Prolog, but like λ-Prolog, Twelf permits nested implication. Operationally, this means that Twelf (and λ-Prolog) can introduce statically scoped variables that may participate in proof search. Twelf (as a logic programming language) corresponds closely to λ-Prolog with a richer type system, but without λ-Prolog's impure features.

program from a Twelf signature by extracting the inference rules, deleting their names, and rewriting them in Prolog syntax. For example, our sample signature corresponds to the Prolog program:

```
sum(0, N, N).
sum(s(N1), N2, s(N3)) :- sum(N1, N2, N3).
```

2.3 Meta-Proofs

An interesting fact about the `sum` predicate is that whenever it is given ground (fully specified) inputs in its first two positions, as in the query above, it always returns a ground result in its third position. (In fact, it always returns exactly one, but that is not important for our present purposes). Thus, `sum` is a *total relation* from its first two positions to its third.

The totality of `sum` means it can be interpreted as a metatheorem, asserting the existence of certain logical objects under certain conditions. In this case, the theorem proved is uninteresting; it states that for any two natural numbers n_1 and n_2, there exists a third natural number n_3.

However, we can write logic programs corresponding to more interesting metatheorems. For example:[4]

```
sum_ident : {N:nat} sum N 0 N -> type.

sum_ident_z : sum_ident 0 sum_z.
sum_ident_s : sum_ident (s N) (sum_s D)
                <- sum_ident N (D : sum N 0 N).

sum_inc : sum N1 N2 N3 -> sum N1 (s N2) (s N3) -> type.

sum_inc_z   : sum_inc sum_z sum_z.
sum_inc_s   : sum_inc
                (sum_s (D : sum N1 N2 N3)
                   : sum (s N1) N2 (s N3))
                (sum_s D'
                   : sum (s N1) (s N2) (s (s N3)))
                <- sum_inc D (D' : sum N1 (s N2) (s N3)).
```

Each of these is a total relation from its first position to its second and therefore corresponds to a metatheorem. The first, `sum_ident`, states that for any natural number n, there exists a derivation of $n + 0 = n$. The second, `sum_inc`, states that if there exists a derivation of $n_1 + n_2 = n_3$, then there exists a derivation of $n_1 + s(n_2) = s(n_3)$.

The names of logical inference rules are significant, since we will wish to work with them in the metalogic. However, provided we do not plan to prove any meta-meta-theorems (*i.e.,* theorems regarding the behavior of meta-theorems), we will never need to refer to the cases of a metatheorem, so the names we give to those

[4]To make the logic of the proof more clear, we have added a number of type annotations. These are not necessary for Twelf to check the proof, but are very useful for human readers.

cases are never significant except for Twelf's error reporting. Therefore we will save space in this paper by eliding them and replacing them all with the symbol `!`.

Using `sum_ident` and `sum_inc` as lemmas, we may prove a mildly interesting metatheorem:

```
sum_commute : sum N1 N2 N3
               -> sum N2 N1 N3 -> type.

! : sum_commute (sum_z : sum 0 N N) D
     <- sum_ident N (D : sum N 0 N).
! : sum_commute
     (sum_s (D : sum N1 N2 N3)
        : sum (s N1) N2 (s N3))
     D''
     <- sum_commute D (D' : sum N2 N1 N3)
     <- sum_inc D' (D'' : sum N2 (s N1) (s N3)).
```

Observe that `sum_commute` is total from its first position to its second.

2.4 Meta-proof Checking

So far we have been informal in our justification of totality, but of course, Twelf must be able to check whether a relation is total in order to determine if it is a valid metatheorem. Type checking ensures that no relation fails with a run-time type error, which leaves three ways a relation can fail to be total: mode failure, termination failure, and coverage failure. We illustrate each form of failure by an invalid proof of the false meta-proposition `sum_zero`:

```
sum_zero : {N:nat} sum N N 0 -> type.
```

—In mode failure, the relation returns a non-ground (*i.e.,* not fully specified) result, or passes a non-ground argument as an input to a lemma. For example, the non-proof:

```
! : sum_zero N (D : sum N N 0).
```

is well-typed, but does no actual work. Its return value is entirely unspecified, and hence is non-ground. The slightly more sophisticated version:

```
! : sum_zero N D'
     <- sum_commute (D : sum N N 0) D'.
```

is similar; it passes a non-ground value as an input to a lemma.

—In termination failure, the relation may loop forever. This corresponds to an invalid induction. For example:

```
!: sum_zero N D
    <- sum_zero N (D : sum N N 0).
```

—In coverage failure, not all cases of the theorem are covered. For example, the incomplete proof:

```
    ! : sum_zero 0 (sum_z : sum 0 0 0).
```

correctly proves the meta-proposition in the case when `N` is zero, but leaves out the nonzero case for which it fails.

Twelf checks the totality of relations with some assistance from the programmer. First, the programmer states a relation's inputs and outputs with a `%mode` declaration:

```
%mode sum_ident +N -D.
%mode sum_inc +D1 -D2.
%mode sum_commute +D1 -D2.
```

The `+` modes indicate inputs and the `-` modes indicate outputs (the variable names are insignificant). Since the mode declaration is essential to reading a relation as a metatheorem, henceforth we will always include the mode declaration with every metatheorem statement.

Second, the programmer directs Twelf to verify the relation to be total, and provides some information to help it do so (*e.g.,* the induction argument with which to show termination). We omit those declarations in the interest of brevity. For this paper, the reader may distinguish metatheorems from ordinary relations by the presence of a mode declaration.

3. THE SAFETY POLICY

The main body of our safety policy, TSP, is an operational semantics for the concrete architecture. For our work we have chosen to use the Intel IA-32 architecture, as it is the architecture used by the greatest number of potential grid participants. We have begun with a fairly small number of instructions, but new instructions are easy to add to the superstructure we have built. In the interest of brevity, the discussion here is largely generic in regard to the architecture; we do not discuss any of the issues peculiar to the IA-32.

Our operational semantics includes only safe operations; unsafe operations (for example, a store or jump to address `0x0`) are simply omitted from the semantics. This means that no transition exists out of any machine state in which an unsafe operation is about to be performed. In the usual parlance, such states are *stuck.*

A second, minor component of the safety policy is a definition of the possible initial states of the machine when loaded with a given program. With these two components, we can define the overall safety policy:

> A program P is *safe* if no stuck state is reachable from an initial state of P.

The code formalizing this is given in Section 3.6. It follows that in no state reachable by P is an unsafe operation *about to be* performed, and consequently no unsafe operation ever *is* performed.

3.1 Indeterminism

An important issue complicating the operational semantics is *indeterminism.*[5] In some cases it is infeasible to determine the outcome of an operation, either because the outcome is fundamentally unknowable (*e.g.,* an input operation), or because the operation's semantics is too complex for a complete specification to be practical (*e.g.,* garbage collection resulting from an allocation operation). In such cases, our semantics must assume that any possible transition could be taken, and require that all of them are safe. The formal safety of the machine cannot depend on the alternative the machine takes at runtime.

The most obvious approach to indeterminism is to make the semantics' transition relation non-functional, that is, to allow it to relate states to multiple following states. This simple approach does not work well for two reasons. First, for the purpose of developing safety proofs, it turns out to be *much* more convenient to work with a deterministic (functional) relation (Section 5.4).

Second, recall that any state that might perform an unsafe operation must be given no transitions at all; it is *not* sufficient simply to omit the unsafe transition, as then the state would not be stuck, and would therefore be deemed safe. To design the relation in this manner is subtle and error-prone, and leads to the likelihood of subtle errors in the safety policy.

Instead, we force the transition relation to be deterministic by adding an imaginary oracle to the state. When the outcome of an operation cannot be determined, the semantics simply consults the oracle. All possible outcomes are covered by this mechanism because the definition of initial states (Section 3.6) quantifies over all finite oracles, and any safety violation will happen within a finite amount of time.

3.2 Data

We use two notions of numbers in our semantics. One is the natural numbers from Section 2.1, which are used for various auxiliary purposes. The other is `binary N`, which contains N-bit binary numbers, the sort of number actually manipulated by the architecture. Given these, we can define our principal data types:

```
bw  : nat = 8.   %% byte width in bits
ww  : nat = 4.   %% word width in bytes
wwb : nat = 32.  %% word width in bits

%abbrev abyte = binary bw.   %%  "actual byte"
%abbrev aword = binary wwb.  %%  "actual word"
%abbrev address = aword.

string : nat -> type.
#      : string 0.
/      : abyte -> string N -> string (s N).
```

[5]We use the term "indeterminism" to emphasize that a program is safe only if *all* possible executions are safe. In contrast, "nondeterminism" does not seem to have a consistent definition, but often refers to an *existential* quantification over executions.

```
%abbrev word = string ww.
%abbrev byte = string 1.
```

The most common data type in the semantics is `string`, which contains strings of bytes constructed using `/` for cons[6] and `#` for nil. It is convenient for the type of strings to indicate the string's length; thus we may define words as `string ww`. When we do not care about the length of a string, we may say `string _`, using the Twelf wildcard.[7]

3.3 State

The state of the architecture consists of a memory, a register file, a flag register (containing the IA-32's condition codes), an instruction pointer, and an oracle.

Main memory (or, more precisely, the accessible portion of main memory) is divided into *sections.* A section is a piece of memory that is known to be contiguous, and offsets into it may be used to access particular locations. The sections are arranged in the address space of representable pointers, that is, it lies between 0 and $2^{32} - 1$. Memory is maintained as a list of sections ordered by their start addresses.

Sections are classified into *valid* and *reserved.* We say that an address is valid if it falls within a valid section, and similarly reserved if it falls within a reserved section. We say that an address is *undefined* if it falls within no section.

Valid sections are available to the program; they allow loads, stores, and jumps to any address within them. Valid sections contain a string that indicates their contents and a segment descriptor. The contents string can be altered (of course), but may not change in length. The segment descriptor may be one of `hs` (heap segment), `cs` (code segment), and `ss` (stack segment). The segments differ in how they are treated by the garbage collector (Section 3.5).

Reserved sections, on the other hand, are largely unavailable to the program; they do not allow loads or stores to any address within them. Reserved sections are used for a variety of purposes. For example, one use of reserved sections is to represent functions provided to the program by a runtime library. The program is permitted to jump to the beginning of such a section, but no other jumps to reserved addresses are permitted.

The principal difference between reserved sections and ranges of undefined addresses (other than special properties that reserved sections may have) is that a reserved section will remain for the duration of the program. However, any undefined range may be selected by the memory allocator and thereby become a valid section.

[6]The curious choice of name for the cons constant is justified by making `/` infix, so that a word may be written $byte_1$ `/` $byte_2$ `/` $byte_3$ `/` $byte_4$ `/` `#`

[7]There is an important distinction between the wildcard `_` and implicit arguments such as `N`. Implicit arguments are universally quantified, whereas wildcards are existentially quantified. Therefore, implicit arguments may not unify with each other or with constants, but wildcards may unify with anything, including implicit arguments.

```
segment : type.

ss  : segment.
cs  : segment.
hs  : segment.

rsection : nat -> type.  %% "reserved section"

section  : nat -> type.

section_valid    : segment -> string N -> section N.
section_reserved : rsection N -> section N.

memory : type.

mnil  : memory.
mcons : address -> {n:nat} section n -> memory -> memory.
          %% start address, size, contents, rest of memory
```

The register file is a list of contents of registers. It has a fixed length (eight[8]), and contains words instead of arbitrary strings.

```
numregs : nat = 8.

regs      : nat -> type.
regs_nil  : regs 0.
regs_cons : word -> regs N -> regs (s N).
```

The full register file then has type `regs numregs`.

The flag register has four bits representing the carry, overflow, zero and sign flags. Other flags such as the direction and parity flags are currently not supported.

```
flags : type.

flags_ : bit        % cf
           -> bit   % zf
           -> bit   % sf
           -> bit   % of
           -> flags.
```

The instruction pointer is a simple address. These components are assembled into the machine state:

[8]We include only the general purpose registers (including `esp`). The `EFLAGS` and `EIP` register are handled specially, and the segment registers are omitted (we assume they are all set to the same segment, providing a flat address space). Floating point and the SIMD features of later IA-32 models are also currently unsupported.

```
state  : type.
state_ : memory
           -> regs numregs
           -> flags
           -> address  %% instruction pointer
           -> oracle -> state.
```

A few operations halt execution of the program, but are still considered safe. A simple example is when the program finishes and exits; more interesting examples are processor exceptions that the runtime can trap (*e.g.,* stack overflow or divide-by-zero). We say that these operations transition to a "stopped" state:

```
stopped : state.
```

3.4 The Transition Relation

The transition relation is defined by three rules. In the ordinary case, the semantics fetches the next instruction and then executes it:

```
fetch : state -> inst -> type
transition' : inst -> state -> state -> type.
...

transition  : state -> state -> type.
transition_main : transition ST ST'
                   <- fetch ST IN
                   <- transition' IN ST ST'.
```

Here, `fetch` is an auxiliary relation that fetches and decodes the next instruction. Due to the baroque encoding on instructions on the IA-32, `fetch` contains much of the uninteresting complexity of our safety policy. The interesting content of the safety policy is largely contained in the auxiliary relation `transition'`, which carries out an instruction in a particular state.

In the event that the instruction pointer is at the beginning of a function provided by the runtime library, `fetch` is unable to fetch any instructions, so the rule `transition_main` does not apply. (Recall that loads from reserved sections are not permitted.) Instead, the rule `transition_runtime` (given below) applies. The auxiliary relation `at_runtime_address` indicates that a runtime facility is to be executed, and that facility is executed by the auxiliary relation `transition_runfac`, which is defined in a similar manner to `transition'`.

```
at_runtime_address : state -> runfac -> type
transition_runfac : runfac -> state -> state -> type.
...

transition_runtime : transition ST ST'
                      <- at_runtime_address ST RF
                      <- transition_runfac RF ST ST'.
```

The semantics does not attempt to represent the intermediate states of a runtime function; it treats runtime functions as if they were atomic operations. Note that a `jmp` or `call` instruction directed to a runtime function does not immediately invoke the runtime function; it merely sets the instruction pointer to the beginning of that function. The function itself is executed in the next transition. Therefore there is no special case in the semantics `jmp` or `call` for calls to the runtime.

Since our simple safety policy expects all safe states to make transitions, and in particular makes no allowance for safe terminal states, we have need of a “stopped” state that transitions to itself:

```
transition_stopped : transition stopped stopped.
```

The main work is done by the helper relations `transition'` and `transition_runfac`. We give two cases by way of example:

```
trans_add :
   transition' (ii_add E O) ST ST'
      <- load ST E W1
      <- oload ST O W2
      <- add W1 W2 W3 RF
      <- store ST E W3 ST1
      <- store_result_flags ST1 RF ST2
      <- next ST A
      <- puteip ST2 A ST'.

trans_jmp :
   transition* (ii_jmp O) ST ST'
      <- oload ST O W
      <- implode_word W A
      <- puteip ST A ST'.
```

An IA-32 add instruction takes two arguments: an effective address `E` that is the destination for the result and one of the summands, and an operand `O` providing the other summand. An effective address is either a register or a memory location; an operand (in this case) is either an effective address or an immediate value.

To perform the add, we load the values of `E` and `O`, obtaining the words `W1` and `W2`. We then add the summands, obtaining a result `W3` and some result flags `RF` (*e.g.,* the carry and zero flags). We then store `W3` back into `E` and the result flags into the flag register, obtaining state `ST2`. Finally, we compute the address `A` of the next instruction and store it into the instruction pointer, obtaining the final state `ST'`.

An IA-32 jump instruction takes only an operand `O`, indicating the target of the jump. It is performed by resolving the operand to obtain the target address `W`. The address `W` is rephrased[9] as a 32-bit address `A`, which is then written into the instruction pointer.

[9]Recall from Section 3.2 that a `word` (the result type of `oload` in this case) is defined to be a sequence of four 8-bit numbers, rather than a single 32-bit number.

3.5 Garbage Collection

Our operational semantics must specify the behavior, not only of the instruction set itself, but also of the operations provided by the runtime library. Most notable of the runtime operations is memory allocation. Memory allocation is largely straightforward (we simply add a new section to the memory at an address determined by the oracle), except that any allocation may invoke the garbage collector. Thus we must define the possible behaviors of the garbage collector.

The actual collection process is carried out simply by deleting a set of sections from the heap. The complexity arises in show that set is determined. To fully specify the behavior of a garbage collector would be a daunting task, and, moreover, it is not clear that a safe program should depend on the fine details of the collector's implementation in any case. Instead, we specify a collection of possible behaviors for the garbage collector that will include the collector's actual behavior, among others. Exactly which behavior will take place is determined by the oracle, so a safe program must be prepared for any of them.

The possible behaviors are exactly those in which an *unreachable* set (as defined below) of heap sections is selected to be deleted. Note that this definition makes no promise of liveness. The empty set is always unreachable, so it is always permissible for it to collect nothing.

Our definition of unreachability is based on that in Crary [2003]. Informally, given some current state, a set S of heap sections is reachable if there exists a pointer into S either from the root set or from the complement of S (relative to the set of all heap sections). A set is unreachable if it is not reachable. The root set consists of all registers and the stack (that is, the region of the stack segment[10] above the stack pointer).

Our runtime uses the Boehm-Demers-Weiser conservative collector [Boehm and Weiser 1988], so no invariants are maintained to assist in pointer identification. Any number that equals a legitimate pointer value is deemed to be a pointer. This greatly simplifies the semantics of garbage collection, but it does prevent the use of non-conservative collection. Relative to non-conservative collection, our notion of reachability is weaker, and therefore our notion of unreachability is stronger. This in turn means that fewer collection behaviors are considered possible, and therefore that more programs are considered safe, including some that would not be safe with a non-conservative collector.

3.6 The Safety Policy

The final element needed by the safety policy is a definition of the possible initial states of a program. The relation `initial_state S ST` relates a program to its possible initial states. A possible initial state is obtained by performing the following operations:

(1) placing the program into memory (at an arbitrary start address),

(2) choosing arbitrary places in memory to place the runtime functions (including

[10]It is an invariant of the semantics that there exists exactly one section in the heap segment. The semantics requires that the stack pointer point into that section at the beginning of collection; otherwise the program will be stuck.

a "global offset table" [Tool Interface Standards Committee 1995] to contain them),
(3) choosing an arbitrary size for the stack and filling it with arbitrary values,
(4) setting the `ESP` register to the end of the stack area and the `EBX` register to the address of the global offset table,
(5) setting the instruction pointer to the beginning of the program,
(6) filling the flags and remaining registers with arbitrary initial values, and
(7) choosing an arbitrary value for the oracle.

We can now state the safety policy. First, we say that a program `S` (*i.e.,* a string of bytes) can reach a state `ST` if `ST` is reachable in zero or more transitions from an initial state of `S`:

```
reachable   : string _ -> state -> type.
reachable_z : reachable S ST
                 <- initial_state S ST.
reachable_s : reachable S ST2
                 <- reachable S ST1
                 <- transition ST1 ST2.
```

Second, we declare, but do not define, a type of advice and a predicate `check` that takes a program and advice as inputs.

```
advice : type.

check : advice -> string _ -> type.
```

The code supplier is responsible for filling in the definitions of both `advice` and `check`.

The purpose of advice is to provide a means by which program-specific information can be supplied to the checker, so that the checker itself need not be specialized to a particular program. No guarantees are made regarding the quality of the advice; they serve only as hints. However, one expects that most checkers will fail if given faulty advice.

Finally, we declare, but do not prove, the safety metatheorem:

```
safety : check _ S
          -> reachable S ST
          -> transition ST ST' -> type.
%mode safety +D1 +D2 -D3.
```

This metatheorem says that whenever there exists a derivation of `check _ S` (*i.e.,* there is some advice such that `S` with the advice passes the code supplier's safety checker), and there exists a derivation of `reachable S ST` (*i.e.,* `S` can reach state `ST`), then there exists a derivation of `transition ST ST'` (*i.e.,* `ST` is not stuck).

Recall that the code supplier is responsible for filling in the proof of `safety`. In so doing he or she establishes the soundness of his or her safety checker.

The safety policy consists of 3557 lines of Twelf code, including comments. A breakdown appears in Table I.

TSP Components	
lines	purpose
32	Top-level safety policy
110	Initial states
130	Architecture data and state
191	Natural number arithmetic
247	ALU operations
248	GC
527	Instructions and transitions
622	Memory and register utilities
645	Instruction decode
805	Binary number arithmetic
3557	Total

Table I. Safety Policy Breakdown

4. TALT

As a particular instance of a correct safety checker, we have implemented a safety checker based on the typed assembly language TALT [Crary 2003], and developed safety proofs for it that satisfy TSP. We briefly discuss the checker here, and then focus on its safety proof in the next section.

Our checker does not use TALT directly. Like most declarative specifications of programming languages, it would not behave well were it taken directly as a logic program. So at the least, we would need to reformulate the type system with execution in mind. In fact, we must go a bit further: TALT is optimized for theoretic elegance, not for typechecking, and it is likely that its typechecking problem is undecidable. Consequently, we first develop a variant of TALT called XTALT ("explicitly typed TALT") that does enjoy decidable typechecking, and, moreover, we formulate it to behave well as a logic program.

A complete discussion of XTALT is beyond the scope of this paper. Briefly, it makes typechecking syntax directed by adding explicit type annotations and by requiring explicit coercions in place of uses of subtyping.

The XTALT checker is defined using two main relations:

```
parse_program : string _ -> program -> type.
check_program : program -> type.
```

The relation `parse_program S P` determines that the binary `S` corresponds to the XTALT program `P`, and the relation `check_program P` determines that `P` is well-typed. The XTALT safety checker passes a binary `S` provided it corresponds to a XTALT program `P` (supplied as advice), and `P` is well-typed:

```
%{ advice : type }%
advice_ : program -> advice.

%{ check : advice -> string _ -> type. }%
check_ : check (advice_ P) S
          <- parse_program S P
          <- check_program P.
```

XTALT Checker Components	
lines	purpose
95	Top-level checking
185	Checking coercions
231	Checking values and operands
304	Checking individual instructions
388	XTALT syntax
617	Checking type validity and equivalence
1051	Parsing programs
2871	Total

Table II. Safety Checker Breakdown

Recall that `advice` and `check` are pre-declared by the safety policy, so we have only to fill in their definitions.

The XTALT checker consists of 2871 lines of Twelf code, including comments. A breakdown appears in Table II.

5. THE SAFETY PROOF

The proof that the XTALT checker satisfies TSP is structured in three stages:

(1) **Static Stage:** a purely static portion that proves the soundness of the XTALT typechecker relative to the TALT type system.
(2) **Abstract Stage:** a type safety proof for TALT relative to an abstract machine, using the usual tools of progress and type preservation.
(3) **Concrete Stage:** a type-free simulation argument that the TALT abstract machine is realized by the IA-32 architecture.

We will discuss the first two stages only briefly, and focus our attention on the third.

5.1 The Static Stage

In the first stage, we define the TALT type system and prove that the XTALT checker is sound with respect to it. At the top level, TALT defines a type `machine` of abstract machines, and a well-typedness predicate `machineok` on them.

Next we define an elaboration relation that formalizes the correspondence between XTALT and TALT programs. At the top level it relates XTALT programs to TALT machines:

```
elab_program : program -> machine -> type.
```

With these definitions made, we prove two main theorems. The first states the soundness of the XTALT type checker:

```
sound_program : elab_program P M
                  -> check_program P
                  -> machineok M -> type.
%mode sound_program +D1 +D2 -D3.
```

This says that if the XTALT program `P` elaborates to the TALT abstract machine `M`, and the XTALT checker says that `P` is well-typed, then `M` is also well-typed.

The second theorem states the soundness of the binary parser:

```
initial_impl : parse_program S P
                -> initial_state S ST
                -> elab_program P M
                -> implements ST M -> type.
%mode initial_impl +D1 +D2 -D3 -D4.
```

This says that if the binary `S` corresponds to the XTALT program `P`, and `ST` is an initial state for `S`, then `P` elaborates to some TALT abstract machine `M` that is implemented by `ST`. (The relation `implements` is discussed in Section 5.3.)

These two theorems are folded into the final safety proof in Section 5.5.

5.2 The Abstract Stage

In the second stage, we give TALT an operational semantics in terms of a low-level abstract machine. We then prove type safety for that abstract semantics. The development of this stage is given in detail in Crary [2003] and we will not repeat it here except to summarize its top-level results.

The TALT type system is summarized by the predicate `machineok` (as discussed above), and the operational semantics is given by the relation `stepsto`. Given these, the final results of Crary are two safety theorems:[11]

```
progress : machineok M
            -> stepsto M M' -> type.
%mode progress +D1 -D2.

preservation : machineok M
                -> stepsto M M'
                -> machineok M' -> type.
%mode preservation +D1 +D2 -D3.
```

These are the standard type safety lemmas [Wright and Felleisen 1994; Harper 1994]: `progress` states that when the abstract machine state `M` is well-typed, it takes a step to some `M'`; and `preservation` states that when `M` is well-typed and steps to `M'`, then `M'` is well-typed.

Unlike the concrete semantics of TSP, which we forced to be deterministic, the TALT abstract machine is indeterministic.[12]

That is, a TALT machine state may step to more than one following state. This is because the actual result of an operation is sometimes determined by information

[11]Crary [2003] also proved a third theorem stating that well-typedness is preserved by garbage collection, but that theorem is now folded into the type preservation theorem.

[12]The alert reader may observe an apparent inconsistency in our methodology. In Section 3.1 we argued that using a non-functional relation in TSP's operational semantics would be subtle and error-prone, but we are willing to use one for TALT's operational semantics. The difference between the two situations is that TALT's operational semantics is proven correct relative to another system (TSP), but TSP is merely stated; we cannot show it correct relative to anything.

that it abstracts away. For example, when doing pointer arithmetic, the condition codes are determined by the actual numeric values of the operands, but in the abstract machine, pointers are taken as abstractions, so their numeric values are not available. In addition, the abstract machine has no access to the oracle, so any instruction that consults the oracle is indeterministic in the abstract machine. The simulation in the concrete stage must manage this mismatch between the two semantics.

5.3 The Concrete Stage

In the third stage we show that the abstract operational semantics of TALT maps correctly onto the semantics of the concrete machine. This argument is type-free, as all type-theoretic issues are dealt with in the abstract proofs, but it is still fairly involved due to the myriad technicalities of the concrete architecture. We present the high-level structure of the proof, without going into the technicalities.

First we define a relation `implements ST M`, which states that the concrete state `ST` implements the abstract state `M`. Second, we define a multi-step transition relation `transitions N ST ST'`, which states that `ST` transitions to `ST'` in exactly `N` steps:

```
transitions   : nat -> state -> state -> type.
transitions_z : transitions 0 ST ST.
transitions_s : transitions (s N) ST1 ST3
                 <- transition ST1 ST2
                 <- transitions N ST2 ST3.
```

5.3.1 *Simulation.* One main lemma of the concrete stage is simulation:

```
simulate : implements ST M
            -> stepsto M M1
            -> transitions (s N) ST ST'
            -> stepsto M M2
            -> implements ST' M2 -> type.
%mode simulate +D1 +D2 -D3 -D4 -D5.
```

This lemma is read as follows: If `ST` implements `M`, and `M` steps abstractly to some `M1`, then there exists another abstract machine `M2` (possibly the same as `M1`) and a concrete machine `ST'` implementing `M2`, such that `ST` transitions to `ST'` in one or more steps, and `M` steps to `M2`. Note that the lemma allows for an abstract step to correspond to multiple concrete transitions.

Observe how this lemma addresses the indeterminism of the abstract machine. The fact that `M` steps to some `M1` indicates that `M` is safe. However, it is not necessarily the case that `M1` corresponds to the transition that actually takes place, so the `simulate` theorem returns a second machine `M2` that does correspond to the actual transition.

5.3.2 *Determinism.* The other main lemma of the concrete stage is determinism:

```
state_eq    : state -> state -> type.
state_eq_   : state_eq ST ST.

determinism : transition ST ST1
                -> transition ST ST2
                -> state_eq ST1 ST2 -> type.
```

The relation `state_eq ST1 ST2` holds exactly when `ST1` and `ST2` are identical. Therefore the lemma is read as follows: If `ST` transitions to `ST1`, and `ST` transitions to `ST2`, then `ST1` and `ST2` are identical.

5.4 Concrete Progress and Preservation

We say that a concrete state `ST` is *ok* if `ST` transitions in zero or more steps to some `ST'` that implements a well-typed abstract state:

```
ok  : state -> type.
ok_ : ok ST
       <- transitions _ ST ST'
       <- implements ST' M
       <- machineok M.
```

We can now prove *concrete* progress and preservation, using `ok` as the relevant notion of typeability. The machine-readable proofs of these and the other remaining theorems are given in Appendix A.

Lemma 5.1.

```
iprogress : ok ST -> transition ST ST' -> type.
%mode iprogress +D1 -D2.
```

Proof: Since `ST` is okay, it steps to some `ST'` in some `N` steps. If `N` ≥ 1, the result is immediate. Otherwise `ST` = `ST'`, so `implements ST M` and `machineok M`. By `progress`, `stepsto M M'`, and therefore by `simulate`, `ST` takes a step. □

Lemma 5.2.

```
ipreservation : ok ST
                 -> transition ST ST'
                 -> ok ST' -> type.
%mode ipreservation +D1 +D2 -D3.
```

Proof: Since `ST` is ok, it steps to some `ST''` (which implements a well-typed abstract state) in some `N` steps. Suppose `N` ≥ 1. Then `transition ST ST1` and `transitions _ ST1 ST''`. By `determinism`, `ST'` = `ST1`, and `ST1` is ok, so `ST'` is also ok.

Suppose `N` $= 0$. Then `ST` implements a well-typed abstract state `M`. By `progress`, we have `stepsto M M1`. By `simulate`, `transitions _ ST ST''`, `stepsto M M2`, and `implements ST'' M2`. By `preservation`, `M2` is well-typed, so `ST''` is ok. Finally, by `determinism`, `ST'` = `ST''`, so `ST'` is ok. □

5.5 Safety

It remains to put the pieces together to prove safety. First we show that every initial state of a program that passes the safety checker is ok:

LEMMA 5.3.

```
initial_ok : check _ S
               -> initial_state S ST
               -> ok ST -> type.
%mode initial_ok +D1 +D2 -D3.
```

Proof: By inversion on `check`, there exists an XTALT program `P` such that `parse_program S P` and `check_program P`. By `initial_impl`, there exists an abstract machine `M` such that `elab_program P M` and `implements ST M`. Then by `sound_program`, we have `machineok M`. Therefore, since `ST` transitions to itself in zero steps, we conclude `ok M`. □

We may now prove that any state reachable from a program that passes the safety checker is ok:

LEMMA 5.4.

```
safety' : check _ S
            -> reachable S ST
            -> ok ST -> type.
%mode safety' +D1 +D2 -D3.
```

Proof:

—(Case `reachable_z`) Suppose `initial_state S ST`. By `initial_ok`, `ST` is ok.

—(Case `reachable_s`) Suppose `reachable S ST'` and `transition ST' ST`. By induction, `ST'` is ok, so by `ipreservation`, `ST` is ok. □

Using `iprogress`, `safety` is an immediate consequence of `safety'`. This completes the proof.

The complete safety proof for the safety checker consists of 59,896 lines of Twelf code, including comments. A breakdown appears in Table III. It takes 126 seconds to check in Twelf 1.5 on a Pentium 4 with one gigabyte of RAM.

6. CONCLUSION

Using the metalogical approach we advocate here, one may work conveniently with derivations in logics, including type systems and safety policies. This enables relatively rapid development of foundational certified code.

However, there are some costs to the Twelf metalogical approach, at least as things stand today. First, in the Twelf metalogic one is limited to Π-1 reasoning (*i.e.,* reasoning involving only propositions of the form $\forall x_1 \ldots \forall x_m \exists y_1 \ldots \exists y_n . P$ where P is quantifier-free). Using Skolemization, propositions can often be cast in this form, so this is rarely an obstacle. However, some proof techniques (notably logical relations) cannot be cast in Π-1 form and therefore cannot be employed. The

Proof Components	
lines	purpose
8250	Static stage
2086	TALT type system
21051	Abstract stage
1085	TALT abstract machine
22712	Concrete stage
4712	Shared
59896	Total

Table III. Proof Breakdown

Twelf developers are exploring ways to relax this restriction, but none are available at this time.

Second, since checking the validity of a meta-proof involves more than just type-checking (which is all that is required for checking the validity of a proof within a logic), the proof checker for the Twelf metalogic is larger and more complicated than checkers for simpler logics can be (*e.g.,* Appel *et al.* [Appel et al. 2002]). As a result, it can expected to take longer to develop the same degree of trust in our system. However, recall that our purpose in developing an foundational system is more to improve flexibility and extensibility by eliminating trusted components that may prove unsatisfactory in the future, and less to improve confidence by minimizing the size of the remaining trusted components.

Despite these limitations, we believe the advantages of the Twelf metalogical approach are compelling. In addition to the practical benefit of rapid development, metalogic also holds the promise of making it easier to draw connections between distinct certified code systems (which in practice are all expressed in distinct formal systems). For example, one might show that one safety policy implies another, and in so doing make it possible to unify two lines of development of certified code systems. We plan to explore this in the future.

APPENDIX

A. TWELF PROOFS

```
ok_resp : state_eq ST ST' -> ok ST -> ok ST' -> type.
%mode ok_resp +D1 +D2 -D3.

! : ok_resp state_eq_ D D.


iprogress : ok ST -> transition ST ST' -> type.
%mode iprogress +D1 -D2.

! : iprogress (ok_ Dmok Dimpl transitions_z) Dtrans
     <- progress Dmok Dstep
     <- simulate Dimpl Dstep (transitions_s _ Dtrans) _ _.
! : iprogress (ok_ _ _ (transitions_s _ Dtrans)) Dtrans.
```

```
ipreservation : ok ST -> transition ST ST' -> ok ST' -> type.
%mode ipreservation +D1 +D2 -D3.

! : ipreservation (ok_ Dmok Dimpl transitions_z) Dtrans Dok
     <- progress Dmok Dstep
     <- preservation Dmok Dstep Dmok'
     <- simulate Dimpl Dstep (transitions_s Dmtrans Dtrans') Dcoll Dimpl'
     <- determinism Dtrans' Dtrans Deq
     <- collect_ok Dcoll Dmok' Dmok''
     <- ok_resp Deq (ok_ Dmok'' Dimpl' Dmtrans) Dok.
! : ipreservation (ok_ Dmok Dimpl (transitions_s Dmtrans Dtrans')) Dtrans Dok
     <- determinism Dtrans' Dtrans Deq
     <- ok_resp Deq (ok_ Dmok Dimpl Dmtrans) Dok.


initial_ok : check _ S -> initial_state S ST -> ok ST -> type.
%mode initial_ok +D1 +D2 -D3.
! : initial_ok (check_ Dcheck Dparse) Dinitial
     (ok_ Dmok Dimpl transitions_z)
     <- initial_impl Dparse Dinitial Delab Dimpl
     <- sound_program Delab Dcheck Dmok.


safety' : check _ S -> reaches S ST -> ok ST -> type.
%mode safety' +D1 +D2 -D3.

! : safety' Dcheck (reaches_z Dinitial) Dok
     <- initial_ok Dcheck Dinitial Dok.
! : safety' Dcheck (reaches_s Dtrans Dreach) Dok'
     <- safety' Dcheck Dreach Dok
     <- ipreservation Dok Dtrans Dok'.


safety : check _ S -> reaches S ST -> transition ST ST' -> type.
%mode safety +D1 +D2 -D.

! : safety Dcheck Dreaches Dtrans
     <- safety' Dcheck Dreaches Dok
     <- iprogress Dok Dtrans.
```

REFERENCES

Appel, A. W. and Felty, A. P. 2000. A semantic model of types and machine instructions for proof-carrying code. In *Twenty-Seventh ACM Symposium on Principles of Programming Languages*. Boston, 243–253.

Appel, A. W., Michael, N., Stump, A., and Virga, R. 2002. A trustworthy proof checker. Tech. Rep. TR-647-02, Department of Computer Science, Princeton University. Apr.

BERNARD, A. AND LEE, P. 2002. Temporal logic for proof-carrying code. In *Eighteenth International Conference on Automated Deduction.* Lecture Notes in Artificial Intelligence, vol. 2392. Springer-Verlag, Copenhagen, Denmark, 31–46.

BOEHM, H.-J. AND WEISER, M. 1988. Garbage collection in an uncooperative environment. *Software Practice and Experience 18,* 9 (Sept.), 807–820.

BUYYA, R. AND BAKER, M., Eds. 2000. *First International Workshop on Grid Computing.* Lecture Notes in Computer Science, vol. 1971. Springer-Verlag, Bangalore, India.

CHANG, B.-Y. E., CRARY, K., DELAP, M., HARPER, R., LISZKA, J., VII, T. M., AND PFENNING, F. 2002. Trustless grid computing in ConCert. In *Third International Workshop on Grid Computing.* Lecture Notes in Computer Science, vol. 2536. Baltimore, Maryland, 112–125.

COLBY, C., LEE, P., NECULA, G., AND BLAU, F. 2000. A certifying compiler for Java. In *2000 SIGPLAN Conference on Programming Language Design and Implementation.* Vancouver, British Columbia, 95–107.

CRARY, K. 2003. Toward a foundational typed assembly language. In *Thirtieth ACM Symposium on Principles of Programming Languages.* New Orleans, Louisiana, 198–212.

HAMID, N., SHAO, Z., TRIFONOV, V., MONNIER, S., AND NI, Z. 2002. A syntactic approach to foundational proof-carrying code. In *Seventeenth IEEE Symposium on Logic in Computer Science.* Copenhagen, Denmark, 89–100.

HARPER, R. 1994. A simplified account of polymorphic references. *Information Processing Letters 51,* 4, 201–206. Follow-up note in *Information Processing Letters*, 57(1), 1996.

HARPER, R., HONSELL, F., AND PLOTKIN, G. 1993. A framework for defining logics. *Journal of the ACM 40,* 1 (Jan.), 143–184.

Intel Corporation 2001. *IA-32 Intel Architecture Software Developer's Manual.* Intel Corporation. Order numbers 245470–245472.

LEE, C., Ed. 2001. *Second International Workshop on Grid Computing.* Lecture Notes in Computer Science, vol. 2242. Springer-Verlag, Denver, Colorado.

LINDHOLM, T. AND YELLIN, F. 1996. *The Java Virtual Machine Specification.* Addison-Wesley.

MORRISETT, G., CRARY, K., GLEW, N., GROSSMAN, D., SAMUELS, R., SMITH, F., WALKER, D., WEIRICH, S., AND ZDANCEWIC, S. 1999. TALx86: A realistic typed assembly language. In *Second Workshop on Compiler Support for System Software.* Atlanta.

MORRISETT, G., CRARY, K., GLEW, N., AND WALKER, D. 2002. Stack-based typed assembly language. *Journal of Functional Programming 12,* 1 (Jan.), 43–88.

MORRISETT, G., WALKER, D., CRARY, K., AND GLEW, N. 1999. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems 21,* 3 (May), 527–568. An earlier version appeared in the 1998 Symposium on Principles of Programming Languages.

NECULA, G. 1997. Proof-carrying code. In *Twenty-Fourth ACM Symposium on Principles of Programming Languages.* Paris, 106–119.

NECULA, G. AND LEE, P. 1996. Safe kernel extensions without run-time checking. In *Second Symposium on Operating Systems Design and Implementation.* Seattle, 229–243.

NECULA, G. C. 1998. Compiling with proofs. Ph.D. thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania.

PARASHAR, M., Ed. 2002. *Third International Workshop on Grid Computing.* Lecture Notes in Computer Science, vol. 2536. Springer-Verlag, Baltimore, Maryland.

PAULIN-MOHRING, C. 1993. Inductive definitions in the system coq—rules and properties. In *International Conference on Typed Lambda Calculi and Applications.* Lecture Notes in Computer Science, vol. 664. Springer-Verlag.

PFENNING, F. 1991. Logic programming in the LF logical framework. In *Logical Frameworks*, G. Huet and G. Plotkin, Eds. Cambridge University Press, 149–181.

PFENNING, F. AND ROHWEDDER, E. 1992. Implementing the meta-theory of deductive systems. In *Eleventh International Conference on Automated Deduction.* Lecture Notes in Computer Science, vol. 607. Springer-Verlag, Saratoga Springs, New York, 537–551.

Pfenning, F. and Schürmann, C. 1999. System description: Twelf — a meta-logic framework for deductive systems. In *Sixteenth International Conference on Automated Deduction*. Lecture Notes in Computer Science, vol. 1632. Springer-Verlag, Trento, Italy, 202–206.

Pfenning, F. and Schürmann, C. 2002. *Twelf User's Guide, Version 1.3R4*. Available electronically at `http://www.cs.cmu.edu/~twelf`.

Pientka, B. and Pfenning, F. 2000. Termination and reduction checking in the logical framework. In *Workshop of Automation of Proofs by Mathematical Induction*.

Rohwedder, E. and Pfenning, F. 1996. Mode and termination checking for higher-order logic programs. In *European Symposium on Programming*. Lecture Notes in Computer Science, vol. 1058. Springer-Verlag, Linköping, Sweden, 296–310.

Schürmann, C. 2000. Automating the meta theory of deductive systems. Ph.D. thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania.

SETI@Home. 2000. `http://setiathome.ssl.berkeley.edu`.

Tool Interface Standards Committee. 1995. *Executable and Linking Format (ELF) specification*. `http://x86.ddj.com/ftp/manuals/tools/elf.pdf`.

Wright, A. K. and Felleisen, M. 1994. A syntactic approach to type soundness. *Information and Computation 115*, 38–94.