Foundational Certified Code in a Metalogical Framework

Karl Crary and Susmit Sarkar

Carnegie Mellon University

Abstract. 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.

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 [26] in 1997, a handful of projects have successfully made profitable use of idle computational resources on the Internet. More recently, this paradigm, now called *grid computing,* has elicited serious interest among academics [5, 14, 22] 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 [6] 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 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 [15, 20, 16] 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 [19, 18, 17, 21], 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 system's trusted elements,[1] such as the account of Appel and Felty [1], and that of Hamid *et al.* [10]. 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 [3] eliminate the verification condition generator from the trusted elements of the SpecialJ PCC system [7], and Appel *et al.* [2] seek to minimize the number of lines of code in an LF proof checker [12]. 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 type system, a proof that the given program passes the type system, and 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.

proof that all programs passing that type system are safe. In each case, the main development is in the latter, the type system's safety proof.

A closer analysis reveals important methodological differences. The account of Appel and Felty [1] and their collaborators 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, the account of Hamid, *et al.* [10] is operational, with safety given by conventional type preservation and progress lemmas [28, 11].

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 [12], and for the Hamid *et al.*, the calculus of inductive constructions [23]), 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.

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

1. For our safety policy, 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] [13], but it should be adaptable to other architectures.
2. The code supplier is invited to provide a second logic defining a *safety condition.* The safety condition could identify a single program, but in practice is more likely to be a type system accepting many programs.
3. The final component of the safety policy is the statement of a metatheorem: for any program P for which the safety condition holds, if a machine loaded with P eventually runs to a state S, then S transitions to some state S'. It follows (by construction of the operational semantics) 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 verified by a meta-proof checker.
5. Finally, the code supplier also provides a derivation establishing that the program of interest satisfies the safety condition. This derivation will typically be computed automatically by a type checker.

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

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

The importance of this difference is a pragmatic one. Using the Twelf metalogical framework [24, 25], 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 entire system, including the foundational safety proof for an expressive typed assembly language called TALT ("TAL Two") [8], in less than one-and-a-half 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 its pragmatic advantage.

The remainder of the paper is structured as follows: In Section 2 we present our system's architecture semantics and safety policy, that is, the elements of a certified code system imposed by the code consumer. In Section 3 we outline our meta-proof that TALT is one particular instance of a system satisfying that policy. Concluding remarks appear in Section 4. We assume familiarity with the Twelf metalogic [25]. The companion technical report [9] contains a brief tutorial on Twelf.

2 The Safety Policy

The main body of the safety policy 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. Due to space considerations, 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.

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. This code formalizing this, which concludes the safety policy, is given in Section 2.5.

Indeterminism An important issue complicating the operational semantics is *indeterminism.*[3] In some cases it is impossible to determine the outcome of an operation. This may happen become the outcome is fundamentally unknowable (*e.g.,* an input operation), or because the operation is too complex for a full specification to be feasible (*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 most obvious implementation of indeterminism is to make the relation defining the semantics non-functional, that is, to allow it to relate a state to multiple next states. This simple approach does not work well for two reasons. First, for the purpose of developing safety proofs, it is *much* more convenient to work with a deterministic (functional) relation, as we will see. Second, with our notion of safety, it is necessary that any state that might possibly perform an unsafe operation be given no transitions at all; it is *not* sufficient simply to omit the unsafe transition, as would happen with the most natural crafting of the relation. Designing the relation in this manner is very subtle to do, which leads to the possibility of insidious errors in the safety policy that undermine the entire system.

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 2.5) universally quantifies over all finite oracles, and any safety violation happens within a finite amount of time.

Abstraction The main form of data manipulated by the operational semantics consists of literal bytes. Additionally, the semantics deals with some abstract forms of data. The main example of abstract data are pointers. Because it is not feasible to formalize the entire behavior of our allocation and garbage collection (based on the Boehm-Demers-Weiser conservative collector [4]), it is impossible to determine the concrete value of an allocated pointer.

Instead, we break up memory into *sections.* A section is a contiguous area of memory, but distinct sections appear in an unknown order in memory, possibly with intervening gaps. A pointer is then viewed as a pair of a section identifier and an offset into that section. Note that the offset need not fall within the bounds of the section, so it is still quite possible to construct ill-formed pointers.

Our semantics also includes two other forms of abstract data. One is a special distinguished pointer to a *global offset table* that grid applications use to access

[3] 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.

the facilities (such as allocation) provided by the runtime [27]. The other is a special "don't know" value used to represent various ill-formed, abstract data, such as the sum of two pointers.

2.1 Data

We use two notions of numbers in our semantics. One is the natural numbers. The other is `binary N`, which contains N-bit binary numbers. Given these, we can define our data types:

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

section : type.
sect    : nat -> section.

address : type.
addr    : section -> binary wwb -> address.

byte : type.
act  : binary bw -> byte. %% actual
dk   : byte.              %% don't know
boa  : address
        -> nat -> byte.   %% byte of address
boga : nat -> byte.       %% byte of GOT addr.

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

The four forms of byte are discussed above. In order to work in terms of bytes instead of words, the pointer forms include an index into the pointer, so `boa A N` represents the `N`th byte of address `A`. Thus, the pointer `A` is represented as the sequence of bytes `boa A 0`, ..., `boa A 3`.

The most common data type in the semantics is `string`, which contains strings of bytes constructed using `/` for cons[4] 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.

---

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

2.2 State

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

The memory consists of a mapping from section identifiers to strings. As given in Section 2.1, section identifiers are values of the form `sect N` where `N` is a natural number. Therefore, we may represent the memory as a list of strings, and obtain `sect N` by extracting the `N`th element of the list:

```
memory : type.
mnil   : memory.
mcons  : string _ -> memory -> memory.
minv   : memory -> memory.
```

A section may become invalidated, such as when it is garbage collected. Since sections are looked up by their position in the list of sections, we use `minv` as a placeholder for invalid sections, in order to ensure the invalidation of one section does not change the positions of later sections in the list.

The register file is similar to the memory except that it has a fixed length (eight[5]), it contains words instead of arbitrary strings, and registers cannot become invalid:

```
numregs : nat = 8.

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

The full register file then has type `regs numregs`. Due to space considerations, we omit the definitions of the flag register (type `flags`) and the oracle (type `oracle`). The program counter is a simple address. These components are assembled into the machine state:

```
state  : type.
state_ : memory
          -> regs numregs
          -> flags
          -> address  %% program counter
          -> oracle -> state.
```

[5] 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.

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.
```

2.3 The Transition Relation

The transition relation is defined by two rules. In the ordinary case, the semantics fetches the next instruction and then executes it using the auxiliary relation `transition'`:

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

The stopped state simply transitions to itself:

```
trans_stopped : transition stopped stopped.
```

Instruction fetching is performed by loading the string beginning at the current program counter and decoding it to obtain the instruction `IN`. This process is unsurprising, but is fairly involved due to the complexity of the IA-32's instruction encoding.

The main work is done by the helper relation `transition' IN ST ST'`, which says that executing `IN` in state `ST` results in state `ST'`. We give one case 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
      <- putpc ST2 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 program counter, obtaining the final state `ST'`.

2.4 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 system. Most notable of the operations provided by the runtime is memory allocation. Memory allocation is largely straightforward (we simply add a new section to the memory and return an address constructed from its section identifier), except that any allocation may invoke the garbage collector. Garbage collection causes an important complication to the semantics. Although space considerations preclude a full discussion of our approach to modelling garbage collection, we briefly summarize our approach here.

Following Crary [8], we define a notion of *unreachability.* Roughly speaking, a set S of section identifiers is unreachable in a state if that state contains no pointers into sections in S except from other sections in S. Consequently, collection of S leaves no dangling pointers.

Note that a state will typically have many unreachable sets; in particular, the empty set is always unreachable. The definition is crafted in this manner because we cannot predict what objects will actually be collected by a conservative collector. Instead, we use the oracle to cover all possible actions the collector might take.

The semantics garbage-collects by selecting an unreachable set (using the oracle, since many possibilities will exist) and invalidating every section in that set. This is done as part of every operation that might invoke garbage collection (*e.g.,* allocation). Therefore, safe programs must always be prepared for the possibility of a GC in such operations.

2.5 The Safety Policy

To state the safety policy, we need an additional type definition for input programs. We view a program simply as a list of (actual) bytes, giving us the definition:

```
astring : type.
##      : astring.
!       : binary bw -> astring -> astring.
```

Here, astring stands for "actual string"; the operator ! takes on the role of cons and ## that of nil.

A relation initial_state AS ST relates a program to its possible initial states. An initial state is obtained by (1) placing the program into memory, (2) choosing an arbitrary size for the stack, (3) filling the stack and flags with junk values and the registers with appropriate initial values, (4) setting the program counter to the beginning of the program, and (5) choosing an arbitrary value for the oracle.

We can now state the safety policy. First, we say that a program AS can reach a state ST, if ST is reachable in zero or more transitions from an initial state of AS:

```
reachable   : astring -> state -> type.
reachable_z : reachable AS ST
                <- initial_state AS ST.
reachable_s : reachable AS ST2
                <- reachable AS ST1
                <- transition ST1 ST2.
```

Second, we declare, but do not define, a predicate good on programs:

```
good : astring -> type.
```

Recall that the code supplier is responsible for filling in the definition of good.

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

```
safety : good AS
           -> reachable AS ST
           -> transition ST ST' -> type.
%mode safety +D1 +D2 -D3.
```

This metatheorem says that whenever there exists a derivation of good AS (*i.e.,* AS passes the code supplier's safety condition), and there exists a derivation of reachable AS ST (*i.e.,* AS 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 definition of the safety condition good.

The complete safety policy consists of 2404 lines of Twelf code, including comments.

3 A Safety Proof

Next we discuss our proof that TALT adheres to the safety policy above, that is, that TALT provides a safety condition good satisfying the theorem safety.

It will prove to be convenient to discuss the underlying machinery of the proof first, and conclude by discussing TALT's definition for the predicate `good`.

Following Hamid, *et al.* [10], we structure our foundational safety proof for TALT in two stages: an abstract, type-theoretic portion, and a concrete, type-free portion.

3.1 The Abstract Stage

In the first stage, we define the TALT type system and give it an operational semantics in terms of a low-level abstract machine. We then prove type safety for that abstract semantics. The development of the first stage is given in detail in Crary [8] and we will not repeat it here except to summarize its top-level results.

The TALT type system is summarized by the predicate `machineok`, indicating well-typed machine states, and the operational semantics is given by the relation `stepsto`. Crary also defines a relation `collect` specifying garbage collection for the abstract machine (`collect M M'` indicates when GC is invoked in state `M`, a possible result is state `M'` (recall Section 2.4)). Given these, the final results of Crary are three safety theorems:

```
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.

collect_ok : collect M M'
               -> machineok M
               -> machineok M' -> type.
%mode collect_ok +D1 +D2 -D3.
```

The first two are standard safety results [28, 11]: `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. The third, `collect_ok`, asserts a fact about garbage collection: if `M` is well-typed and may garbage-collect to `M'` by deleting an reachable set of sections (recall Section 2.4), then `M'` is also well-typed.

3.2 The Concrete Stage

We complete our foundational safety proof by combining the abstract safety theorems above with a simulation argument showing that the abstract operational semantics maps correctly onto the concrete architecture. The simulation argument is entirely 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 do not attempt to present those technicalities here, and instead give the high-level structure of the proof.

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.
```

Simulation One main lemma of the concrete stage is simulation:

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

This lemma is read as follows: If `ST` implements `M`, and `M` steps to `M'`, then there exists `ST'` such that `ST` transitions to `ST'` in one or more steps, and `M'` garbage-collects to some `M''` that `ST'` implements.

In most cases, the transition from `ST` to `ST'` takes just one step, but TALT supports a few instructions (*e.g.,* `cmpjcc`) that expand to multiple instructions. Also, in most cases, when garbage collection is not invoked, `M'` and `M''` are identical.

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.

3.3 Safety

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 now prove *concrete* progress and preservation, using `ok` as the relevant notion of typeability:

Lemma 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 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` and `preservation`, we have `stepsto M M'` and `machineok M'`. By `simulate`, `transitions _ ST ST''`, `collect M' M''`, and `implements ST'' M''`. By `collect_ok`, `M''` is well-typed, so `ST''` is ok. Finally, by `determinism`, `ST'` $=$ `ST''`, so `ST'` is ok. □

It remains to define a safety condition `good` such that for good programs `AS`, whenever `initial_state AS ST` we have that `ST` implements a well-typed abstract state. This is not difficult, but the details depend on the definition of `implements`, so we cannot present them here. The resulting lemma is:

Lemma 3.

```
initial_ok : good AS
             -> initial_state AS ST
             -> implements ST M
             -> machineok M -> type.
%mode initial_ok +D1 +D2 -D3 -D4.
```

We may now prove that any state reachable from a good program is ok:

Lemma 4.

```
safety' : good AS
          -> reachable AS ST
          -> ok ST -> type.
%mode safety' +D1 +D2 -D3.
```

Proof: (Case `reachable_z`) Suppose `initial_state AS ST`. By `initial_ok`, `ST` implements a well-typed abstract state. Since `ST` transitions in zero steps to itself, `ST` is ok. (Case `reachable_s`) Suppose `reachable AS 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 (first and second stage) for TALT consists of 40370 lines of Twelf code, including comments. It takes approximately 75 seconds to check in Twelf 1.4 on a Pentium 4 with one gigabyte of RAM.

4 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 $\forall x_1 \dots \forall x_m \exists y_1 \dots \exists y_n.P$ 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 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.* [2]). 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 benefits 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.

References

1. Andrew W. Appel and Amy P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *Twenty-Seventh ACM Symposium on Principles of Programming Languages*, pages 243–253, Boston, January 2000.
2. Andrew W. Appel, Neophytos Michael, Aaron Stump, and Roberto Virga. A trustworthy proof checker. Technical Report TR-647-02, Department of Computer Science, Princeton University, April 2002.
3. Andrew Bernard and Peter Lee. Temporal logic for proof-carrying code. In *Eighteenth International Conference on Automated Deduction*, volume 2392 of *Lecture Notes in Artificial Intelligence*, pages 31–46, Copenhagen, Denmark, July 2002. Springer-Verlag.
4. Hans-J. Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, September 1988.
5. Rajkumar Buyya and Mark Baker, editors. *First International Workshop on Grid Computing*, volume 1971 of *Lecture Notes in Computer Science*, Bangalore, India, December 2000. Springer-Verlag.
6. Bor-Yuh Evan Chang, Karl Crary, Margaret DeLap, Robert Harper, Jason Liszka, Tom Murphy VII, and Frank Pfenning. Trustless grid computing in ConCert. In *Third International Workshop on Grid Computing*, volume 2536 of *Lecture Notes in Computer Science*, pages 112–125, Baltimore, Maryland, November 2002.
7. Christopher Colby, Peter Lee, George Necula, and Fred Blau. A certifying compiler for Java. In *2000 SIGPLAN Conference on Programming Language Design and Implementation*, pages 95–107, Vancouver, British Columbia, June 2000.
8. Karl Crary. Toward a foundational typed assembly language. In *Thirtieth ACM Symposium on Principles of Programming Languages*, pages 198–212, New Orleans, Louisiana, January 2003.
9. Karl Crary and Susmit Sarkar. Foundational certified code in a metalogical framework. Technical Report CMU-CS-03-108, Carnegie Mellon University, School of Computer Science, 2003.
10. Nadeem Hamid, Zhong Shao, Valery Trifonov, Stefan Monnier, and Zhaozhong Ni. A syntactic approach to foundational proof-carrying code. In *Seventeenth IEEE Symposium on Logic in Computer Science*, pages 89–100, Copenhagen, Denmark, July 2002.

11. Robert Harper. A simplified account of polymorphic references. *Information Processing Letters*, 51(4):201–206, 1994. Follow-up note in *Information Processing Letters*, 57(1), 1996.
12. Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, January 1993.
13. Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual*, 2001. Order numbers 245470–245472.
14. Craig Lee, editor. *Second International Workshop on Grid Computing*, volume 2242 of *Lecture Notes in Computer Science*, Denver, Colorado, November 2001. Springer-Verlag.
15. Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.
16. Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. In *Second Workshop on Compiler Support for System Software*, Atlanta, May 1999.
17. Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. *Journal of Functional Programming*, 12(1):43–88, January 2002.
18. Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999. An earlier version appeared in the 1998 Symposium on Principles of Programming Languages.
19. George Necula. Proof-carrying code. In *Twenty-Fourth ACM Symposium on Principles of Programming Languages*, pages 106–119, Paris, January 1997.
20. George Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Second Symposium on Operating Systems Design and Implementation*, pages 229–243, Seattle, October 1996.
21. George Ciprian Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania, September 1998.
22. Manish Parashar, editor. *Third International Workshop on Grid Computing*, volume 2536 of *Lecture Notes in Computer Science*, Baltimore, Maryland, November 2002. Springer-Verlag.
23. Christine Paulin-Mohring. Inductive definitions in the system coq—rules and properties. In *International Conference on Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.
24. Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logic framework for deductive systems. In *Sixteenth International Conference on Automated Deduction*, volume 1632 of *Lecture Notes in Computer Science*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag.
25. Frank Pfenning and Carsten Schürmann. *Twelf User's Guide, Version 1.3R4*, 2002. Available electronically at `http://www.cs.cmu.edu/~twelf`.
26. SETI@Home. `http://setiathome.ssl.berkeley.edu`, November 2000.
27. Tool Interface Standards Committee. *Executable and Linking Format (ELF) specification*, May 1995. `http://x86.ddj.com/ftp/manuals/tools/elf.pdf`.
28. Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1994.