Abstract

Foundational certified code systems seek to prove untrusted programs to be safe relative to safety policies given in terms of actual machine architectures. 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 theTwelf 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 and 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 [30] 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, 13, 21] 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.

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 [14, 19, 15] 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 [18, 17, 16, 20], 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, such as the account of Appel and Felty [1], and that of Hamid et al. [9]. 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 [11]. 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 ma-
chine 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 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. [9] is operational, with safety given by conventional type preservation and progress lemmas [32, 10]. However, both systems are similar in that the safety policy and the entire proof are given in a single logic; for the former, higher-order logic encoded in LF [11], and for the latter, the calculus of inductive constructions [22].

**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\(^1\) [12], 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.

The importance of this difference is largely pragmatic. Using the Twelf metalogical framework [25, 26], 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.

\(^1\)Popularly known as the “x86” architecture.

The remainder of the paper is structured as follows: In Section 2 we discuss how to express logics, metatheorems, and meta-proofs in Twelf. In Section 3 we present our system’s architecture semantics and safety policy. In Section 4 we outline our meta-proof that TALT satisfies that policy. Concluding remarks appear in Section 5.

## 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 [23] and were developed further in a variety of papers on Twelf and its predecessor Elf [24, 28, 27, 29, 26]. We assume familiarity with logic programming, and with the methodology of encoding logics in LF [11], 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 given third number. This logic is given in standard mathematical notation in Figure 1.

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

\[ n ::= 0 \mid s(n) \]

\[ 0 + n = n \quad s(n_1) + n_2 = s(n_3) \]

**Figure 1: Natural Numbers**

\[ \begin{align*}
    s & : \text{nat} \rightarrow \text{nat} \\
    0 & : \text{nat} \\
    \text{sum} & : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat} \rightarrow \text{type}
\end{align*} \]

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:

\[ \text{sum} \ (s \ 0) \ (s \ (s \ 0)) \ (s \ (s \ (s \ 0))) \]

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

\[ \text{sum_s sum_z} \]
Implicit arguments. Note Twelf’s use of implicit arguments to express the rules $\text{sum}_z$ and $\text{sum}_s$ in a convenient form. In explicit form, the implicit argument $N$ to $\text{sum}_z$ (for example) would be explicit, resulting in the type:

$$\{N:\text{nat}\} \text{ sum O N N}$$

(where $\{N:\text{nat}\}$ is the Twelf rendering of the universal quantifier $\forall N:\text{nat}$, and every use of $\text{sum}_z$ 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 $\text{sum}_z$ is used.

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

$$\text{sum}_s 0 (s (s 0)) (s (s 0)) (\text{sum}_z (s 0))$$

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 inhabitable).

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

?- $\text{sum} (s z) (s (s z)) N$.
Solving...
$N = s (s z)$.

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 $\text{sum}_z$, $N$ 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, we may obtain a Prolog 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:

$$\text{sum}(0, N, N).$$
$$\text{sum}(s(N1), N2, s(N3)) :- \text{sum}(N1, N2, N3).$$

2.3 Meta-Proofs

An interesting fact about the $\text{sum}$ 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, $\text{sum}$ is a total relation from its first two positions to its third.

The totality of $\text{sum}$ 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:

$$\text{sum}_{\text{ident}} : \{N:\text{nat}\} \text{ sum O N N} \rightarrow \text{type}.$$  
$$\text{sum}_{\text{ident}} z : \text{sum}_{\text{ident}} 0 \text{ sum}_{\text{z}}.$$  
$$\text{sum}_{\text{ident}} s : \text{sum}_{\text{ident}} (s N) (\text{sum}_{\text{s}} D) \leftarrow \text{sum}_{\text{ident}} N D.$$  
$$\text{sum}_{\text{inc}} : \text{sum} N1 N2 N3 \rightarrow \text{sum} N1 (s N2) (s N3) \rightarrow \text{type}.$$  
$$\text{sum}_{\text{inc}} z : \text{sum}_{\text{inc}} \text{ sum}_{\text{z}} \text{ sum}_{\text{z}}.$$  
$$\text{sum}_{\text{inc}} s : \text{sum}_{\text{inc}} (\text{sum}_{\text{s}} D) (\text{sum}_{\text{s}} D') \leftarrow \text{sum}_{\text{inc}} D D'.$$

Each of these is a total relation from its first position to its second and therefore corresponds to a metatheorem. The first, $\text{sum}_{\text{ident}}$, states that for any natural number $n$, there exists a derivation of $n + 0 = n$. The second, $\text{sum}_{\text{inc}}$, 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, the names we give to the cases of a metatheorem are never significant except for Twelf’s error reporting. Therefore we will save space in this paper by eliding them and replacing them with the symbol $\&$.

Using $\text{sum}_{\text{ident}}$ and $\text{sum}_{\text{inc}}$ as lemmas, we may prove a mildly interesting metatheorem:

$$\text{sum}_{\text{commute}} : \text{sum} N1 N2 N3 \rightarrow \text{sum} N2 N1 N3 \rightarrow \text{type}.$$  
$$\& : \text{sum}_{\text{commute}} (\text{sum}_{\text{z}} : \text{sum} O N N) D \leftarrow \text{sum}_{\text{ident}} N (D : \text{sum} N O N).$$  
$$\& : \text{sum}_{\text{commute}} ((\text{sum}_{\text{s}} D) : \text{sum} (s N1) N2 (s N3)) D' \leftarrow \text{sum}_{\text{commute}} D (D' : \text{sum} N2 N1 N3) \leftarrow \text{sum}_{\text{inc}} D D'.$$

$\text{sum}_{\text{commute}}$ is total from its first position to its second. 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.

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 $\text{sum}_{\text{zero}}$:  

$$\text{sum}_{\text{zero}} : \rightarrow \text{type}.$$
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.

  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.

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

  &: sum_zero 0 sum_z.

  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 simply from ordinary relations by the presence of a mode declaration.

3 The Safety Policy

The main body of the safety is an operational semantics for the concrete architecture. For our work we have chosen to use the Intel IA-32 architecture, as it the architecture used by the greatest number of potential grid participants. We have begun with a fairly small number of instructions, but new instruction are easy to add to the superstructure we have built.

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 thus no unsafe operation is performed. This code formalizing this, which concludes the safety policy, is given in Section 3.5.

Indeterminism An important issue complicating the operational semantics is indeterminism. In some cases it is impossible to determine the outcome of an operation. This may happen because 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). 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 defying 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 program states provides every possible finite oracle, 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 the facilities (such as allocation) provided by the runtime [31]. The other is
a special “don’t know” value used to represent various ill-informed, abstract data, such as the sum of two pointers.

### 3.1 Data

We use two notions of numbers in our semantics. One is the natural numbers from Section 2.1. The other is binary \( N \), which contains \( N \)-bit binary numbers. Given these, we can define our data types:

- \( \text{bw} : \text{n} = 8 \). % % byte width in bits
- \( \text{ww} : \text{n} = 4 \). % % word width in bytes
- \( \text{wwb} : \text{n} = 32 \). % % word width in bits

\( \text{section} : \text{type} \)
\( \text{sect} : \text{n} \rightarrow \text{section} \)

\( \text{address} : \text{type} \)
\( \text{addr} : \text{section} \rightarrow \text{binary} \text{wwb} \rightarrow \text{address} \)

\( \text{byte} : \text{type} \)
\( \text{act} : \text{binary} \text{bw} \rightarrow \text{byte} \). % % actual
\( \text{dk} : \text{byte} \). % % don’t know
\( \text{boa} : \text{address} \rightarrow \text{n} \rightarrow \text{byte} \).

\( \text{boga} : \text{n} \rightarrow \text{byte} \). % % byte of GOT addr.

\( \text{string} : \text{n} \rightarrow \text{type} \)
\( \# : \text{string} \ ) 0.
\( / : \text{byte} \rightarrow \text{string} \text{ww} \rightarrow \text{string} (\text{s} \text{ww}) \).

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. Thus, the pointer \( A \) is represented as the sequence of bytes \( \text{boa} A 0, \ldots, \text{boa} A 3 \).

The most common data type in the semantics is string, which contains strings of bytes constructed using \( / \) for cons\(^3\) 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.\(^4\)

### 3.2 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 3.1, 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:

\( \text{memory} : \text{type} \)
\( \text{mnnil} : \text{memory} \)
\( \text{mcons} : \text{string} _ \rightarrow \text{memory} \rightarrow \text{memory} \)
\( \text{minv} : \text{memory} \rightarrow \text{memory} \)

---

\( ^3\)The curious choice of name for the cons constant is justified by making / infix, so that a word may be written byte1 / byte2 / byte3 / byte4 / #.

\( ^4\)There is an important distinction between the wildcard _, and implicit arguments such as \( \$ \). 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.

A section may become invalidated, such as when it is garbage collected. Since sections are looked up by their position, we use minv as a placeholder for invalid sections.

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:

\( \text{numregs} : \text{n} = 8 \)
\( \text{regs} \rightarrow \text{type} \)
\( \text{regs Nil} : \text{regs} \)
\( \text{regs cons} : \text{string} \text{ww} \rightarrow \text{regs} \text{n} \rightarrow \text{regs (s} \text{n}) \).

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

\( \text{state} : \text{type} \)
\( \text{state} \rightarrow \text{memory} \)
\( \rightarrow \text{regs numregs} \)
\( \rightarrow \text{flags} \)
\( \rightarrow \text{address} % % \text{program counter} \)
\( \rightarrow \text{oracle} \rightarrow \text{state} \).

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:

\( \text{stopped} : \text{state} \).

### 3.3 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\(^*\):

\( \text{transition} : \text{state} \rightarrow \text{state} \rightarrow \text{type} \)
\( \text{transition} : \text{transition} \text{ST} \text{ST'} \)
\( \leftarrow \text{fetch} \text{ST} \text{IN} \)
\( \leftarrow \text{transition} \text{IN} \text{ST} \text{ST'} \).

The stopped state simply transitions to itself:

\( \text{trans stopped} : \text{transition stopped stopped} \).

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 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:

\( ^5\)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.
trans_add : 
transition* (ii_add E 0) ST ST' 
<- load ST E W1 
<- oload ST 0 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 0 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 0, 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'.

3.4 Garbage Collection

Garbage collection causes an important complication to the semantics. Although space considerations preclude a full discussion of our approach to 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.

3.5 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 bv -> 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.  
reachables : reachable AS ST2  
<- reachable AS ST1  
<- transition ST1 ST2.  
```

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

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

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  
<- stepsto M M' -> 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). The code supplier is responsible for filling in the proof of safety, and in so doing establishes the soundness of his or her safety condition.

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

4 A Safety Proof

As in Hamid, et al. [9], our foundational safety for TALT is broken into two stages, an abstract, type-theoretic portion, and a concrete, type-free portion.

4.1 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 final results of Crary [8] 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.  
```

6
The first two are standard safety results [32, 10]: 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, asserts a fact about garbage collection: if \( M \) is well-typed and garbage-collects to \( M' \) by deleting an reachable set of sections (recall Section 3.4), then \( M' \) is also well-typed.

### 4.2 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 \( \text{implements} \ ST \ M \), which states that the concrete state \( ST \) implements the abstract state \( M \). Second, we define a multi-step transition relation \( \text{transitions} \ N \ ST \ ST' \), which states that \( ST \) transitions to \( ST' \) in exactly \( N \) steps:

\[
\begin{align*}
\text{transitions} & : \text{nat} \to \text{state} \to \text{state} \to \text{type}. \\
\text{transitions}_z & : \text{transitions} \ 0 \ ST \ ST. \\
\text{transitions}_s & : \text{transitions} \ (s \ N) \ ST_1 \ ST_3 \\
& \quad \leftarrow \text{transition} \ ST_1 \ ST_2 \\
& \quad \leftarrow \text{transitions} \ N \ ST_2 \ ST_3. 
\end{align*}
\]

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

\[
\begin{align*}
\text{simulate} & : \text{implements} \ ST \ M \\
& \quad \to \text{stepsto} \ M \ M' \\
& \quad \to \text{transitions} \ (s \ N) \ ST \ ST' \\
& \quad \to \text{collect} \ M' \ M'' \\
& \quad \to \text{implements} \ ST' \ M''' \to \text{type}. \\
\text{%mode simulate} & \ +D1 \ +D2 \ -D3 \ -D4 \ -D5.
\end{align*}
\]

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 instruction (\( e.g. \), 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:

\[
\begin{align*}
\text{state_eq} & : \text{state} \to \text{state} \to \text{type}. \\
\text{state_eq} & : \text{state_eq} \ ST \ ST. \\
\text{determinism} & : \text{transition} \ ST \ ST_1 \\
& \quad \to \text{transition} \ ST \ ST_2 \\
& \quad \to \text{state_eq} \ ST_1 \ ST_2 \to \text{type}. 
\end{align*}
\]

The relation \( \text{state_eq} \ ST_1 \ ST_2 \) holds exactly when \( ST_1 \) and \( ST_2 \) are identical. Therefore the lemma is read as follows: If \( ST \) transitions to \( ST_1 \), and \( ST \) transitions to \( ST_2 \), then \( ST_1 \) and \( ST_2 \) are identical.

### 4.3 Safety

We say that a concrete state \( ST \) is \textit{ok} if \( ST \) transitions in zero or more steps to some \( ST' \) that implements a well-typed abstract state:

\[
\begin{align*}
\text{ok} & : \text{state} \to \text{type}. \\
\text{ok_} & : \text{ok} \ ST \\
& \quad \leftarrow \text{transitions} \ ST \ ST' \\
& \quad \leftarrow \text{implements} \ ST' \ M \\
& \quad \leftarrow \text{machineok} \ M.
\end{align*}
\]

We now prove \textit{concrete} progress and preservation, using \textit{ok} as the relevant notion of typeability:

**Lemma 4.1**

\[
\begin{align*}
\text{iprogress} & : \text{ok} \ ST \\
& \quad \to \text{transition} \ ST \ ST' \to \text{type}. \\
\text{%mode iprogress} & \ +D1 \ -D2.
\end{align*}
\]

**Proof:** Since \( ST \) is ok, it steps to some \( ST' \) in some \( N \) steps. If \( N \geq 1 \), the result is immediate. Otherwise \( ST = ST' \), so \( \text{implements} \ ST \ M \) and \( \text{machineok} \ M \). By \text{progress}, \text{stepsto} \ M \ M' \), and therefore by \text{simulate}, \( ST \) takes a step. \( \square \)

**Lemma 4.2**

\[
\begin{align*}
\text{ipreservation} & : \text{ok} \ ST \\
& \quad \to \text{transition} \ ST \ ST' \\
& \quad \to \text{ok} \ ST' \to \text{type}. \\
\text{%mode ipreservation} & \ +D1 \ +D2 \ -D3 \ -D4.
\end{align*}
\]

**Proof:** Since \( ST \) is ok, it steps to some \( ST' \) (which implements a well-typed abstract state) in some \( N \) steps. Suppose \( N \geq 1 \). Then \( \text{transition} \ ST \ ST_1 \) and \( \text{transitions} \ ST \ ST' \). By determinism, \( ST' = ST_1 \), and \( ST_1 \) is ok, so \( ST' \) is also ok.

Suppose \( N = 0 \). Then \( ST \) implements a well-typed abstract state \( M \). By \text{progress} and \text{preservation}, we have \text{stepsto} \ M \ M' \) and \text{machineok} \ M' \). By \text{simulate}, \( \text{transitions} \ ST \ ST' \), \text{collect} \ M' \ M'' \), and \text{implements} \ ST' \ M'' \). By \text{collect_ok}, \( M'' \) is well-typed, so \( ST'' \) is ok. Finally, by determinism, \( ST' = ST'' \), so \( ST' \) is ok. \( \square \)

The machine-checkable proofs of \textit{iprogress} and \textit{ipreservation}, as well as safety* and safety (below), appear in Appendix A.

It remains to define a safety condition \textit{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 \text{implements}, so we cannot present them here. The resulting lemma is:

**Lemma 4.3**

\[
\begin{align*}
\text{initial\_ok} & : \text{good} \ AS \\
& \quad \to \text{initial\_state} \ AS \ ST \\
& \quad \to \text{implements} \ ST \ M \\
& \quad \to \text{machineok} \ M \to \text{type}. \\
\text{%mode initial\_ok} & \ +D1 \ +D2 \ -D3 \ -D4.
\end{align*}
\]

We may now prove that any state reachable from a good program is ok:
Lemma 4.4
\[
\text{safety}^* : \text{good AS} \\
\rightarrow \text{reachable AS ST} \\
\rightarrow \text{ok ST} \rightarrow \text{type}.
\]
\text{Proof:} (Case reachable) 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) Suppose reachable AS ST' and transition ST' ST. By induction, ST' is ok, so by ipreservation, ST is ok.

Using ipprogress, 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.

5 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 metalogical approach, at least as things stand today. First, in the Twelf metalogic one is limited to II-1 reasoning (i.e., reasoning involving only propositions of the form \( \forall x_1 \ldots \forall x_n \exists y_1 \ldots \exists y_m \, P \) 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 II-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.

Nevertheless, we believe the benefits of the 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 certified code systems. For example, one might show that one safety policy (even one not expressed in metalogic) 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


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 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.

safety* : good AS -> reachable AS ST -> ok ST -> type.
%mode safety* +D1 +D2 -D3.
&: safety* Dgood (reachable_z Dinitial) (ok_ Dmok Dimplements transitions_z)
  <- initial_ok Dgood Dinitial Dimplements Dmok.
&: safety* Dgood (reachable_s Dtrans Dreach) Dok'
  <- safety* Dgood Dreach Dok
  <- ipreservation Dok Dtrans Dok'.

safety : good AS -> reachable AS ST -> transition ST ST' -> type.
%mode safety +D1 +D2 -D3.
&: safety Dgood Dreach Dtrans
  <- safety* Dgood Dreach Dok
  <- iprogress Dok Dtrans.