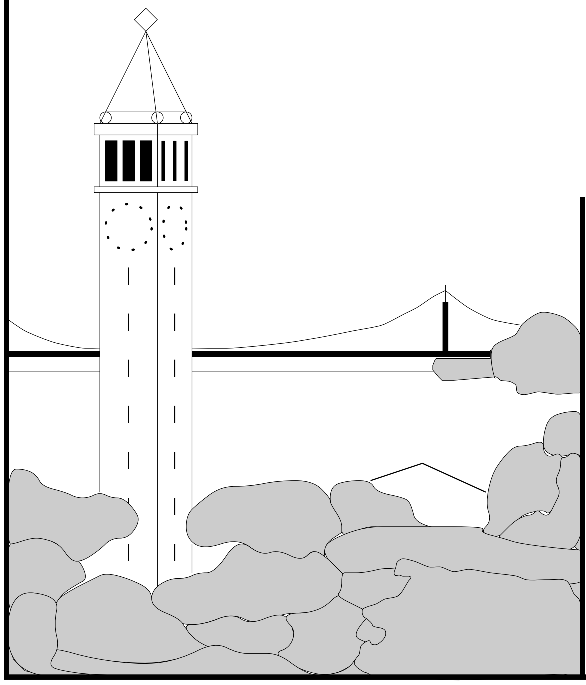
An Untrusted Verifier for Typed Assembly Language

Adam Chlipala



Report No. UCB/ERL-M04/41

December 2004

Computer Science Division (EECS)
University of California
Berkeley, California 94720

An Untrusted Verifier for Typed Assembly Language*

Adam Chlipala
adamc@cs.berkeley.edu
Department of Electrical Engineering and Computer Science
University of California, Berkeley

December 2004

Abstract

I present the results of constructing a fully untrusted verifier for memory safety of Typed Assembly Language programs, using the Open Verifier architecture. The verifier is untrusted in the sense that its soundness depends only on axioms about the semantics of a concrete machine architecture, not on any axioms specific to a type system. This experiment served to evaluate both the expressiveness of the Open Verifier architecture and the quality of its support for simplifying the construction of verifiers. I discuss issues of proof generation that are generally not the focus of previous efforts for foundational checking of TAL, and I contrast with these past approaches the sort of logical formalization that is natural in the context of the Open Verifier. My approach is novel in that it uses direct reasoning about concrete machine states where past approaches have formalized typed abstract machines and proved their correspondence with concrete machines. I also describe a new approach to modeling higher-order functions that uses only first-order logic.

1 Introduction

It is increasingly common for today's computing systems to run software produced by potentially untrustworthy sources. Such *mobile code* may run in many different settings, from fast personal computers to heavily resource-constrained mobile phones. Naturally, it is important that users be able to place some degree of trust in mobile code if they are to be expected to run it.

The Java Virtual Machine Language (JVML) [LY99] and the Microsoft Common Intermediate Language (CIL) [GS01, Gou02], two popular mobile code platforms, use

*This research was supported in part by NSF Grants CCR-0326577, CCR-0081588, CCR-0085949, CCR-00225610, and CCR-0234689; NASA Grant NNA04CI57A; a Sloan Fellowship; an NSF Graduate Fellowship; an NDSEG Fellowship; and California Microelectronics Fellowships. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

access control mechanisms [WF98] that depend on type correctness of code. It is straightforward to guarantee this by running these high-level bytecode programs within a “sandbox” that forces them to interact with the host environment through authorized means only. However, especially on small mobile and embedded hardware platforms with very limited battery life, the cost of interpretation can be prohibitive. Even on high-end PC’s, there is much efficiency to be gained by avoiding interpretation.

Just-in-time (JIT) compilation is a popular answer to these concerns. A program in a safe high-level bytecode language like JVML or CIL is compiled to native code before being executed. The JIT compiler is trusted to preserve the desired security properties of programs it translates. While the full range of native code programs might be able to circumvent the access control mechanisms of the mobile code platform, end users trust that the JIT compiler will never generate such code. JIT compiler implementation involves enough subtleties that this can be a risky proposition. It is also true that end users tend to be impatient. They will likely not be willing to wait the time it would take a JIT compiler to produce optimized native code on par with that produced by a good compiler for the native platform. This difference in efficiency can mean a crucial savings of battery power for resource-constrained platforms.

It is also the case that these high-level bytecode languages inevitably sacrifice expressiveness for tractability of checking. To avoid dealing with complicated low-level details, they will do things such as have separate instructions for different varieties of function calls. Despite CIL’s eight call instructions, there are still requests for the addition of new instructions to support efficient compilation of different language features [Sym01]. Yet all of these types of calls have traditionally been implemented with just a few basic machine instructions as building blocks.

Proof-carrying code (PCC) [Nec97] was developed to address these deficiencies. PCC provides a means for evaluating directly the trustworthiness of native code: each such program comes with a *proof* of the properties that the end user requires hold. The original PCC approach relied on trusted proof rules specific to a type system for a safe source language. A subsequent approach known as *Foundational PCC* (FPCC) [App01] opts instead to trust only concrete machine semantics and the safety policy. Nonetheless, FPCC work to date has tended to focus on attempting to construct a single, though untrusted, type system usable to certify the safety of all source languages.

The *Open Verifier* project takes another approach to foundational-style verification of native code. Instead of attempting to produce compilers for all source languages into machine code describable by a single low-level type system, we allow the use of different verification strategies for different compilers. Instead of fixing the *mechanism* for code safety, such as a particular type system, we fix only the *policy*, such as memory safety. The Open Verifier architecture is designed to allow easy adaptation of existing verifiers written in the style of the Java Bytecode Verifier, while still maintaining the full power of PCC. An *extension* for a particular safety mechanism answers queries from the trusted core of the Open Verifier to guide the verification process. The extension must provide proofs of the soundness of its actions where necessary. The result is that a code consumer need only trust the core of the Open Verifier, not any particular extension.

Previous work on the Open Verifier [Sch04] had been successful in implementing extensions for a Java-like language, verified using abstract interpretation on types; and annotated C code, verified in traditional PCC style. To test the expressiveness of our architecture as compared to previous foundational systems, I constructed an extension for a Typed Assembly Language (TAL) [MWCG99], the preferred subject of these systems. I have found the architecture to be quite sufficient and natural for this task. In the process of implementing the extension, I also discovered a way of encoding function calls that allows the removal of the most complex element of the architecture.

In the following section, I provide more detailed background on TAL, proof-carrying code, and the Open Verifier. Next, I introduce the logical formalization and describe the verifier extension that uses it. I generalize one aspect of the extension, its handling of control flow, to extract the fundamental pieces of the approach. Finally, I describe the implementation and conclude.

2 Background

2.1 Typed Assembly Language

Morrisett et al. [MWCG99] proposed the use of *typed assembly languages* in certifying compilation. A typed assembly language (or *TAL*) is an extension of the native assembly language of a particular architecture. A typical TAL introduces a few macroinstructions not found in the native assembly language and adds type annotations and coercions to standard instructions. By choosing its type system and instructions carefully, it is possible to design a TAL that may be type-checked for safety using small variations on well-known techniques applicable to higher level languages.

My work has focused on the TALx86 variant [MCG+99]. The implementation handles a subset of the TAL supported by the first release of the TALC tools [MCG+03] from Cornell. To simplify verification, I have ignored all of TALx86's features for modular verification, including kinds and abstract types. Prior foundational work on TALs has not focused on such features, either. A straightforward linking process can combine TAL sources that use these features into a single file that does not.

2.1.1 Basic blocks

The majority of a TALx86 program is a set of labeled basic blocks. Each block comes with a precondition stating assumed types for a subset of the registers. These preconditions are expressed as code types which may use universal quantification, as discussed in more detail below.

For example, consider:

$\texttt{lab1} : \forall\alpha.\mathsf{code}\{\mathsf{eax} : \mathsf{word}, \mathsf{ebx} : \alpha\}$
MOV $\mathsf{ebx} := \mathsf{eax}$
JUMP $\texttt{lab2}$

This block of code, named `lab1`, requires that register eax hold a value of type word and ebx hold a value of some parameter type α whenever it is reached. The universally quantified variable α is bound in the block's body.

2.1.2 Types

Basic types

$$\tau \quad ::= \quad \mathsf{word}$$

The most basic TALx86 type is word, the type of arbitrary machine words. These are 32-bit integers that fit inside registers but are not subject to any other constraints. In particular, 32-bit values that address memory will be given different types.

Stack types

$$\tau \quad ::= \quad \ldots \mid [] \mid \tau_1 :: \tau_2 \mid \mathsf{stackptr}(\tau)$$

In addition to types that describe values that fit into registers, stack-based TALs [MCGW98] like TALx86 also include types to describe entire stacks. The [] constructor describes an empty stack, and $\tau_1 :: \tau_2$ describes a stack that is like τ_2 but has a value of type τ_1 pushed onto its top. The type $\mathsf{stackptr}(\tau)$ describes a word-sized pointer to a stack of type τ.

The full TALx86 uses a system of *kinds* to invalidate nonsensical types like $\mathsf{word} :: \mathsf{word}$ and $\mathsf{stackptr}(\mathsf{word})$, but it turns out that such types are safe to allow for whole-program verification. It will simply be impossible to prove that any value has such a type.

Code pointer types

$$\tau \quad ::= \quad \ldots \mid \mathsf{code}\{r_1 : \tau_1, ..., r_n : \tau_n\}$$

The type $\mathsf{code}\{r_1 : \tau_1, ..., r_n : \tau_n\}$ denotes a pointer to a segment of program code that expects each register r_i to contain a value of type τ_i. Such types will be sufficient to describe functions with a wide variety of calling conventions. For instance, suppose that we wish to type a function that takes a word argument in eax, performs some computation that leaves its result in ecx, and returns to the calling address saved in ebx. Using continuation-passing style, we can give that function the type:

$$\mathsf{code}\{\mathsf{eax} : \mathsf{word}, \mathsf{ebx} : \mathsf{code}\{\mathsf{ecx} : \mathsf{word}\}\}$$

Universal types

$$\begin{array}{rcl} \alpha, \beta & & \text{Type variables} \\ \tau & ::= & \ldots \mid \forall \alpha . \tau \end{array}$$

$\forall \alpha . \tau$ is analogous to the types assigned to parametrically polymorphic functions in high level languages like ML. By definition, any value of this type also has type $[\tau'/\alpha]\tau$ for any type τ', where the preceding notation denotes the result of substituting τ' for every free occurrence of α in τ. The full TALx86 annotates each universal type with a kind requirement on its bound variable, but this again turns out to be unnecessary for whole-program verification. Since TALx86 only permits universal quantifiers around code types, it is safe to allow any instantiations, since the worst that can happen is that a value is proved to have a code type with a nonsensical condition on register types that cannot possibly be met.

Besides being useful in compilation of source languages with parametric polymorphism, universal types are also critical in encoding calling conventions. TALx86 uses them to encode both the stack discipline and the preservation of callee-save registers for the standard x86 C calling conventions. For example, this type describes a procedure of no arguments and no return value that treats ecx as a callee-save register and returns with the stack in the same state as when the procedure was called:

$$\forall\alpha.\forall\beta.\mathsf{code}\{\mathsf{ecx}:\alpha,\mathsf{esp}:\mathsf{stackptr}(\mathsf{code}\{\mathsf{ecx}:\alpha,\mathsf{esp}:\mathsf{stackptr}(\beta)\}::\beta)\}$$

α stands for an arbitrary type for the value of ecx on entry to the procedure. Like in ML, the only valid operation on a value with a parametrically bound type is to copy it from one place to another. This allows the procedure to use whatever method it deems best for preserving the value of ecx across the call, including storing it in another register, in the stack, or in the heap. However, the procedure can never "counterfeit" the value; it is impossible to construct any new value and prove it has type α.

Similarly, β stands for the type of the "irrelevant" part of the stack that the procedure will not need to read, but whose value it must nonetheless preserve. The procedure expects esp to point on entry to a stack that begins with the return pointer, followed by this unknown suffix β. The return pointer uses the bound type variables to require that ecx and the stack have the same values as when the procedure was called.

Product types

$$\begin{array}{rcl} i & ::= & \mathsf{init} \mid \mathsf{uninit} \\ f & ::= & \tau^i \\ \tau & ::= & \ldots \mid f_1 \times \ldots \times f_n \end{array}$$

Product types denote mutable tuples with components of potentially heterogeneous types. We assume that newly allocated product values contain useless junk values. *Initialization flags* i indicate whether a field of a product has yet been assigned a useful value. A *field* f describes a product field with its type and initialization flag.

For example, the following TAL code allocates and initializes a product. Note the use of the macro-instruction MALLOC that is not found in the real x86 instruction set. One of its parameters is a list of types for the fields of the new product.

```
MALLOC eax : word × word    ; Allocate a new pair of words, storing its address in eax
MOV [eax] := 1              ; Initialize the first field
MOV [eax + 4] := 2          ; Initialize the second field
```

Recursive types

$$\begin{array}{lll} \ell & & \text{Type labels} \\ \tau & ::= & \ldots \mid \ell \end{array}$$

TALx86 programs may define sets of global, mutually recursive named types. Labels ℓ are used to refer to these types in instructions. The unroll and roll coercions replace a label with its definition and vice versa, respectively.

Other types The implementation supports a few more of TALx86's features, including sum, array, and existential types and parametric recursive types. I will not describe them in detail, as the types given so far are sufficient to present the results of this work.

2.2 Proof-carrying code

Necula and Lee's original proof-carrying code formulation [Nec97] was based on the idea of a trusted *verification condition generator* (VCGen). The VCGen takes a native program as input and constructs a formula that implies the safety of that program with respect to some safety policy. The code consumer calls an untrusted proof-generating theorem prover to prove this verification condition. If a trusted proof checker says that the proof is a valid proof of the verification condition, then the code consumer believes that the program is safe.

Naturally, it is impractical to expect any algorithm to generate safety proofs for arbitrary safe machine code. Instead, PCC is used with a subset of possible programs, those generated by custom *certifying compilers* [NL98]. Certifying compilers preserve enough information throughout the compilation process to be able to package additional annotations with their final machine code. These annotations provide hints that can be used by the code consumer to make safety checking more efficient.

For example, the original PCC implementation involved a certifying compiler that determined logical invariants for all loops and included them with its output. Providing these invariants is analogous to giving away the induction hypotheses in a proof by induction; the remaining deduction is often routine. Indeed, armed with these invariants, the theorem prover is able to fill in the remaining details with good efficiency.

Traditional PCC requires the code consumer to trust in the soundness of a logical development of significant size. For instance, the standard certifying compiler derives its annotations from typing derivations for the safe source language it compiles. The first PCC systems asserted without proof the truth of a large number of non-trivial lemmas about these type systems. The type system and compilation strategy in general were hard-coded into many parts of the trusted system. For example, the verification condition generator of the Touchstone compiler [CLN$^+$00] incorporated knowledge of how a particular Java compiler implemented exceptions. One might expect that it is unlikely that such a large theory could be formalized soundly without mechanical checking. Indeed, League [LST03] discovered such a soundness bug in this early PCC implementation.

More recent PCC research has focused on reducing the size of the *trusted computing base* (TCB) to reduce the chance that such flaws will go undetected. Along these lines, Appel and Felty [AF00] proposed *foundational proof carrying code* (FPCC). In an FPCC system, the trusted computing base consists only of a model of machine semantics and the safety policy, along with a trusted proof checker. The unwieldy VCGen and all facts specific to a safe compilation strategy have been removed.

FPCC work has focused on certifying compilers that emit a variety of TALs. Appel et al. chose to use a “semantic” approach in modeling TAL type systems. Although in all cases they only wanted to check code using specific, fixed type systems, they opted to use a more general notion encompassing all possible type systems. They defined a type as a predicate on machine states and values that satisfies a set of well-formedness conditions. While this approach is very expressive, it introduces many complications and can lead to very large proofs. To handle both recursive types and mutable references, Appel and McAllester [AM01] were forced to propagate the notion of an *index* through most of their judgments. Intuitively, a judgment indexed by natural number i states that a certain property is true for at least i more steps of execution. Strangely, this idea is not familiar from standard descriptions of type-checking, and it had no analogue in traditional PCC.

Hamid et al. [HST$^+$02] proposed an alternate approach called *syntactic FPCC*. They reason about code safety relative to a fixed set of possible types, defined syntactically as in a compiler. Both varieties of FPCC are based around a notion of *progress* and *preservation* theorems, lifted from standard techniques for formalizing high-level languages. They define notions of abstract machine states, similar to those maintained in traditional TAL type-checkers; and define *global invariant* predicates that give well-formedness conditions on those states. The safety policy is modeled by defining an abstract operational semantics that disallows unsafe instruction executions.

A *progress* theorem shows that a safe transition is possible from any valid state. A *preservation* theorem shows that any transition from a valid state leads to another valid state. To construct a foundational safety proof, a typing derivation is constructed for the program in question. The typing rules cannot be arbitrary; they are proved, assuming only the facts about the native machine and the safety policy that are included in the trusted computing base. With a typing derivation, the progress and preservation theorems show the safety of the program with the abstract semantics. A final key theorem connects the abstract and concrete semantics to derive foundational safety.

Because they use a more restricted model, Hamid et al. are able to avoid much of the complexity of Appel’s approach. Like in standard deductive systems for high-level type systems, their judgments depend on a *context* that assigns types to memory locations. This “breaks the cycles” that motivate Appel’s use of indexed judgments. Since a particular memory location may be declared to contain a value of a particular recursive type, there is no need for complicated judgments based only on the “shape” of a portion of memory. This style can be called *intensional*, in contrast to Appel’s *extensional* style.

Though these approaches succeed in removing facts about particular type systems from the TCB, in practice, their presentations have been based around typed assembly or machine languages that are meant to be good targets for multiple source languages. These low-level languages are often much more expressive than the JVM and CIL bytecode languages, but none is an ideal compilation target for all source languages.

TALs often make it awkward to encode language features not previously considered, as evidenced by an attempt to compile Java to the standard TAL of a syntactic FPCC system [LST02]. Very general systems like the typed machine language of Appel et al. [CWAF03] require the use of the complicated logical machinery of semantic FPCC, even when it might be easier to construct a customized first-order logic formalization from scratch.

In general, FPCC research has not focused on the feasibility of constructing complete verification systems for different source languages and compilation strategies. As previously mentioned, targeting one low-level format from many source languages can be tedious and awkward. If it becomes awkward enough to be infeasible from a development or performance standpoint, the compiler writer faces the daunting task of redoing a great deal of work. He must rebuild an alternate version of the formal development associated with the unsuitable target language.

We can simplify this process a great deal by extracting parts of a formalization that we expect to be useful in many other cases. The Open Verifier project starts with this idea and works towards the goal of providing an infrastructure that simplifies the construction of new foundational verifiers as much as possible.

2.3 The Open Verifier

The Open Verifier [CCNS05] is an architecture providing a means for untrusted code producers to guide the verification of their code. A code consumer fixes a safety policy of interest without specifying how the property should be enforced or proved for individual programs. Providing the enforcement mechanism is the responsibility of the code producer, who includes with his code an executable *extension* that provides hints on how to verify the code.

The basic idea behind the architecture is that past verification schemes for safety properties can be viewed as using one general strategy. On a given verification run, each scheme produces a set $\mathcal{E}$ of completely local *invariants*, which are predicates on execution states at particular program points. To prove that a safety policy is respected, one of these schemes proceeds to show three things:

- Every allowable initial state for the program satisfies some member of $\mathcal{E}$.
- Every execution step possible from any state satisfying a member of $\mathcal{E}$ avoids violating the safety policy.
- Every such execution step leads to a state also satisfying some member of $\mathcal{E}$.

During verification, the trusted core of the Open Verifier queries the extension about how to proceed in constructing $\mathcal{E}$. The extension may begin by specifying an arbitrary initial set of invariants $\mathcal{E}_0$. This set could be one that already has the properties listed above, but it is more common to provide a smaller set and construct the final $\mathcal{E}$ in an online manner. For instance, the TALx86 extension chooses the set of all function entry points. The extension uses a special language of *scripts* to show that some concrete initial invariant implies that some member of $\mathcal{E}_0$ is true. Scripts allow the use of a set of sound actions that transform an invariant into another that is logically weaker or equivalent. Some of the actions require formal proofs of properties that guarantee their soundness.

Once $\mathcal{E}_0$ is specified, the trusted core of the Open Verifier can perform a standard abstract interpretation. If the interpretation reaches a fixed point $\mathcal{E}$ with the properties above, then we know that the program in question is safe. The Open Verifier determines a transition relation on invariants using a trusted instruction decoder and strongest postcondition generator. This transition relation is dictated by the precise execution semantics of the underlying machine, which means that it will maintain much more information than any practical verification strategy requires. For most programs, infinitely many states are reachable from $\mathcal{E}_0$ via the concrete transition relation.

The Open Verifier copes with this by computing transitions in a two-step process. To compute the logical successor states of a particular state, the concrete transition is first applied, using the strongest postcondition operation. Next, the extension is queried about *which information it would like to forget* in each of these successors. This "forgetting" can be viewed as a logical weakening of invariants.

An extension answers queries in a special language of *scripts*. Scripts allow such sound weakening actions as abstracting a value using existential quantification, forgetting a particular logical fact, and replacing the value of a register with a provably equal value. There are also facilities for case analysis, registering reached states by name, and asserting that a transition needn't be followed because it leads to a particular registered state. I will present the specifics of scripts by example as they come up in later sections.

One of our goals in the Open Verifier project is to simplify the effort required to create new foundational verifiers. To this end, we have tried to keep simple the logical formalisms that are exposed to the end user. Ideally, a good fraction of industrial software engineers should be able to use a polished version of our framework to construct verifiers for the outputs of their companies' compilers. One way that we aid this is through building into our system the skeleton of a verifier, as described above. To our knowledge, all previous foundational verification strategies fit within this framework. However, we also introduce two additional simplifications: a restricted language of invariants and a restricted language of weakening proofs.

We restrict every invariant to be a predicate of the form:

$$\exists x_1 : \tau_1, ..., x_n : \tau_n. \bigwedge_i \mathsf{r_i} = e_i \wedge \bigwedge_i p_i(a_{i1}, ..., a_{ik_i})$$

The body of the predicate has two main parts. First, an expression e_i is specified as the value of every register $\mathsf{r_i}$. Second, a set of ground predicates is given, where each is a constant predicate symbol applied to a sequence of argument expressions. These predicates occur inside a sequence of existential quantifiers binding typed *existential variables*. The ground predicates may not refer to the machine registers directly. Instead, an existential variable may both be given as the value of a register and appear in a ground predicate.

For example, a TAL register file typing has a straightforward encoding as an Open Verifier invariant. An existential variable is used to stand for the value of each register, and we use one ground predicate stating "the current value of register r has type τ" for each register r and the type τ assigned to it by the register file. I will return to the issue of encoding TAL states in more detail in Section 3.

We could have chosen to allow extensions to give unrestricted natural deduction proofs in weakening the invariants produced by the strongest postcondition generator. Instead, we introduce a special scripting language. In keeping with our goal of accessibility for this framework, we designed the script language to look a lot like code you might expect to find in a conventional program verifier, like the Java Bytecode Verifier or the TALx86 type checker. In this way, an engineer who understands how these systems work can transfer much of that knowledge directly to the Open Verifier.

The script language has two main differences from this ideal state. First, instead of using custom data structures to keep track of state, Open Verifier extensions must encode all safety-critical state as predicates inside of invariants. It is still safe to use custom structures for any state or metadata that serve only as "hints" to guide proof search and improve verification performance. It is fairly straightforward to transform traditional verifier code to meet this form. Once this is done, what remains is to provide *adequacy proofs* along with some state transformation operations that were assumed sound in the standard verifier. For instance, adding a new ground predicate to an invariant ought to require giving a proof that it really holds.

Naturally, it is not obvious that it is tractable to use the Open Verifier to develop verifiers with sufficient flexibility and efficiency. Part of the motivation for my work on the TAL extension is to discover how suitable our abstractions are for constructing a realistic extension. It seems that a working TAL extension would show that our system is at least as practical for real use as existing FPCC systems, since to date all have focused primarily on verifying TAL code.

3 The TAL Extension

I built the TAL extension around the first release of the TALC tools [MCG$^+$03] from Cornell University. TALC includes a type-checker for TALx86 code, as well as compilers to TALx86 from Popcorn (a safe C dialect) and mini-Scheme.

I wrote a straightforward translator that compiles TALx86 code into native x86 assembly code, removing type annotations and expanding macro-instructions. It is actually these translated files that the Open Verifier checks, with the original TAL source as meta-data.

In this section, I will build up a picture of how the standard TALx86 type checker works. In parallel, I will show the evolution of my first-order logic formalization.

3.1 TALx86 states

The original formal presentation of TALs [MWCG99] divides the state used in type-checking into three components:

- A *heap type* Ψ that maps locations in the assembly code to types. For instance, Ψ provides a type for every function label of the program.

- A *type environment* Δ, a set of type variables.

- A *register file* Γ that maps x86 registers to TALx86 types.

Of these components, Ψ is constant throughout verification, while Δ and Γ will vary. Ψ does not change to reflect dynamic memory allocation. Dynamically allocated objects are reachable only through registers, so the register file provides enough typing information to check any accesses to them.

The original presentation defines TAL type checking using a traditional typing relation, in the style common in formalizations of languages like ML. Here I will use a single-step, operational style static semantics, to make the connection to future developments clearer. Among other benefits, this choice makes the flow-sensitive aspects of TAL type checking more apparent.

For example, say that we want to type check one instruction of a TAL program. Let pc be the address of a TALx86 instruction $\mathsf{MOV\ eax} := 0$. We can describe how this instruction should be type checked with:

$$\frac{\mathsf{program}(pc) = \mathsf{MOV\ r} := n}{\langle \Delta, \Gamma, pc \rangle \xrightarrow{\Psi} \langle \Delta, \Gamma[\mathsf{r} \leftarrow \mathsf{word}], pc + 1 \rangle}$$

This means that, in this abstract semantics of program execution, the effect of the instruction $\mathsf{MOV\ eax} := 0$ is to advance the program counter to the next location and assign a value of type word to eax.

For example, fix some Ψ containing the fixed type information for the program under analysis. Assume that the instruction occurs in a context with no bound type variables, so $\Delta = \emptyset$; that pc points to an instruction $\mathsf{MOV\ eax} := 0$; and that nothing interesting is known about any of the other register values, so Γ maps every register to word. We get $\langle \emptyset, \Gamma, pc \rangle \xrightarrow{\Psi} \langle \emptyset, \Gamma, pc + 1 \rangle$.

Previous efforts to foundationalize TALs have taken a relation like this as a starting point. They prove safety theorems about the semantics defined by the relation, and then they prove that this relation abstracts concrete machine semantics in a suitably conservative way. One of my goals was to explore an alternate approach.

The Open Verifier has a built-in notion of states. The trusted core handles many kinds of reasoning about register values, state-specific logical assertions, and finding fixed points of abstract interpretations. Previous TAL formalizations handle these tasks using customized methods. In the work I present here, I have tried to take advantage of what the Open Verifier provides, in part to evaluate what engineering benefits its architecture may bring.

Recall that an Open Verifier *invariant*, or logical state, is a first-order logical predicate of the form

$$\exists x_1 : \tau_1, ..., x_n : \tau_n. \bigwedge_i \mathsf{r_i} = e_i \wedge \bigwedge_i p_i(a_{i1}, ..., a_{ik_i})$$

We see here that part of the reasoning about register files is already present. I will exploit this in my logical encoding. For now, consider only the Γ part of states. If

$\mathsf{r_1}, ..., \mathsf{r_n}$ is an enumeration of the x86 registers and $\Gamma(\mathsf{r_i}) = \tau_i$ for each i, then the following captures the meaning of Γ as an Open Verifier invariant:

$$\exists x_1 : \mathsf{val}, ..., x_n : \mathsf{val}.\ \bigwedge_i \mathsf{r_i} = x_i \wedge \bigwedge_i x_i : \tau_i$$

Since invariants of this form are very common, I will often use a shorthand notation where a register name stands for a unique existential variable that stands for the value of that register:

$$\bigwedge_i \mathsf{r_i} : \tau_i$$

Actual invariants label logical assertions with names, to make it easier to reference them later. Taking this into account, I will define the form of TAL invariants to consist of an assumption $h_{\mathsf{r_i}}$ for each register $\mathsf{r_i}$, declaring $\mathsf{r_i} : \tau_i$ for some τ_i.

Now I consider how to translate the above transition rule into a form that the Open Verifier understands. It must be expressed as a script describing how to transform an invariant.

$$\begin{array}{l}
\textbf{abstract}\ 0\ \textbf{as}\ eax_1\ \textbf{in} \\
\textbf{set}\ \mathsf{eax} = eax_1\ \textbf{by}\ \mathsf{Id}\ \textbf{in} \\
\textbf{assert}\ h_{\mathsf{eax}} : (eax_1 : \mathsf{word})\ \textbf{by}\ \mathsf{WordIntro}\ \textbf{in} \\
\textbf{collect as}\ t
\end{array}$$

Step-by-step, the effect of the script is to:

1. Declare a new existential variable eax_1 to stand for the new value of eax. This is not necessary for soundness; it would be possible to record that eax is exactly 0. However, I have chosen to maintain that every register's value is named uniquely by an existential variable, since the TALx86 type-checker itself ignores register values. It is logical weakenings like this that restrict the verification state space to a tractable size.

2. eax is set to its new value. For this to be sound, we must prove that the new value is equal to the old. The reflexivity proof rule Id suffices to show this, since the proof is checked with the instantiations of any new existentials substituted in. The instantiations are only used in checking this script; they will be forgotten for later stages.

3. The assumption about eax's type must be changed. A proof rule WordIntro, which declares that any value has word type, is used.

4. The **collect** instruction registers a name t for the new invariant and instructs the Open Verifier to continue checking its successors. If verification ever reaches this precise state again, the name t may be used to declare that the state's successors have already been explored.

This script always translates an invariant of the form described above into another of the same form. All of the scripts used by the TAL extension behave similarly, so that they define an implicit abstract transition relation. However, in contrast to previous work, the verifier described here relies on the reusable core of the Open Verifier to handle the overall structure.

3.2 Bringing memory into the picture

We start to see problems with the simplistic encoding from the last section when dealing with memory operations. For example, say that eax has type $\mathsf{word}^{\mathsf{init}} \times \mathsf{word}^{\mathsf{init}}$, a pointer to a pair of words in the heap. Say that we reach the instruction

$$\mathsf{MOV\ ebx} := [\mathsf{eax}]$$

The abstract transition relation rule might be:

$$\frac{\mathsf{program}(pc) = \mathsf{MOV\ r_1} := [\mathsf{r_2}] \quad \Delta;\Gamma \vdash \mathsf{r_2} : \tau_1^{\mathsf{init}} \times \tau_2^{\mathsf{init}}}{\langle \Delta, \Gamma, pc\rangle \xrightarrow{\Psi} \langle \Delta, \Gamma[\mathsf{r_1} \leftarrow \tau_1], pc+1\rangle}$$

One first cut at an Open Verifier script could be:

> **abstract** $\mathsf{sel}(\mathsf{mem}, \mathsf{eax})$ **as** ebx_1 **in**
> **set** $\mathsf{ebx} = ebx_1$ **by** Id **in**
> **assert** $h_{\mathsf{ebx}} : (ebx_1 : \mathsf{word})$ **by** $\mathsf{PairElim1}(h_{\mathsf{eax}})$ **in**
> **collect as** t

This looks much like the script from the last section, with two differences. First, instead of a constant, the new value of ebx is the result of reading the word at address eax of the current memory. Second, the truth of the new typing assertion about ebx is justified with a unary proof rule PairElim1, which depends on a proof that $\mathsf{eax} : \mathsf{word}^{\mathsf{init}} \times \mathsf{word}^{\mathsf{init}}$. The old typing assumption about eax should be an appropriate argument.

While this script looks workable, we run into problems when trying to formalize the PairElim1 proof rule. We must prove something about the contents of a memory, despite the fact that we have no information on that memory. This highlights our first fundamental difference from the standard TAL type checker. While a soundness argument about that type checker must reason about memory contents, the actual implementation leaves such information implicit.

This is quite sensible, since most interesting programs have infinitely many different reachable heap configurations. I do the same in the verification-time portion of the TAL extension. I keep precise heap information in the logical model and rely on existential quantification to hide the details at verification time. In this way, a state with an existential variable standing for the heap can model an infinite number of more concrete states. This is essentially a formalization of what the standard type checker does.

I introduce a type context, whose values will describe the current memory state, as well as other pieces of useful information that I will describe later. I revise the form of

TAL invariants to be:

$$\exists\Sigma : \mathsf{context}.\ \mathsf{GlobalInv}(\Sigma, \mathsf{mem}) \wedge \bigwedge_i \Sigma \vdash \mathsf{r_i} : \tau_i$$

The differences from the previous form are:

- Each invariant involves an existentially quantified context Σ. Among other things, Σ can be thought of as a partial mapping from memory locations to types.
- A *global invariant* GlobalInv connects Σ with the current memory. For instance, whenever Σ maps a location a to a type τ, the value currently in cell a of memory ought to have type τ.
- Σ is added as a parameter of the main typing judgment, since the truth of some judgments depends on the current state of memory.

Say that the global invariant assumption is always named h_{inv}. This modification of the earlier script now does the job:

$$\begin{array}{l}
\textbf{abstract}\ \mathsf{sel}(\mathsf{mem}, \mathsf{eax})\ \textbf{as}\ ebx_1\ \textbf{in} \\
\textbf{set}\ \mathsf{ebx} = ebx_1\ \textbf{by}\ \mathsf{Id}\ \textbf{in} \\
\textbf{assert}\ h_{\mathsf{ebx}} : (\Sigma \vdash ebx_1 : \mathsf{word})\ \textbf{by}\ \mathsf{PairElim1}(h_{inv}, h_{\mathsf{eax}})\ \textbf{in} \\
\textbf{collect as}\ t
\end{array}$$

Now that a proof of PairElim1 may assume that the memory agrees with Σ, the proof rule is indeed derivable, using suitable definitions of the predicates. I will provide more detail on these definitions later.

3.3 Instructions that change the context

The last example shows one advantage of using the Open Verifier: the script needs only to refer to aspects of the state that the current instruction changes. For instance, the typing assertions for the registers besides eax and ebx are carried along unchanged. Past foundational TAL verifiers have used custom congruence rules to express this idea for their abstract machines.

However, using the Open Verifier doesn't remove the need for "congruence rule" style reasoning. For instance, in the TAL invariant format from the last section, every typing assertion depends on the current context. If an instruction requires that the context be modified, then we must reprove any typing assertions that we want to remember.

To take a concrete example, consider an instruction MALLOC eax : word $\times$ word. This TALx86 macro-instruction allocates an integer pair in the heap, storing a pointer to the new object in eax. A general rule could be:

$$\frac{\mathsf{program}(pc) = \mathsf{MALLOC}\ \mathsf{eax} : \vec{\tau}}{\langle \Delta, \Gamma, pc \rangle \xrightarrow{\Psi} \left\langle \Delta, \Gamma[\mathsf{r} \leftarrow \tau_1^{\mathsf{uninit}} \times \ldots \times \tau_n^{\mathsf{uninit}}], pc + 1 \right\rangle}$$

The change to the heap allocation state is completely implicit. This works out fine for the informal TALx86 type checker, but we need some account of MALLOC's effect on the heap to foundationalize it. Since the TAL program probably makes use of eax's new type, we need to update the context to one in which we can make useful assertions about eax. I define a function $\mathsf{contextMalloc} : \mathsf{context} \times \mathsf{type\ list} \to \mathsf{context}$ that transforms a context appropriately, by adding heap mappings for a new record with components of the specified types.

The MALLOC instruction is compiled into a sequence of x86 instructions. For all but the last one, the TAL extension can give an empty script, asking the Open Verifier to use precise strongest postconditions to build successor states. Say that these instructions involve incrementing a heap pointer by the appropriate amount to allocate a new record and storing this new pointer in eax. Let HeapPointer be the statically allocated address of a memory cell that serves as the heap pointer.

The following script transforms an invariant appropriately after a MALLOC operation. mem_0 is the existential variable that was the value of the memory before beginning the allocation.

$$
\begin{array}{l}
\mathbf{abstract}\ \mathsf{sel}(mem_0, \mathsf{HeapPointer})\ \mathbf{as}\ eax_1\ \mathbf{in} \\
\mathbf{set}\ \mathsf{eax} = eax_1\ \mathbf{by}\ \mathsf{Id}\ \mathbf{in} \\
\mathbf{abstract}\ \mathsf{contextMalloc}(\Sigma, [\mathsf{word}, \mathsf{word}])\ \mathbf{as}\ \Sigma\ \mathbf{in} \\
\mathbf{assert}\ h_{\mathsf{r_1}} : (\Sigma \vdash \mathsf{r_1} : \tau_1)\ \mathbf{by}\ \mathsf{HasTypeMalloc}(h_{inv}, h_{\mathsf{r_1}})\ \mathbf{in} \\
\ldots \\
\mathbf{assert}\ h_{\mathsf{r_n}} : (\Sigma \vdash \mathsf{r_n} : \tau_n)\ \mathbf{by}\ \mathsf{HasTypeMalloc}(h_{inv}, h_{\mathsf{r_n}})\ \mathbf{in} \\
\mathbf{assert}\ h_{\mathsf{eax}} : (\Sigma \vdash \mathsf{eax} : \mathsf{word}^{\mathsf{uninit}} \times \mathsf{word}^{\mathsf{uninit}})\ \mathbf{by}\ \mathsf{NewMalloc}(h_{inv})\ \mathbf{in} \\
\mathbf{assert}\ h_{inv} : (\mathsf{GlobalInv}(\Sigma, \mathsf{mem}))\ \mathbf{by}\ \mathsf{GlobalInvMalloc}(h_{inv})\ \mathbf{in} \\
\mathbf{collect}\ \mathbf{as}\ t
\end{array}
$$

The script starts by abstracting the new value of eax, which is the starting value of the heap pointer; and the new context, which is built from the old with contextMalloc. Following that, every register typing assertion is re-proved for the new context, using the following proof rule:

$$
\frac{\mathsf{GlobalInv}(\Sigma, mem) \quad \Sigma \vdash e : \sigma}{\mathsf{contextMalloc}(\Sigma, \vec{\tau}) \vdash e : \sigma}\ \mathsf{HasTypeMalloc}
$$

Next, eax is asserted to be a pointer to a new record of the expected type, using:

$$
\frac{\mathsf{GlobalInv}(\Sigma, mem)}{\mathsf{contextMalloc}(\Sigma, \vec{\tau}) \vdash \mathsf{sel}(mem, \mathsf{HeapPointer}) : \tau_1^{\mathsf{uninit}} \times \ldots \times \tau_n^{\mathsf{uninit}}}\ \mathsf{NewMalloc}
$$

Finally, the script asserts that the new context and the new memory are compatible. The new memory is the old memory with the heap pointer incremented to reflect the allocation.

$$
\frac{\mathsf{GlobalInv}(\Sigma, mem)}{\begin{array}{c}\mathsf{GlobalInv}(\mathsf{contextMalloc}(\Sigma, \vec{\tau}), \\ mem[\mathsf{HeapPointer} \leftarrow \mathsf{sel}(mem, \mathsf{HeapPointer}) + 4 \times \mathsf{length}(\vec{\tau}))\end{array}}\ \mathsf{GlobalInvMalloc}
$$

3.4 Control flow

The last major type of code to be handled is that dealing with control flow. TAL programs are sets of basic blocks that end with jump instructions. I have already shown how the non-jump instructions are handled with a transition relation. The jump instructions are handled differently, using a kind of assume-guarantee reasoning familiar from program verification. When a jump is encountered, execution is assumed to proceed safely as long as the jump target has a code type with a precondition that the current state satisfies. An informal soundness argument for this technique is based on two ideas:

- The TAL type checker is built always to check every basic block.
- For whole program verification, the type system is defined so that basic block entry points are the only values of code pointer type.

With these in mind, we can construct a simple inductive argument on the number of steps of execution of a TAL program that shows that it never executes an unsafe instruction.

The following rule captures this idea operationally:

$$\frac{\mathsf{program}(pc) = \mathsf{JUMP}\ e \quad \Delta; \Gamma \vdash e : \mathsf{code}\{\Gamma'\} \quad \Delta \vdash \Gamma \leq \Gamma'}{\langle \Delta, \Gamma, pc \rangle\ \mathsf{safe}^{\Psi}}$$

It says that a jump is safe if it is to a value with type $\mathsf{code}\{\Gamma'\}$ and if the current register file Γ is a kind of subtype of the precondition Γ'. What this means is that, for each register assigned a type τ by Γ', Γ must assign that register a subtype of τ.

The TAL extension uses almost the same strategy in the foundational setting. However, in comparison to previous formalizations, mine takes advantage of the Open Verifier architecture to avoid a good amount of boilerplate code. Past projects prove new theorems about how their abstract machines admit assume-guarantee reasoning. Since the Open Verifier has such reasoning built in, I am able to deal only with the interesting, TAL-specific parts.

Point number 2 from the beginning of this subsection is imported easily. Ignoring polymorphism for the moment, consider some code type $\mathsf{code}\{\Gamma\}$. There is a known subset $\ell_1, ..., \ell_n$ of basic block labels assigned this type. Thus, I define:

$$\Sigma \vdash e : \mathsf{code}\{\Gamma\} \equiv \bigvee_{i=1}^{n} e = \ell_i$$

It is straightforward to define this relationship as a function of an arbitrary code type, as well as to take polymorphism into account.

It is not much harder to model point 1, which is the critical hypothesis for the assume-guarantee reasoning. Recall from the introduction that an extension is allowed to specify a set $\mathcal{E}_0$ of verification root invariants. The trusted core of the Open Verifier uses these as the basis for an exhaustive state space exploration, guided by the abstractions that the extension requests. Since the trusted core "promises" to check the safety of every root invariant, it is sound to cut off state space exploration whenever one of these invariants is encountered as a successor state.

For example, say that a TAL program contains a function identity, whose entry label is given the type $\tau = \mathsf{code}\{\mathsf{eax} : \mathsf{word}, \mathsf{ebx} : \mathsf{code}\{\mathsf{eax} : \mathsf{word}\}\}$. This type could describe a calling convention where the function expects an argument in eax and a return pointer in ebx, and where the return value will be stored in eax. The extension would declare this as a verification root with the following invariant, named identity:

$$\exists\Sigma : \mathsf{context}.\ \mathsf{GlobalInv}(\Sigma, \mathsf{mem}) \wedge\ \Sigma \vdash \mathsf{eax} : \mathsf{word} \wedge\ \Sigma \vdash \mathsf{ebx} : \mathsf{code}\{\mathsf{eax} : \mathsf{word}\}$$

Now consider that an instruction JUMP e is reached where e has type τ. Using a definition of having code type in the style sketched above, we will be able to derive a result like $\Sigma \vdash e : \tau \Leftrightarrow e = \mathsf{identity} \vee \bigvee_i e = \ell_i$ for zero or more other labels ℓ_i with compatible types. Let $disj$ be some proof of this fact. The following Open Verifier script proves this instruction execution safe:

$$\begin{array}{l}
\mathbf{cases}\ disj\ \mathbf{of} \\
\mid h_{\mathsf{identity}} : (e = \mathsf{identity}) \Rightarrow \\
\quad \mathbf{set}\ \mathsf{pc} = \mathsf{identity}\ \mathbf{by}\ h_{\mathsf{identity}}\ \mathbf{in} \\
\quad \mathbf{match}\ \mathsf{identity} \\
\mid h_{\ell_1} : (e = \ell_1) \Rightarrow \ldots \\
\ldots \\
\mid h_{\ell_n} : (e = \ell_n) \Rightarrow \ldots
\end{array}$$

The script considers the possible cases for e's identity, one for each disjunct in the proposition proved by $disj$. Each case binds a proof of the corresponding disjunct to a local hypothesis name. In the case of identity, the **set** directive is used to rewrite the program counter from e to the now provably equal identity. After this change, the current invariant should match the root identity declared at the start of the verification. The **match** instruction declares this fact, notifying the trusted core that it need not explore the successor states in this case. The cases for the other compatible labels ℓ_i would use analogous scripts.

Considering polymorphism, the outline of this technique remains the same, but the details become quite a bit more involved. It is necessary to create a new context with instantiations of the target label's type variables, and then to prove that the typing precondition holds in this new context. This requires changing the definition of having code type. "e has code type τ" is changed from meaning "e is one of the labels assigned type τ" to "e is a label assigned type $\forall\alpha_1, ..., \alpha_n.\ \sigma$, where some substitution for the type variables in σ is equal to τ." Now a critical component in handling jumps is an inversion lemma that proves disjunctions of this kind for particular code types.

This technique works in a logical formalization, but it has some undesirable properties at verification time. The disjunction at every function call site can add a quadratic factor to verification time, when we would hope to verify TAL programs in almost linear time. The standard TAL type checker has no such problem. It considers a jump safe whenever the jump satisfies the premises of the rule given earlier. This is safe because the checker maintains the invariant that every code pointer points to a basic block that will be verified eventually. To obtain similar performance from the TAL extension, I encode this assumption explicitly.

The Open Verifier architecture is designed to remove any need for explicit reasoning about safety of states. Instead, the notion of invariants is taken as a starting point, with built-in machinery for constructing safety proofs. One consequence of this is that it may not be clear how to formalize a particular argument about safety. In the case of this safe jump property, I was able to use a feature of the Open Verifier called an *indirect invariant*. While standard invariants apply to particular program locations, indirect invariants remove this restriction, making them general predicates about program states. The Open Verifier allows any named indirect invariant to be declared safe at any time, as long as it is accompanied with a script that proves its safety by appealing to existing invariants.

What is needed is an indirect invariant like:

$$\begin{aligned}&\exists \Sigma : \mathsf{context}, \Gamma : \mathsf{regFile}, v_1 : \mathsf{val}, \ldots, v_n : \mathsf{val}.\ \mathsf{r_1} = v_1 \wedge \ldots \wedge \mathsf{r_n} = v_n \\ &\quad \wedge \mathsf{GlobalInv}(\Sigma, \mathsf{mem}) \wedge \Sigma \vdash \mathsf{pc} : \mathsf{code}\{\Gamma\} \wedge \Sigma \vdash \{\mathsf{r_1} = v_1, \ldots, \mathsf{r_n} = v_n\} : \Gamma\end{aligned}$$

The last conjunct uses a new judgment, which says that a value assignment to the registers is compatible with a given register file typing. The overall meaning of the invariant can be phrased as a description of a set of states. This invariant describes all states such that

- The current memory is compatible with some context Σ.
- In Σ, the program counter has a code type with precondition Γ.
- The current register values satisfy that precondition in Σ.

Now any well-typed jump can be proved safe in constant time through an appeal to this indirect invariant. Most of the components should already be present in a valid local invariant. It only remains to prove that the program counter has a code type and that the current register values satisfy the corresponding precondition, which is easily done if this is the case.

The last part of the approach left to explain is how to justify the safety of the indirect invariant. This is actually straightforward, following the method described earlier for proving jumps safe. The difference is that we must consider all possible jump targets, not just those compatible with a given code type. The script is a case analysis over the possibilities, like before. In each case, the sub-script must transform the invariant into the entry invariant for a particular basic block, given the knowledge that the program counter points to that entry point. All of the existential variables match up with those that are expected, and the assumption about the global invariant is already in the right form. All that remains are the register type assumptions, which can be proved straightforwardly from the fact that the register values satisfy that basic block's precondition.

3.5 Some details of local invariants

The preceding subsections have demonstrated the basic ideas behind my approach. In the remainder of this section, I will sketch some of the finer details.

The format for local invariants in the implementation is actually more complicated than I have presented so far. In particular:

- I use values of a type `program` to represent all program-specific information, like types of code labels and definitions of named types. Every invariant contains an assertion that the context Σ is associated with the proper program.

- Every invariant lists explicitly which type variables are bound in the current context. This is important to know for situations where new type variables are introduced. If an old variable were re-used, some typing relations could be invalidated.

- There is an unfortunately large effort associated with keeping track of the stack. The TAL typing relation has the desirable property that it is *monotonic*, meaning that old typing relationships on values are never invalidated in the course of a program. This is a useful abstraction to have in the presence of such things as heap allocation, where we expect a garbage collector to be careful to maintain this property. The exception is the stack, where popping values can be thought of as invalidating old typing relationships. TAL handles this by treating `stackptr` types as giving no information on anything but the *length* of a stack that they point to, to prevent them from becoming invalidated. The current type of the stack is tracked separately, using a separate typing notion. This type is changed whenever a push or pop occurs. Therefore, each TAL extension invariant also carries an assumption about a stack type.

3.6 Contexts

The precise definition of contexts Σ in my formalization is as records containing:

- A program value, which contains a partial mapping from program labels to types and from type names to definitions.

- A mapping from bound type variables to instantiations.

- A memory, a partial map from memory addresses to *fields*. Making this a map to types wouldn't work using the standard TAL type system, which relies on the idea of a field to deal with issues of aliasing. Recall that a field provides both a type and an initialization flag. When a memory location is assigned an uninitialized field of type τ, it is acceptable to write a value of type τ to it, but not to write a value of any other type. When a record is first allocated, all of its fields are uninitialized, but their eventual types are locked in. Because of the way fields are handled, it is safe to pass this record handle around; there is no danger that, say, two functions with pointers to the record will be able to write values of incompatible types into the same slot.

- Beginning, end, and current allocation pointers for the heap.

- Beginning, end, and current allocation pointers for the stack.

3.7 The global invariant

$\mathsf{GlobalInv}(\Sigma, \mathsf{mem})$ asserts a number of facts. Here I will sketch what these facts are, and I will try to give an informal idea of why each is needed.

First, the context's program must satisfy two conditions:

1. The definition of every named type must not have any free type variables. Since the named types exist on a per-program level, it would not make sense to allow them to refer to type variables, which are bound only in local contexts.

2. The type of every code block must be similarly closed.

It is easy to come up with other program well-formedness conditions, such as that no undefined named type should be referenced or that recursive type definitions should meet well-definedness conditions similar to those imposed in languages like ML. However, I have not found it necessary to impose these conditions in order to be able to prove memory safety of TAL programs. For example, when a named type is defined with ill-founded recursion, the typing rules will never allow a (necessarily finite) derivation assigning that type to any value. Allowing a type whose values are impossible to construct has no ill effect on memory safety.

The global invariant also places requirements on the parts of a context that vary during verification:

1. Every type variable instantiation is closed. There are sensible formalizations that would allow type variables to refer to others in their definitions, but this makes it harder to deal with jumps to other code blocks, where some old type variables' definitions may mention out-of-scope variables and cease to make sense.

2. The boundaries of the heap and stack make sense: They are all word-aligned, the heap and stack defined have positive size, and there is sufficient space between the two regions for a "guard page" of at least a word. This is critical for proving the soundness of allocation. It must be known that the heap and stack may be grown upward and downward, respectively, by a program that takes care not to skip past the guard page that divides them. Without this condition, one of the heap or stack might overflow into the other.

3. Every address between the heap start and stack end is known to be valid for reading or writing, since TAL programs will assume that all of these addresses are fair game for allocation.

4. The stack pointer and heap allocation pointer denote valid addresses within the boundaries of their respective regions. This provides a connection between these run-time values and the information about memory region layout.

5. Every address in the area between the heap and stack regions is in the guard page, so that accesses to it will safely abort execution.

6. The heap allocation pointer is stored in the first cell of the heap, and no heap address after the heap allocation pointer has yet been assigned a type. The first condition provides a run-time method for determining the heap allocation pointer. (Unfortunately, since TALx86 allows user programs to use all of the x86's small set of registers, it does not seem possible to use a register to store this information.) The second condition prevents "double allocation" of the same memory region to two conceptually different objects of incompatible type.

7. Finally, we need to know that the memory mem respects Σ's type assignments to memory regions. More formally, whenever the memory maps an address a to τ^{init}, $\Sigma \vdash \mathsf{sel}(\mathsf{mem}, a) : \tau$. This is the obvious property relating a context's allocation information with the real state of a memory.

The choice of this set of properties is crucial for the global invariant to be useful. There is no need to expend too much effort in minimizing their complexity. Since the extension will only reason about the global invariant using a set of well-chosen Horn logic lemmas, the precise definition will be hidden at verification time and will not contribute to time or space inefficiency.

4 Generalizing the control-flow approach

The TAL extension's control flow strategy can be adapted to work in a variety of other contexts. Here I present a re-formulation that is parameterized so as to be generally useful.

The formulation imposes the following restrictions on extensions:

1. A notion of *abstract states* is defined. Each invariant is created by translating the abstract state to a concrete state. For example, register typings Γ are the TAL extension's reified abstract states.

2. The possible jump targets in any program to be verified are known in advance. Each is pre-labelled with a precondition describing the allowable abstract states on entry to it.

3. Checking whether an abstract state satisfies a precondition may be prohibitively expensive. However, for every abstract state satisfying a given precondition, there should be a *witness* that indicates this in an easily checkable way. For the TAL extension, preconditions are the types of basic blocks, which may use universal quantification. Witnesses are substitutions for type variables, which explain how to instantiate those quantifiers.

To begin, I define the machine-specific notion of a concrete state.

$$\mathsf{concState} \quad : \quad \mathbf{Type}$$

To use this approach, we must first come up with a notion of abstract machine states that describe the information of interest in verification. This may look like a step away from one of the goals of this work; I have tried to take as much advantage as possible of the Open Verifier infrastructure, which has meant avoiding defining an explicit notion of abstract states. However, this abstract state is only conceptual. The extension developer uses it implicitly, remaining free to take advantage of what the Open Verifier provides.

$$\mathsf{absState} \quad : \quad \mathbf{Type}$$

An extension based on this formalism must maintain an abstract state associated with each invariant, and the extension must ensure that the abstract state corresponds with the family of satisfying concrete states in an appropriate way. I formalize this correspondence with the idea of a denotation function:

$$\mathsf{invariantOf} \quad : \quad \mathsf{absState} \rightarrow (\mathsf{concState} \rightarrow \mathbf{Prop})$$

The invariantOf function makes explicit the strategy that the extension uses to construct invariants. I represent invariants in the more general form of predicates on states, though the Open Verifier implementation uses a more restricted form.

Now we need the set of allowable jump targets:

$$\mathsf{targets} \quad : \quad \mathcal{P}(\mathbb{Z})$$

For the TAL extension, targets is the set of basic block labels. Each target must have an associated precondition. For example, preconditions for TAL can be computed from basic block types and type variable substitutions in the way I have shown informally in previous sections. I model the general notion as a predicate on existential witnesses and abstract states:

$$\begin{array}{rcl} \mathsf{witness} & : & \mathbf{Type} \\ \mathsf{precondition} & \equiv & \mathsf{witness} \rightarrow \mathsf{absState} \rightarrow \mathbf{Prop} \\ \mathsf{preconditionOf} & : & \mathsf{targets} \rightarrow \mathsf{precondition} \end{array}$$

I also need a notion of a "point of view switch" that determines what abstract state corresponds to a jump target entered with a particular witness. For the TAL extension, this operation is responsible for instantiating the target basic block's type variables. The witness (a type variable substitution) provides exactly the information needed to do this.

$$\mathsf{enter} \quad : \quad \mathsf{targets} \rightarrow \mathsf{witness} \rightarrow \mathsf{absState}$$

Now I can state concisely the key property that these relations should have. An extension implementor must prove this for his specific choices of definitions.

$$\frac{(\mathsf{invariantOf}\ abs\ conc) \quad e \in \mathsf{targets} \quad (\mathsf{preconditionOf}\ e\ wit\ abs)}{(\mathsf{invariantOf}\ (\mathsf{enter}\ e\ wit)\ conc)}$$

This lemma provides a way to deduce that a particular concrete state $conc$ satisfies the invariant of a jump target e. We assume that an abstract state abs corresponding to

$conc$ is known. It should be the case that if abs satisfies e's precondition using some witness wit, then $conc$ satisfies the local invariant associated with e, as evidenced by the witness wit.

The verification root invariant for every jump target must be chosen such that every valid `enter` output involving that target satisfies it. In the case of the TAL extension, we have the entry invariant for every basic block in a standard form, asserting the existence of a context Σ for which assertions about the global invariant and register types hold. Each block's precondition is of the form described in Section 3.4, which implies all of the needed facts.

5 Results

5.1 The logical formalization

I used the proof assistant Coq [Coq02] to formalize the logical development that I've outlined here. My implementation is about 19,000 lines long. It includes both definitions and proofs of lemmas about them. I was learning how to use Coq while developing this formalization, so I expect that I could produce a considerably more concise version if I started again from scratch. Nonetheless, I did encounter a few fundamental issues that are worth noting.

The biggest such issue was the unexpectedly large amount of work required to deal with type variables. The majority of the formalization is related to these, including definitions of notions of when a type has no free variables and proofs of lemmas about substitution.

Past efforts that use the logical framework Twelf [CWAF03, Cra03] get around this problem by encoding variable bindings using *higher order syntax*. Instead of defining types in a purely first-order form, where a type variable is just another syntactic element, they use the meta-language's function construct. A type can be viewed as a function from its free variables to a closed type.

Coq doesn't admit this strategy, since its well-formedness condition on inductive type definitions doesn't allow the type being defined to appear in a function argument position in its own definition. I don't view this as a conclusive reason to choose Twelf for future work, though. It seems likely that past projects that made that decision have paid a price in the amount of time it takes to build a complete formalization. Coq's support for human-assisted proof automation was a huge help for me in this work, and I think that the restrictions imposed on the language make this kind of automation more tractable.

5.2 The extension

The TAL extension successfully verifies memory safety for all of the test cases included with the TALC 1.0 distribution. The TAL code for the examples is produced by sample compilers from source code in Popcorn (a safe C dialect) and mini-Scheme.

There are a few provisos to this statement. First, as mentioned earlier, the TAL extension currently only supports whole program verification. Examples that span multi-

Compiler	**Test**	**Time** (seconds)	**Slowdown**	**Instructions**
Popcorn	fact	0.240	-	252
	fact2	0.230	23	254
	fib	0.270	27	273
	list	1.110	27.75	1079
	queue	0.270	-	338
	test3	1.300	32.5	1149
Scheme	fact	0.350	17.5	473
	foo	4.580	30.533	5324
	mergesort	1.650	23.571	2210
	print-int	1.230	24.6	1680
	printit	3.530	29.416	4195
	test	0.340	17	438
	test0	0.170	17	170
	test1	0.300	15	399
	test2	0.430	21.5	536
	test20	6.610	30.045	7203
	test3	0.390	19.5	497
	test4	0.140	14	122

Table 1: Statistics on tests run on a 1.7 GHz Pentium 4

ple source files cannot currently be handled directly. A straightforward linking process that combines these TAL sources into a single file allows them to be verified.

Secondly, my simple compiler from TAL into real assembly code uses a trivial and unrealistic runtime system. Allocation is handled by incrementing a pointer into the heap, and no garbage collection is performed. Also, none of the standard runtime functions are supported. The extension can verify foundationally any programs that don't call any of these functions, but, of course, such programs are of only limited use, though they can exercise all of the interesting aspects of TAL. Checking programs that link in untrusted runtime systems, including garbage collectors, remains for future work. The current extension checks realistic programs by allowing them to assume that runtime system functions are safe.

Table 1 presents results on using the TAL extension to verify these tests. The "Slowdown" column gives the ratio of the running time of the foundational verifier versus the conventional verifier, which is a thin wrapper around the type checker distributed with TALC 1.0. "-" entries in that column denote cases where the conventional verifier finished more quickly than the resolution of the timer, so the ratio is not well-defined. The "Instructions" column gives the number of assembly instructions in the program to be verified. These are not x86 instructions, but rather instructions in a simplified assembly language to which the input assembly program is translated before beginning verification. This inflates the instruction count by a small constant factor.

It is clear from these results that we have a long way to go to make the TAL extension practical. One would expect that a realistic foundational framework would allow verifiers running with only a small constant factor more time than conventional verifiers. These numbers are not really cause for pessimism, though, since performance has not been our primary concern so far. I summarize at the end of the conclusion some promising strategies for achieving small constant factor slowdowns.

6 Comparison

The TAL extension demonstrates a number of benefits of using the Open Verifier compared to previous approaches.

First, we have some comparatively minor savings that result from the larger TCB of the Open Verifier. Its trusted core handles reasoning about registers and machine instruction effects. There is no need for a logical development to handle the details of decoding a machine's instruction format, for example. When a state's successor can be constructed using the built-in strongest postcondition operation, an extension requires very little "boilerplate" code to do so. This kind of engineering convenience can ease the extension writer's burden noticeably.

There are also more fundamental differences. Previous formalizations [App01, HST$^+$02, Cra03] have followed the standard technique of reasoning about an abstract operational semantics during verification. The abstract semantics is connected to the machine's concrete semantics via a bisimulation in a way sufficient for proving memory safety.

In the Open Verifier, the core infrastructure handles these details. The TAL extension's formalization includes no notion of abstract machine states or an abstract semantics. Instead, it uses direct statements about the concrete machine state, parameterized by a context.

One benefit of this approach is that instructions that have only local effects can be handled using very local reasoning. In the register setting example of Section 3.1, for instance, there was no use of reasoning with the flavor of a "congruence rule" to combine the unchanged portions of the state with the changes. This reduces the conceptual complexity of the verification scheme, as well as the effort required to formalize it.

Allowing local reasoning also simplifies compositional construction of verifiers. For instance, the TAL extension could be combined with an extension for verifying another high-level language to produce an extension for verifying hybrid programs. Alternatively, we can view interaction between TAL code and a C runtime system in the same way. By reasoning in a way that abstracts irrelevant assumptions, we get some compositionality "for free." The extensions in charge of TAL code and the runtime system could maintain their own separate assumptions throughout verification, with a comparatively small effort in maintaining assumptions that connect the two worlds.

Of course, purely local reasoning can make things harder in some cases. For example, later versions of TALx86 have a type for "unique pointers" known to reside nowhere in a machine state but a single register, at most. The Open Verifier saves the extension from having to reason about whole machine states, but the most straightforward handling of unique pointers seems to require bringing more of the state back into the picture. There are other ways of modeling this, but preserving the same semantics seems unduly hard. The TALx86 1.0 system that I use in the TAL extension doesn't lead to this problem, because it uses subtyping and the idea of uninitialized record fields to achieve the same effect.

Previous work on verification of language interoperation by Hamid and Shao [HS04] suggested a somewhat asymmetric approach where low-level routines have preconditions parameterized on arbitrary state predicates. These predicates encode aspects of the interaction specific to a particular high-level language that could call the routines. The Open Verifier uses a simpler and more general system that requires no more than first-order reasoning. Hamid and Shao also do not suggest general methods to use in allowing the two aspects of verification to maintain their own state across several interactions; they only present a mechanism to prevent the runtime system from breaking TAL invariants.

In fact, their method involves assigning invariants like ours to each program point. They do some work to map their old bisimulation-based formalism onto this system. The Open Verifier can be viewed as skipping this extra step and using the more basic approach as the starting point.

7 Conclusion

To my knowledge, my implementation is the first foundational verifier in the literature to handle a complete, practical typed assembly language. It also appears to be the first such verifier to work with existing source-level compilers developed years ago by a different group of authors. The success of the project provides evidence that the Open Verifier succeeds in meeting one of its primary goals: simplifying the engineering effort involved in creating new foundational verifiers.

The Open Verifier forces verifier extensions to work within a very specific regime. It is not at all obvious what practical effect this has on expressiveness. I have shown that our framework does allow natural construction of a verifier for the "gold standard" of the FPCC community, a realistic typed assembly language. This is by no means a "completeness" result. It is quite clear that the general FPCC idea admits strategies that the Open Verifier does not. However, this experiment suggests that we likely do include enough mechanisms for handling of strategies that would be chosen in practice.

This work also lead to the development of the method I have described for handling control flow. Previous work on the Open Verifier [Sch04] relied on a complicated mechanism built into the trusted system. It allowed the use of nested invariants, and a proof of its soundness required an argument about indexed predicates. My approach allows the trusted computing base to be reduced while at the same time reducing the complexity of developing an extension. By using no "higher order" reasoning but that present in soundness proofs for the Open Verifier's trusted fixed point module, I make a strong case that simple logical mechanisms really are sufficient and natural for the construction of practical foundational verifiers.

A final contribution of this work comes from the insights it provides into a new way of thinking about and proving the soundness of typed assembly languages. Instead of starting from an abstract semantics and arguing that it corresponds with the concrete machine's semantics, I reason with direct propositions about machine states. This provides a natural kind of modularity that insulates some parts of the abstract state from the effects of changes to others. As a result, the formal results that must be proved decompose more naturally, and less verification-time proof effort is required for instructions whose effects can be formalized in a suitably incremental way. It is too early to make any strong statements about advantages of this viewpoint over the traditional one, but the new formalization is valuable in any case as an example of a novel way to use existing formalisms.

There must be considerable engineering effort before these techniques are feasible for wide use. Our current performance figures for different Open Verifier extensions are unacceptably slow. Efficiency of verification has yet to become a focus in the FPCC community, but we believe that we can make serious improvements. We hope to reap large benefits from applying proof representation and checking optimizations like those used in the original PCC work [NL97]. We have also concentrated to date on logical formalizations of the safety mechanisms of different extensions, relying on a generic Prolog interpreter for proof generation. It is likely that using a custom, optimized proof generator for each extension would also bring large savings. I am currently studying the problem of constructing a domain-specific extension language and an optimizing compiler for it that produces extension code of this type. Such a language brings the added benefit of simplifying the development of extensions.

Acknowledgments. This work was possible thanks to close collaboration with George Necula, Robert Schneck, Evan Chang, and Kun Gao, as part of the Open Verifier project. I'd also like to thank George Necula and Robert Schneck for their feedback on this report.

References

[AF00] Andrew W. Appel and Amy P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL-00)*, pages 243–253. ACM Press, January 2000.

[AM01] Andrew W. Appel and David A. McAllester. An indexed model of recursive types for foundational proof-carrying code. *Programming Languages and Systems*, 23(5):657–683, 2001.

[App01] Andrew W. Appel. Foundational proof-carrying code. In *Logic in Computer Science*, 2001.

[CCNS05] Bor-Yuh Evan Chang, Adam Chlipala, George C. Necula, and Robert R. Schneck. The Open Verifier framework for foundational verifiers. In *Proc. of the 2nd ACM Workshop on Types in Language Design and Implementation (TLDI'05)*, January 2005.

[CLN+00] Christopher Colby, Peter Lee, George C. Necula, Fred Blau, Mark Plesko, and Kenneth Cline. A certifying compiler for Java. In *ACM SIGPLAN*

'00 Conference on Programming Language Design and Implementation (PLDI), pages 95–107. ACM Press, May 2000.

[Coq02] Coq Development Team. The Coq proof assistant reference manual, version 7.3. May 2002.

[Cra03] Karl Crary. Toward a foundational typed assembly language. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL-03)*, volume 38(1) of *ACM SIGPLAN Notices*, pages 198–212. ACM Press, January 15–17 2003.

[CWAF03] Juan Chen, Dinghao Wu, Andrew W. Appel, and Hai Fang. A provably sound tal for back-end optimization. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 208–219. ACM Press, 2003.

[Gou02] John Gough. *Compiling for the .NET Common Language Runtime*. .NET series. Prentice Hall, Upper Saddle River, New Jersey, 2002.

[GS01] Andrew D. Gordon and Don Syme. Typing a multi-language intermediate code. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL-01)*, pages 248–260, London, United Kingdom, January 2001.

[HS04] Nadeem A. Hamid and Zhong Shao. Interfacing Hoare logic and type systems for Foundational Proof-Carrying Code. In *17th International Conference on Theorem Proving in Higher-Order Logics (TPHOLs2004)*, September 2004.

[HST+02] Nadeem A. Hamid, Zhong Shao, Valery Trifonov, Stefan Monnier, and Zhaozhong Ni. A syntactic approach to foundational proof-carrying code. In *Proceedings of the Seventeenth Annual IEEE Symposium on Logic in Computer Science*, pages 89–100, Copenhagen, Denmark, July 2002.

[LST02] Christopher League, Zhong Shao, and Valery Trifonov. Type-preserving compilation of Featherweight Java. *ACM Transactions on Programming Languages and Systems*, 24(2):112–152, 2002.

[LST03] Christopher League, Zhong Shao, and Valery Trifonov. Precision in practice: A type-preserving Java compiler. In *Proc. 12th International Conference on Compiler Construction (CC'03)*, Warsaw, Poland, April 2003.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2nd edition, 1999.

[MCG+99] 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. 1999.

[MCG+03] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. Talc releases, 2003. URL: http://www.cs.cornell.edu/talc/releases.html.

[MCGW98] G. Morrisett, K. Crary, N. Glew, and D. Walker. Stack-based typed assembly language, 1998.

[MWCG99] 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, 1999.

[Nec97] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, pages 106–119, Paris, January 1997.

[NL97] George C. Necula and Peter Lee. Efficient representation and validation of logical proofs. Technical Report CMU-CS-97-172, Computer Science Department, Carnegie Mellon University, October 1997.

[NL98] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *ACM SIGPLAN'98 Conference on Programming Language Design and Implementation*, pages 333–344, June 1998.

[Sch04] Robert R. Schneck. *Extensible Untrusted Code Verification*. PhD thesis, University of California, Berkeley, May 2004.

[Sym01] Don Syme. ILX: Extending the .NET common IL for functional language interoperability. *Electronic Notes in Theoretical Computer Science*, 59(1), 2001.

[WF98] Dan S. Wallach and Edward W. Felten. Understanding Java stack inspection. In *IEEE Symposium on Security and Privacy*, pages 52–63, May 1998.