Comparing Semantic and Syntactic Methods in Mechanized Proof Frameworks

C.J. Bell Robert Dockins Aquinas Hobor
Andrew W. Appel David Walker

Computer Science
Princeton University
Princeton, USA

Abstract

We present a comparison of semantic and syntactic proof methods for reasoning about typed assembly languages in Coq. We make available our complete Coq developments for a simple and easily understood benchmark system presenting both styles of soundness proof to the same interface. The syntactic proof is standard subject reduction; the semantic proof uses Gödal Löb modal logic, shallowly embedded in Coq. The proof style of the modal logic is flexible and facilitates experimental modifications to the underlying machine. As an example of this flexibility, we discuss how to add fault tolerance to the list machine. In addition, we discus how the the choice of proof methodology affects the trusted computing base of a typed assembly language system.

Keywords: theorem proving, typed assembly language, list-machine benchmark, Coq

1 Introduction

We believe that semantic methods are useful for reasoning about typed assembly languages but few head-to-head comparisons have been performed. We will remedy that now. We choose a very simple typed assembly language (TAL), fix the type system, and demonstrate both syntactic and semantic proofs of soundness in the same logical framework.

We will show how the semantic proofs can be constructed in a modular and extensible way. Then, we will demonstrate that extensibility by showing how to build a fault-tolerant TAL (FTTAL) and prove it sound.

While this toy example will be wonderful for comparing how these different systems work, it is too small to tell us about the relative behavior of syntactic and semantic styles as they scale up the industrial strength systems. Therefore, in section 4 we will present measurements of industrial-strength systems and draw conclusions about how they scale.

All of the type-safety proofs we will describe start with operational semantics on an untyped language. The type system is some set of inference rules. A *syntactic*

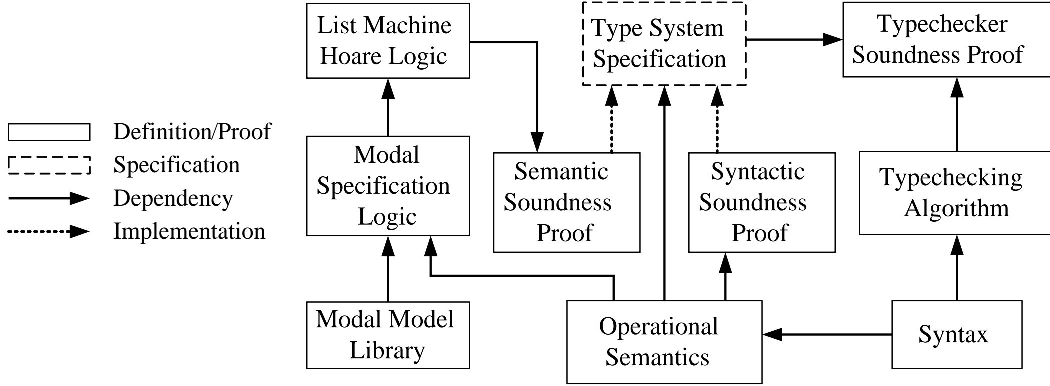

Fig. 1. Proof Organization

proof of type safety defines the typing operators as an inductive syntax and defines the inference rules of the type system as an inductive definition and proceeds by subject reduction [23]. A *semantic* proof gives a denotation of each operator of the type system as some sort of predicate on the state of the operational semantics, and proceeds by proving each typing rule as a derived lemma from the denotations [3]. There are also intermediate styles in which the syntax of types is inductive but the rules are proved semi-semantically. We demonstrate the purely syntactic and the purely semantic styles, but it is a strength of our benchmark Coq development that Adam Chlipala found it easy to demonstrate his mixed syntactic/semantic approach in our framework [11].

2 A Semantic Type-Safety Proof

Appel and Leroy suggested the list-machine benchmark as a way to compare machine-verified proofs involving TALs [4]. The original benchmark was used to compare two soundness proofs, using the syntactic method, in the proof assistants Twelf and Coq. For the research in this paper we have produced an updated version of this benchmark;[1] one of our major goals was to examine the difference between semantic and syntactic proof methods (in Coq). We provide fully-worked machine-checked proofs in both the syntactic and semantic styles. We expect that most readers will be familiar with the syntactic subject-reduction approach, and thus we will focus here on the semantic proof.

The overall proof organization is given in Figure 1. The syntax of the 2.0 list machine is given in Figure 2. We write $\alpha \rightharpoonup \beta$ to indicate a finite partial map with keys of type α and elements of type β, and $A(x) = y$ when the map A maps key x to a value y. The labels ($\mathbf{L_i}$) and values ($\mathbf{v_i}$) are assumed to be distinct countable sets. The zero label $\mathbf{L_0}$ plays two special roles; it represents both the entry label to the program and also a distinguished "nil" value for the **branch-if-nil** instruction.

The operational semantics of the list machine are listed in Figure 3. The judgment $(\rho, \iota) \overset{\Psi}{\mapsto} (\rho', \iota')$ means that the machine state (ρ, ι) takes a single step to the modified state (ρ', ι') in the program Ψ. The operational semantics are written in Wright-Felleisen style, such that illegal operations have no successor and "get

[1] The entire proof development is available at http://www.cs.princeton.edu/~appel/listmachine/2.0.

			$\iota ::=$	**jump** v	indirect jump to register v
$l ::=$	$\mathbf{L_0}, \mathbf{L_1}, \mathbf{L_2}, \ldots$	labels	$\mid$	**get-label** $l\ v$	load label l into v
$v ::=$	$\mathbf{v_0}, \mathbf{v_1}, \mathbf{v_2}, \ldots$	variables	$\mid$	**branch-if-nil** $v\ l$	if $v = \mathbf{L_0}$ go to l
$a ::=$	l	label values	$\mid$	**fetch-field** $v\ 0\ v'$	fetch the head of v into v'
$\mid$	$\mathrm{cons}(a_0, a_1)$	cons cell vaues	$\mid$	**fetch-field** $v\ 1\ v'$	fetch the tail of v into v'
$\rho ::=$	$v \rightharpoonup a$	register banks	$\mid$	**cons** $v_0\ v_1\ v'$	make a cons cell in v'
$\Psi ::=$	$l \rightharpoonup \iota$	programs	$\mid$	**halt**	stop executing
			$\mid$	$\iota_0\ ;\ \iota_1$	sequential composition

Fig. 2. Syntax of listmachine programs

stuck." For example, attempting to dereference a label or jump to a cons cell will get stuck. The safely-halted state is represented by $(\rho, \mathbf{halt})$ for any ρ. We say that a program state (ρ, ι) is *safe in program* Ψ (written safe-state$_\Psi$ (ρ, ι)) iff after stepping the state forward an arbitrary number of steps, it is either in the halt state or can take another step. We then say that a *program* Ψ is safe iff jumping to the entry label $\mathbf{L_0}$ with an arbitrary ρ results in a state that is safe in Ψ. In other words, safe programs never get stuck.

We present a type system in Figure 4; we will prove this sound w.r.t. program safety. The main judgment of the type system has the form $\Pi \vdash_{\text{blocks}} \Psi$, which means that the whole-program typing Π holds on program Ψ. For the purposes of our soundness proof, we consider the type constructors and the rules relating them a *specification* of the typing discipline. This notion is made formal via the Coq module system by defining a module signature. [2] Implementers of this module signature are required to prove a safety theorem, that the whole-program typing judgment is sufficient for program safety:

$$\forall \Psi\, \Pi,\ \Pi \vdash_{\text{blocks}} \Psi \rightarrow \text{safe}\, \Psi$$

We separately define a typechecking *algorithm* and prove that the algorithm is sound with respect to the type system; that is, each step of the algorithm is justified by one of the rules of the type system. This portion of the proof is agnostic to the proof method we have chosen (semantic or syntactic). We formalize this indifference by placing the proof in a *module functor* which is parameterized by an implementation of the type system signature. The main lemma says that whenever the typechecking algorithm returns true, there is a whole-program typing derivation:

$$\forall \Psi\, \Pi,\ \text{check}(\Pi,\ \Psi) = \text{true} \rightarrow \Pi \vdash_{\text{blocks}} \Psi$$

We can easily combine this lemma with any implementation of the type system soundness signature to obtain the theorem, $\forall \Psi\, \Pi,\ \text{check}(\Pi,\ \Psi) = \text{true} \rightarrow \text{safe}\, \Psi$.

All that remains is to implement the type system specification (including soundness proof) to obtain a complete foundational TAL. We have done this in two different ways: first, using syntactic progress-and-preservation (adapted from Leroy's Coq development [4]), and second, using a semantic argument.

We build the semantic proof is as follows: (i) build a generic library for defining modal specification logics for programming languages; (ii) use the library to define

[2] A signature is the type of a module. A signature may include parameters, definitions, inductive types, and the *statements* of theorems (but not their proofs). Parameters and the proofs of stated theorems must be provided by modules implementing the signature.

$$\frac{}{(\rho,\ (\iota_1;\iota_2);\iota_3) \overset{\Psi}{\mapsto} (\rho,\ \iota_1;(\iota_2;\iota_3))}\ \text{step-seq}$$

$$\frac{\rho(v) = l \quad \Psi(l) = \iota'}{(\rho, \textbf{jump}\ v) \overset{\Psi}{\mapsto} (\rho, \iota')}\ \text{step-jump}$$

$$\frac{\rho(v) = \text{cons}(a_0, a_1) \quad \rho[v' := a_0] = \rho'}{(\rho, (\textbf{fetch-field}\ v\ 0\ v';\iota)) \overset{\Psi}{\mapsto} (\rho', \iota)}\ \text{step-fetch-0}$$

$$\frac{\rho[v := l] = \rho'}{(\rho, \textbf{get-label}\ l\ v;\iota) \overset{\Psi}{\mapsto} (\rho', \iota)}\ \text{step-getlabel}$$

$$\frac{\rho(v) = \text{cons}(a_0, a_1) \quad \rho[v' := a_1] = \rho'}{(\rho, (\textbf{fetch-field}\ v\ 1\ v';\iota)) \overset{\Psi}{\mapsto} (\rho', \iota)}\ \text{step-fetch-1}$$

$$\frac{\rho(v) = \text{cons}(a_0, a_1)}{(\rho, (\textbf{branch-if-nil}\ v\ l;\iota)) \overset{\Psi}{\mapsto} (\rho, \iota)}\ \text{step-no-branch}$$

$$\frac{\rho(v_0) = a_0 \quad \rho(v_1) = a_1 \quad \rho[v' := \text{cons}(a_0, a_1)] = \rho'}{(\rho, (\textbf{cons}\ v_0\ v_1\ v';\iota)) \overset{\Psi}{\mapsto} (\rho', \iota)}\ \text{step-cons}$$

$$\frac{\rho(v) = \mathbf{L_0} \quad \Psi(l) = \iota'}{(\rho, (\textbf{branch-if-nil}\ v\ l;\iota)) \overset{\Psi}{\mapsto} (\rho, \iota')}\ \text{step-branch}$$

Fig. 3. Small-step operational semantics

a specification logic specific to list-machine programs; (*iii*) define the Hoare triple within the specification logic, prove the soundness of Hoare rules for each instruction, and demonstrate that the definition of the Hoare judgment is sufficient for program safety; and (*iv*) show that the rules of the type system are specializations of the Hoare rules and the main typing judgment is sufficient for safety.

Step (i): Modal Model Library. We define a *shallow* embedding of a heterogeneous multimodal logic [13] to use as the specification logic. We define statements of the specification logic to be *predicates on worlds*. In Coq, this means statements of the logic are defined as $\mathbf{W} \rightarrow$ `Prop`, where `Prop` is the type of Coq propositions and $\mathbf{W}$ is some abstract type, which is later instantiated by the library consumer.

Using this technique we obtain a powerful system which includes: all the usual logical connectives of higher-order logic including impredicative quantification; a suite of useful modal operators; and a powerful recursion operator. Among the modal operators defined by the library is the important "later" operator (written $\triangleright$), where $\triangleright p$ means that the proposition p holds at all times strictly in the future. This operator is important for defining recursive propositions and for solving certain kinds of circularity problems that arise from, e.g., impredicative reference types [5]. The library also defines a related operator called "necessarily" (written $\Box$) which means that a predicate holds now *and* in the future.

Step (ii): Modal Specification Logic. We specialize the generic library to the list-machine setting by defining a suitable set of worlds. For the list machine, we define worlds as tuples of the form (n, ρ, a), where the natural number n is the "age" of the world, ρ is the register bank, and a is a value. The age is a proof artifact that interacts with the operators $\triangleright$ and $\Box$; the register bank and value components allow our logic to express statements of interest in the problem domain.

We can view predicates that state properties of values as types. In other words, we use the specification logic to give a direct semantics for the types of our type system. For example, we can define the semantics of list types as follows:

$$\begin{aligned}
\text{nil} &:= \lambda(n, \rho, v).\, v = \mathbf{L_0} \\
\text{pair}\ p\ q &:= \lambda(n, \rho, v).\, \exists v_1\ v_2,\ v = \text{cons}(v_1, v_2) \wedge \forall \rho',\ (\triangleright p)(n, \rho', v_1) \wedge (\triangleright q)(n, \rho', v_2) \\
\text{list}\ t &:= \mu X.\, \text{nil} \mid\mid \text{pair}\ t\ X \\
\text{listcons}\ t &:= \text{pair}\ t\ (\text{list}\ t)
\end{aligned}$$

$\tau ::=$	nil	The value $\mathbf{L_0}$
$\mid$	list τ	List of τ
$\mid$	listcons τ	Non-nil list of τ
$\Gamma ::=$	$v \rightharpoonup \tau$	Type environments
$\Pi ::=$	$l \rightharpoonup \Gamma$	Program typings

Subtyping

$$\frac{}{\tau \subset \tau}\ \text{sub-refl} \qquad \frac{\tau \subset \tau'}{\text{listcons}\,\tau \subset \text{list}\,\tau'}\ \text{sub-listmixed}$$

$$\frac{}{\text{nil} \subset \text{list}\,\tau}\ \text{sub-nil}$$

$$\frac{\tau \subset \tau'}{\text{list}\,\tau \subset \text{list}\,\tau'}\ \text{sub-list} \qquad \frac{\tau \subset \tau'}{\text{listcons}\,\tau \subset \text{listcons}\,\tau'}\ \text{sub-listcons}$$

$$\frac{\forall l, \tau_0.\ \Gamma_0(l) = \tau_0 \rightarrow \exists \tau_1.\ \Gamma_1(l) = \tau_1 \wedge \tau_0 \subset \tau_1}{\Gamma_0 \subset \Gamma_1}\ \text{sub-env}$$

Instruction typings. Individual instructions are typed by a judgement $\Pi \vdash_{\text{instr}} \Gamma\{\iota\}\Gamma'$. The intuition is that, under program-typing Π, the Hoare triple $\Gamma\{\iota\}\Gamma'$ relates precondition Γ to postcondition Γ'.

$$\frac{\Pi \vdash_{\text{instr}} \Gamma\{\iota_1\}\Gamma' \quad \Pi \vdash_{\text{instr}} \Gamma'\{\iota_2\}\Gamma''}{\Pi \vdash_{\text{instr}} \Gamma\{\iota_1;\iota_2\}\Gamma''}\ \text{chk-seq} \qquad \frac{\Gamma[v := \text{nil}] = \Gamma'}{\Pi \vdash_{\text{instr}} \Gamma\{\textbf{get-label}\ \mathbf{L_0}\ v\}\Gamma'}\ \text{chk-getnil}$$

$$\frac{\Gamma(v) = \text{list}\,\tau \quad \Pi(l) = \Gamma_1 \quad \Gamma' \subset \Gamma_1 \qquad \Gamma[v := \text{nil}] = \Gamma' \quad \Gamma[v := \text{listcons}\,\tau] = \Gamma''}{\Pi \vdash_{\text{instr}} \Gamma\{\textbf{branch-if-nil}\ v\ l\}\Gamma''}\ \text{chk-br-list}$$

$$\frac{\Gamma(v) = \text{listcons}\,\tau \quad \Gamma[v' := \tau] = \Gamma'}{\Pi \vdash_{\text{instr}} \Gamma\{\textbf{fetch-field}\ v\ 0\ v'\}\Gamma'}\ \text{chk-fetch-0}$$

$$\frac{\Gamma(v) = \text{listcons}\,\tau \quad \Gamma[v' := \text{list}\,\tau] = \Gamma'}{\Pi \vdash_{\text{instr}} \Gamma\{\textbf{fetch-field}\ v\ 1\ v'\}\Gamma'}\ \text{chk-fetch-1}$$

$$\frac{\Gamma(v) = \text{listcons}\,\tau}{\Pi \vdash_{\text{instr}} \Gamma\{\textbf{branch-if-nil}\ v\ l\}\Gamma}\ \text{chk-br-cons}$$

$$\frac{\Gamma(v) = \text{nil} \quad \Pi(l) = \Gamma_1 \quad \Gamma \subset \Gamma_1}{\Pi \vdash_{\text{instr}} \Gamma\{\textbf{branch-if-nil}\ v\ l\}\Gamma'}\ \text{chk-br-nil}$$

$$\frac{\Gamma(v_0) = \tau_0 \quad \tau_0 \subset \tau' \quad \tau_1 \subset \text{list}\,\tau' \qquad \Gamma(v_1) = \tau_1 \quad \Gamma[v := \text{listcons}\,\tau'] = \Gamma'}{\Pi \vdash_{\text{instr}} \Gamma\{\textbf{cons}\ v_0\ v_1\ v\}\Gamma'}\ \text{chk-cons}$$

Block typings. A *block* is an instruction that does not (statically) continue with another instruction, because it ends with a halt or a jump.

$$\frac{}{\Pi; \Gamma \vdash_{\text{block}} \textbf{halt}}\ \text{chk-block-halt} \qquad \frac{\Pi \vdash_{\text{instr}} \Gamma\{\iota_1\}\Gamma' \quad \Pi; \Gamma' \vdash_{\text{block}} \iota_2}{\Pi; \Gamma \vdash_{\text{block}} \iota_1; \iota_2}\ \text{chk-block-seq}$$

$$\frac{\Pi(l) = \Gamma_1 \quad \Gamma \subset \Gamma_1 \quad v \notin \text{dom}(\Gamma_1)}{\Pi; \Gamma \vdash_{\text{block}} (\textbf{get-label}\ l\ v; \textbf{jump}\ v)}\ \text{chk-block-jump}$$

Program typings. The judgement $\Pi \vdash_{\text{blocks}} \Psi$ means that the blocks Ψ are well-typed in the program-typing Π.

$$\frac{\forall l, \iota.\ \Psi(l) = \iota \rightarrow \exists \Gamma.\ \Pi(l) = \Gamma \wedge \Pi; \Gamma \vdash_{\text{block}} \iota}{\Pi \vdash_{\text{blocks}} \Psi}\ \text{chk-blocks}$$

Fig. 4. Simple type system

Here "nil" is a predicate that states that the value component of the world is exactly the value $\mathbf{L_0}$, and "pair $p\ q$" is a predicate that states that the value component of the world is a cons cell (v_0, v_1) where p holds on v_0 and q holds on v_1. Lists are then defined in a straightforward way via the recursion operator (here the symbol $||$ is disjunction at the level of the embedded logic). More elaborate predicates can be built up in a similar way for, e.g., continuations and whole-program typings.

Step (iii): List-machine Hoare Logic. Now that we have defined the specification logic and the semantics of the necessary type constructors, we need a way to show that the typing rules are sound. To do this we first define a general Hoare logic. It turns out that we can define the Hoare triple for this language via reduction to a more primitive notion of guarding [2]. We say that a predicate p "guards" instruction i if it is safe to execute i whenever p holds (in program Ψ). Concretely

(where $\Rightarrow$ stands for implication lifted into the specification logic):

$$\text{safe-instr}_\Psi\ i := \lambda(n, \rho, v).\, \text{safe-state}_\Psi(\rho, i)$$
$$\text{guards}_\Psi\ p\ i := p \Rightarrow \text{safe-instr}_\Psi i$$
$$\text{hoare}_{\Psi,\hat{\Pi}}\ p\ i\ q := \triangleright\hat{\Pi} \Rightarrow (\forall i',\ \text{guards}_\Psi\ (\Box q)\ i' \Rightarrow \text{guards}_\Psi\ (\Box p)\ (i; i'))$$

Recall that $\text{safe-state}_\Psi\ (\rho, i)$ holds when the state (ρ, i) cannot lead to a stuck state. In the definition of the Hoare triple, $\hat{\Pi}$ stands for a proposition stating preconditions for jumping to labels; it is the semantic counterpart of the whole-program typing Π and is used when reasoning about control-flow instructions. Notice that we only assume that these preconditions are correct "later." This restriction is important for proving whole-program correctness in the presence of arbitrary mutually-recursive control flow (see section 11.1 of Appel et. al. [5]).

Now we can prove lemmas corresponding to the Hoare rules for the list machine. The proofs for straight-line instructions proceed by unfolding the definition of the Hoare triple and showing how executing the instruction i, when p holds, is sufficient to show that q holds on the resulting state. Control-flow instructions use the preconditions stated in $\hat{\Pi}$, relying on the fact that executing a jump consumes time, which allows us to "unpack" the $\hat{\Pi}$ from under the later operator.

Step (iv): Semantic Soundness Proof. Once we have proved the correctness of the Hoare rules, we can use them to justify the rules of our type system. This almost always involves using a weakening lemma to show that the typing rule is just a weaker form of the Hoare rule; the proofs are usually straightforward.

Finally, we rely on the semantic definitions of the type judgments to conclude that the type system is sufficient for safety. Because we defined the Hoare triple in terms of guarding (which in turn is defined in terms of program safety), it is easy to show that derivations constructed using the type system (and therefore the underlying Hoare logic) are sufficient for program safety.

One could be forgiven for thinking that this approach is needlessly complicated for proving the soundness of a simple typechecker. It is true that the syntactic approach to proving soundness for this typechecker is quite a bit simpler and shorter. However, the semantic proof accomplishes more; it proves soundness for a general Hoare logic. This general logic can be *directly* reused for proofs involving other type systems. In other words, steps (*i*), (*ii*) and (*iii*) in the above list are *reusable.* This, we claim, is a major advantage of the semantic methods.

As an example, we have proved the soundness of a more advanced system than the simple one presented here; the advanced type system is able to typecheck more flexible uses of label values and typechecks a strictly larger set of programs. Its soundness proof directly reuses the Hoare logic and differs from the soundness proof for the simple type system by less than 200 lines of proof script. [3]

One notable difference between the semantic and syntactic approaches is that, in a syntactic system, typing rules are *axioms* (typically introduced by an inductive definition) whereas in a semantic proof, typing rules are simply *lemmas* proved via the definition of the typing judgment. This may have a significant impact on the trusted computing base of the system, as we discuss in section 4.

[3] This count does not include the soundness proof for the typechecking *algorithm* because it is independent of the proof method.

While constructing the simple type system soundness proofs, we observed an interesting difference in the areas which required the most mental energy. In the syntactic proof, intermediate lemmas proceed by structural induction. Finding the correct induction hypothesis can sometimes be difficult and may require tricks like generalizing the hypothesis, complete induction on depths, proving two related lemmas simultaneously, etc. All these techniques are well known, but discovering the correct application to prove a theorem of interest is something of an art.

In contrast, mental energy in the the semantic proof is concentrated primarily on getting definitions correct. For example, the definition of the "pair" predicate is rather subtle; it includes application of the $\triangleright$ operator to ensure contractiveness, thus guaranteeing a fixed point. Unlike inductive proofs, the bag of tricks for "semantic" proofs is not well known. Perhaps a useful battery of techniques will emerge as we acquire additional experience with this proof method.

3 An Example: Fault-Tolerant List Machine

A transient hardware fault occurs when an energetic particle strikes a transistor, causing it to change state. As a consequence, bits in registers, memory, or other processing components may be corrupted [9]. While infrequent on most current hardware, such faults have resulted in serious damage in well-publicized cases [15,9,24].

In order to provide software reliability in the face of transient faults, a number of researchers have proposed developing compilers that insert additional checking code into programs for the purpose of guaranteeing fault tolerance [16,20]. Unfortunately, it is difficult to validate the correctness of these compilers. One cannot tell by running the program whether it is properly fault tolerant because, in general, it will appear to work just fine.

Therefore, Perry *et al.* [18] developed a special kind of proof-carrying code that guarantees that the output of a compiler is properly fault tolerant. That is, they developed an FTTAL. They prove this sound, on paper, using syntactic techniques.

In this section, we will explain how to extend the operational semantics and type system for the list machine so as to model and check fault-tolerant code. Our fault-tolerant type system is simpler and weaker than the work done by Perry [18]. Nevertheless, the extension is quite a radical departure from the originally anticipated goals of the list machine, and consequently, it provides an interesting and challenging test case for the experimental framework.

3.1 Overview

The first step in the development of a framework for fault tolerance is to decide upon the fault model. In our case, we will adopt the Single Event Upset (SEU) model, which assumes that no more than one fault will occur during a run of the program. In addition, we will assume that faults only occur in machine registers. These are relatively standard assumptions in the literature (single-bit errors in memory are adequately detected by ECC). These assumptions will be modeled by altering the operational semantics of the list machine, as discussed in the upcoming sections.

Once we understand the fault model, we must come up with a fault-tolerant

$c ::= Green \mid Blue$
$\iota ::= \textbf{jump}\ v_G\ v_B \mid \textbf{get-label}\ c\ l\ v \mid \textbf{branch-if-nil}\ v_G\ v_B\ l \mid \textbf{fetch-field}\ v\ 0\ v' \mid \textbf{fetch-field}\ v\ 1\ v'$
$\quad \mid \textbf{cons}\ v_0\ v_1\ v' \mid \textbf{halt} \mid \textbf{fault} \mid \iota_0\ ;\ \iota_1$

Fig. 5. Syntax of fault-tolerant list machine

$$\frac{\rho(v_G) = l \quad \rho(v_B) = l \quad \Psi(l) = \iota'}{(\rho, \textbf{jump}\ v_G\ v_B) \overset{\Psi}{\mapsto} (\rho, \iota')}\ \text{step-jump} \qquad \frac{\rho(v_G) = \text{cons}(a_0, a_1) \quad \rho(v_B) = \text{cons}(a_0, a_1)}{(\rho, (\textbf{branch-if-nil}\ v_G\ v_B\ l; \iota)) \overset{\Psi}{\mapsto} (\rho, \iota)}\ \text{step-no-branch}$$

$$\frac{\rho(v_G) = l \quad \rho(v_B) = l' \quad l \neq l'}{(\rho, \textbf{jump}\ v_G\ v_B) \overset{\Psi}{\mapsto} (\rho, \textbf{fault})}\ \text{step-jump-fault} \qquad \frac{\rho(v_G) = \mathbf{L_0} \quad \rho(v_B) = \mathbf{L_0} \quad \Psi(l) = \iota'}{(\rho, (\textbf{branch-if-nil}\ v_G\ v_B\ l; \iota)) \overset{\Psi}{\mapsto} (\rho, \iota')}\ \text{step-branch}$$

$$\frac{\rho[v := l] = \rho'}{(\rho, \textbf{get-label}\ c\ l\ v; \iota) \overset{\Psi}{\mapsto} (\rho', \iota)}\ \text{step-getlabel} \qquad \frac{\rho(v_G) \neq \rho(v_B)}{(\rho, (\textbf{branch-if-nil}\ v_G\ v_B\ l; \iota)) \overset{\Psi}{\mapsto} (\rho, \textbf{fault})}\ \text{step-branch-fault}$$

Fig. 6. Small-step fault tolerant operational semantics. *Only the modified rules are shown.*

solution. In our case, the solution is to compute every observable program result twice and to check the two results against one another. If we detect a difference between the two results, we abort the program and signal an error. [4] In order for this solution to work properly, neither of the two redundant computations must depend upon the other. For example, if computation 1 produces a value, then computation 2 is not allowed to use that value. Otherwise, a single fault can percolate to both computations and avoid detection. Hence, the central job of the type system for fault tolerance is to guarantee the independence of the two computations. If it does so correctly, it will guarantee a strong reliability property: If a *fault-free* run of a program successfully executes to completion and delivers a final value a, then in any *faulty* run of that program, either the fault will be (i) detected and the program will terminate in the special "fault state", or (ii) completely masked and the program will terminate successfully and deliver the correct final value a.

We have the type system assign a color, either blue or green, to each computation. Blue values are created and manipulated only by computation 1 while green values are created and manipulated only by computation 2, ensuring that the two redundant computations are independent.

3.2 The Modified List Machine

We use a slightly modified syntax for the fault-tolerant list machine in Figure 5. Aside from branching instructions, most instructions remain the same because we generally ensure fault tolerance by duplicating instructions rather than relying on hardware support. Control flow, however, cannot be protected via this mechanism.

Consider a source language statement, **if** $e \neq$ **nil then** A **else** B. In list-machine assembly language, there will be a block of instructions computing e into some register v, followed by **branch-if-nil** $v\ l$. In order to protect this branch from the influence of a fault, we compute another value v' independently of v. In the SEU

[4] More sophisticated systems both detect and recover from faults. We have chosen to omit recovery procedures in our toy example.

$\Gamma ::= \quad v \rightharpoonup (C, \tau) \quad$ Type environments

$$\frac{\Gamma(v_G) = (G, \text{listcons}\,\tau) \quad \Gamma(v_B) = (B, \text{listcons}\,\tau)}{\Pi \vdash_{\text{instr}} \Gamma\{\textbf{branch-if-nil}\ v_G\ v_B\ l\}\Gamma} \text{ chk-br-cons}$$

$$\frac{\Gamma(v) = (c, \text{listcons}\,\tau) \quad \Gamma[v' := (c, \tau)] = \Gamma'}{\Pi \vdash_{\text{instr}} \Gamma\{\textbf{fetch-field}\ v\ 0\ v'\}\Gamma'} \text{ chk-fetch-0}$$

$$\frac{\Gamma(v) = (c, \text{listcons}\,\tau) \quad \Gamma[v' := (c, \text{list}\,\tau)] = \Gamma'}{\Pi \vdash_{\text{instr}} \Gamma\{\textbf{fetch-field}\ v\ 1\ v'\}\Gamma'} \text{ chk-fetch-1}$$

$$\frac{\Gamma(v_G) = (G, \text{nil}) \quad \Pi(l) = \Gamma_1 \quad \Gamma(v_B) = (B, \text{nil}) \quad \Gamma \subset \Gamma_1}{\Pi \vdash_{\text{instr}} \Gamma\{\textbf{branch-if-nil}\ v_G\ v_B\ l\}\Gamma'} \text{ chk-br-nil}$$

$$\frac{\Gamma(v_0) = (c, \tau_0) \quad \tau_0 \subset \tau' \quad \tau_1 \subset \text{list}\,\tau' \quad \Gamma(v_1) = (c, \tau_1) \quad \Gamma[v := (c, \text{listcons}\,\tau')] = \Gamma'}{\Pi \vdash_{\text{instr}} \Gamma\{\textbf{cons}\ v_0\ v_1\ v\}\Gamma'} \text{ chk-cons}$$

$$\frac{\Gamma[v := (c, \text{nil})] = \Gamma'}{\Pi \vdash_{\text{instr}} \Gamma\{\textbf{get-label}\ c\ \mathbf{L_0}\ v\}\Gamma'} \text{ chk-getnil}$$

$$\frac{\Gamma(v_G) = (G, \text{list}\,\tau) \quad \Pi(l) = \Gamma_1 \quad \Gamma[v_G := (G, \text{nil})][v_B := (B, \text{nil})] = \Gamma' \quad \Gamma(v_B) = (B, \text{list}\,\tau) \quad \Gamma' \subset \Gamma_1 \quad \Gamma[v_G := (G, \text{listcons}\,\tau)][v_B := (B, \text{listcons}\,\tau)] = \Gamma''}{\Pi \vdash_{\text{instr}} \Gamma\{\textbf{branch-if-nil}\ v_G\ v_B\ l\}\Gamma''} \text{ chk-br-list}$$

$$\frac{\Pi(l) = \Gamma_1 \quad \Gamma \subset \Gamma_1 \quad v_G \notin \text{dom}(\Gamma_1) \quad v_B \notin \text{dom}(\Gamma_1)}{\Pi; \Gamma \vdash_{\text{block}} (\textbf{get-label}\ G\ l\ v_G; \textbf{get-label}\ B\ l\ v_B; \textbf{jump}\ v_G\ v_B)} \text{ chk-block-jump}$$

Fig. 7. Fault-tolerant type system. *Only the modified rules are shown.*

model, only one of these registers can be corrupt; if the two registers are equal, then no fault could have taken place. We wish to test $v = v'$ for fault detection and we wish to test $v = \mathbf{L_0}$ to implement the program logic. If we do these tests sequentially, we might be so unlucky as to experience a bit flip in between the two instructions. Therefore the *fault-tolerant list machine* has a special branch instruction of the form **branch-if-nil** v_G v_B l, where v and v' are now v_G and v_B, to do both comparisons at once. It can have three outcomes as shown in Figure 6. If $v_G \neq v_B$ then it transitions to the **fault** state (`step-branch-fault`). If $v_G = v_B = \mathbf{L_0}$, the instruction will jump to label l (`step-branch`), otherwise it will fall through (`step-no-branch`). The **fault** state is reached only when both a bit flip has occurred and when some fault-tolerant code has detected the resulting inconsistency. With the introduction of **fault**, we extend the definition of a safe-state$_\Psi$ to include $(\rho, \textbf{fault})$ so that detecting a fault is considered safe. While our notation implies that v_G and v_B should be colored green and blue respectively, this is enforced by the type system and not the machine. The same concept applies to the jump instruction.

In Figure 5, we also annotate the **get-label** instruction with a color even though it does not control program flow. This color is ignored by the machine but is used by the typechecker to infer the register's color.

3.3 The Modified Typechecking Algorithm

Section 3.1 introduced the need to have two independent, but equal, computations to detect faults. This is achieved by assigning a color to each of the two computations, resulting in an extended typing environment where each register maps to a color in addition to its stored type, shown in Figure 7. A register receives its color when a literal value is loaded: rule `chk-getnil`. Due to a simplification in the type system, we do not assign a color to registers in the `chk-block-jump` rule because the registers are not added to the typing environment.

For other instructions, we maintain the independence of one computation from

$$\frac{(\rho,\iota) \overset{\Psi}{\mapsto} (\rho',\iota')}{(\rho,\iota) \overset{\Psi}{\mapsto}_z (\rho',\iota')}\ \text{step-no-fault} \qquad \frac{\rho[v := a] = \rho' \quad (\rho',\iota) \overset{\Psi}{\mapsto} (\rho'',\iota')}{(\rho,\iota) \overset{\Psi}{\mapsto}_z (\rho'',\iota')}\ \text{step-fault}$$

$$\frac{(\rho_1,\iota_1) \overset{\Psi}{\mapsto} (\rho_1',\iota_1') \quad (\rho_2,\iota_2) \overset{\Psi}{\mapsto}_z (\rho_2',\iota_2')}{(\rho_1,\iota_1,\rho_2,\iota_2) \overset{\Psi}{\leadsto} (\rho_1',\iota_1',\rho_2',\iota_2')}\ \text{step-ft}$$

Fig. 8. Additional operational semantics to model faults

the other with typing rules that require the source and destination registers to have the same color. An example of this is rule `chk-cons`. Since we require branch instructions to use a value computed twice independently, their typing rules, such as `chk-br-list`, require that one register be green and the other blue. However, the typechecker does not guarantee that the two values, while independent, will be equal in the absence of a fault.

3.4 Fault Tolerance

For a well-typed program, we must prove two main theorems: (i) a fault does not induce undefined behavior and (ii) faulty executions *simulate* fault-free executions. Simulation is satisfied when the register stores of a faulty and fault-free run are equal except for, in the presence of a fault, registers of a particular color (the color of the register that was corrupted by the fault).

In order to prove fault tolerance, which is a simulation relation between faulty and non-faulty runs of a program, we construct new operational semantics for the proofs. These new semantics are presented in Figure 8. In most proofs, we replace $(\rho,\iota) \overset{\Psi}{\mapsto} (\rho',\iota')$ with $(\rho_1,\iota_1,\rho_2,\iota_2) \overset{\Psi}{\leadsto} (\rho_1',\iota_1',\rho_2',\iota_2')$ such that the proofs deal with both fault free and *potentially* faulty steps at the same time. When considering multiple steps, we require that $\iota_1 = \iota_2$ and either $\iota_1' = \iota_2'$ or $\iota_2' = \textbf{fault}$. That is, until a fault is detected, the sequence of instructions executed by the faulty run is the same as the fault-free run. Finally, ρ_1' must simulate ρ_2' if ρ_1 simulates ρ_2.

As a result, we must modify various safety proofs to deal with both faulty and fault-free computations. This mostly results in adding redundant cases to each proof except where the presence of a fault may be detected. So far, this has been a straighforward process. However, these proofs have yet to be completed; the remainder is a work in progress.

4 How Semantic and Syntactic Methods Scale

How do the syntactic and semantic methods scale to industrial-strength TALs? We cannot use the list-machine TAL to answer that question, as it's far too simple. Instead we analyze two large projects, the Princeton Foundational Proof-Carrying Code (FPCC) project [1], and the Carnegie Mellon ConCert project [12]. Each of these projects included a TAL for ML compiled to a real machine (Sparc, and a subset of x86, respectively). A big difference between the two approaches, however, was that FPCC used semantic methods to prove its type system sound, whereas ConCert used syntactic methods.

It is difficult to compare the complexities of two large software artifacts. Large

projects can differ in many ways: goals, design choices (such as the choice of semantic or syntactic methods), implementation language, "coding" styles, and many others. Fortunately, FPCC and ConCert are similar enough to provide a useful comparison. The stated goal of both FPCC and ConCert was to guarantee a memory safety property for untrusted code. Each project was developed in the Twelf proof assistant, so it is reasonable to make quantitative measures of the developments.

What should be measured? One obvious and important metric is the manpower required to produce the projects. FPCC took more graduate students more time to produce, but quantifying how much more time is difficult due to a lack of accurate timekeeping methods. One key reason why it took the FPCC researchers longer to complete their proof obligations was that it was necessary to invent completely new semantic techniques, including a semantic model to handle the combination of mutable references, contravariant recursive types, and impredicative polymorphism. In contrast, ConCert was able to rely on "off-the-shelf" syntactic techniques, developed by the programming language community over the last ten to fifteen years to handle such features. The ability to use such off-the-shelf methods gives syntactic methods an advantage at the present time, but continuing research on semantic methods may mitigate this advantage in the future. In particular, while the FPCC model was challenging and time-consuming to develop the first time, many of the ideas and even the proofs themselves are highly reusable. Already, the Concurrent C minor project [14] has seen great cost savings from the ability to re-use semantic libraries.

The nature of the projects themselves suggests another possibility. Consider a proof P of a theorem T, in a logic L, checked by a machine M. The size and difficulty of producing the proof P is important because it affects the human cost of verification, and was discussed in the previous paragraph. However, unlike T, L and M, P does not have to be trusted – errors in P will be caught by M. On the other hand, errors in T (the statement of the theorem), L (the rules of a logic), or M (the implementation of a checker) will not be caught by any mechanical verifier. These three components form the system's trusted computing base (TCB).

Fortunately, both FPCC and ConCert are organized such that the size of T, L and M are orders of magnitude smaller than P, and hence, relative to the overall size of the projects, the TCB is small. Indeed, unlike most other PCC systems, both ConCert and FPCC were foundational systems, meaning that they were developed to *minimize* the difficulty of trusting the TCB. Therefore, it is interesting to compare the TCBs of the two systems and we do so here.

FPCC and ConCert both guarantee a memory safety property. That is, the theorem T is memory safety (and thus this theorem must include a specification of the instruction-set semantics of the target machine). Both systems guarantee memory safety by providing a type system and proving that well-typed programs obey the memory policy. That is, the specification of the type system is part of the proof P, and is not part of the specification of T. In other words, the type system was only used to prove a program was consistent with the memory policy, meaning that the policy had to be trusted, but not the type system. In each system, T includes: axioms relevant to machine computation (*e.g.* the definition of modular arithmetic); a machine definition (of the SPARC for FPCC or a subset of x86 for

	Axioms	Machine Def.	Policy	Runtime	Checker
FPCC	6,136	16,624	1,198	197	4,353
ConCert	2,808	16,577	524	3,973	326,937

Fig. 9. Token count of TCB components

ConCert), a policy (the statement of of the memory safety property), and a runtime system (used by secured programs).

The logic L_{FPCC} is LF; the logic L_{ConCert} is LF *metatheory*, which is not the same thing! Although FPCC was developed in Twelf, its theorems are stated as LF types, and all proofs are constructed as λ-expressions satisfying those types. That is, proof checking is just LF typechecking. That is, M_{FPCC} is just a typechecker. As part of the Princeton FPCC project, the Flit typechecker was developed. Flit is a very minimal LF typechecker written in 800 lines of C using no libraries [6]. In contrast, ConCert uses the metatheory of Twelf as its logic: modes, totality, and coverage checking. Thus, M_{ConCert} is a substantially larger software system.

At the moment, no minimal metatheorem checker comparable to Flit exists. However we speculate that since, in general, checking metatheorems is harder than checking theorems, such a system would be substantially larger than Flit.

The simplest meaningful measurement of TCB complexity is a token count (ignoring comments). While this measure is not ideal, it does help to correct for differences in style more than line or character counts do. Moreover, the size of a software artifact is clearly proportional, *ceteris paribus,* to the difficulty in trusting that artifact. The results of this count are contained in Figure 9.

Before taking into account the checkers, the TCBs of the two systems are comparable in size. For further analysis, the TCBs have been broken into roughly comparable parts. The axioms of FPCC are approximately twice as long as for ConCert [5], but this is due to the fact that ConCert is taking advantage of a much more powerful checker. The machine definitions are almost exactly the same size. FPCC has a moderately larger policy than ConCert. ConCert has a substantially larger runtime system, but it supports more features, such as a garbage collector and various grid computing primitives. Therefore, before the checker is taken into account, the two are almost exactly equal (24,155 for FPCC vs. 24,001 for ConCert).

From memory safety to type safety. What additions would be required if we wished to enforce a richer policy than memory safety, such as guaranteeing that modules obeyed their type interfaces? Two basic changes are required. First, we must update the policy to add this additional property. Furthermore, the policy now makes reference to the type system: if we want to say that a certain value is an "int→bool", we must know how ints and bools are represented, as well as the calling conventions for functions. So we must include a definition of the type system in the trusted base (in T, not in P). Would it be better to include a semantic definition or a syntactic definition of the type system?

A key observation is that the definition size of the two methods scales differently. In general, a semantic system requires a new definition (the denotation of a type operator) whenever a new type constructor is added, whereas a syntactic system

[5] Of course, strictly speaking ConCert has no axioms in the sense that FPCC does; but we write "axioms" to include foundational definitions such as the construction of modular arithmetic.

Semantic FPCC	Syntactic FPCC	ConCert (XTALT)	ConCert (TALT)
14,063	29,983	22,464	25,129

Fig. 10. Token count for type system definitions

requires a new definition (a typing rule) whenever a new expression constructor is added. In a toy system, it is common for the number of expression constructors to be limited to just those which can showcase a new typing feature, but in a real setting such as the SPARC processor, the number of expression constructors dwarfs the number of type constructors [6]. Since FPCC defines its type system semantically, whereas ConCert defines its syntactically, we can see how these factors actually balance in a large system.

To produce this measurement, we identified the type operators (for FPCC) and typing rules (for ConCert). We developed a proof-dependency tool to isolate all of the definitions on which those definitions depended, and removed all unneeded definitions and comments before measuring. For additional comparison, we measured the FPCC type system as though it were syntactic by taking the typing lemmas (that is, the typing rules of the FPCC type system), stripping the soundness proofs to get typing rules as typically defined in syntactic systems, and measuring the resulting "axioms". The resulting measurements are contained in Figure 10.

The apples-to-apples comparison is between Syntactic FPCC and TALT: both of these are syntax-directed type systems; and indeed they have similar sizes [7]. The difference between 30k and 25k for the syntactic systems is likely explained by the different SML compilers and machine subsets that they cover.

The FPCC semantic definitions are about half the size of the syntactic definitions. We believe that this is explained by the scaling law. Furthermore, suppose each system (FPCC and ConCert) were to be extended to cover more machine instructions generated by a fancier compiler. The scaling law would drive the comparison even more in favor of the semantic methods, *since the syntactic type system definitions would grow (more typing rules for more instructions) while the semantic ones would remain constant.* Therefore we conclude that in large, real systems where there is a desire to guarantee a typing policy, defining a type system semantically instead of syntactically may result in a substantially smaller TCB.

5 Related Work

Perhaps the best-known work in the area of mechanized programming language metatheory is the POPLmark challenge [8]. The concrete problem posed in the POPLmark challenge differs from the work considered here considerably; POPLmark focuses on the metatheory of $F_{<:}$ whereas we consider TALs. However, both are different means to the same end, the mechanization of metatheory.

[6] One might think that an average semantic type definition would be larger than the average typing judgment. This is often the case for toy systems; however, in realistic TALs the typing rules often have a large number of complicated premises and it is not clear that the rules are simpler than the type definitions.

[7] XTALT is simpler than TALT because it is not obliged to be syntax-directed. On the other hand there is no soundness proof for TALT; instead, the untrusted TALT is used (for each program) to produce a derivation in XTALT, and it is XTALT that has a soundness proof. Since FPCC is both sound and syntax-directed, derivations are not necessary in this way.

Many groups have investigated software techniques for improving program reliability in the presence of transient faults [7,22,17,10,16,21,20]. In an effort to better understand major semantic issues, formal models of the problem have been studied in the context of the lambda calculus [20], and TALs [16]. The fault-tolerant list machine presented here follows in a similar vein.

6 Conclusion

The list-machine 2.0 benchmark serves as a vehicle for communicating in an accessible and concrete way how semantic proofs work, facilitates a direct head-to-head comparison between different proof methods, and provides an experiment testbed for innovations like fault tolerance.

After our experiments with semantic FTTAL proofs, we hope to extend these techniques beyond the list machine to more realistic systems and real compilers that generate fault-tolerant code. Perry *et al.* have described more realistic FTTALs, but with paper proofs instead of machine-checked proofs [18,19].

The semantic proof used by the list machine is potentially more scalable for larger, more complex systems, as demonstrated by the comparison between FPCC and ConCert. Syntactic proof techniques have been much studied over the past 10-15 years. As a consequence, the proof techniques are well known and it is relatively easy for experts to hammer out proofs of many complex systems. On the other hand, semantic techniques have been less well studied over the same time period. Consequently, coming up with unified semantic models for various features including references, polymorphism and concurrency was initially very challenging and time-consuming for the FPCC framework. However, once the overhead of implementing these basic semantic models is complete, extending semantic proofs requires less work. With these new techniques in hand, as well as reusable, higher-level libraries, such as Appel et al.'s modal model [5], future projects will have great advantages in terms of both TCB size and proof engineering effort.

Semantic models for features such as references usually require a syntactic element. Yet certain advanced type-theoretic concepts, like dependent types, resist an entirely syntactic analysis. In the end, complex future systems may well see a blending of both semantic and syntactic techniques.

Acknowledgments. We thank the program committee of the PCC workshop for excellent feedback on our work. This research is funded in part by NSF awards CNS-0627650 and CCF-0540914. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

References

[1] A. W. Appel. Foundational proof-carrying code. In *Symp. on Logic in Computer Science (LICS '01)*, pages 247–258. IEEE, 2001.

[2] A. W. Appel and S. Blazy. Separation logic for small-step C minor. In *20th International Conference on Theorem Proving in Higher-Order Logics (TPHOLs 2007)*, 2007.

[3] A. W. Appel and A. P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of*

Programming Languages, pages 243–253, 2000.

[4] A. W. Appel and X. Leroy. A list-machine benchmark for mechanized metatheory. Technical Report RR-5914, INRIA, May 2006.

[5] A. W. Appel, P.-A. Melliès, C. D. Richards, and J. Vouillon. A very modal model of a modern, major, general type system. In *Proc. 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)*, pages 109–122, Jan. 2007.

[6] A. W. Appel, N. G. Michael, A. Stump, and R. Virga. A trustworthy proof checker. *Journal of Automated Reasoning*, 31:231–260, 2003.

[7] A. Avizienis and L. Chen. On the implementation of nversion programming for software fault tolerance during execution. In *COMPSAC*, pages 149–155, 1977.

[8] B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The POPLmark challenge. In *International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, August 2005.

[9] R. C. Baumann. Soft errors in commercial semiconductor technology: Overview and scaling trends. In *IEEE 2002 Reliability Physics Tutorial Notes, Reliability Fundamentals*, pages 121_01.1–121_01.14, April 2002.

[10] J. Chang, G. A. Reis, and D. I. August. Automatic instruction-level software-only recovery methods. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, June 2006.

[11] A. Chlipala. E-mail communication via the POPLmark mailing list. http://lists.seas.upenn.edu/pipermail/poplmark/2008-April/000411.html.

[12] K. Crary. Toward a foundational typed assembly language. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 198–212, New York, NY, USA, 2003. ACM.

[13] R. Dockins, A. W. Appel, and A. Hobor. Multimodal separation logic for reasoning about operational semantics. In *Proc. 24th Conference on the Mathematical Foundations of Programming Semantics*, May 2008. To appear.

[14] A. Hobor, A. W. Appel, and F. Zappa Nardelli. Oracle semantics for concurrent separation logic. In *Proc. European Symp. on Programming (ESOP 2008)*, 2008. to appear.

[15] S. E. Michalak, K. W. Harris, N. W. Hengartner, B. E. Takala, and S. A. Wender. Predicting the number of fatal soft errors in los alamos national labratorys asc q computer. In *IEEE Transactions on Device and Materials Reliability*, volume 5, pages 329–335, September 2005.

[16] N. Oh and E. J. McCluskey. Low energy error detection technique using procedure call duplication. In *Proceedings of the 2001 International Symposium on Dependable Systems and Networks*, 2001.

[17] N. Oh, P. P. Shirvani, and E. J. McCluskey. Error detection by duplicated instructions in super-scalar processors. In *IEEE Transactions on Reliability*, volume 51, pages 63–75, March 2002.

[18] F. Perry, L. Mackey, G. A. Reis, J. Ligatti, D. I. August, and D. Walker. Fault-tolerant typed assembly language. In *ACM SIGPLAN Conference on Programming Language Design and Implemenation (PLDI)*, June 2007.

[19] F. Perry and D. Walker. Reasoning about control flow in the presence of transient faults. In *International Static Analysis Symposium*, July 2008.

[20] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. Swift: Software implemented fault tolerance. In *CGO '05: Proceedings of the international symposium on Code generation and optimization*, pages 243–254, Washington, DC, USA, 2005. IEEE Computer Society.

[21] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. S. Mukherjee. Software-controlled fault tolerance. *ACM Trans. Archit. Code Optim.*, 2(4):366–396, 2005.

[22] P. P. Shirvani, N. Saxena, and E. J. McCluskey. Software implemented edac protection against seus. In *IEEE Transactions on Reliability*, volume 49, pages 273–284, 2000.

[23] A. K. Wright and M. Felleisen. In *A Syntactic Approach to Type Soundness*, volume 115, pages 38–94, 1994.

[24] J. F. Ziegler and H. Puchner. Ser - history, trends, and challenges: A guide for designing with memory ics. 2004.