Typed Machine Language and its Semantics

Kedar N. Swadi
Princeton University
kswadi@cs.princeton.edu

Andrew W. Appel
Princeton University
appel@cs.princeton.edu

ABSTRACT

We present TML, a new low level typed intermediate language for the proof-carrying code framework. The type system of TML is expressive enough to compile high level languages like core ML to and can be guaranteed sound. It is also flexible enough to provide a lot of freedom for low-level data representations. We can model real machine instructions in TML, and thus avoid high-level opaque operations like memory allocation and perform provably safe optimisations like array bounds check eliminations. Most important, TML has a semantic model.

1. INTRODUCTION

Proof-carrying code (PCC) [11] is a framework for the generation of provably safe programs. In this framework, an untrusted program is accompanied by a proof of its compliance with some predefined safety policy (resource access, type safety, or memory safety). The host mechanically checks the proof for correctness to ensure safety of execution of the untrusted code. Though this technique is general enough to encode a very wide range of safety properties, existing PCC systems [10] suffer from a lack of flexibility with respect to the high-level source languages they translate from and the target machine architectures they compile to.

Building on a semantic model of machine-level types [4], we design TML, a new low-level language that is general enough so that a wide variety of high-level languages may be compiled to it, and is also retargetable to different machines.

TML makes the following contributions :

- We have a semantic model for TML types based on a very small set of axioms. Our trusted computing base comprises the rules of higher-order logic augmented by some elementary facts about number arithmetic. All TML types are modelled as (defined) predicates, and type inference rules are lemmas for which we provide machine-checkable proofs. Previous approaches to low-level semantics [9, 11] did not provide models, but only syntactic consistency results.
- Low-level type constructors for type intersections and address arithmetic give front-end compilers more freedom in data layout than high-level (TAL-style) records or objects would.
- TML is really a machine language, with instructions specified by integer opcodes; thus we do not need to trust an assembler.
- Unlike TAL [9] or DTAL [16], we can model each machine instruction in TML, even those occuring in sequences allocating a heap record. While TAL and DTAL have an atomic "malloc" instruction that expands into several real machine instructions, TML is expressive enough to reason about the intermediate states in the malloc subroutine; similarly for multi-instruction case-discrimination sequences. Each TML instruction corresponds to a single instruction on a real machine like SPARC. This allows a compiler to perform provably safe optimisations and the trusted computing base is also made smaller as a result.
- We can encode complex dataflow facts within the type system. Tracking dataflow allows us to reason about intermediate machine states and thus make operations like sum-type discrimination nonatomic in a safe way.
- Like other PCC systems, TML has no decision procedure: we assume that a type-preserving compiler from a safe source language produces hints and invariants useful in constructing machine-checkable proofs. However, because each typing rule is proved separately as a derived lemma (not through a global syntactic metatheorem), we find it easy to add ad-hoc rules for the convenience of the prover or compiler (in the case where no change is necessary to the semantic model).

2. TYPED MACHINE LANGUAGE

TML is stratified into the levels of *kinds*, *types* and *values*. To reduce complexity of the system we have only two kinds in TML. There are no other kind constructors in TML like functions or pairing. This simplifies the semantic model. The type constructors are sufficiently low-level to allow translation from a wide range of source languages into TML. We cannot model quantification over higher-order kinds, which are required by the polymorphic typed lambda calculus F_ω. We can extend TML to higher kinds such as $\Omega \rightarrow \Omega \rightarrow \Omega$, but we can't quantify over these higher kinds; still, this is enough to do type constructors in core ML.

2.1 TML Kinds

To reason about machine-level programs with machine-level types, in TML we have kinds for *instructions* and *type maps*. *Type maps* are judgements on a vector of values associating them with their types, and have the kind Ω. In our semantic model, we can use type maps also to represent scalar types (judgements on a single value, not a vector) and also to represent numbers. The main motivation to unify the three kinds into one was to have a smaller set of quantification operators and also fewer rules for specifying static semantics.

Preliminary version, July 2001

2.2 Types ($\tau : \Omega$)

In TML, the state comprises a register bank R and a memory M. All accessible values live in registers of R. These values may point to data in memory M. If x is a number then $\mathsf{R}(x)$ is the value of the xth register; informally, we write r_x. Unlike TAL, which distinguishes address values from integer values, our machine-level calculus uses numbers to index memory.

As in the Appel-Felty semantic model of types [4], types are encoded as predicates on values. Then, $\mathsf{M} \vdash r_x : \tau$ if M and $\mathsf{R}(x)$ satisfy the predicate τ. TML has the following primitive types:

- $\top$, $\bot$: Every value has type $\top$; no value has type $\bot$.
- id : The identity type-constructor i.e., $\mathsf{id}(\tau) = \tau$.
- $\mathsf{box}(\tau)$: The box type constructor is for memory references. $v : \mathsf{box}(\tau)$ if the content of $\mathsf{M}(v)$ satisfies the predicate τ. box makes immutable references. TML can accommodate mutable references using the semantic model of Ahmed, Appel, and Virga [2], but we will not show details here.
- $\mathsf{rec}(\tau)$: This constructor allows us to define recursive types.
- $\mathsf{offset}(i)(\tau)$: The value v has this type if $v+i$ satisfies τ. This constructor is useful wherever address arithmetic is required. For example, if location $\mathsf{R}(x)$ contains a pointer to a record with two fields $[0 : \tau_1, 1 : \tau_2]$, we could say r_x : $\mathsf{offset}(1)\ (\mathsf{box}\ \tau_2)$. A convenient abbreviation, $\mathsf{field}\ c\ \tau$ is the same as $\mathsf{offset}(c)\ (\mathsf{box}\ \tau)$.
- $\tau_1 \cap \tau_2$ is the type of a value satisfying predicates for both τ_1 and τ_2, while a value of type $\tau_1 \cup \tau_2$ satisfies predicates for at least one of types.
- $\forall, \exists$: These are the universal and existential quantifiers. We allow quantification over Ω but not over *INSTR*. We use de Bruijn indices [6] (see Section 3.1) rather than variables, so $\forall$ or $\exists$ implicitly binds a de Bruijn variable. Informally, however, we will often show variables with the quantifiers.
- rec : General (covariant, contravariant) recursive types are written as $\mathsf{rec}\ \tau$, where rec implicitly binds a de Bruijn index that may be free in τ. For appropriate ("contractive") expressions τ, we have $\mathsf{rec}\ \tau$ is a fixed point of τ, which is written as $\mathsf{rec}\ \tau = \tau[\mathsf{rec}\ \tau \cdot \mathsf{id}]$ in the calculus of explicit substitutions.
- $\mathsf{codeptr}$: The judgement $v : \mathsf{codeptr}\ \phi$ holds if v is a code pointer with formal parameters ϕ. That is, it is safe to jump to location v if the registers satisfy ϕ. The formal parameters are modelled as a type map (Section 2.4).
- $\mathsf{at}(\iota, s)$: If at memory location l, we have a TML instruction ι of size s, then $l : \mathsf{at}(\iota, s)$.

2.3 Integers ($n : \Omega$)

In TML, we can specify the types of integers, and constant and bounded integers.

- $\mathsf{int}_\pi(i)$: The type of integers x such that $x\ \pi\ i$. For example, $\mathsf{int}_<100$ is the type of integers less than 100, and $\mathsf{int}_=(c)$ is the singleton type of integers equal to c. For example, the type of machine integers is $\mathsf{int}_{32} = \mathsf{int}_\geq 0 \cap \mathsf{int}_< 2^{32}$.
- $+$: This type captures the result of additions. After executing $r_i \leftarrow r_j + r_k$ where $r_j : \mathsf{int}_= n_1$ and $r_k : \mathsf{int}_= n_2$, we infer r_i : $\mathsf{int}_=(n_1+n_2)$ Other arithmetic operators like subtraction or multiplication on integers can also be easily defined, as can modular arithmetic.

These constructors allow us a great deal of flexibility for data representation. Consider, for example, a list-of-integers datatype. List cells could have untagged or tagged representations. In an untagged scheme (Figure 2a), if r_1 is a pointer to a list, assuming a 4-byte word, we represent this as

$$r_1 : \mathsf{int}_=0 \cup (\mathsf{int}_{\neq}0 \cap (\mathsf{field}\ 0\ \mathsf{int}_{32}) \cap (\mathsf{field}\ 4\ \tau))$$

where the left union type is the nil case and the right is the cons case. If r_1 contains a non-zero value it points to a record of two fields, the first being the data and the second being the pointer to the next cell. This makes pointers non-abstract in TML. (This example is a simplification which avoids dealing with the recursive nature of the list data type; see section 3.1 for a discussion of recursive types.)

	Kinds	
κ	$::=$	Ω
	$\mid$	*INSTR*
	Types (Ω)	
τ, ϕ, Γ	$::=$	$\top \mid \bot$
	$\mid$	$\mathsf{codeptr}\ \phi$
	$\mid$	$\mathsf{offset}\ (n, \tau)$
	$\mid$	$\mathsf{id}\ \tau$
	$\mid$	$\mathsf{box}\ \tau$
	$\mid$	$\mathsf{rec}\ \tau$
	$\mid$	$\tau \cap \tau' \mid \tau \cup \tau'$
	$\mid$	$\forall\ \tau \mid \exists\ \tau$
	$\mid$	$\{n : \tau\}$
	$\mid$	$\phi \backslash n$
	$\mid$	$\mathsf{int}_\pi n$
	$\mid$	$n_1 + n_2$
	$\mid$	$\mathsf{at}(\iota, n)$
	$\mid$	$\underline{n}$
	Instructions (*INSTR*)	
ι	$::=$	$\mathsf{instr}(\Gamma, \phi, \phi')$
	$\mid$	$\forall_\iota\ \iota$
	$\mid$	$\exists_\iota\ \iota$
π	$::=$	$> \mid \geq \mid < \mid \leq \mid = \mid \neq$

Figure 1: TML Syntax : Kinds and Types

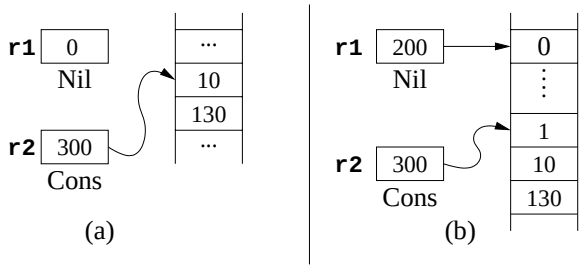

Figure 2: Data Representation

To get a tagged representation (Figure 2b) we write

$$r_1 : \quad (\mathsf{field}\ 0\ \mathsf{int}_=0) \cup (\mathsf{field}\ 0\ \mathit{int}_=1 \cap \mathsf{field}\ 4\ \mathsf{int}_{32} \cap \mathsf{field}\ 8\ \tau)$$

In this case, a nil cell has a tag of 0, and the cons cell has a nonzero tag field followed by the data and the next-cell pointer.

2.4 Type Maps ($\phi : \Omega$)

A type map is a collection of typing judgements for registers, associating each register r_x to some type τ_x. For most purposes, type maps serve the function of environments. If any register is not explicitly referenced in the environment, it is assumed unconstrained.

- The empty type map is $\top$.
- $\{i : \tau\}$: This is a *singleton* type map, where the only judgement is that register r_i has type τ.
- $\phi_1 \cap \phi_2$: If register bank $\mathsf{R} : \phi_1$ and $\mathsf{R} : \phi_2$, we have $\mathsf{R} : \phi_1 \cap \phi_2$. We also write $\{i : \tau_1, j : \tau_2\}$ for $\{i : \tau_1\} \cap \{j : \tau_2\}$.
- $\phi \backslash i$ contains the type judgements for all registers in ϕ except for the r_i.

Existential and singleton types allow us to capture dataflow information which can be used to relate the contents of two registers. For example, if $\mathsf{R}(i) = \mathsf{R}(j)$, we can express this fact as

$$\exists n.\{i : \mathsf{int}_{=} n, j : \mathsf{int}_{=} n\}$$

A more general type map, relate, captures a wider class of such register relations.

$$\begin{array}{l} \mathsf{relate}_{\pi}(F, G)(i, j) \equiv \\ \quad \exists n. \\ \quad \{r_i : F\ (\mathsf{int}_{\pi}(n))\} \cap \{r_j : G\ (\mathsf{int}_{=}(n))\} \end{array}$$

We use the following example to illustrate the use of relate. Let $\phi_1 = \{r_i : \mathsf{box}\ \tau\}$. We fetch the contents of the memory (M) at $\mathsf{R}(i)$ into register r_j, ($r_j \leftarrow \mathsf{M}[r_i]$). This operation results in a new environment, ϕ_2. Trivially, $\mathsf{R}(j) = \mathsf{M}[\mathsf{R}(i)]$ in ϕ_2. This fact is expressed using the relate constructor as $\mathsf{relate}_{=}(\mathsf{box}, \mathsf{id})(i, j)$. In the definition above, the existential is instantiated with $\mathsf{M}[\mathsf{R}(i)]$. This instantiation can be shown to satisfy the two sub-environments. We now have
$\phi_2 = \{i : \mathsf{box}\ \tau\} \cap \mathsf{relate}_{=}(\mathsf{box}, \mathsf{id})(i, j)$. From this environment, we have lemmas which can infer that $r_j : \tau \in \phi_2$. We can also use relate to reason about the safety of certain optimisations. For example, Section 7.3 explains how to perform provably safe sum-type discriminations using relate.

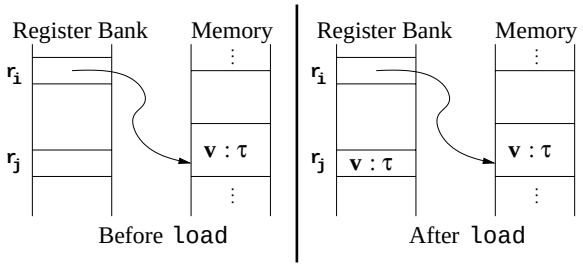

Figure 3: relate **Example**

2.5 Code Context ($\Gamma : \Omega$)

Program safety can be proved from type safety. Thus, we ensure that certain typing judgements hold before and after the execution of every instruction. Consider an instruction at location l which requires its arguments to be of certain types. For example, a load instruction might require its source register to be of a field type. In terms of Hoare logic, if the register bank R satisfies some precondition, it is safe to jump to location l. This precondition on R is represented as the type map ϕ . Thus we have l being of type $\mathsf{codeptr}(\phi)$. This fact itself can be encoded as the type map $\{l : \mathsf{codeptr}(\phi)\}$, though l is an address indexing into the code and not the register bank. This type map describes the local typing invariants at location l. Using intersection ($\cap$), we collect the local invariants at all program locations. This resultant map is Γ.

As a running example, consider the SPARC program in Figure 5 which adds up the elements of a list.

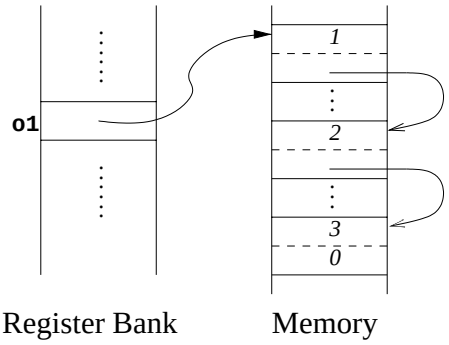

Figure 4: List Representation in Memory

In this program, register o1 is a list value, which is either 0 or a pointer to a two-word record containing an integer and a list value. A length-3 list containing $[1, 2, 3]$ is shown in Figure 4. Register o2 is a temporary to hold the data till it is added to the accumulator in register o3. SPARC uses delay slots after branch instructions. Instructions which go in delay slots are shown prefixed with a "*", and they are executed whether or not the branch is taken.

For this program to be safe, it must start with register o1 having the type of a pointer to a list, and register o7 should have the return address, i.e. it should have the type of a code pointer which expects register o3 to have an integer. Assuming definitions for high-level types like list_ty for lists (as shown at the end of Section 3.1), we have the precondition ϕ_0 for address 0 to be

$$\phi_0 = \{\mathtt{o1} : \mathsf{list\_ty}\} \cap \{\mathtt{o7} : \mathsf{codeptr}\{\mathtt{o3} : \mathsf{int}_{32}\}\}$$

If this condition holds, then it is safe to jump to location 0. The compiler provides invariants ϕ_l for each location l, which are combined into

$$\begin{array}{rl} \Gamma = & \{0 : \mathsf{codeptr}(\phi_0)\} \cap \\ & \{4 : \mathsf{codeptr}(\phi_4)\} \cap \\ & \vdots \\ & \{36 : \mathsf{codeptr}(\phi_{36})\} \end{array}$$

One of the goals of TML is to be able to work with multiple target machines. Unlike TAL or PCC, which assume the correctness of the assembler, we wish to have evidence that the code was assembled correctly. Hence, proofs of safety of programs must also include proofs that each word in the code part of the memory matches its semantic specification. For each machine, we need to know the instruction semantics, or how instructions in that machine change the state. Assuming that program location l has value v, we must prove that this value decodes to some instruction (described in Section 2.6) ι, of size s, which has the required semantics. Informally, we must prove that the contents of the "opcode" column in Figure 5 implement the contents of the "pseudo code" column. Δ encodes the contents of the part of the memory containing the program as intersections of singleton type maps $\{l : \mathsf{box}(\mathsf{int}_{=} v)\}$

Address		*Program*	*Pseudocode*	*opcode*
0		`tst %o1`	$\mathsf{R}[o1] == 0$?	`0x80900009`
4		`mov 0, %o2`	$\mathsf{R}[o2] \leftarrow 0$	`0x94102000`
8		`ba entry`	*goto* `entry`	`0x10800005`
12	$*$	`mov 0, %o3`	$\mathsf{R}[o3] \leftarrow 0$	`0x96102000`
16	`loop:`	`ld [%o1],%o2`	$\mathsf{R}[o2] \leftarrow \mathsf{M}[\mathsf{R}[o1]]$	`0xd4024000`
20		`ld [%o1+4],%o1`	$\mathsf{R}[o1] \leftarrow \mathsf{M}[\mathsf{R}[o1]+4]$	`0xd2026004`
24		`tst %o1`	*if* $\mathsf{R}[o1] <> 0$?	`0x80900009`
28	`entry:`	`bne loop`	*goto* `loop`	`0x12bffffd`
32	$*$	`add %o3, %o2, %o3`	$\mathsf{R}[o3] \leftarrow \mathsf{R}[o2]+\mathsf{R}[o3]$	`0x9602c00a`
36		`retl`	*return*	`0x81c3e008`
40	$*$	`nop`		`0x01000000`

Figure 5: SPARC program to add the contents of an integer list

for each address l in the program. For our example program,

$$\begin{aligned}\Delta \;=\; & \{0 : \mathsf{box}(\mathsf{int}_{=}80900009_{16})\} \cap \\ & \{4 : \mathsf{box}(\mathsf{int}_{=}94102000_{16})\} \cap \\ & \quad\vdots \\ & \{40 : \mathsf{box}(\mathsf{int}_{=}01000000_{16})\}\end{aligned}$$

If we shuffle $\Delta \cap \Gamma$ to interleave the contents of Γ and Δ to look like

$$\begin{aligned}& \{0 : \mathsf{codeptr}(\phi_0)\} \cap \\ & \{0 : \mathsf{box}(\mathsf{int}_{=}80900009_{16})\} \cap \\ & \{4 : \mathsf{codeptr}(\phi_4)\} \cap \\ & \{4 : \mathsf{box}(\mathsf{int}_{=}94102000_{16})\} \cap \\ & \vdots \\ & \{40 : \mathsf{box}(\mathsf{int}_{=}01000000_{16})\}\end{aligned}$$

the connection to traditional Hoare logic style presentation becomes obvious. The ϕ's are the program annotations, while the other singletons form the program instructions.

2.6 Instructions

An instruction in TML is a relation on ϕ_1, the type map which holds before its execution, and ϕ_2, the one that holds after its execution. The tuple (ϕ_1, ϕ_2) is the Hoare-logic style specification of an instruction with ϕ_1 being the precondition and ϕ_2 being the postcondition for that instruction.

Unlike Hoare logic, we have explicit branches and jumps. For branches we need to know the preconditions of all possible branch targets. This is provided by the code context Γ. We therefore have the instruction constructor instr depending on three type maps, (Γ, ϕ_1, ϕ_2).

For example, consider the `add` $(r_i \leftarrow r_j + r_k)$ instruction in code context Γ. Addition requires the source values to be integers. So we have the precondition $\phi_1 = (\phi \cap \{j : \mathsf{int}_{32}\} \cap \{k : \mathsf{int}_{32}\})$. After addition, all previous typing judgements about r_i are invalidated. Additionally, in ϕ_2, r_i gets the integer type, encoded as $\{r_i : \mathsf{int}_{32}\}$. Therefore, $\phi_2 = (\phi_1 \backslash i) \cap \{i : \mathsf{int}_{32}\}$.

We can actually have a stronger formulation for `add`.If we know that in ϕ_1, $\{j : \mathsf{int}_{=}n, k : \mathsf{int}_{=}m\}$, we have the postcondition $\phi_2 = (\phi_1 \backslash i) \cap \{i : \mathsf{int}_{=}(n+m)\}$ In TML, we use the instr instruction constructor to encode `add` for ϕ_1 and ϕ_2 under context Γ as $\texttt{add} = \mathsf{instr}(\Gamma, \phi_1, \phi_2)$. We see that `add` is polymorphic over all environments ϕ and contexts Γ. Therefore, in writing TML-instruction constructors we need quantifications over Ω, using operators $\forall_\iota$, $\exists_\iota$. We usually wish to prove that an instruction at some location satisfies some semantic specifications. Subtyping on instructions, "$\subset_\iota$" gives us a convenient way of expressing this. Therefore, we have

$$\begin{aligned}\texttt{add}(i,j,k) \quad & \subset_\iota \; \forall \Gamma, \phi, n, m. \\ & \mathsf{instr}(\Gamma, (\phi \cap \{j : \mathsf{int}_{=}n\} \cap \{k : \mathsf{int}_{=}m\}), \\ & \qquad ((\phi \cap \{j : \mathsf{int}_{=}n\} \cap \{k : \mathsf{int}_{=}m\}) \backslash i \\ & \qquad\quad \cap \{i : n+m\}))\end{aligned}$$

The form of instruction for addition given above is the most general form. However, `add` may be called in other contexts, most importantly for address arithmetic needed for subsequent loads and stores. This form of `add` is significantly different in its semantic specification and we must prove that the most general form is a subtype of the form below.

$$\begin{aligned}\texttt{add}(i,j,k) \quad & \subset_\iota \; \forall \Gamma, \phi, \tau, c. \\ & \mathsf{instr}(\Gamma, (\phi \cap \{j : \mathsf{field}\ c\ \tau\} \cap \{k : \mathsf{int}_{=}c\}), \\ & \qquad ((\phi \cap \{j : \mathsf{field}\ c\ \tau\} \cap \{k : \mathsf{int}_{=}c\}) \backslash i \\ & \qquad\quad \cap \{i : \mathsf{box}(\tau)\}))\end{aligned}$$

Unlike `add` which made no reference to the code context (Γ), given the instruction `jmp` r_1, we need to make sure that it is type-safe to jump to the branch target (given by $\mathsf{R}(1)$, the contents of r_1) with the current register environment, ϕ. Hence, ϕ should satisfy the precondition for code address $\mathsf{R}(1)$. Thus the TML instruction characterising a jump should have a code context $\Gamma \cap \{\mathsf{R}(1) : \mathsf{codeptr}(\phi_1)\}$. The instruction subtyping rule for a jump is written as

$$\begin{aligned}\texttt{jmp}(i) \quad & \subset_\iota \; \forall \Gamma, \phi, n. \\ & \mathsf{instr}(\Gamma \cap \{n : \mathsf{codeptr}(\phi)\}, \\ & \qquad \phi \cap \{i : \mathsf{int}_{=}n\}, \\ & \qquad \phi \cap \{i : \mathsf{int}_{=}n\})\end{aligned}$$

3. TML STATIC SEMANTICS

TML static semantics are given in terms of rules for the well-formedness of types, type maps and instructions (Figure 10), subtyping rules on types in all these kinds (Figure 12), and well-formedness of programs.

3.1 Well-Formedness of Types

Some types like $\forall$ and $\exists$ take type arguments and could be written as $\forall\,(\lambda x.\ F\ x)$. But type functions using λ would require higher kinds, complicating the semantic model. We avoid this complication by having quantifiers implicitly bind de Bruijn indices [6], represented as $\underline{n}$ instead of named variables in the type terms. The above term, for example, looks like $\forall\,(F\ \underline{0})$. We use explicit substitution [1] rules given in Figure 11 to manipulate the terms.

For reasoning about recursive types, we need to know which types are contractive (as in the ideal model [7] or the indexed model [5]). The type $\alpha = \mathsf{offset}(3,\alpha)$ is not meaningful because the operator offset is not contractive, but

$$list = \mathsf{int}_{=}0 \cup (\mathsf{int}_{\neq}0 \cap (\mathsf{field}\ 0\ \mathsf{int}_{32}) \cap (\mathsf{field}\ 4\ list))$$

is meaningful because field is contractive.

To define well-formedness, we use a context formed by W, a list mapping indices in the type term to their contractiveness, and – in effect – we reason about type-functions, not types. If the n^{th} element in W is **Y**, it means that $\underline{n}$ may be assumed to be contractive for determining the contractiveness of the term containing it. If it is **N**, the contractiveness of the term is not dependent on the contractiveness of $\underline{n}$. Whenever we introduce any new variable, it gets the de Bruijn index 0. We shift all other indices by 1. Therefore, we add the **Y** or **N** to the head of the context W. We use the notation $W^{\mathbf{Y}}$ and $W^{\mathbf{N}}$ to force all entries in W to **Y** and **N** respectively. Figure 10 gives the complete set of rules to determine the well-formedness of type terms. We list a few of these below.

$$\frac{W \vdash \tau : \Omega \quad W \vdash n : \Omega}{W \vdash \mathsf{offset}\ (n,\tau) : \Omega}\ \text{WF\_OFFSET}$$

$$\frac{\mathbf{N}, W \vdash \tau : \Omega}{W \vdash \mathsf{rec}\ \tau : \Omega}\ \text{WF\_REC}$$

$$\frac{\mathbf{Y}, W \vdash \tau : \Omega}{W \vdash \forall\, \tau : \Omega}\ \text{WF\_}\forall \qquad \frac{\mathbf{Y}, W \vdash \tau : \Omega}{W \vdash \exists\, \tau : \Omega}\ \text{WF\_}\exists$$

$$\frac{W[n] = \mathbf{Y}}{W \vdash \underline{n} : \Omega}\ \text{WF\_INDEX} \qquad \frac{}{W \vdash \mathsf{int}_{=}(i) : \Omega}\ \text{WF\_CONST}$$

For example, consider the term $\mathsf{rec}(\lambda\alpha(\mathsf{offset}\ 3\ \alpha))$, written as $\mathsf{rec}(\mathsf{offset}\ 3\ \underline{0})$ using de Bruijn indices. This term is not well-formed since offset is not a contractive constructor. The derivation tree for the well-formedness of the term under context W is

$$\dfrac{\dfrac{\dfrac{???}{\mathbf{N}, W \vdash \underline{0} : \Omega} \qquad \dfrac{}{\mathbf{N}, W \vdash 3 : \Omega}\ \text{WF\_CONST}}{\mathbf{N}, W \vdash \mathsf{offset}\ 3\ \underline{0} : \Omega}\ \text{WF\_OFFSET}}{W \vdash \mathsf{rec}(\mathsf{offset}\ 3\ \underline{0}) : \Omega}\ \text{WF\_REC}$$

After using the WF_OFFSET rule, we use the WF_CONST rule to show well-formedness of 3. Since $\{\mathbf{N}, W\}[0] = \mathbf{N}$, we cannot use the WF_INDEX rule to show the well-formedness of the $\underline{0}$ subterm, and hence the derivation fails.

The integer-list data type of Figure 2a is,

$$\mathsf{list\_ty} = \mathsf{rec}(\mathsf{int}_{=}0 \cup (\mathsf{int}_{\neq}0 \cap (\mathsf{field}\ 0\ \mathsf{int}_{32}) \cap (\mathsf{field}\ 4\ \underline{0})))$$

and this can be proved well-formed with a syntax-directed derivation.

3.2 Subtyping

In our proofs for derivations of program safety, we are often required to prove that any value of type ϕ is also a value of type ϕ'. Subtyping provides a convenient way of expressing this, and we express the above condition as $\phi \subset \phi'$. For instructions, we often wish to say that an instruction ι implements another instruction ι' (like add implementing pointer arithmetic in Section 2.6), express this syntactically as $\iota \subset_\iota \iota'$. A few illustrative rules are given below. A larger set of rules is in Figure 12.

$$\frac{}{\tau \subset \top}\ \subset\text{\_}\top$$

$$\frac{}{\mathsf{int}_{=}i \subset \mathsf{int}_{\geq}(i)}\ \subset\text{\_INT\_}\pi \qquad \frac{}{\mathsf{int}_{=}i \subset \mathsf{int}_{\leq}(i)}\ \subset\text{\_INT\_}\pi$$

$$\frac{\tau_1 \subset \tau_2}{\mathsf{box}\ \tau_1 \subset \mathsf{box}\ \tau_2}\ \subset\text{\_BOXED}$$

$$\frac{\tau \subset \tau_3 \quad \tau \subset \tau_4}{\tau \subset \tau_3 \cap \tau_4}\ \subset\text{\_}\cap\text{R}$$

$$\frac{\tau_1 \subset \tau_2}{\{i : \tau_1\} \subset \{i : \tau_2\}}\ \subset\text{\_SINGLE}$$

$$\frac{}{\tau \subset \tau \backslash i}\ \subset\text{\_}\backslash$$

$$\frac{\tau_1' \subset \tau_1 \quad \tau_2' \subset \tau_2 \quad \tau_3 \subset \tau_3'}{\mathsf{instr}(\tau_1,\tau_2,\tau_3) \subset_\iota \mathsf{instr}(\tau_1',\tau_2',\tau_3')}\ \subset\text{\_INSTR}$$

4. REAL MACHINE INSTRUCTIONS

We want to run real programs on real machines; we will take a usable subset of the SPARC architecture as our prototype example. We omit floating-point instructions, register windows, and privileged instructions; this means that we can still type-check (prove safe) integer-only programs generated by compilers that don't shift register windows, of which the FLINT compiler [13] is an example. But for this paper we illustrate only a subset of our subset.

For each machine, we must augment our type system with types and typing rules that are necessary to describe that machine. For SPARC, we need to add kinds and types necessitated by the architecture details.

4.1 SPARC Instruction Syntax

For the subset of SPARC instructions we consider, there are three classes of instructions, the ALU instructions, branch instructions, and memory access instructions. In TML, we model these instructions as subtypes of instructions formed by the instr construct in TML, as shown in Figure 15. Wellformedness rules for SPARC instructions are given in Figure 14.

SPARC arithmetic instructions take the form $s_1 \oplus s_2 \rightarrow d$, where s_1 must be one of the 32 registers, s_2 may be a register or an immediate sign-extended 13-bit constant, and d must be a register. To describe the second operand we have a kind *RegImm*, which allows us to discriminate between register-mode RMode type arguments and immediate-mode IMode type instructions.

Figure 13 shows the kinds and constructors for operators. A general ALU instruction is specified by $\mathsf{alu}(x,c,oper)$, where *oper* (of kind *Oper*) is the operation to be performed, x (0 or 1) specifies whether carry-in is to be performed as part of the operation, and c (0 or 1) specifies whether the condition codes are to be modified by the operation. Thus, for example, the standard add instruction is $\mathsf{alu}(0,1,\mathsf{add\_oper}) : \mathit{INSTR}$.

Condition codes on architectures such as SPARC and Pentium, are modelled as if they live somewhere in the register bank, perhaps in one or more processor-status registers. The fact that the condition codes in some state (R,M) match the condition "not equal" is written $\mathsf{M} \vdash \mathsf{R} : \mathsf{cc\_ne}$, and we see that cc_ne occupies the position of a type map. The SPARC has sixteen predicates on the condition codes, of which "not equal" is one; thus, we have sixteen primitive type maps, of which cc_ne is one. We also have a type map op-

erator $\backslash^{cc}$, which is analogous to our type map operator $\backslash$; that is, $\phi\backslash^{cc}$ is a type map similar to ϕ but with any information about the condition-code settings removed, e.g., $\phi \subset \phi\backslash^{cc}$.

We define the complement operator on condition-code type maps as follows:

$$\begin{array}{lll} \overline{\mathsf{cc\_a}} = \mathsf{cc\_n} & \overline{\mathsf{cc\_n}} = \mathsf{cc\_a} & (\text{always}, \text{never}) \\ \overline{\mathsf{cc\_g}} = \mathsf{cc\_le} & \overline{\mathsf{cc\_le}} = \mathsf{cc\_g} & (>, \leq) \\ \vdots \qquad \vdots & & \end{array}$$

Some arithmetic and logical operations on the SPARC yield condition-code settings, and these settings depend not only on the result but in some cases on how the result is computed (i.e. on the arguments). We model this with an operator $\mathsf{setcc}(oper, n_1, n_2)$ which computes the appropriate condition-code type maps. For example,

$\mathsf{setcc}(\mathsf{add\_oper}, 3, -7) =$
$\quad \mathsf{cc\_a} \cap \mathsf{cc\_ne} \cap \mathsf{cc\_le} \cap \mathsf{cc\_l} \cap$
$\quad \mathsf{cc\_gu} \cap \mathsf{cc\_cc} \cap \mathsf{cc\_ne} \cap \mathsf{cc\_vc}$

A conditional branch instruction is specified by $\mathsf{ibranch}(cond, a, i)$ where *cond* is a condition-code type map, *a* specifies whether the delay-slot instruction should be annulled if the branch is not taken, and *i* is the displacement, an integer to be added to the program counter if the comparison yields true.

For SPARC load instructions of the form $d_1 \leftarrow \mathsf{M}[s_1 + c]$ or store instructions of the form $\mathsf{M}[s_1 + c] \leftarrow s_2$, the *c* argument is formed with the IMode constructor. The store instruction has involved semantics which are tied to our allocation model. We describe this model and the store instruction in detail in Section 7.1.

5. SEMANTIC MODEL OF TYPES

In this section, we briefly describe the semantic model we use for types. Our model is built upon the model described by Appel and McAllester [5].

In this model, a *ty-pred* is a predicate on tuples (k, v). Informally, a value *v* belongs to a type τ to index *k*, $(v :_k \tau)$, it is safe to run a program expecting a value of type τ on argument *v* for *k* instructions. Since τ is a predicate on values, $\tau\ k\ v$ means that (k, v) belongs to the type τ.

A value *v* is a tuple (s, x). The first component, *s*, describes the state of the machine. *s* itself is complicated and contains other relations like $\mathsf{readable}$, $\mathsf{writable}$, $\mathsf{allocated}$, which describe the sets of readable, writable and allocated parts of the machine memory respectively. Since types involving memory references would require to know about the currently readable parts of memory, for example, we need to have the state as a component of the value.

The Appel-McAllester model is too weak to specify dependent types (e.g., relate). In our new model, the second component *x* of a value (s, x) is a vector of integers (i.e., a finite function on integers). For example, Figure 2a illustrates a value (s, r) where the memory component of *s* contains $\{300 \mapsto 10, 304 \mapsto 130\}$ and the "root vector" *r* contains $\{1 \mapsto 0, 2 \mapsto 300\}$.

Ty-preds are written as predicates on the tuple (k, v). To continue our example, the value $(s, r(2))$ might or might not be a list, because in this illustration we don't know the contents of memory location 130; but to approximation 1 it's a list, meaning that if we execute for at most 1 instruction from $(s, r(2))$ the program can't notice that it fails to be a list. We write this as $\mathsf{list\_ty}(1, (s, r(2)))$.

The type expressions of TML are not simply modelled by ty-preds, because we must also represent open de Bruijn indices: an an open term is (implicitly) parameterized by a substitution providing values for all the unbound variables. So the type map of TML is

$$(N \rightarrow \textit{ty-pred}) \rightarrow \textit{ty-pred}.$$

For example, a singleton type map constructor (call it $\{1 : \tau\}$) is defined in the following way:

$$\begin{array}{ll} \{1:\tau\} = & \lambda\sigma. \\ & \lambda(k, (s_v, x_v)). \\ & \tau(\sigma)\ k\ (s_v, \lambda i. x_v(1)) \end{array}$$

Since τ may contain de Bruijn indices, we provide an environment σ to interpret them. In this type map, we wish the first vector entry to satisfy the type τ. Hence, we extract the first component of *v*, before again putting it into a tuple with the state. Then τ is applied to the new tuple with index *k*.

As mentioned before, type scalars are also modelled as type maps. For example, the definition of the character type "char" is given below.

$$\begin{array}{ll} \mathsf{char} = & \lambda\sigma.\lambda(k, (s_v, x_v)). \\ & 0 \leq (\lambda i. x_v\ 0) < 256 \end{array}$$

We apply 0 to x_v to convert it into a scalar (call it c_v). The set defining "char" should only allow values between 0 and 256. Therefore, we check that c_v satisfies this condition.

A number $n : \Omega$ can be represented by the type $\mathsf{int}_=(n) : \Omega$.

In this model, we think of programs both as sequences of opcodes Δ and as collections of code pointers Γ. Assuming a program starts at location 100 with an environment ϕ describing the types of values in registers, we must show that location 100 has type $\mathsf{codeptr}(\phi)$ for any index *k*. We prove this by induction on *k*.

The base case is easy enough; every value has type τ for any τ to approximation 0. For the induction step, we prove that

$$\forall k, v.\ (\Delta \cap \Gamma)(k, v) \Rightarrow \Gamma(k+1, v).$$

That is, if the program satisfies predicate $\Delta \cap \Gamma$ to approximation *k*, then it also satisfies Γ to degree $k+1$. Appel and McAllester [5] explain in detail what this means.

To avoid mentioning indices *k* in the TML system, we abstract $\forall k, v.\ \phi(k, v) \Rightarrow \phi'(k+1, v)$ as $\phi \Subset \phi'$. Thus, the previous formula is written as $\Delta \cap \Gamma \Subset \Gamma$, and we give syntactic rules for introducing and eliminating $\Subset$ in Figure 6:

$$\frac{\tau \Subset \tau' \quad \tau \Subset \tau''}{\tau \Subset (\tau' \cap \tau'')}\ \Subset\_\cap \qquad \frac{\tau \Subset \tau'}{\tau \subset \tau'}\ \Subset\_\subset \qquad \frac{}{\Gamma \Subset \top}\ \Subset\_\top$$

$$\frac{\begin{array}{c}\tau \subset \{l : \mathsf{at}(\mathsf{instr}(\tau', \tau_1, \tau''), s)\} \\ \tau \subset (\{l+s : \mathsf{codeptr}(\tau'')\})\end{array}}{\tau \Subset (\{l : \mathsf{codeptr}(\tau_1)\})}\ \Subset\_\mathrm{INSTR}$$

$$\frac{}{\Gamma \Subset (\{l : \mathsf{codeptr}(\bot)\})}\ \Subset\_\mathrm{CPTR}$$

$$\frac{\tau \cap \tau' \Subset \tau'}{\tau \subset \tau'}\ \Subset\_\mathrm{INDUC}$$

Figure 6: $\Subset$-Rules

6. PROVING PROGRAM SAFETY

Let *A* be the allocated set which includes the part of the memory that has the program code and any allocated data. *A* is not primitive; it can be expressed as a predicate on the memory M. Registers r_a and r_l will point to the boundaries of the allocation area, that is,

$\mathsf{avail}(\mathsf{R}) = \{\mathsf{R}(a), \mathsf{R}(a)+1, \ldots, \mathsf{R}(l)-1\}$ are locations available for future allocation. To check the safety of the program under memory M and register bank R, we have the following rule :

$$\frac{\begin{array}{ll} A; \mathsf{M} \vdash \mathsf{R} : \phi & (1) \\ A \subset \mathsf{readable} & (2) \\ \mathsf{avail}(\mathsf{R}) = \subset (\mathsf{readable} \cap \mathsf{writable}) & (3) \\ A \cap \mathsf{avail}(\mathsf{R}) = \emptyset & (4) \\ A; \mathsf{M} \vdash \mathsf{R}(pc) : \mathsf{codeptr}(\phi) & (5) \end{array}}{\mathsf{safe}(\mathsf{R}, \mathsf{M})}$$

This program could be just a function which is called by another piece of code with some parameters. Hence, the initial program location is a continuation with formal parameters ϕ. Therefore, the first condition says that we start out with the environment ϕ. We do not wish to have any assumptions about the type system of the calling program. We might like the called function to use any complex type system which the caller might not know about. Hence, we expect ϕ to be very primitive in nature and easily provable. This condition ensures that we start out in the correct state.

Premises (2) says that code and allocated data should be readable, (3) says that free space should also be writable and (4) ensures that allocated set is disjoint from the free space.

For the example program in Figure 5, let the $\mathsf{readable}$ set be $\{0 \ldots 1000\}$, the $\mathsf{writable}$ set be $\{200 \ldots 1000\}$, and the allocated set be $\{0 \ldots 300\}$. Proving (2),(3), and (4) is fairly easy.

The last condition says that we start at a location which is a valid code pointer of type ϕ. It is this condition which captures the main proof of safety of the program.

$$\frac{\mathsf{loaded}(\Delta, \mathsf{M}) \quad \Delta \subset \Gamma \quad \Gamma \subset \mathsf{R}(i) : \tau}{A; \mathsf{M} \vdash \mathsf{R}(i) : \tau}$$

$$\frac{\Delta = \bigcap_{i \in \mathtt{dom}\ p} \{i : \mathsf{box}(\mathsf{int}_{=} p(i))\} \quad \forall i \in \mathtt{dom}\ p. \mathsf{M}(i) = p(i)}{\mathsf{loaded}(\Delta, \mathsf{M})}$$

Figure 7: Program-loading rules

Figure 7 lists the inference rules on values. The first rule shows that proving (5) involves proving three facts:

- We first show that the program loaded in memory is described by Δ
- The second condition involves decoding Δ to get TML instructions. Then we must show that these instructions respect the invariants mentioned in the code context Γ. The proof of this fact requires us to traverse the program and check that type safety is preserved at each point in the execution of the program. This proof captures progress by showing that there is always another instruction (the next instruction for non-branches, and the branch target instruction otherwise) that may be safely executed.
- Finally, for the program counter (r_{pc}), we must show that the Γ from the second condition satisfies $\{r_{pc} : \mathsf{codeptr}(\phi)\}$. This can be proved using the subtyping rules shown in Figure 12. For example, if

$$\Gamma = \{0 : \mathsf{codeptr}(\phi_0)\} \cap \{4 : \mathsf{codeptr}(\phi_4)\} \cap \ldots \cap \{40 : \mathsf{codeptr}(\phi_{40})\}$$

then $\Gamma \subset \{0 : \mathsf{codeptr}(\phi_0)\}$ is trivial.

Proving $\Delta \subset \Gamma$ is nontrivial and we give a schematic description below. Figure 8 gives part of the proof tree for the second condition. The two most interesting stages in the proof are labelled S1 and S2. At stage S1, we need to prove facts like $\Delta \cap \Gamma \Subset \{l : \mathsf{codeptr}(\phi_l)\}$ for every location l in the program. This means that can always take another well-typed execution step at any point in the program. These are the typing judgements which implement the induction proof (mentioned in the earlier section) in our underlying model.

In stage S2 proving these facts for each location l is reduced to

1. proving that some instruction is present at location l whose precondition is ϕ_l specified by the environment at l and post-condition is some ϕ', and
2. proving that the next instruction is a codepointer which expects the environment to satisfy ϕ'.

The connection to traditional Hoare-logic proofs is again evident. Below is the Hoare-logic style proof for our list sum example.

We start out with the Δ and Γ as described in 2.5.

To prove the first fact, we begin with decoding the contents of Δ. Each program code location, l, contains some number n. The first step involves proving that n decodes into a TML instruction $\iota = \mathsf{instr}(\Gamma, \phi, \phi')$. We prove this using instruction decoding techniques described by Michael and Appel [8, 3]. After this, we prove that $\phi_l \subset \phi$. This allows us to execute ι under the environment ϕ_l safely.

Proving the second part involves showing that the resultant environment ϕ' is compatible with the given environment at next location, $l+4$ (for non branch instruction). This is done by showing that $\phi' \subset \phi_{l+4}$.

We list the environment supplied at each point in the program along with the decoded instructions.

$\phi_0 = \{\mathtt{o1} : \mathsf{list}\}$
0 SPARC_TST
$\phi_4 = \{\mathtt{o1} : \mathsf{list}\}$
4 SPARC_MOV
$\phi_8 = \phi_4 \cap \{\mathtt{o2} : \mathsf{int}_{32}\}$
8 SPARC_BRANCH
$\phi_{12} = \phi_8 \cap \{\mathtt{o3} : \mathsf{int}_{32}\}$
12 SPARC_MOV
$\phi_{16} = \phi_{12} \cap \{\mathtt{o1} : \mathsf{int}_{\neq} 0\}$
16 SPARC_LOAD
$\phi_{20} = \phi_{12} \cap \{\mathtt{o1} : \mathsf{int}_{\neq} 0\} \cap \{\mathtt{o2} : \mathsf{int}_{32}\}$
20 SPARC_LOAD
$\phi_{24} = \phi_{12} \cap \{\mathtt{o2} : \mathsf{int}_{32}\}$
24 SPARC_TST
$\phi_{28} = \phi_{24}$
28 SPARC_BRANCH
$\phi_{32} = \phi_{28}$
32 SPARC_ALU
$\phi_{36} = \{\mathtt{o1} : \mathsf{list}\} \cap \{\mathtt{o2} : \mathsf{int}_{32}\} \cap \{\mathtt{o3} : \mathsf{int}_{32}\}$

We can always execute the instruction at location 0, since the precondition for SPARC_TST has no constraints. From subtyping rule $\subset$REFL, we can prove that $\phi_0 \subset \phi$. The postcondition

$$\phi' = \phi_0 \cap ((\mathtt{cc\_ne} \cap \{\mathtt{o1} : \mathsf{int}_{\neq} 0\}) \cup (\mathtt{cc\_e} \cap \{\mathtt{o1} : \mathsf{int}_{=} 0\}))$$

ϕ' is stronger than ϕ_4, and step (2) in stage S2 can be easily proven by subtyping rules. This postcondition helps us relate the value of $\mathtt{o1}$ to the condition code when we take a branch depending on it. We can always execute the two $\mathtt{mov}$ instructions (at location 4 and at the delay slot at location 12) as there are no source register constraints. After the branch instruction at location 8, we have postcondition

$$\dfrac{\dfrac{\dfrac{\vdots}{\Delta\cap\Gamma\subset\{0:\mathsf{at}(\mathsf{instr}(\Gamma,\phi_0,\phi_4),4)\}\quad \Delta\cap\Gamma\subset\{4:\mathsf{codeptr}(\phi_4)\}}\quad \dfrac{\vdots}{\Delta\cap\Gamma\subset\{4:\mathsf{at}(\mathsf{instr}(\Gamma,\phi_4,\phi_8),4)\}\quad \Delta\cap\Gamma\subset\{8:\mathsf{codeptr}(\phi_8)\}}\quad \dfrac{\vdots}{\Delta\cap\Gamma\subset\{36:\mathsf{at}(\mathsf{instr}(\Gamma,\phi_{36},\phi_{40}),4)\}\quad \Delta\cap\Gamma\subset\{40:\mathsf{codeptr}(\phi_{40})\}}\ \text{S2}}{\Delta\cap\Gamma\Subset\{0:\mathsf{codeptr}(\phi_0)\}\quad \Delta\cap\Gamma\Subset\{4:\mathsf{codeptr}(\phi_4)\}\ldots\Delta\cap\Gamma\Subset\{40:\mathsf{codeptr}(\phi_{40})\}}\ \text{S1}}{\dfrac{\Delta\cap\Gamma\Subset\Gamma}{\Delta\subset\Gamma}\ \Subset\text{\_INDUC}}\ \Subset\text{\_}\cap$$

Figure 8: Part of Syntactic Proof Tree for Program Safety

$$\phi_{ba}=\phi_4\cap\{\mathtt{o2}:\mathsf{int}_{=}0\}\cap\{\mathtt{o3}:\mathsf{int}_{=}0\}$$

This environment can be shown to be stronger than ϕ_{28}, the target of the unconditional branch, i.e. $\phi_{ba}\subset\phi_{28}$. At this point, depending on the condition codes, we either go to location 12 or location 36 after executing the add instruction at location 32. Since types of `o2` and `o3` are int_{32}, we can show that this SPARC_ALU addition instruction in the delay slot can be executed.

The postcondition of the conditional branch, if taken, is

$$\phi_y=\phi_0\cap((\mathtt{cc\_ne}\cap\{\mathtt{o1}:\mathit{int}_{\neq}0\})\cap\{\mathtt{o2}:\mathsf{int}_{32}\}\cap\{\mathtt{o3}:\mathsf{int}_{32}\}$$

This environment satisfies the condition of the pointer in `o1` being non-null. We can prove that it is stronger than the ϕ_{16}, the target of the conditional branch using rules in Figure 12.

Since we are using the untagged representation of a list, and we have $\{\mathtt{o1}:\mathsf{list\_ty}\}$ and $\{\mathtt{o1}:\mathsf{int}_{\neq}0\}$. We can infer $\{\mathtt{o1}:\mathsf{field}\ 0\ \mathsf{int}_{32}\}\cap\{\mathtt{o1}:\mathsf{field}\ 4\ \mathsf{list\_ty}\}$.

This would satisfy the precondition for the `ld` instruction at locations at 20 and 24, since the types of `o2` and `o3` are the same, but the type of `o1` is stronger than just list. The next load can be similarly shown to be safe.

If the branch at location 28 is not taken, we have the postcondition

$$\phi_n=\phi_0\cap((\mathtt{cc\_e}\cap\{\mathtt{o1}:\mathit{int}_{=}0\})\cap\{\mathtt{o2}:\mathsf{int}_{=}0\}\cap\{\mathtt{o3}:\mathsf{int}_{32}\}$$

This is stronger than the requirement that we return with $\{\mathtt{o3}:\mathsf{int}_{32}\}$. Thus the entire function can be shown to be safe.

7. NO ATOMIC OPERATIONS

In TML, we do not use opaque high-level instructions for allocation or array accesses which expand into multiple instructions on real machines. Our type system allows us to argue about intermediate machine states within these instructions. Below we explain how to perform safe memory allocation, array bounds check elimination, and sum type discriminations in TML.

7.1 Memory Allocation

We assume that memory is allocated from a contiguous region in order. i.e. every location is allocated after all preceding locations have been allocated. As shown in figure 9, the region begins at "Start" and ends at "Limit", where all locations till "Boundary" are allocated.

Most typed assembly languages treat memory allocation as an atomic operation. This increases the trusted computing base, making us trust the safety of the allocation subroutine. This also disallows the compiler making certain optimisations or rearrangement of instructions in the code that involves memory allocation. In TML, we use bookkeeping registers to get rid of the atomicity of allocation. The contiguous allocation space starts at address pointed

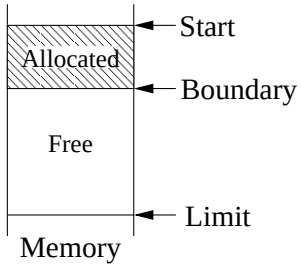

Figure 9: Memory Allocation

to by the register r_{ap}. r_b points to the beginning of the unallocated set of locations.i.e., all locations from r_{ap} to r_b-1 are allocated. r_l points to the end of the memory allocation region. All of these are virtual registers, and may not be used as register arguments to any instructions.

We wish to allocate a tuple of two integers. Let these integers be stored in r_1 and r_2 and let r_3 point to the resultant tuple. Before allocation, we must start out with a guarantee that some space is avilable. Assume r_{ap} pointing to location 100, r_b pointing to location 120, and r_l pointing to location 200. The atomic malloc program would be split into the following steps:

- Check for the availability of space : The condition $\mathsf{R}(b)+4\leq\mathsf{R}(l)$ checks that we have space for two integers.
- Store contents of r_1 at address $\mathsf{R}(b)$.
- Update $\mathsf{R}(b)$ to $\mathsf{R}(b+4)$.

For SPARC, the store instruction `sto i, j, c` allows us to split malloc . It uses the first type map to encodes the intial checks by having $\mathsf{relate}_{\leq}(\mathsf{offset}\ 4,\mathsf{id})(b,l)$ This ensures space for one integer taking up one machine word. By having $\mathsf{relate}_{=}((\mathsf{offset}\ c),\mathsf{id})(i,b)$, we know the address for storing is exactly the same as the beginning of the unallocated space. The postcondition correctly assigns the type to i. It also updates the r_b to point to the next unallocated location, and maintains the "$\leq$" relation between the r_b and the r_l. Thus, it guarantees that the store is safe. The second store is similarly guaranteed safe. At the end of the two stores, we have $\{i:(\mathsf{field}\ 4\ \mathsf{int}_{32})\}$ from the first `sto` and $\{i:(\mathsf{field}\ 8\ \mathsf{int}_{32})\}$ from the second. Taking the intersection of these two types, we have r_i having exactly the type for a tuple of two integers.

Our allocation method is not completely general. For example, we still have the restriction of making allocations in a linear order without holes.

7.2 Array bounds check elimination

We can define n-length arrays in TML as

$$\mathsf{array}(n,\tau)=\forall i.\mathsf{field}(((4*i)\cap\mathsf{int}_{\geq}0\cap\mathsf{int}_{<}n),\tau)$$

The allocation for arrays is similar to the example above. We assume a register r_i which has type τ. We make up an array of type τ and size n by initialising n locations using the scheme above to the value in r_i within a loop. Due to lack of space, we cannot go through a complete example.

We have a rich set of constuctors over integer values and singleton types in TML which allow us to perform safe array bounds check eliminations.

Loc	*Program*	*Pseudo code*
100	loop:subcc %o2, %o4, %g0	R[*o*2] == R[*o*4] ?
104	be done; nop	*goto* done
112	add %o1, %o4, %o5	R[*o*5] ← R[*o*1] + R[*o*4]
116	ld [%o5], %o6	R[*o*2] ← M[R[*o*5]]
120	add %o6, %o3, %o3	R[*o*3] ← R[*o*3] + R[*o*6]
124	add %o4, 4, %o4	R[*o*4] ← R[*o*4] + 4
128	ba foo; nop	*goto* loop
136	done:	

Consider the program above which adds the elements of an integer array. Register o1 contains the pointer to the array, o2 contains length of the array. o3 is the accumulator for the sum of elements. o4 contains an integer used to index into the array. We start with the environment

$$\phi_{100} = \{\texttt{o1} : \mathsf{array}\ n\ \mathsf{int}_{32} \cap \mathsf{int}_{=} 4l\} \cap \{\texttt{o2} : \mathsf{int}_{=} 4n\} \cap \{\texttt{o4} : \mathsf{int}_{>} 0 \cap \mathsf{int}_{\leq} 4n \cap \mathsf{int}_{=} 4m\}$$

From the types in ϕ_{100}, the starting point for the program, we can infer the fact that $\mathsf{R}(\texttt{o4}) \leq \mathsf{R}(\texttt{o2})$ and that $\mathsf{R}(\texttt{o4})$ and $\mathsf{R}(\texttt{o1})$ are divisible by 4. After the subcc instruction at 100, using SPARC_ALU_2 (which considers condition codes), instantiated with op = sub, i = o2, j = o4 and k = g0, we have the environment ϕ_{104} corresponding to the postcondition having $\mathsf{set\_cc}(\texttt{sub}, \mathsf{R}(\texttt{o2}), \mathsf{R}(\texttt{o4}))$. The branch instruction SP_BRANCH requires the postcondition that the "equal" flag is not set for the fall-through instruction. Therefore, we can determine the precondition of instruction at 112 to include $\mathsf{R}(\texttt{o2}) \neq \mathsf{R}(\texttt{o4})$. These two facts allow us to further infer that $\mathsf{R}(\texttt{o4}) < 4n$. The add instruction at address 112 computes the address of the array element to be accessed. This access is safe only if the address is between the array base $\mathsf{R}(\texttt{o1})$ and the array limit ($= \mathsf{R}(\texttt{o1}) + 4n$), and if it is on a word boundary. Since $0 < \mathsf{R}(\texttt{o4}) < 4n$, the effect of the add instruction is that $\mathsf{R}(\texttt{o1}) < \mathsf{R}(\texttt{o5}) < \mathsf{R}(\texttt{o1}) + 4n$. The sum is also divisible by 4. We expect a medium-sized set of arithmetic lemmas to be sufficient for deriving most of the inferences required for such reasoning. All these conditions can be shown to be implied by the postcondition we get for the add instruction. Therefore, the next ld instruction at location 116 can be shown to load from an address on a word boundary which is within the array limits. Hence the access is safe. After the add instruction at location 128, we can easily prove that $\mathsf{R}(\texttt{o4})$ remains to be divisible by 4.

7.3 Sum type discrimination

Consider the following ML datatype.

```
datatype foo = A of int * int
             | B of t * t * int
```

This has a machine level representation where the A and B cases are differentiated on the basis of a tag. A zero tag implies case A and the next two words contain integers. A nonzero tag implies B tag where it is safe to access the third word after the tag.

	Code	Pseudocode
100	ld [%o1], %o2	R[*o*2] ← M(R(*o*1))
104	tst %o2	if *o*2 ≠ 0
108	bne b_case;nop	B tag case
116	ba a_case; nop	else A tag case
124 b_case:	ld [%o1+12], %o3	R[*o*3] ← M(R(*o*1) + 12)
⋮		
140 a_case:	ld [%o1+4], %o3	R[*o*3] ← M(R(*o*1) + 4)
⋮		

Assume that register o1 points to a foo cell. Instruction 100 loads the tag into register o2. If o2 is not zero, (B case) then we branch to the b_case label. Else, (A case) we go to the a_case label. In instruction 124, we access the third field. This access may be ensured to be safe in the following way. The load instruction relates o2 and o1 such that $\mathsf{M}[\texttt{o1}] = \texttt{o2}$ (Fact 1). On jumping to b_case after the test, we have another fact $\texttt{o2} \neq 0$ (Fact 2) due to the TML instruction SPARC_TST. At location 124, both facts 1 and 2 continue to be true. From these two facts, we use simple arithmetic lemmas to conclude that $\mathsf{M}[\texttt{o1}] \neq 0$. This combined with the fact $\{\texttt{o1} : \mathsf{box}(\texttt{foo})\}$ allows us to conclude that o1 has the B variant of foo. Therefore, the access in location 124 can be proven safe.

8. RELATED WORK

The PCC system described by Necula [11] lays the foundation for this research. However, it has a very large trusted computing base in terms of the type inference rules and the "VCgen" or the verification condition generator. By giving semantic model to types and machine semantics, we have a much smaller trusted computing base. Also, their implementations are specially geared towards 'C' or Java.

TAL [9] uses opaque high level operations to handle allocations and array accesses. TAL is also specialised for compilation to x86 architectures. DTAL seems to be more expressive than TAL, though it also handles allocation in an opaque way. DTAL also requires an extension to the system to handle sum type discrimination. However, due to the presence of dependent types, DTAL makes array accesses transparent and allows us to perform safe array-bounds check eliminations.

Finally, PCC and TAL have had no semantic models, only syntactic metatheorems.

9. FUTURE WORK

We are using Twelf [12] to encode TML. Though most of our model for types has been encoded, we still have to encode the model for instructions and many lemmas which are required to complete the proofs of program safety. Our allocation model is also not fully general and we hope to extend it to handle regions [14]. This could allow us to use provably safe garbage collection schemes as shown by Wang and Appel [15].

We are also using TML to construct a semantic model for a TAL-like calculus, completing the end-to-end path from ML source code to a foundational safety proof.

10. ACKNOWLEDGEMENTS

We would like to thank Adriana Compagnoni, Amy Felty, Zhong Shao, Roberto Virga, David Walker and Dan Wang for many helpful comments and suggestions. We would also like to thank Roberto Virga for helping us design the wellformedness judgements.

11. REFERENCES

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Levy. Explicit substitutions. In *Seventeenth Annual ACM Symp. on Principles of Prog. Languages*, pages 31–46. ACM Press, Jan 1990.

[2] Amal Ahmed, Andrew W. Appel, and Roberto Virga. Semantics of general references by a hierarchy of Gödel numberings. July 2001.

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

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

[5] Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. Technical Report TR-629-00, Princeton University, October 2000.

[6] N. G. deBruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indag. Math.*, 34:381–92, 1972.

[7] David MacQueen, Gordon Plotkin, and Ravi Sethi. An ideal model for recursive polymophic types. *Information and Computation*, 71(1/2):95–130, 1986.

[8] Neophytos G. Michael and Andrew W. Appel. Machine instruction syntax and semantics in higher-order logic. In *17th International Conference on Automated Deduction*, June 2000.

[9] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. In *POPL '98: 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 85–97. ACM Press, January 1998.

[10] George Necula. Proof-carrying code. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, New York, January 1997. ACM Press.

[11] George Ciprian Necula. *Compiling with Proofs*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, September 1998.

[12] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In *The 16th International Conference on Automated Deduction*. Springer-Verlag, July 1999.

[13] Zhong Shao. An overview of the FLINT/ML compiler. In *Proc. 1997 ACM SIGPLAN Workshop on Types in Compilation*, June 1997.

[14] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Twenty-first ACM Symposium on Principles of Programming Languages*, pages 188–201. ACM Press, January 1994.

[15] Daniel C. Wang and Andrew W. Appel. Type-preserving garbage collectors. In *POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 166–178. ACM Press, January 2001.

[16] Hongwei Xi and Robert Harper. A dependently typed assembly language. Technical Report OGI-CSE-99-008, Computer Science and Engineering Department, Oregon Graduate Institute, July 1999.

APPENDIX

$$\frac{}{W \vdash \top : \Omega}\ \text{WF\_}\top \qquad \frac{}{W \vdash \bot : \Omega}\ \text{WF\_}\bot \qquad \frac{}{W \vdash \mathsf{c}\ (i) : \Omega}\ \text{WF\_CONST} \qquad \frac{W \vdash i : \Omega}{W \vdash \mathsf{int}_\pi i : \Omega}\ \text{WF\_INT\_}\pi$$

$$\frac{W \vdash n_1 : \Omega \quad W \vdash n_2 : \Omega}{W \vdash n_1 + n_2 : \Omega}\ \text{WF\_+} \qquad \frac{W \vdash \tau : \Omega}{W \vdash \mathsf{id}\ \tau : \Omega}\ \text{WF\_ID} \qquad \frac{W \vdash \tau : \Omega}{W^{\mathbf{N}} \vdash \mathsf{codeptr}\ \tau : \Omega}\ \text{WF\_CPTR}$$

$$\frac{W \vdash \tau : \Omega \quad W \vdash n : \Omega}{W \vdash \mathsf{offset}\ (n, \tau) : \Omega}\ \text{WF\_OFFSET} \qquad \frac{W \vdash \tau : \Omega}{W^{\mathbf{N}} \vdash \mathsf{box}\ \tau : \Omega}\ \text{WF\_BOXED} \qquad \frac{\mathbf{N}, W \vdash \tau : \Omega}{W \vdash \mathsf{rec}\ \tau : \Omega}\ \text{WF\_REC}$$

$$\frac{W_1, W_2^{\mathbf{Y}} \vdash \tau_1 : \Omega \quad W_1^{\mathbf{Y}}, W_2 \vdash \tau_2 : \Omega}{(W_1, W_2) \vdash \tau_1 \cap \tau_2 : \Omega}\ \text{WF\_}\cap \qquad \frac{W \vdash \tau_1 : \Omega \quad W \vdash \tau_2 : \Omega}{W \vdash \tau_1 \cup \tau_2 : \Omega}\ \text{WF\_}\cup$$

$$\frac{\mathbf{Y}, W \vdash \tau : \Omega}{W \vdash \forall\, \tau : \Omega}\ \text{WF\_}\forall \qquad \frac{\mathbf{Y}, W \vdash \tau : \Omega}{W \vdash \exists\, \tau : \Omega}\ \text{WF\_}\exists \qquad \frac{W[n] = \mathbf{Y}}{W \vdash \underline{n} : \Omega}\ \text{WF\_INDEX}$$

$$\frac{W \vdash i : \Omega \quad W \vdash \tau : \Omega}{W \vdash \{i : \tau\} : \Omega}\ \text{WF\_SINGLE} \qquad \frac{W \vdash \tau : \Omega \quad W \vdash i : \Omega}{W \vdash \tau \backslash i : \Omega}\ \text{WF\_}\backslash$$

$$\frac{W \vdash \iota : \mathit{INSTR} \quad W \vdash s : \Omega}{W \vdash \mathsf{at}(\iota, s) : \Omega}\ \text{WF\_at} \qquad \frac{W \vdash \tau, \tau', \tau'' : \Omega}{W \vdash \mathsf{instr}(\tau, \tau', \tau'') : \mathit{INSTR}}\ \text{WF\_INSTR}$$

$$\frac{\mathbf{Y}, W \vdash \iota : \mathit{INSTR}}{W \vdash \forall_\iota \iota : \mathit{INSTR}}\ \text{WF\_}\forall_\iota \qquad \frac{\mathbf{Y}, W \vdash \iota : \mathit{INSTR}}{W \vdash \exists_\iota \iota : \mathit{INSTR}}\ \text{WF\_}\exists_\iota \qquad \frac{W, \mathbf{N}, W' \vdash \tau : \Omega}{W, \mathbf{Y}, W' \vdash \tau : \Omega}\ \text{WF\_WEAKEN}$$

Figure 10: Static Semantics : Wellformedness of Types

$$\begin{array}{lll}
\textit{Terms } \tau, \iota & ::= & \underline{n} \mid \top \mid \bot \mid \mathsf{codeptr}\ \tau \mid \mathsf{offset}(n, \tau) \mid \mathsf{id}\ \tau \mid \mathsf{box}\ \tau \mid \\
 & & \tau \cap \tau' \mid \tau \cup \tau' \mid \forall \tau \mid \exists \tau \mid \mathsf{rec}\ \tau \mid \{n : \tau\} \mid \tau \backslash n \mid \\
 & & \mathsf{relate}_\pi(\tau_1, \tau_2)(n_1, n_2) \mid \mathsf{c}\ \ n \mid \mathsf{int}_{32} \mid \mathsf{int}_\pi n \mid n_1 + n_2 \mid \\
 & & \mathsf{at}(\iota, n) \mid \mathsf{instr}(\tau, \tau', \tau'') \mid \forall_\iota \iota \mid \exists_\iota \iota \mid \tau[s] \\
\textit{Substitutions } r, s & ::= & \mathbf{id} \mid \uparrow \mid \tau \cdot s \mid r \circ s
\end{array}$$

$$\frac{}{\underline{0}\ [\mathbf{id}] =_\tau \underline{0}}\ \text{es\_VarId} \qquad \frac{}{\underline{0}\ [\tau \cdot s] =_\tau \tau}\ \text{es\_VarCons} \qquad \frac{}{(\tau_1 \tau_2)[s] =_\tau (\tau_1[s])(\tau_2[s])}\ \text{es\_App} \qquad \frac{}{\forall\, \tau[s] =_\tau \tau[\underline{0} \cdot (s \circ \uparrow)]}\ \text{es\_}\forall$$

$$\frac{}{\tau[s_1]\,[s_2] =_\tau \tau[s_1 \circ s_2]}\ \text{es\_Clos} \qquad \frac{}{\tau[\mathbf{id} \circ s] =_\tau \tau[s]}\ \text{es\_IdL} \qquad \frac{}{\tau[\uparrow \circ \mathsf{id}] =_\tau \tau[\uparrow]}\ \text{es\_ShiftId}$$

$$\frac{}{\tau[\uparrow \circ (\tau' \cdot s)] =_\tau \tau[s]}\ \text{es\_ShiftCons} \qquad \frac{}{\tau_1 \cap \tau_2[s] =_\tau \tau_1[s] \cap \tau_2[s]}\ \text{es\_}\cap \qquad \frac{}{\tau_1 \cup \tau_2[s] =_\tau \tau_1[s] \cup \tau_2[s]}\ \text{es\_}\cup$$

$$\frac{}{\tau[(\tau' \cdot s) \circ s'] =_\tau \tau[\tau'[s'] \cdot (s \circ s')]}\ \text{es\_Map} \qquad \frac{}{\tau[s \circ (s' \circ s'')] =_\tau \tau[(s \circ s') \circ s'']}\ \text{es\_Ass}$$

$$\frac{}{\mathsf{int}_\pi \tau[s] =_\tau \mathsf{int}_\pi(\tau[s])}\ \text{es\_int}_\pi \qquad \frac{}{\mathsf{int}_{32}[s] =_\tau \mathsf{int}_{32}}\ \text{es\_int}_{32} \qquad \frac{}{\mathsf{c}\ n[s] =_\tau \mathsf{c}\ n}\ \text{es\_c}$$

Figure 11: Static Semantics : Explicit Substitution Rules

$$\frac{}{\tau \subset \tau}\ \subset\text{\_REFL} \qquad \frac{\tau \subset \tau'' \quad \tau'' \subset \tau'}{\tau \subset \tau'}\ \subset\text{\_TRANS} \qquad \frac{\tau \subset \tau' \quad \tau' \subset \tau}{\tau =_\tau \tau'}\ \subset\text{\_EQ}$$

$$\frac{}{\tau \subset \top}\ \subset\text{\_}\top \qquad \frac{}{\bot \subset \tau}\ \subset\text{\_}\bot \qquad \frac{}{\mathsf{int}_{=} i \subset \mathsf{int}_{\geq} i}\ \subset\text{\_INT\_}\pi \qquad \frac{}{\mathsf{int}_{=} i \subset \mathsf{int}_{\leq} i}\ \subset\text{\_INT\_}\pi$$

$$\frac{\tau_1 \subset \mathsf{int}_{\leq} n_1 \quad \tau_2 \subset \mathsf{int}_{\leq} n_2 \quad c(n_2 + n_2) \subset \mathsf{int}_{32}}{+(\tau_1, \tau_2) \subset \mathsf{int}_{32}}\ \subset\text{\_}+ \qquad \frac{\tau_1 \subset \tau_2}{\mathsf{id}\ \tau_1 \subset \mathsf{id}\ \tau_2}\ \subset\text{\_ID}$$

$$\frac{\phi_2 \subset \phi_1}{\mathsf{codeptr}(\phi_1) \subset \mathsf{codeptr}(\phi_2)}\ \subset\text{\_CPTR} \qquad \frac{\tau_1 \subset \tau_2}{\mathsf{offset}\ (i)\ \tau_1 \subset \mathsf{offset}\ (i)\ \tau_2}\ \subset\text{\_OFFSET} \qquad \frac{\tau_1 \subset \tau_2}{\mathsf{box}\ \tau_1 \subset \mathsf{box}\ \tau_2}\ \subset\text{\_BOXED}$$

$$\frac{}{\tau_1 \cap \tau_2 \subset \tau_1}\ \subset\text{\_}\cap\text{L1} \qquad \frac{}{\tau_1 \cap \tau_2 \subset \tau_2}\ \subset\text{\_}\cap\text{L2} \qquad \frac{\tau \subset \tau_3 \quad \tau \subset \tau_4}{\tau \subset \tau_3 \cap \tau_4}\ \subset\text{\_}\cap\text{R}$$

$$\frac{}{\tau_1 \subset \tau_1 \cup \tau_2}\ \subset\text{\_}\cup\text{R1} \qquad \frac{}{\tau_2 \subset \tau_1 \cup \tau_2}\ \subset\text{\_}\cup\text{R2} \qquad \frac{\tau_1 \subset \tau \quad \tau_2 \subset \tau}{\tau_1 \cup \tau_2 \subset \tau}\ \subset\text{\_}\cup\text{L}$$

$$\frac{\forall x.\ (\tau_1\ x) \subset (\tau_2\ x)}{\forall \tau_1 \subset \forall \tau_2}\ \subset\text{\_}\forall \qquad \frac{\forall x.(\tau_1\ x) \subset (\tau_2\ x)}{\exists \tau_1 \subset \exists \tau_2}\ \subset\text{\_}\exists \qquad \frac{\tau_1 \subset \tau_2}{\{i : \tau_1\} \subset \{i : \tau_2\}}\ \subset\text{\_SINGLE} \qquad \frac{}{\tau \subset \tau \backslash i}\ \subset\text{\_}\backslash$$

$$\frac{}{(\{i : F(\mathsf{int}_{=} n)\} \cap \mathsf{relate}_\pi(F, G)(i, j)) \subset \{j : G(int_\pi(n))\}}\ \subset\text{\_RELATE} \qquad \frac{\iota_1 \subset_\iota \iota_2}{\mathsf{at}(\iota_1, s) \subset \mathsf{at}(\iota_2, s)}\ \subset\text{\_at}$$

$$\frac{\forall x.\ ((\iota_1\ x) \subset_\iota (\iota_2\ x))}{\forall_\iota \iota_1 \subset_\iota \forall_\iota \iota_2}\ \subset\text{\_}\forall_\iota \qquad \frac{\forall_\iota x.\ ((\iota_1\ x) \subset_\iota (\iota_2\ x))}{\exists_\iota \iota_1 \subset_\iota \exists_\iota \iota_2}\ \subset\text{\_}\exists_\iota \qquad \frac{\tau_1' \subset \tau_1 \quad \tau_2' \subset \tau_2 \quad \tau_3 \subset \tau_3'}{\mathsf{instr}(\tau_1, \tau_2, \tau_3) \subset \mathsf{instr}(\tau_1', \tau_2', \tau_3')}\ \subset\text{\_INSTR}$$

Figure 12: Static Semantics : Subtyping Rules

		SPARC Kinds	
κ	::=	*Oper*	ALU operators
	|	*RegImm*	instruction modes
	|	Ω	types
		SPARC ALU Operators (*Oper*)	
op	::=	add(add_cc)	integer add (set codes)
	|	addc(addc_cc)	logical add/carry (set codes)
	|	and(and_cc)	logical AND (set codes)
	|	andn(andn_cc)	logical NAND (set codes)
	|	xor(xor_cc)	logical XOR (set codes)
	|	xnor(xnor_cc)	logical XNOR (set codes)
		SPARC Instruction Modes (*RegImm*)	
ri	::=	RMode n	Register mode
	|	IMode n	Immediate mode
		SPARC types (Ω)	
τ_s	::=	calc(op, n_1, n_2)	ALUop result
	|	cc_a | cc_n | cc_ne | ...	Condition codes
	|	$\backslash^{cc}$	Restrict on condition codes
	|	set_cc(op, n_1, n_2)	ALUop cc result
		SPARC instructions (*INSTR*)	
ι_s	::=	alu(x, c, op)(i, ri, k)	ALU instruction
	|	ibranch(τ_s, a, n_1)	integer branch instruction
	|	load(i, ri, j)	load instruction
	|	store(i, ri, j)	store instruction

Figure 13: TML Syntax : SPARC Kinds and Types

$$\frac{1 \le i < 32}{\vdash \mathsf{rmode}(i) : \Omega_{regimm}} \qquad \frac{-2^{12} \le i < 2^{12}}{\vdash \mathsf{imode}(i) : \Omega_{regimm}}$$

$$\frac{x \in \{0,1\} \quad c \in \{0,1\} \quad \vdash oper : Oper \quad \vdash regimm : RegImm \quad i,k : \Omega_{num}}{\vdash \mathsf{alu}(x,c,oper)(i,regimm,k) : \Omega_\iota}$$

Figure 14: Wellformedness of SPARC Instructions

$$\frac{}{\begin{array}{l}\mathsf{alu}(0,0,\mathsf{op})(i,\mathsf{RMode}(j),k) \subset_\iota \\ \forall \Gamma,\phi,n_1,n_2. \\ \quad \mathsf{instr}(\Gamma,\phi \cap \{i:\mathsf{int}_=n_1, j:\mathsf{int}_=n_2\}, \\ \quad\quad ((\phi \cap \{i:\mathsf{int}_=n_1, j:\mathsf{int}_=n_2\})\backslash i) \\ \quad\quad \cap \{k:\mathsf{calc}(\mathsf{op},n_1,n_2)\})\end{array}} \text{SPARC\_ALU\_1}$$

$$\frac{}{\begin{array}{l}\mathsf{alu}(0,1,\mathsf{op})(i,\mathsf{RMode}(j),k) \subset_\iota \\ \forall \Gamma,\phi,n_1,n_2. \\ \quad \mathsf{instr}(\Gamma,\phi \cap \{i:\mathsf{int}_=n_1, j:\mathsf{int}_=n_2\}, \\ \quad\quad ((\phi \cap \{i:\mathsf{int}_=n_1, j:\mathsf{int}_=n_2\})\backslash i\backslash^{\mathsf{cc}}) \\ \quad\quad \cap \{k:\mathsf{calc}(\mathsf{op},n_1,n_2)\} \\ \quad\quad \cap \mathsf{set\_cc}(\mathsf{op},n_1,n_2))\end{array}} \text{SPARC\_ALU\_2}$$

$$\frac{\begin{array}{l}\iota \subset_\iota \mathsf{instr}(\Gamma,\phi \cap \tau_\mathsf{s}, \phi \cap \tau_\mathsf{s}) \\ \iota \subset_\iota \mathsf{instr}(\Gamma,\phi \cap \overline{\tau_\mathsf{s}}, \phi \cap \overline{\tau_\mathsf{s}})\end{array}}{\begin{array}{l}\{l:\mathsf{at}(\mathsf{ibranch}(\tau_\mathsf{s},0,i),4), l+4:\mathsf{at}(\iota,4)\} \subset_\iota \\ \{l:\mathsf{at}(\mathsf{instr}(\Gamma \cap \{l+i:\mathsf{codeptr}(\phi_t)\},\phi,\phi_f),8)\}\end{array}} \text{SPARC\_BRANCH}$$

$$\frac{}{\begin{array}{l}\mathsf{load}(i,\mathsf{IMode}(c),j) \subset_\iota \\ \forall \Gamma,\phi,\tau. \\ \quad \mathsf{instr}(\Gamma,\phi \cap \{j:\mathsf{field}\ c\ \tau\}, \\ \quad\quad ((\phi \cap \{j:\mathsf{field}\ c\ \tau\})\backslash i) \cap \mathsf{relate}_=(\mathsf{id},\mathsf{field}\ c)(i,j)\end{array}} \text{SPARC\_LOAD}$$

$$\frac{}{\begin{array}{l}\mathsf{store}(i,\mathsf{IMode}(c),j) \subset_\iota \\ \forall \Gamma,\phi,\tau. \\ \mathsf{instr}(\Gamma,\phi \cap \ \{i:\tau\} \cap \ \mathsf{relate}_=(\mathsf{offset}\ c,\mathsf{id})(j,b) \\ \quad\quad\quad\quad \mathsf{relate}_\le(\mathsf{offset}\ 4,\mathsf{id})(b,l), \\ \quad\quad\quad \phi \cap \ \{i:\tau\}\backslash r_b \cap \mathsf{relate}_=(\mathsf{offset}\ (c{+}4),\mathsf{id})(j,b) \\ \quad\quad\quad\quad \mathsf{relate}_\le(\mathsf{id},\mathsf{id})(b,l)\end{array}} \text{SPARC\_STORE}$$

$$\frac{}{\mathsf{tst}(i) \subset_\iota \forall \Gamma,\phi.\ \ \mathsf{instr}(\Gamma,\phi \cap \ \{i:\mathsf{int}_{32}\}, \ \phi \cap ((\mathsf{cc\_ne} \cap \{\mathsf{o1}:\mathsf{int}_{\neq}0\}) \cup (\mathsf{cc\_e} \cap \{\mathsf{o1}:\mathsf{int}_=0\})))} \text{SPARC\_TST}$$

$$\frac{}{\mathsf{mov}(i) \subset_\iota \forall \Gamma,\phi.\ \ \mathsf{instr}(\Gamma,\phi,(\phi\backslash j) \cap \{j:\mathsf{int}_=c\})} \text{SPARC\_MOV}$$

Figure 15: Static Semantics : SPARC Instructions in TML