A Provably Sound TAL for Back-end Optimization

Juan Chen Dinghao Wu Andrew W. Appel Hai Fang

Princeton University
{juanchen, dinghao, appel}@cs.princeton.edu

Yale University
hai.fang@yale.edu

ABSTRACT

Typed assembly languages provide a way to generate machine-checkable safety proofs for machine-language programs. But the soundness proofs of most existing typed assembly languages are hand-written and cannot be machine-checked, which is worrisome for such large calculi. We have designed and implemented a low-level typed assembly language (LTAL) with a semantic model and established its soundness from the model. Compared to existing typed assembly languages, LTAL is more scalable and more secure; it has no macro instructions that hinder low-level optimizations such as instruction scheduling; its type constructors are expressive enough to capture dataflow information, support the compiler's choice of data representations and permit typed position-independent code; and its type-checking algorithm is completely syntax-directed.

We have built a prototype system, based on Standard ML of New Jersey, that compiles most of core ML to Sparc code. We explain how we were able to make the untyped back end in SML/NJ preserve types during instruction selection and register allocation, without restricting low-level optimizations and without knowledge of any type system pervading the instruction selector and register allocator.

Categories and Subject Descriptors

F.3.1 [**Theory of Computation**]: Logics and Meanings of Programs—*specifying and verifying and reasoning about programs*; D.3.4 [**Software**]: Programming Languages—*processors, compilers*

General Terms

Languages, Verification

Keywords

Typed Assembly Language, Proof-Carrying Code

To appear in PLDI 2003: ACM SIGPLAN Conference on Programming Language Design and Implementation. This research was supported in part by DARPA award F30602-99-1-0519.

1. INTRODUCTION

The idea of Proof-Carrying Code (PCC) [18] is that the compiler should produce machine code accompanied by a proof of safety. A weakness of previous PCC systems is that the proof-checking infrastructure is too complex to prove sound. We have built the first compiler that produces machine code accompanied by safety proofs that are machine-checkable in a simple logic from minimal axioms.

Most PCC compilers, including ours, are based on typed intermediate languages or Typed Assembly Language (TAL) [17], which provide a way to generate safety proofs automatically. TAL has a soundness guarantee: If a TAL program type-checks and there is no bug in the assembler, the machine code is safe to execute. Soundness is proved as a metatheorem outside of the proving system; the proof is hand-written and not machine-checkable. The typing rules and the type checker are in the trusted computing base (TCB), that is, bugs in these components can let unsafe code slip past the checker. There have been many variants of TAL [15, 23, 16], which rely on similar soundness metatheorems. A recent variant [10] has a machine-checkable metatheorem.

It is hard to manage the soundness proofs and avoid errors when scaling up to realistic type systems for real compilers. The goal of the Foundational Proof-Carrying Code (FPCC) [2] project at Princeton is to build machine-checkable safety proofs for machine-code programs from the minimal set of axioms. We have designed a low-level typed assembly language (LTAL) to be the interface between the compiler and the checker: the compiler compiles a source program to machine code annotated by an LTAL program.

This paper focuses on the LTAL interface and the compiler. Our design and implementation has the following desirable properties, some of which are shared by some other TAL and PCC systems (see Figure 1 and Appendix A):

Compiles a "real" source language. We have built a compiler for almost all of core ML—a full-scale source language with polymorphic higher-order functions, disjoint-sum recursive datatypes, and so on.

Compiles to a real target machine. We generate high-quality Sparc code.

Foundational specification. We have a concise logical specification, independent of any type system, of the safety property guaranteed by our system: in our prototype we guarantee memory safety and that only a certain subset of Sparc instructions will be executed [2]. Furthermore, our specification relates to the actual *machine language* to be executed—not assembly language—we model (and check) instruction encodings explicitly.

Key: ○ partially ⊙ nearly ● completely	Compiles “real” source language	Compiles to real target machine	Foundational specification	Machine-checked soundness proof	Minimal checker	Atomicity	Compiler can choose data reprs.	Dataflow analysis	Position-independent code	Basic blocks	Syntax-directed checking
	1	2	3	4	5	6	7	8	9	10	11
SpecialJ [9]	●	●				○		●			
TALx86 [17]	⊙	●				○	●	○		○	○
DTAL [23]								●			
FTAL [12]			⊙	●							●
TALT [10]		●	⊙	⊙			●	●	○		
Our LTAL	⊙	●	●	⊙	●	⊙	●	●	●	●	●

Figure 1: Comparison of typed assembly languages (see Appendix A)

Machine-checked proof. We have a machine-checked proof (mostly finished) of the soundness of our system—that is, if the LTAL type-checks, the machine code is safe. Unlike any other TAL or PCC system, our proof is with respect to a minimal set of axioms, the largest part of which is a specification (in logic) of the instruction set architecture of the Sparc processor.

Minimal checker. Just in case you are worried about bugs (or Trojan horses) in proof checkers, our soundness proof is checkable in a very minimal logic: the trusted base of our system (including axioms, machine specification, and a C program implementing LF checking) is less than 2700 lines of code [5, 22], an order of magnitude smaller than other systems.

Atomicity. Some other TALs have “macro” instruction sequences (or even worse, calls to the runtime system) for compare-and-branch, or datatype tag-checking, or memory allocation. This inhibits optimizations such as hoisting and scheduling.[1] Each of our LTAL instructions corresponds to at most one machine instruction.

Compiler can choose data representations. For data structures such as tagged disjoint sums, a compiler may want to exercise discretion in choosing data layouts, unhampered by assumptions built into a typed assembly language. LTAL permits this flexibility; some other TALs do not.

Dataflow & induction analysis. LTAL includes existential and singleton types that are powerful enough to permit dataflow-based safety proofs of optimized machine code (though our prototype compiler does not exploit all of this power yet).

Position-independent code. To avoid the need to trust a linker, we show how to check typed position-independent code—even in the presence of long jumps and of operations that move code addresses into pointer variables and closures.

Basic blocks. LTAL groups instructions into basic blocks, making it easy for an optimizing compiler to reorder blocks to optimize cache placement or shorten span-dependent instructions.

Syntax-directed. Typechecking LTAL is syntax-directed; that way, if a compiler generates a well-typed LTAL program it doesn't have to worry about whether the checking algorithm will be smart enough to find a proof.

2. OVERVIEW OF FPCC

Necula's PCC system [18] constructs for untrusted code a verification condition (VC), which has the property that if VC holds with regard to the logic axioms and the typing rules, the program is safe. A VC generator (VCGen) is used by both the code producer and the code consumer to construct VCs. VCGen examines a machine-code program instruction by instruction and calculates the weakest preconditions for each instruction in Hoare-logic style. This VC-based verification builds the type system and machine instruction semantics into the algorithm for formulating the safety predicate. VCGen must be trusted to generate the right formula, but it is a large program (23,000 lines of C code [6]), thus difficult to guarantee bug-free.

2.1 FPCC

The motivation of Foundational PCC is to make the TCB as small as possible, without committing to any specific type system. We believe that the smaller the TCB, the more confidence PCC users can have. Our TCB consists of the specification of the safety policy, machine instruction semantics, and the proof checker. In the current implementation, it is less than 2,700 lines of code [5, 22], of which more than half is the specification of the Sparc instruction set architecture. To make the TCB minimal, we choose Church's higher-order logic with a few axioms of arithmetic, give types a semantic model to move the type system out of the TCB, and model machine instructions by a step relation between machine states; we avoid VCGen entirely [3].

In order to support contravariant recursive datatypes and mutable fields, we model types as predicates on states, approximation indices [4], and type levels [1]. We have an abstraction layer, Typed Machine Language (TML) [20], to hide the complex semantic models for types. TML provides a rich set of constructors for types, type maps, and instructions, and an orthogonal set of primitive type constructors such as union, intersection, existential and universal quantification, and so on. TML is so expressive that type-checking for it is undecidable; it is more a logic than a type system. However, it is very useful for building semantic models of higher-level, application-specific type systems such as LTAL: we give LTAL constructors a semantic model in terms of TML.

The soundness of LTAL typing rules is proved not by a metatheorem as in TAL, but by their semantic model [21], bottom up: first we use higher-order logic with axioms for arithmetic to prove lemmas about machine instructions and types, then we prove the TML typing rules based on these lemmas, then we prove the soundness of LTAL typing rules in the TML model. Each typing rule is represented as a derived lemma in our logic.

LTAL benefits from its semantic model in many aspects: first, it is more scalable. Adding new rules that can be described in our semantic model generally does not affect the soundness of existing rules, which we found very useful

[1] These optimizations can be done in the assembler, but need to be trusted bug-free, whereas our system does not need to trust them.

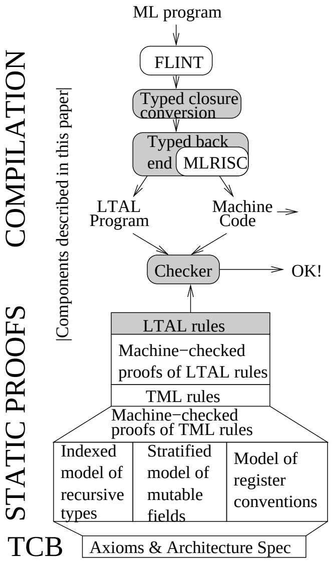

ML program
FLINT
Typed closure conversion
Typed back end
MLRISC
LTAL Program
Machine Code
Checker
OK!
COMPILATION
STATIC PROOFS
TCB
[Components described in this paper]
LTAL rules
Machine−checked proofs of LTAL rules
TML rules
Machine−checked proofs of TML rules
Indexed model of recursive types
Stratified model of mutable fields
Model of register conventions
Axioms & Architecture Spec

Figure 2: Foundational PCC Framework

in evolving the design. Second, it is more secure because the typing rules are moved out of the TCB. Third, TML connects LTAL to real machine instruction semantics, thus bridges the gap between typed assembly language and machine language.

The FPCC framework is shown in Figure 2. A source program is compiled into a machine-code program and an LTAL program. The "code consumer" receives the LTAL rules, along with their soundness proof; checks the soundness proof [5, 22]; and then runs the LTAL checker, which is a simple computation (like Prolog but without backtracking and with only a very limited form of unification).

LTAL is not intended as a universal TAL. Instead, it is extensible. Our semantic modeling technique is very modular. New operators can be added to LTAL (and proved sound) without disturbing the soundess proofs for existing operators, as long as the new operators conform to the assumptions in the semantic model. We started with a very simple model [3], and when we added contravariant recursive types [4] and mutable record fields [1] these changes did violate previous assumptions and require nonmodular rewrites. But now our model is very powerful and general: none of the existing LTAL soundness proofs will need to be touched when we add operators to handle extensible sums, various kinds of exception handling mechanisms, various kinds of multidimensional arrays (with or without pointer indirections), or arbitrary predicates on scalar values.

2.2 FPCC-ML Compiler

Our compiler transforms core ML (ML without the module system) into Sparc code with LTAL annotations. At present our prototype omits exceptions, arrays, and strings. We have built our compiler based on the Standard ML of New Jersey system.

There are several stages: the front end of SML/NJ translates source ML programs to FLINT (a typed intermediate language based on F_ω) [19]; we have reused the FLINT front end. Our newly built typed CPS-conversion and closure conversion phases generate NFLINT (a typed intermediate language like Morrisett's λ_C [17]). The next few phases break down complex instructions, build basic blocks, and insert coercions to get machine-independent LTAL programs. The back end takes machine-independent LTAL, and produces machine code with machine-specific LTAL annotations and some auxiliary information, such as mapping from labels to their addresses.

SML/NJ's back end uses the untyped MLRISC retargetable instruction selection, register allocation, and low-level optimization software [11]. The difficulty is to make MLRISC preserve and manipulate type information, without rewriting the MLRISC or making it dependent on our particular type system. Fortunately, MLRISC already had some support for an annotation mechanism [13] that permits "comments" on the instructions; we have generalized this mechanism and used it to propagate types.

2.3 Checker

Our checker has two main components. First, it uses a simple LF type-checker to check a proof, in higher-order logic, of the soundness of the LTAL typing rules [5, 22]. We can view these LTAL rules as a set of lemmas.

On the other hand, the LTAL rules can be regarded as a set of Prolog-like clauses. Then, because these rules are syntax-directed, the checker can run a very simple subset Prolog interpreter (without backtracking and with only a limited form of unification) on these rules to type-check the machine-language program [5, 22].

The LTAL program is only an untrusted hint so that the checker can take advantage of type and dataflow information from the compiler in proving the safety of the machine code. The process of running the checker on a machine code and the corresponding LTAL program is like type-checking the machine code according to structural information from the LTAL program. The overall goal of the checker is `judge_prog H P` where P is the binary code (a sequence of instruction words) and H is the corresponding LTAL program. The predicate `judge_prog` characterizes well-typedness. The checker solves this goal according to the structure of H. In the underlying semantic model, we can prove that well-typedness implies safety:

```
judge_prog H P -> safe_program P.
```

The predicate `safe_program` is the machine-level safety policy. When the checker succeeds on the goal `judge_prog H P`, we apply this lemma to get a proof of `safe_program P`.

3. LTAL

We have designed our own typed assembly language because we want to generate safety proofs of machine code, with as much flexibility as possible for an optimizing compiler. Thus, even part-way through a sequence of instructions that allocates on the heap or that does datatype-tag discrimination, the LTAL type system must be able to describe the machine state. That is, LTAL has no "macro" instructions: each LTAL instruction corresponds to one Sparc instruction (or is a coercion with no runtime effect). Because no sequence of instructions is unbreakable, low-level optimizations such as instruction scheduling are permissible (however, at present our LTAL does not accommodate the filling of branch-delay slots on the Sparc). Macro instructions in other TALs (such as *malloc* and *test-and-branch*)

	τ	$::= \alpha \mid \top \mid \bot \mid \mathrm{int} \mid \exists\alpha.\tau \mid \mu\alpha.\tau \mid \mathrm{boxed}$	**Types**
		$\mid\ \bar{n} \mid \mathrm{int}_\pi\,\tau \mid \mathrm{field}\ i\ \tau \mid \tau_1 \cap \tau_2 \mid \tau_1 \cup \tau_2$	
		$\mid\ \mathrm{codeptr}[\alpha_1, \ldots, \alpha_j](m, cc, v_1 : \tau_1, \ldots, v_n : \tau_n)$	
		$\mid\ \mathrm{addr}(l) \mid \mathrm{diff}(l_1, l_2) \mid \mathrm{def}\,\mathbb{k}$	
	cc	$::= \mathrm{cc\_cmp}(\tau_1, \tau_2)$	**Cond.Codes**
		$\mid\ \mathrm{cc\_testm}(m) \mid \mathrm{cc\_none}$	
	v	$::= x \mid i \mid l \mid c(v) \mid \mathrm{vdiff}(l_1, l_2)$	**Values**
	c	$::= \mathrm{cid} \mid c_1 \circ c_2 \mid \mathrm{cpack}(\tau_1, \tau_2)$	**Coercions**
		$\mid\ \mathrm{cfold}[\tau] \mid \mathrm{cunfold} \mid \mathrm{crange}[n_1, n_2]$	
		$\mid\ \mathrm{cinj1}\ \tau \mid \mathrm{cinj2}\ \tau \mid \mathrm{cproj1} \mid \mathrm{cproj2}$	
		$\mid\ \mathrm{cunion}(c_1, c_2) \mid \mathrm{cinters}(c_1, c_2) \mid \mathrm{cname} \mid \mathrm{cdef}\,\mathbb{k}$	
	op	$::= + \mid - \mid * \mid /$	**Arith. Ops**
	π	$::= \; = \mid \neq \mid > \mid \geq \mid < \mid \leq$	**Arith. Compares**
			Instructions
	ι	$::= (\alpha, v') = \mathrm{open}(v)$	*no instruction*
		$\mid\ v' = v$	*move, or nop*
		$\mid\ v = v_1\ \ op\ \ v_2$	*ALU instructions*
		$\mid\ v = \mathrm{sethi}(n)$	*sethi*
		$\mid\ \mathrm{store}(v_i, v)$	*store*
		$\mid\ v = \mathrm{record}$	*move*
		$\mid\ \mathrm{inc\_alloc}(v)$	*add*
		$\mid\ v = \mathrm{load}(v_1, v_2)$	*load*
		$\mid\ v = \mathrm{addradd}(v_1, v_2)$	*add*
		$\mid\ \mathrm{call}(v, [\tau_1, \ldots, \tau_n])$	*jump*
		$\mid\ \mathrm{calln}(l, [\tau_1, \ldots, \tau_n])$	*fall through*
$\star$		$\mid\ \mathrm{cmpcc}(v_1, v_2)$	*subcc*
$\star$		$\mid\ (\alpha, v_1') = \mathrm{cmpcci}(v_1, v_2)$	*subcc*
$\star$		$\mid\ \mathrm{testm}(n)$	*subcc*
$\star$		$\mid\ \mathrm{if}(\pi)\ \mathrm{then}\ l_1\ \mathrm{else}\ l_2$	*branch*
$\star$		$\mid\ \mathrm{ifr}(\pi)\{v\}\ \mathrm{then}\ (v_1, l_1)\ \mathrm{else}\ (v_2, l_2)$	”
$\star$		$\mid\ \mathrm{iffull}\ \mathrm{then}\ l_1\ \mathrm{else}\ l_2$	”
$\star$		$\mid\ \mathrm{iftag}(\pi)\{v\}\ \mathrm{then}\ (v_1, l_1)\ \mathrm{else}\ (v_2, l_2)$	”
			Basic block
	B	$::= l[\alpha_1, \ldots, \alpha_j](m, cc, v_1 : \tau_1, \ldots, v_n : \tau_n) = \iota_1; \ldots; \iota_k$	
	LRT	$::= (L, R, T)$	**Environments**
	L	$::= \{l_1 \mapsto a_1, \ldots, l_n \mapsto a_n\}$	*label map*
	R	$::= \{x_1 \mapsto r_1, \ldots, x_n \mapsto r_n\}$	*register map*
	T	$::= \{\mathbb{k}_1 \mapsto \tau_1, \ldots, \mathbb{k}_n \mapsto \tau_n\}$	*type abbrev. map*
	P	$::= (LRT, \vec{B})$	**Program**

Figure 3: LTAL Syntax. Marked $\star$ operators are specific to machines with condition codes.

that expand to a fixed sequence of machine instructions, interfere with low-level optimization.

3.1 Syntax

LTAL is a calculus with conventional features such as variable names and scoping rules. The LTAL syntax is shown in Figure 3. LTAL supports first-order kinds; it has only limited support for higher-order kinds, since TML does not model higher-order kinds in full generality. For core ML, this is enough.

LTAL has a set of standard types: type variables[2], top and bottom types, integer types, existential types, and recursive types. We give type **boxed** to pointers pointing to heap values.

There are low-level constructors to model high-level abstractions, such as singleton integer type $\bar{n}$ and refined integer type $\mathrm{int}_\pi\,\bar{n}$ for integers (i has type $\mathrm{int}_\pi\,\bar{n}$ means $i\,\pi\,n$ is true, where π is a predicate on integers such as $=$ or $\leq$), field types, intersection types and union types for records and user-defined datatypes.

To model basic blocks (with their live variables) and functions (with their formal parameters) we have polymorphic "code pointer" types $\mathbf{codeptr}[\vec{\alpha}](m, cc, v_1 : \tau_1, \ldots, v_n : \tau_n)$. where $\vec{\alpha}$ is a list of type variables, m is the available memory size known at this point, cc is the condition code requirement, and $v_i : \tau_i$ are the input arguments.

For label arithmetic we have type constructors **addr** and **diff**, which will be explained further in Section 3.7.

Type **def** refers to a type expression by a name $\mathbb{k}$; in our implementation, names are just integers. Each program can have a sequence of type abbreviations that give names to type expressions. This mechanism makes LTAL programs concise, and saves the checker some work. The checker expands a name to the type expression it stands for only when such expansion is needed. Otherwise, the checker simply passes the name around, which is more efficient than passing the type expression.

We have a special category cc to capture the condition codes status (on machines with condition codes), which includes **cc_cmp** for comparison, **cc_testm** for memory availability testing, and **cc_none** for arbitrary status.

A value can be a variable x, an integer i, a label l, a coerce value $c(v)$, or a **vdiff** value. We use variables to track aliases of registers. Different variables with different types can be assigned the same register, indicating different views of the same register to the type-checker. Value constructor **vdiff** and type constructors **addr** and **diff** are used for typed position-independent code. Their meaning is explained in Section 3.7.

Coercions are used to change the type of values; all coercions are free of runtime effect, as they follow subtyping relations in the underlying model. Many of these coercions are conventional, such as identity, composition, pack, fold/unfold, inject, and project. Coercion rules are further discussed in Section 3.5.

LTAL has a machine-independent core, which includes: move and ALU instructions, **sethi** for loading large integers, **store**, **record**, and **inc_alloc** for heap allocation, **calln** for "call by fall-through," (which generates no code), and **addradd** for address arithmetic. Each target machine requires the addition of machine-specific operators and rules. The instructions in $\mathrm{LTAL}_{\mathrm{Sparc}}$ that are specific to machines with condition codes are: **cmpcc** compares two integers and sets condition codes; **cmpcci** compares a value with a compile-time-known integer, sets condition codes and refines the type of the value; **testm** tests for out-of-heap; **if** is normal conditional branch without type refinement; **ifr** is conditional branch with type refinement in both branches, and **iffull** and **iftag** specialize type refinement for memory allocation and datatype tag discrimination, respectively.

Function declaration $l[\vec{\alpha}](m, cc, v_1 : \tau_1, \ldots, v_n : \tau_n) = \iota_1; \ldots; \iota_k$ defines a function (basic block) with label l, type parameters $\vec{\alpha}$, formal parameters $v_1 : \tau_1, \ldots, v_n : \tau_n$, and function body $\iota_1 \ldots \iota_k$ which is a sequence of LTAL instructions. The number m specifies how much memory is guaranteed to be available when the function is called. If a function specifies m words and allocates no more than m words, there is no need to test the memory availability. Otherwise, it has

[2] In our implementation we use de Bruijn indices, but in this presentation we will show named variables.

to check explicitly if there is enough memory. The condition-code requirement cc specifies the status of condition codes when the function is called. The function label l is assigned a code pointer type $\mathbf{codeptr}[\vec{\alpha}](m, cc, v_1 : \tau_1, \ldots, v_n : \tau_n)$. Each function is closed in the sense that there are no free type variables or value variables.

Triple LRT represents three environments that keep auxiliary information for type checking: label environment L maps labels to addresses (offset from the beginning of the program); register environment R maps variables to temporaries (registers or spill locations); type abbreviation environment T maps type abbreviations to their expansions.

An LTAL program consists of the above environments and a set of function declarations.

3.2 Static Semantics

The low-level type and term constructors in LTAL make the typing system expressive. Yet we need a decidable and simple type-checking algorithm so that proof generation can be done without a complicated decision procedure or constraint solver. To this end, we have made LTAL completely syntax-directed. There are no subtyping rules; instead, we use coercions to avoid nondeterministic choices during type checking. We explain various typing judgments, and then show some typing rules in this section.

The typing judgment for values $LRT; \rho; \Phi \vdash v : \tau$ means value v has type τ under environment $LRT; \rho; \Phi$. Triple LRT is part of the program. Kind environment ρ is a list of type variables bound so far (in our implementation we use de Bruijn numbers, so ρ is just a number). Value environment Φ maps variables to their types.

The judgment $LRT \vdash (\rho; \hbar; \Phi; cc)\ \{\iota\}\ (\rho'; \hbar'; \Phi'; cc')$ means after instruction ι is executed, environment $(\rho; \hbar; \Phi; cc)$ becomes $(\rho'; \hbar'; \Phi'; cc')$. The construction $\Phi, v : \tau$ augments Φ with a new binding $v : \tau$ and keeps the bindings other than v unchanged. The heap-allocation environment $\hbar$ is explained in Section 3.4. Environment cc specifies the current status of condition codes.

As an example we will show a simplified rule for an LTAL *add* instruction. In Section 3.7 we will show a different typed version of *add*. These two different typed versions of *add* expand to the same Sparc machine instruction. The first rule we show here is useful for compiling a source-language *add* for which no dataflow tracking is needed to prove safety; the second is useful for compiling address arithmetic. Having multiple LTAL instructions for the same machine instruction simplifies type-checking.

$$\frac{LRT; \rho; \Phi \vdash x : \mathrm{int} \quad LRT; \rho; \Phi \vdash y : \mathrm{int}}{LRT \vdash (\rho; \hbar; \Phi; cc)\ \{z = x + y\}\ (\rho; \hbar; \Phi, z : \mathrm{int}; cc)}$$

In fact, this rule is dramatically simplified for clarity. The full version looks like this:

$$\frac{\begin{array}{llll} (1) & LRT; \rho; \Phi \vdash x : \mathrm{int}_{32} & (2) & LRT; \rho; \Phi \vdash y : \mathrm{int}_{32} \\ (3) & \ell' = \ell + 4 & & \\ (4) & \mathrm{rmap}(LRT)(z) = t_z & (5) & \mathrm{rmap}(LRT)(x) = t_x \\ (6) & \mathrm{realreg}(t_z) = r_z & (7) & \mathrm{realreg}(t_x) = r_x \\ (8) & y_m = \mathrm{match\_reg\_or\_imm}(y) & & \\ (9) & \Phi' = \{z : \mathrm{int}_{32}\} \cap (\Phi \backslash z) & & \\ (10) & \mathrm{decode\_list}\ \ell\ \ell' P\ P'\ \mathrm{i\_ADD}(r_x, y_m, r_z) & & \end{array}}{LRT; \Gamma \vdash (\ell; \rho; \hbar; \Phi; cc; P)\{z = x + y\}(\ell'; \rho; \hbar; \Phi'; cc; P')}$$

The first and second premises state that both x and y have type int_{32}, the 32-bit integer type. Address ℓ is the location of current instruction $z = x + y$; ℓ' is the location of the next instruction. Premise (3) specifies that the length of the *add* instruction is 4 bytes.

Premises (4) and (5) relate variables z and x to their temporary numbers, and premises (6) and (7) map temporaries to registers; this rule would not be applicable to operands represented in spill locations (but of course that's true of the actual Sparc *add* instruction too). There are about 1000 temporaries (after register allocation); the first 20 are registers, and the remainder are in the spill area. The per-program *rmap*—the R component of LRT—maps variables to temporaries; the program-independent relations *realreg* and *memtemp* relate temporaries to their machine representation.

Since value y can be either a register or an immediate, we use *match_reg_or_imm* in premise (8) to match either a register or an immediate. So y_m can be either (rmode r_y) for some register r_y or (imode i) for some immediate i.

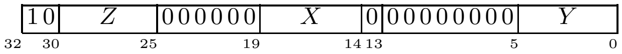
Premise (9) states the relation between the value typing context before and after execution of the current instruction. Before we add the type of variable z into the context, all aliases of z should be killed since they are not live anymore, which is what $\Phi \backslash z$ does.

Premise (10) will be explained in the next subsection.

The conclusion is like a Hoare-logic judgment. In environment LRT, the instruction $z = x + y$ is at location ℓ; the length of the instruction is $\ell' - \ell$; this instruction does not affect type contexts ρ or heap allocation environment $\hbar$; value context Φ becomes Φ' after execution; the machine code at location ℓ' is P'.

3.3 Instruction decoding

The *decode_list* relation in premise (10) maps an instruction word to a higher-level instruction with semantic meaning. Specifically, it says that the instruction word at the beginning of P with length $\ell' - \ell$ is an add instruction $\mathrm{i\_ADD}(r_x, y_m, r_z)$. We check for proper instruction encoding with rules such as the following:



$$\frac{\begin{array}{ll} 32 \cdot 2 + Z = X_9 & 64 \cdot X_9 + 0 = X_7 \\ 32 \cdot X_7 + X = X_6 & 2 \cdot X_6 + 0 = X_4 \\ 256 \cdot X_4 + 0 = X_1 & 32 \cdot X_1 + Y = W \end{array}}{\mathrm{decode}(\mathrm{i\_ADD}(X, \mathrm{rmode}(Y), Z),\ W)}$$

This rule is not an axiom of our system, it is a lemma derived from a more concise and readable definition of instruction encodings [14]. The predicate $A \cdot B + C = D$ shown here is a simplification of an actual predicate that also checks that $C < A$ and that A, B, C, D are natural numbers.

3.4 Heap Allocation

Like SML/NJ, our compiler allocates closures and records in registers or on the heap; we don't push and pop the stack. At present, our type system (like most TALs) also does not accommodate reasoning about garbage collection either. We intend to handle stacks and GC in the future, after we develop a unified theory of stack and heap deallocation (probably based on a region calculus).

As in SML/NJ, with so much heap allocation we need extremely efficient, in-line allocation of records. We model the allocable heap memory as a large contiguous region bounded by two pointers, *allocptr* and *limitptr*. Heap allocation is broken into two steps: first, test whether there is enough memory for allocation; second, initialize memory.

Before the runtime system starts executing a program, it reserves a chunk of memory, and sets the *allocptr* to the lowest address of the memory chunk, and the *limitptr* the highest address (minus a constant $C = 4096$). When the program needs n memory words, where $4n \leq C$, it tests whether $allocptr \leq limitptr$; if so, then at least n words must be available. Then it fills in n words consecutively to addresses from *allocptr* to $allocptr + 4n - 4$, then increases *allocptr* by $4n$.

The following LTAL instruction sequence creates a 3-field record $[v_0, v_1, v_2]$ and assigns it to v. The corresponding Sparc instructions are on the right side of the table (d, d_0, d_1, d_2 are registers assigned to LTAL variables v, v_0, v_1, v_2).

LTAL	Sparc
l_0 :	l_0 :
testm(3)	**subcc** *allocptr*, *limitptr*, $\%g0$
iffull then l_1 else l_2	**bg** l_1
l_2 :	l_2 :
store$(0, v_0)$	**st** d_0, [*allocptr* + 0]
store$(1, v_1)$	**st** d_1, [*allocptr* + 4]
store$(2, v_2)$	**st** d_2, [*allocptr* + 8]
v = record	**mov** *allocptr*, d
inc_alloc 3	**add** *allocptr*, 12, *allocptr*
...	...
l_1 : ...	l_1 : ...

Block l_0 tests if there are at least 3 words in the memory for allocation; after the **testm** comparison the condition-code environment is **cc_testm**(3). Then the branch instruction **iffull** "consumes" this condition code, and statically guarantees 3 words in the fall-through case (memory is not full).

Block l_2 initializes the three newly allocated words. Instruction **store**(i, v_i) initializes the word whose address is $allocptr + 4i$ with v_i. Instruction v = **record** copies *allocptr* to v. Instruction **inc_alloc** n increases *allocptr* by $4n$.

The instruction sequence for allocation is not fixed. The instruction scheduler can shuffle these instructions with others, as long as certain constraints hold.

An allocation environment $\hbar$ is used to check heap allocation. It consists of three parts: the number of words that are guaranteed to be available in the memory, the largest index of initialized fields, and the type of the partial record initialized so far. We don't need the initialization flags used in TALx86 [17].

The typing rules for the allocation instructions are shown in Figure 4. The judgement $LRT; \rho; \hbar; \Phi; cc \vdash_\ell l$ states that the signature of block l matches the current environment. If this judgement holds, it is safe to jump to block l. Instructions **testm** and **iffull** establish the allocation environment in which the **store** instructions type-check. The compiler can (and does) optimize by making one **iffull** cover the sequential allocation of several different records in a control-flow path that covers several basic blocks. The parameter m of **codeptr** conveys the necessary information about how much memory is guaranteed to remain.

$$\frac{0 \leq n \leq 1024}{LRT \vdash (\rho; \hbar; \Phi; cc)\ \{\text{testm(n)}\}\ (\rho; \hbar; \Phi; \text{cc\_testm}(n))}$$

$$\frac{\begin{array}{c} cc = \text{cc\_testm}(n) \qquad LRT; \rho; \hbar; \Phi; cc \vdash_\ell l_1 \\ LRT; \rho; (n, -1, \text{boxed}); \Phi; cc \vdash_\ell l_2 \end{array}}{LRT \vdash (\rho; \hbar; \Phi; cc)\ \{\text{iffull then } l_1 \text{ else } l_2\}\ (\rho; \hbar; \Phi; cc)}$$

$$\frac{\begin{array}{l} LRT; \rho; \Phi \vdash v_i : \text{int}_= i \qquad 0 \leq i < n \qquad m' = max(m, i) \\ LRT; \rho; \Phi \vdash v : t_i \qquad t' = t \cap (\text{field } i\ t_i) \end{array}}{LRT \vdash (\rho; (n, m, t); \Phi; cc)\ \{\text{store}(v_i, v)\}\ (\rho; (n, m', t'); \Phi; cc)}$$

$$\frac{}{LRT \vdash (\rho; (n, m, t); \Phi; cc)\ \{v = \text{record}\}\ (\rho; \hbar; \Phi, v\!:\!t; cc)}$$

$$\frac{LRT; \rho; \Phi \vdash v : \text{int}_= n' \qquad m < n' \leq n}{\begin{array}{c} LRT \vdash (\rho; (n, m, t); \Phi; \text{cc\_testm}(k))\ \{\text{inc\_alloc } v\} \\ (\rho; (n - n', -1, \text{boxed}); \Phi; \text{cc\_none}) \end{array}}$$

$$\frac{LRT; \rho; \Phi \vdash v : \text{int}_= n' \quad m < n' \leq n \quad cc \neq \text{cc\_testm}(k)}{\begin{array}{c} LRT \vdash (\rho; (n, m, t); \Phi; cc)\ \{\text{inc\_alloc } v\} \\ (\rho; (n - n', -1, \text{boxed}); \Phi; cc) \end{array}}$$

Figure 4: Rules for Allocation Instructions

A tuple type $[\tau_0, \tau_1, \ldots, \tau_{n-1}]$ is represented in LTAL as $(\text{field } 0\ \tau_0) \cap (\text{field } 1\ \tau_1) \cap \ldots \cap (\text{field } (n-1)\ \tau_{n-1})$. If v has this type, then the word located at memory address v has type τ_0, at address $v + 4$ type τ_1, etc. (where 4 is the word size). When a field is initialized by a **store** instruction, one more conjunct (a **field** type) is added into the type of the partial record in the allocation environment.

After initialization, the *allocptr* is copied to a variable (with record type) by instruction v = **record**, and then the *allocptr* is adjusted to point to the next available memory word by instruction **inc_alloc**. After instruction **inc_alloc**, the condition codes set by **testm** are invalid because *allocptr* has been changed. So we reset the condition-code environment if it is **cc_testm**.

3.5 Coercions

A coercion only changes the static type of a value; it has no runtime effect. A coercion c defines a type transformation function f_c. If c is applied to value v of type τ, we get another value $c(v)$ of type $f_c(\tau)$. Type τ and $f_c(\tau)$ should be compatible, more accurately, it should be provable in the underlying model that τ is a subtype of $f_c(\tau)$. Coercions simplify type-checking by telling the checker, in effect, where to apply subtyping. However, this can significantly increase the size of the LTAL code.

We list some coercion rules in Figure 5. The coercion typing judgement $\rho; LRT \vdash_c \tau \overset{c}{\hookrightarrow} \tau'$ means that under kind environment ρ and maps LRT, coercion c changes τ to τ'.

Sometimes after applying a coercion we need to use the value both at its old type and its new type. This has been a difficulty in some previous TALs, which assign types to registers: they have to emit a **mov** instruction to handle this case.

We solve this problem by assigning types to variables, not to registers: A variable has only one type, but different variables can be assigned the same register. A move-with-

$$\frac{}{\rho; \mathit{LRT} \vdash_c \tau_1 \overset{\text{cinj1}_{[\tau_1 \cup \tau_2]}}{\hookrightarrow} \tau_1 \cup \tau_2}$$

$$\frac{}{\rho; \mathit{LRT} \vdash_c \tau_2 \overset{\text{cinj2}_{[\tau_1 \cup \tau_2]}}{\hookrightarrow} \tau_1 \cup \tau_2}$$

$$\frac{\tau' = \tau[\mu\alpha.\tau/\alpha]}{\rho; \mathit{LRT} \vdash_c \tau' \overset{\text{cfold}_{[\mu\alpha.\tau]}}{\hookrightarrow} \mu\alpha.\tau}$$

$$\frac{\rho; \mathit{LRT} \vdash_c \tau \overset{c_2}{\hookrightarrow} \tau' \quad \rho; \mathit{LRT} \vdash_c \tau' \overset{c_1}{\hookrightarrow} \tau''}{\rho; \mathit{LRT} \vdash_c \tau \overset{c_1 \circ c_2}{\hookrightarrow} \tau''}$$

$$\frac{\rho; \mathit{LRT} \vdash_c \tau_1 \overset{c_1}{\hookrightarrow} \tau_1' \quad \rho; \mathit{LRT} \vdash_c \tau_2 \overset{c_2}{\hookrightarrow} \tau_2'}{\rho; \mathit{LRT} \vdash_c \tau_1 \cup \tau_2 \overset{\text{cunion}_{(c_1,c_2)}}{\hookrightarrow} \tau_1' \cup \tau_2'}$$

Figure 5: Selected coercion rules

coercion creates a new variable (in the same register) without executing an instruction. In effect, the variable name in an LTAL instruction tells the checker which type to use.

This means that when we "kill" a variable (by assigning a new value to its underlying register), we must also kill all the other variables bound to that register. When adding a new type binding $v : \tau$, we examine each binding $v' : \tau'$ in Φ and remove it from Φ if v' is assigned the same register as v, which means v' should be no longer live. We use $(\Phi \backslash v), v : \tau$ to represent this operation; it can be seen in premise (9) of the big rule in Section 3.2. When there is no ambiguity, it is abbreviated to $\Phi, v : \tau$.

On the other hand, a move-with-coercions such as $v = c(v')$ does not require the application of the $\backslash v$ operator; other aliases of v continue to be active.

3.6 User-defined Datatypes

LTAL's low-level type constructors provide support for various data representations, and extracting and checking tags. The type-checker can check the connection between a sum value and its tag, and refine the type of sum values after tag-checking. We provide flexibility for the compiler writer to choose her preferred style of datatype representation; the representations we describe in this section are not new, but the point is that we can type each aspect of their construction and deconstruction.

For simplicity, we use the notation $[\tau_0, \tau_1, \ldots, \tau_{n-1}]$ for tuple types and use the following two type macros:

- Type $\mathbf{range}(n_1, n_2)$ for type $(\text{int}_{\geq}\, n_1) \cap (\text{int}_{<}\, n_2)$. A sum type is often represented as $\mathbf{range}(0, n) \cup t$. The number n indicates the number of constant constructors, which are represented as integer $0, 1, \ldots, n-1$. Type t is the union of types for the boxed constructors.

- Type $\mathbf{hastag}(\tau_{\text{tag}}, \tau)$ for $(\text{field } 0\ \tau_{\text{tag}}) \cap \tau$. It means that the tag of a sum value has type τ_{tag}, and the sum value is of type τ.

The compiler can choose from different data representations for user-defined datatypes such as *intlist:*

$$\text{datatype } \mathit{intlist} = \mathit{nil} \ \mid \mathit{cons} \text{ of int} * \mathit{intlist}$$

(1) The most straightforward representation is to tag each constructor with a small integer: *nil* is tagged 0, and *cons* tagged 1.

$$\mathit{intlist}_1 = \mu\alpha.([\text{int}_= 0] \cup [\text{int}_= 1, [\text{int}, \alpha]])$$

(2) We assume that small integers can be distinguished from pointers, thus constant data constructors can be represented as small integers: *nil* is represented as integer 0; *cons* is a boxed record with tag 0.

$$\mathit{intlist}_2 = \mu\alpha.(\text{range}(0,1) \cup [\text{int}_= 0, [\text{int}, \alpha]])$$

(3) A datatype with only one value-carrying constructor can be optimized further. If the value-carrying constructor carries an always-boxed value, it need not be tagged. Since *cons* carries a tuple that is always boxed, its tag can be removed.

$$\mathit{intlist}_3 = \mu\alpha.(\text{range}(0,1) \cup [\text{int}, \alpha])$$

Creating Sum Values. We create an empty list of $\mathit{intlist}_1$ by building a 1-element record $v_0 = [0]$, then coercing it to type $\mathit{intlist}_1$:

LTAL	Sparc
$v_0 : [\text{int}_= 0]$ $v_1 = \text{cinj1 } ([\text{int}_= 0] \cup [\text{int}_= 1, [\text{int}, \mathit{intlist}_1]])(v_0)$ $v_2 = \text{cfold}[\mathit{intlist}_1](v_1)$	

The only difference between v_0, v_1 and v_2 is coercions. They are assigned the same register, so no Sparc instruction is emitted for the above LTAL instructions.

By inserting coercions, the type-checker can easily tell that value v_0 can be coerced to be of type $\mathit{intlist}_1$. It simply checks if the type of v_0 is the first part of union type $[\text{int}_= 0] \cup [\text{int}_= 1, [\text{int}, \mathit{intlist}_1]]$ (by the rule of coercion **cinj1**), and if the type of v_1 is exactly the same as $\mathit{intlist}_1$ with type variable α replaced with $\mathit{intlist}_1$ (coercion **cfold**).

The following two LTAL instructions create an empty list of $\mathit{intlist}_3$ by coercing integer 0 to be of type $\mathit{intlist}_3$.

$v_1 = \text{crange}[0,1](0)$ $v_2 = \text{cinj1 } (\text{range}(0,1) \cup [\text{int}, \mathit{intlist}_3])(v_1)$ $v_3 = \text{cfold}[\mathit{intlist}_3](v_2)$	**mov** $0, d_1$

Coercion $\mathbf{crange}[n_1, n_2]$ changes a value of type $\text{int}_= n$ to type $\text{range}(n_1, n_2)$ if $n_1 \leq n < n_2$. In the first instruction the type-checker only needs to check if $0 \leq 0 < 1$ holds.

Eliminating Sum Values. Consider what happens when doing case discrimination on a boxed-tag style of sum type representation, such as is used when there are multiple value-carrying constructors. Given a value x, one fetches the tag into a variable y, then does a conditional branch on y; at this point, the difficulty is in relating the outcome of the conditional branch to the refined type of x. One solution is to use a "macro" TAL instruction to code for the load+compare+branch; we wanted to avoid all such macro instructions. We use type quantification and singleton types to keep track of the implicit dataflow.

User-defined datatype

$$\text{datatype } t = \mathbf{A} \mid \mathbf{B} \mid \mathbf{C} \text{ of int} \mid \mathbf{D} \text{ of int} * t$$

can be represented in LTAL as:

$$\tau = \mu\alpha.(\text{range}(0,2) \cup [\text{int}_= 0, \text{int}] \cup [\text{int}_= 1, \text{int}, \alpha]).$$

"Switching" on sum values in source program

$$\begin{array}{lll} \text{case}(v:t) & \text{of} & \mathbf{A} \Rightarrow e_A \\ & | & \mathbf{B} \Rightarrow e_B \\ & | & \mathbf{C}(x) \Rightarrow e_C \\ & | & \mathbf{D}(x,y) \Rightarrow e_D \end{array}$$

is translated to the following LTAL and Sparc instruction sequence (Variables $v_0, v, v_0', v_1, v_2, v_3, v_3', v_3''$ are all assigned register d, and variable t is assigned d_t):

LTAL	Sparc
$v_0 = \text{cunfold}(v)$	
$(\beta, v_0') = \text{cmpcci}(v_0, 256)$	**subcc** $d, 256$
$\text{ifr}(\geq)\{v_0'\}$ then (v_1, l_{CD})	
else (v_2, l_{AB})	**bge** l_{CD}
$l_{AB} : \ldots$	$l_{AB} : \ldots$
$l_{CD} :$	$l_{CD} :$
$(\alpha_1, v_3) = open(v_1)$	
$t = \text{load}(v_3, 0)$	**ld** $[d], d_t$
$\text{cmpcc}(t, 0)$	**subcc** $d_t, 0, \%g0$
$\text{iftag}(=)\{v_3\}$ then (v_3', l_C)	
else (v_3'', l_D)	**be** l_C
$l_D : \ldots$	$l_D : \ldots$
$l_C : \ldots$	$l_C : \ldots$

We need to generate code that tests v to decide which branch to take. Each test and each branch should be an explicit LTAL instruction. From our assumption that no pointers point to the first 256 words in the memory, if v is a small integer (less than 256), then it is either **A** or **B**, otherwise it is **C** or **D**. Instruction **cmpcci** performs this test and sets condition codes. Instruction **ifr** examines the condition codes and rebinds two fresh variables v_1 and v_2 with refined types for boxed and unboxed cases respectively. Variable v_1 has type $\exists\alpha.\mathbf{hastag}(\alpha, [\text{int}_= 0, \text{int}] \cup [\text{int}_= 1, \text{int}, \tau])$, which means it is tagged (we do not know the tag yet). Variable v_2 has type $\mathbf{range}(0, 2)$, which means it is either 0 or 1. Both v_1 and v_2 are forced to be assigned the same register as v_0, so no machine instruction is needed to move v_0 to v_1 or v_2.

In the unboxed case, we further test if v_2 is 0 or 1, which is easy. In the boxed case, we need to test the tag of v_1. Variable v_1 hides the type of its tag by existential types. We first open v_1 to v_3 and bind a brand new type variable α_1. Again, no Sparc instruction is needed because v_1 and v_3 are assigned the same register. Variable v_3 has type $\mathbf{hastag}(\alpha_1, [\text{int}_= 0, \text{int}] \cup [\text{int}_= 1, \text{int}, \tau])$. Instruction **load** extracts the tag t and gives it type α_1. Then **cmpcc** checks if tag t is 0 and set condition-code environment to be $\mathbf{cc\_cmp}(\alpha_1, \bar{0})$. Instruction **iftag** checks condition codes set by **cmpcc**, rebinds two new variables v_3' and v_3'' as aliases of v_3 and does conditional branch. The type-checker checks in **iftag** instruction that: cc is $\mathbf{cc\_cmp}(\tau_0, \bar{0})$, v_3 is of type $\mathbf{hastag}(\tau_0', \tau)$, and $\tau_0 = \tau_0'$; and it refines the types of v_3' and v_3'' to $[\text{int}_= 0, \text{int}]$ and $[\text{int}_= 1, \text{int}, \tau]$, respectively. This refinement rules out disjuncts by the result of comparing tags with integers. All these rules will be explained in detail in an upcomping thesis [8]. A constraint solver as in DTAL [23] is overkill for our purpose.

The connection between a tagged value and its tag is established by existential types, since every time we open a variable of type $\exists\alpha.\text{hastag}(\alpha, \tau)$ and assign it to some variable v, we get a fresh type variable α', and only v's type contains the new type variable α' in the first conjunct (field 0 α'), and only by instruction $\mathbf{load}(v, 0)$ can we get a variable of type α'.

For simplicity we use linear search here. LTAL also permits binary search; to do an indexed jump we would need to extend LTAL, but our underlying semantic model will permit this in a modular way.

3.7 Don't trust the linker!

To avoid the need to reason about possible bugs in the link-loader, we arrange that each compilation unit needs no link-editing, and links to others using closures, in the style of SML/NJ [7, §3]. We must avoid the need for a linker to do relocation. Our safety policy says, "a program is safe if, no matter where we load it in memory, it will never access an illegal address or execute an illegal instruction" [2].

Position-independent code must use relative addresses instead of absolute ones. The problem arises when we move a label into a register or store it in memory, to make a function-pointer or a closure. The value of the label depends on where the code is loaded.

We adopt the solution that SML/NJ uses, but we show how to type-check it. Each function takes a *base* parameter, which is the start address of its own machine code in the memory. We keep the *base* address of the current function in a register, and calculate the addresses of labels as offsets from *base*. When a function f is called, the address f is passed as its own *base* argument.

In the body of a function f, moving a label g to variable v is implemented as $v = \mathbf{addradd}(base, g{-}f)$, where $g{-}f$ is a constant computed by the compiler. Instruction **addradd** is translated to Sparc *add* instruction, and used only for address arithmetic.

$$\frac{\begin{array}{c} \Phi' = \Phi, v : \text{addr}(g) \\ LRT; \rho; \Phi \vdash v_1 : \text{addr}(f) \\ LRT; \rho; \Phi \vdash v_2 : \text{diff}(g, f) \end{array}}{LRT \vdash (\rho; \hbar; \Phi; cc)\ \{v = \text{addradd}(v_1, v_2)\}\ (\rho; \hbar; \Phi'; cc)}$$

To type-check position-independent code, we introduce type constructors **addr** and **diff**. The former gives a type to *base* and the latter types the difference between two labels. For example, in the above example $v = \mathbf{addradd}(base, g - f)$, variable *base* has type $\mathbf{addr}(f)$; the constant $g - f$, which is represented as a value $\mathbf{vdiff}(g, f)$, has type $\mathbf{diff}(g, f)$; and the typing rule for **addradd** will give type $\mathbf{addr}(g)$ to v.

When a function f is called in a compilation unit other than where it is defined, its label is (statically) unknown at the call site. Then the type of its *base* cannot be **addr**. We use existential types to hide the *base* type; the type of f becomes $\exists\beta.\mathbf{codeptr}[\vec{\alpha}](m, [base : \beta, \ldots])$. To make sure that f itself is passed to its *base* when f is called, we make f have type $\exists\beta.(\beta \cap \mathbf{codeptr}[\vec{\alpha}](m, [base : \beta, \ldots]))$.

As an important optimization, when a function is called only by direct jumps from known locations, it does not need its own *base* argument—it can use the *base* of one of its known callers. This avoids **addradd** instructions in local loops and branches.

4. MAKING AN UNTYPED BACK END PRESERVE TYPES

Our compiler is based on SML/NJ, whose back end uses MLRISC [11], a generic framework for compiler back ends. It can be customized to different source languages and retargeted to different architectures. MLRISC provides instruction selection, register allocation, and instruction scheduling modules parameterized by machine specifications. To generate a compiler back end, users customize these modules for their target machine. MLRISC has been used in many compiler projects including SML/NJ for years, and generates high-performance code.

We did not originally intend to take advantage of MLRISC, because MLRISC is totally untyped while we need type-preserving transformations. When we learned of the annotation mechanism [13], we tried using annotations to connect the typed representation we need and the untyped one MLRISC uses. The experiment turned out to be rewarding: reusing MLRISC this way is much less work than writing a back end from scratch, and has the advantages that MLRISC provides, such as generating code with good performance and being retargetable.

4.1 Annotations

Annotations in MLRISC are like comments; in fact, they are emitted as comments in assembly code. They have no runtime effect. One can annotate cells (pseudo-registers that will be mapped to physical registers or memory words), instructions, code blocks, and compilation units. Each annotation describes a property of the construct it annotates, and a construct can have many annotations addressing different properties. MLRISC provides ways to create, append, extract and remove annotations.

By annotating cells with variables, instructions with LTAL instructions, code blocks with function signatures, compilation units with type definitions, we get a close correspondence between MLRISC code and LTAL programs.

Originally MLRISC developers added this mechanism to propagate type information to code optimization phases. Annotations have been used extensively in MLRISC to pass information without changing existing data structures. Data abstraction hides the representation of client annotations from MLRISC's register allocator and instruction selector.

However, we found that MLRISC did not take care to maintain annotations through every program transformation. For example, sometimes MLRISC removes annotations of instructions, or creates new constructs without annotations because it does not know what annotations to give them. So the main difficulty in using MLRISC is how to restore the missing annotations.

4.2 Basic Blocks

Part of the solution to missing annotations is to design LTAL to provide annotations when MLRISC rewrites instructions.

At first we used extended basic blocks in LTAL: instructions such as $\mathbf{if}(v)\ \mathbf{then}(l, \sigma)\ \mathbf{else}(l', \sigma')$ could appear in the middle of a function body. To avoid long jumps, MLRISC would create a new block for the fall-through case and change the "jump" block to be fall-through case. In the following example, the *neq* case has more than 2^{21} instructions, but the *eq* case does not. MLRISC simply switches the two cases, changing the code in the left column to the one in the right. Label l_3 is for illustration purpose. It does not exist before MLRISC switches branches.

$$\begin{array}{l|l} l_1 : \ldots & l_1 : \ldots \\ \quad \mathbf{be}\ l_2\ (\mathbf{else}\ l_3) & \quad \mathbf{bne}\ l_3\ (\mathbf{else}\ l_2) \\ (l_3 :)\mathit{neq}\ \text{case} & (l_2 :)\mathit{eq}\ \text{case} \\ l_2 : \mathit{eq}\ \text{case} & l_3 : \mathit{neq}\ \text{case} \end{array}$$

Newly created block l_3 needed to be annotationed with an LTAL function signature, which MLRISC could not provide since there was no LTAL function that corresponds to this new block. So we changed LTAL to make each basic block a function. Thus in the above example, when MLRISC moves blocks, the LTAL function signature for l_3 is already there.

The important lesson here is that a good TAL should serve not only as an interface between a compiler and a checker, but also as a useful intermediate language in the back-end phases of the compiler itself. By using basic blocks instead of extended basic blocks, LTAL becomes useful as such an intermediate language.

4.3 Hooks

Our approach needs tight connection between machine code and LTAL annotations. But MLRISC sometimes breaks annotations. This causes problems. For example, MLRISC transformed instruction $d_1 = 3 + d_2$ to $d_1 = d_2 + 3$ (thus one Sparc instruction suffices), and annotated the new instruction with the annotation of the old one. The LTAL annotation of $d_1 = 3 + d_2$ would be like $v_1 = 3 + v_2$ where v_1 and v_2 are assigned register d_1 and d_2 respectively. This annotation is not valid for $d_1 = d_2 + 3$. The checker will try to map 3 to d_2 and v_2 to 3, and fail.

This problem results from the fact that MLRISC does not know the meaning of annotations or the connection between annotations and code, thus could not preserve them. Yet MLRISC should not understand annotations because different users give different annotations. Our solution is that MLRISC users provide hooks (functions) that manipulate annotations, and MLRISC calls those hooks when it transforms the code. The commuting transformation (shown above) will call a function of type $\mathit{annotation} \rightarrow \mathit{annotation}$ to restore annotations before it exchanges the two operands. This function takes the old instruction's annotation and rewrites it to fit the new instruction.

Another case in which MLRISC breaks LTAL annotations is when it splits an instruction into several ones. For example, pseudo-instruction $d = 4097$ is split into two instructions, $d = \mathbf{sethi}\ 4$ and $d = d\ \mathbf{or}\ 1$, where the first instruction loads the high 22 bits of constant 4097 to the high 22 bits of register d, and the second instruction loads the low 10 bits. In our modified back end, MLRISC calls a function to get the annotations for the two new instructions.

5. MEASUREMENTS

The LTAL calculus is a large engineering artifact, just like the compiler that produces it and the Sparc machine that consumes it. It comprises (at the current state of implementation) approximately 1200 operators and rules, including 196 machine-language Sparc instruction constructors (many of which are not used by the compiler and could be deleted from our checker), 263 Sparc instruction decoding rules, 30 coercion operators and 49 coercion rules, 48 explicit-substitution operators and reduction rules, 41 types and constructors for such things as label-maps and register-maps, 27 type operators (union, intersection, field, etc.), 69 rules for type refinement, 98 rules for wellformedness of types, 73 operators and rules for local environment management, 44 operators and rules for static arithmetic calculations, 38 rules for parsing the label, register, and type maps, 50 structural matching heuristics for type expressions, 51 LTAL instruction constructors, and 53 typing rules for instructions.

A typical large rule, such as the one shown in Section 3.2, is quantified over a dozen variables and has a dozen premises. In all, the current LTAL type checker is 3900 lines of (non-blank, non-comment) Prolog-like source code. The machine-checked proof of the soundness of all the LTAL rules (which is nearing completion) is over 98,000 lines of higher-order logic as represented in the Twelf system. The axioms comprise 1850 lines, almost all of which is the specification of the Sparc instruction set.

The compiler from core ML to LTAL+machine code is written in ML; its size (including blank lines and comments) is 50k lines of the Standard ML of New Jersey (110.35) front end (unmodified); 1.8k lines of code copied and modified from the implementation of the SML/NJ interactive top-level loop; 2.7k lines to translate FLINT to NFLINT; 7.8k lines to translate NFLINT to LTAL; 1.2k lines to interface of MLRISC; and approximately 50k lines of the MLRISC system[3] itself, of which 400 lines are new or modified to support our more-general annotation interface.

5.1 Performance

We compared our performance[4] to that of SML/NJ 110.35 on two small benchmarks: Life (adapted from the Standard ML benchmark suite) and RedBlack, which uses balanced trees to do queries on integer sets.

Benchmark	redblack	life
SML/NJ Compile time	0.300	0.490 sec.
SML/NJ Run time	0.013	0.262
FPCC Compile time	0.955	2.998
Safety check time	0.183	0.432
FPCC Run time	0.014	0.407
FPCC/SMLNJ slowdown	1.036	1.555
Sparc instrs.	870	1816
LTAL tokens	34278	57670
Coercion tokens	17%	23%

Our compile time is not competitive (2.998 seconds to compile Life compared to 0.49 seconds for the production release of SML/NJ); we have not engineered our compiler algorithms as necessary for a production compiler. Run time is almost as good as SML/NJ. We do not garbage collect; SML/NJ spends 0.02% of its time garbage-collecting on these benchmarks. SML/NJ's better performance is probably because it has more sophisticated liveness-based closure conversion and fills branch-delay slots.

To measure *Safety check time,* we translate our lemmas into Prolog rules and time the execution in SICStus Prolog. As an alternative, we are building a minimal-size interpreter for syntax-directed lemmas; it is much simpler than Prolog because it doesn't require backtracking or full unification; we have yet to measure the checking speed in that interpreter.

Simple encodings should be able to represent LTAL in a few bits per token, so the LTAL expression should not be significantly bigger than the machine-language program. Eliminating the coercions—thus requiring some backtracking in the checker—could save about 20% in LTAL size. The builders of SpecialJ [9] and TALx86 [15] have devoted substantial effort to reducing proof size—not just removing coercions but getting the checker to reconstruct other data as well. Clearly, there is some engineering to be done in this respect, although we would not want to complicate any part of the checker that is in the trusted base.

6. RELATED WORK AND CONCLUSION

Morrisett *et al.*'s TAL [17] demonstrated the idea of typed assembly language, but was too limited for practical programming languages. Extensions of this work supported stack allocation [16] and implemented a more realistic calculus (TALx86) [15] for compiling a safe C-like language to Intel IA32 assembly language. Xi and Harper's DTAL [23] added a restricted form of dependent types to TAL to support array bound check elimination and datatype tag discrimination. These implementations have soundess proved by hand about abstractions of subsets of the system that is actually implemented; the proofs cannot be machine-checked. These TALs each have a macroinstruction "malloc" for heap allocation (and TALx86 has another macro "btagi" which tests tags and branches).

Hamid *et al.* proposed a syntactic approach to build machine-checkable foundational proofs. They designed Featherweight Typed Assembly Language (FTAL) [12], mapped each valid machine state to a well-typed FTAL program, and related transition of machine states to evaluation of FTAL programs by a machine-checked syntactic metatheorem. It is not clear whether syntactic metatheorems scale very well, or can be made as modular as our semantic-modelling approach; FTAL is too featherweight to tell. Crary [10] has built a more substantial TALT, with a machine-checked syntactic metatheorem proving progress and preservation; he is now working on the machine-checked metatheorem relating his typed calculus to the "bare machine" untyped step relation.

We have designed a syntactic low-level typed assembly language with a semantic model that backs up its soundness with a machine-checkable proof. The semantic modelling technique makes LTAL easily and safely extensible. It has a rich set of expressive constructors, yet its type-checking is decidable and simple. We have implemented a prototype compiler that transforms core ML programs to Sparc code annotated with LTAL programs. In our compiler an untyped back end preserves types by annotations and hooks.

7. ACKNOWLEDGEMENTS

We would like to thank Noam Zeilberger for his improvements on the implementation of calling conventions and run-time exception controls. Also, we are grateful to many reviewers who gave us very constructive comments on earlier versions of this paper.

8. REFERENCES

[1] Amal J. Ahmed, Andrew W. Appel, and Roberto Virga. A stratified semantics of general references embeddable in higher-order logic. In *17th Annual*

[3] The MLRISC software has several other analyses, optimizations, and target machine specifications that we did not use and that we don't count here.

[4] Measured on Sun UltraSparc E250, 400 MHz.

IEEE Symposium on Logic in Computer Science (LICS 2002), pages 75–86. IEEE, June 2001.

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

[3] 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.

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

[5] Andrew W. Appel, Neophytos Michael, Aaron Stump, and Roberto Virga. A trustworthy proof checker. In Iliano Cervesato, editor, *Foundations of Computer Security workshop*, pages 37–48. DIKU, July 2002. diku.dk/publikationer/tekniske.rapporter/2002/02-12.pdf.

[6] Andrew W. Appel and Daniel C. Wang. JVM TCB: Measurements of the trusted computing base of Java virtual machines. Technical Report CS-TR-647-02, Princeton University, April 2002.

[7] Matthias Blume and Andrew W. Appel. Lambda-splitting: A higher-order approach to cross-module optimizations. In *Proc. ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*, pages 112–124, New York, June 1997. ACM Press.

[8] Juan Chen. *A Sound Typed Assembly Language for Real Languages on Real Machines*. PhD thesis, Princeton University, 2003. In preparation.

[9] Christopher Colby, Peter Lee, George C. Necula, Fred Blau, Ken Cline, and Mark Plesko. A certifying compiler for Java. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '00)*. ACM Press, June 2000.

[10] Karl Crary. Toward a foundational typed assembly language. In *POPL '03: The 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, page (to appear). ACM Press, January 2003.

[11] Lal George. MLRISC: Customizable and reusable code generators. Technical report, Bell Laboratories, May 1997.

[12] Nadeem Hamid, Zhong Shao, Valery Trifonov, Stefan Monnier, and Zhaozhong Ni. A syntactic approach to foundational proof-carrying code. In *Proc. 17th Annual IEEE Symposium on Logic in Computer Science (LICS'02)*, pages 89–100, July 2002.

[13] Allen Leung and Lal George. *MLRISC Annotations*. http://cm.bell-labs.com/cm/cs/what/smlnj/compiler-notes/annotations.ps.

[14] Neophytos G. Michael and Andrew W. Appel. Machine instruction syntax and semantics in higher-order logic. In *CADE-17: 17th International Conference on Automated Deduction*, pages 7–24. Springer-Verlag, June 2000. LNAI 1831.

[15] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. In *Second ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25–35, Atlanta, GA, 1999. INRIA Technical Report 0288, March 1999.

[16] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. *J. Functional Programming*, 12(1):43–88, January 2002.

[17] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999.

[18] 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.

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

[20] Kedar N. Swadi and Andrew W. Appel. Typed machine language and its semantics. www.cs.princeton.edu/~appel/papers, July 2001.

[21] Gang Tan, Kedar Swadi, Dinghao Wu, and Andrew W. Appel. Construction of a semantic model for a typed assembly language. January 2003.

[22] Dinghao Wu, Andrew W. Appel, and Aaron Stump. Foundational proof checkers with small witnesses. March 2003.

[23] Hongwei Xi and Robert Harper. A dependently typed assembly language. In *2001 ACM SIGPLAN International Conference on Functional Programming*, pages 169–180. ACM Press, September 2001.

APPENDIX

A. COMPARISON OF TAL SYSTEMS

Figure 1 makes sweeping claims about many competing proof-carrying-code (PCC) and typed-assembly-language (TAL) systems. Here we explain the basis for these claims, based on our understanding of the various cited works. Boxed numerals $\boxed{i}$ reference the columns of the table.

SpecialJ [9] is a PCC compiler from Java byte code $\boxed{1}$ to x86 machine language $\boxed{2}$. It includes a verification condition (VC) generator that scans the Java .class files (that contain compiled information about the data layout and formal parameters of methods) and the machine code, extracting a formula that is supposed to imply the correctness of the machine code. The closest that SpecialJ comes to a foundational specification $\boxed{3}$ is a reference [9, §4.2] to a proof about a different VC generator in an earlier system. Assumptions about instruction encodings and about the safety policy are implicit in the VC generator (a C program tens of thousands of lines long), and there is no machine-checked soundness proof $\boxed{4}\boxed{5}$.

SpecialJ mostly treats instructions atomically $\boxed{6}$: for example, comparisons that set condition codes can be separated from branches that depend on them. However, memory allocation is done by a call to the runtime system. Data representations are fixed by the surrounding Java runtime $\boxed{7}$. The VC generator and prover can accommodate dataflow-based optimizations $\boxed{8}$. Position-independent code does not seem to have been a design goal $\boxed{9}$. Since the VC is entirely a post-compilation pass, support for basic blocks as a tool for back-end optimization is beside the point $\boxed{10}$. There is no syntax-directed calculus at the interface between the SpecialJ compiler and the prover/checker $\boxed{11}$.

TALx86 [17, 16, 15] is the typed assemply language of the Popcorn compiler, which compiles a superset of a subset of C $\boxed{1}$ to Pentium code $\boxed{2}$. Several papers about different subsets of TALx86 each specify overlapping subsets of the safety property, but there is no formal specification of the safety achieved by the actual implementation $\boxed{3}$. There is no machine-checked soundness proof $\boxed{4}\boxed{5}$. TALx86 has nonatomic instruction sequences for compare-and-branch and for memory allocation $\boxed{6}$; later versions of TALx86 can separate the compare from the branch. Disjoint-sum datatype tag-checking is done in separable atomic instructions in the implementation $\boxed{6}$, and with some flexibility in choosing representations $\boxed{7}$. There is limited support for dataflow-based reasoning on integer variables $\boxed{8}$ or for position-independent code $\boxed{9}$. "Blocks" in TALx86 are extended basic blocks, not basic blocks—this would hamper optimizations that reorder blocks—but in the implementation there is a pseudoinstruction to handle "fall-through" that could mitigate this problem $\boxed{10}$. TALx86 (as implemented) is approximately syntax-directed, but omits many coercions to save space; there is no formal specification of what a compiler-writer must do to guarantee that proofs will be checkable $\boxed{11}$.

DTAL [23] is a Dependently Typed Assembly Language. DTAL has been demonstrated with a toy source language $\boxed{1}$ and no translation to any particular target machine $\boxed{2}$. Except for the lack of correspondence to a real machine, there is a formal specification $\boxed{3}$ of the safety property; there is no machine-checked proof $\boxed{4}\boxed{5}$. Atomicity of instructions is impossible to judge in the absence of a translation to a real machine $\boxed{6}$. There is only one tagged 2-way sum datatype $\boxed{7}$. The dependent types permit reasoning about dataflow analysis on user program variables and other quantities $\boxed{8}$. There is no position-independent code $\boxed{9}$. Extended basic blocks are used instead of basic blocks $\boxed{10}$. Type-checking is not syntax-directed, but requires a more sophisticated decision procedure $\boxed{11}$.

Featherweight Typed Assembly Language, FTAL [12], demonstrates an approach to foundational proof-carrying code using machine-checked soundness proofs based on syntactic operational semantics. There's no compiler for a source language nor translation to any target machine $\boxed{1}\boxed{2}$, but there is a type-independent specification of the safety property, encoded in the Stratified Calculus of Inductive Constructions (CiC), with a machine-checked proof in the Coq system $\boxed{4}$. However, the safety specification does not include instruction encodings or other "real-machine" issues $\boxed{3}$. Existing checkers for CiC are at least an order of magnitude larger than our minimal LF checker $\boxed{5}$. Atomicity of instructions is impossible to judge in the absence of a translation to a real machine $\boxed{6}$. There is not enough support for data structures to judge whether the compiler has flexibility to choose data representations $\boxed{7}$. Although the possibility is mentioned of adding existentials and singletons to support dataflow-based reasoning, this has not been done $\boxed{8}$. There is no position-independent code $\boxed{9}$. Extended basic blocks are used instead of basic blocks $\boxed{10}$. Typechecking is syntax-directed $\boxed{11}$.

TALT [10] is a "Foundational" typed assembly language, meaning that, like ours, it is intended to support machine-checked proofs from first principles. It has not yet been demonstrated (as far as we can tell) in a compiler for any source language $\boxed{1}$, but its application to Pentium assembly language is quite explicit $\boxed{2}$. There is a near-foundational specification of safety, but it does not model instruction decoding $\boxed{3}$. Soundness of TALT type-checking will be proved by machine-checked metatheorems of progress, type preservation, and a simulation relation between the typed calculus and the untyped machine calculus; machine-checked progress and preservation proofs have been built, but work on the simulation proof is still ongoing $\boxed{4}$. However, the implementation of Twelf's metatheorem checker (in which these theorems are written) is two orders of magnitude larger than our minimal LF checker $\boxed{5}$. TALT has nonatomic compare-branch and memory-allocation instructions $\boxed{6}$. Its use of explicit unions and tags for representing disjoint sums appears to allow a compiler flexibility in choosing representations $\boxed{7}$. Its use of singleton types (like ours) gives the power to reason about dataflow $\boxed{8}$. TALT does not support position-independent code $\boxed{9}$, but it does support reasoning about relative addressing that is almost enough for that purpose. TALT has no notion of blocks (neither extended nor basic) $\boxed{10}$. TALT has no syntax-directed type-checking algorithm (type-checking is not even decidable), so typing derivations must be sent from compiler to checker $\boxed{11}$.

The LTAL system described in this paper achieves almost all of our goals. However, we don't yet compile all of core ML $\boxed{1}$; we expect to do all of core ML in a few months. Our soundness proofs based on the semantic model are constructed by hand and checked by machine, but the machine-checked proofs are not finished $\boxed{4}$; we expect to finish in a few months. All of our instructions are atomic except for the delayed branch on the Sparc processor, which we bundle with a nop instruction in the delay slot $\boxed{6}$.