Toward a Foundational Typed Assembly Language

Karl Crary

Carnegie Mellon University

Abstract

We present the design of a typed assembly language called TALT that supports heterogeneous tuples, disjoint sums, and a general account of addressing modes. TALT also implements the von Neumann model in which programs are stored in memory, and supports relative addressing. Type safety for execution and for garbage collection are shown by machine-checkable proofs. TALT is the first formalized typed assembly language to provide any of these features.

Categories and Subject Descriptors

D.3 [**Programming Languages**]: Language Constructs and Features;
F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs.

General Terms

Languages, Security, Verification

Keywords

Typed assembly language, proof-carrying code

1 Introduction

The proliferation of low-cost computing hardware and the ubiquity of the Internet has created a situation where a huge amount of computing power is both idle and—in principle—accessible to developers. The goal of exploiting these idle computational resources has existed for years, and, beginning with SETI@Home [28] in 1997, a handful of projects have successfully made profitable use of idle computational resources on the Internet. More recently, this paradigm, now called *grid computing,* has elicited serious interest among academics [5, 16] and in industry as a general means of conducting low-cost supercomputing.

This material is based on work supported in part by NSF grants CCR-9984812 and CCR-0121633. Any opinions, findings, and conclusions or recommendations in this publication are those of the authors and do not reflect the views of this agency.

Despite the increasing interest in grid computing, a remaining obstacle to its growth is the (understandable) reluctance of computer owners to download and execute software from developers they do not know or trust, and may not have even heard of. This has limited the practical use of grid computing to the small number of potential users that have been able to obtain the trust of thousands of computer owners they do not know.

The ConCert project at CMU [9] is seeking to overcome this obstacle by developing a system for trustless dissemination of software. In the ConCert framework, a machine owner installs a "steward" program that ensures the safety of any downloaded software. When a new grid application is obtained for execution (other parts of the ConCert framework determine when and how this takes place), that application is expressed in the form of *certified code,* in which the executable code is accompanied by a certificate proving that the code is safe. The steward then verifies that the certificate is valid before permitting the code to be executed.

The form of certified code used in the prototype ConCert system [7] is Typed Assembly Language (TAL) [20, 19], as implemented in the TALx86 system [18], which was developed primarily at Cornell. TALx86 specializes TAL to the Intel IA-32 architecture[1] [15], and enhances it with a number of constructs not supported in the theoretical system.

Although certified code eliminates the need to trust the developers of grid applications, there remains the need to trust the soundness of the steward. For TALx86 (other production-quality certified code systems such as the Java VM [17] or Proof-Carrying Code (PCC) [8] are similar), the need for trust stems from four issues:

1. TALx86's safety is justified informally (by analogy to TAL and other work), not by any rigorous proof.

2. The safety of TAL (and TALx86 by analogy) is given in terms of an abstract machine. Although this abstract machine is very low-level for a type system, in a few ways it is still somewhat high-level compared to the concrete architecture, and obscures some important issues thereof.

3. The safety proofs that exist are given in written form, and consequently checking their veracity is error-prone and can be done only by experts.

4. One must trust that the TALx86 type checking software correctly implements its type system.

To a certain extent, the need to trust the steward under these conditions is not a serious obstacle. Machine owners routinely trust the safety of a great variety of applications

[1]Popularly known as the "x86" architecture.

and have far less formal basis for doing so. Moreover, a certain amount of trust is required in any case, since very few participants will personally inspect any of the components of the system before using it. Nevertheless, minimizing the system's trusted computing base is always a laudable goal.

For grid computing, however, the desire to minimize trust in the steward goes beyond such general considerations. The key issue is *extensibility.* Our aim is to enable the establishment of a decentralized grid computing fabric available for a wide array of participants. For this purpose, it is implausible to imagine that a single type system will suffice for all potential grid applications. Furthermore, it is also implausible (and indeed undesirable) that a single trusted agent would check the soundness of all proposed new type systems. Thus, it is necessary that the basic steward can be safely extended with new type systems from untrusted sources.

The natural way to achieve this is using *foundational certified code,* in which the entire safety argument, down to a safety policy expressed in terms of the concrete architecture, is given in machine-checkable form. In its incarnation as foundational PCC [1], a binary is accompanied by an complete proof of its safety. In our proposed foundational TAL, a program's safety argument is broken into two parts: The first portion, a self-certified type system, is the specification of a type system and a machine-checkable proof that it satisfies an established safety policy. This portion would generally be independent of any particular application. The second portion is the application's typing derivation in a certified type system.

In this model, any grid application developer could devise his or her own type system sufficient to certify his or her applications. Whenever a steward encountered a certified application referring to an unknown type system, the steward would automatically download the type system, verify its safety proof, and thereafter accept applications with valid typing derivations in that type system.

This model addresses each of the four issues demanding trust of the new type system. The first three issues are clearly dealt with by the use of complete, machine-checkable proofs. By packaging applications with typing derivations, as opposed to type annotations, the fourth issue is also addressed: a single proof checker can check derivations for any type system, and no new checker need ever be employed.

The aim of this work is to implement a first example of foundational typed assembly language for the Intel IA-32 architecture, and also to build a flexible foundation on which a variety of other type systems may be built. Accordingly, we have broken the development of our system into two stages (in the model of Hamid, *et al.'s* development of "Featherweight TAL" [13]):

- The first stage develops a general typed assembly language that is not (very) specific to any architecture. This general language is given an operational semantics, and its *abstract* safety is established by machine-checkable proofs of type preservation and progress. The safety of garbage collection is also established at this level (Section 4).
- The second stage shows that the abstract operational semantics maps correctly onto the concrete architecture. The general type system is designed to account for all those issues that pertain to the type system (including nearly all central issues), so this second stage is a type-free simulation argument.

This topic of this paper is the first stage of this effort. The second is currently underway.

1.1 TALT

This paper presents a new typed assembly language called TALT ("TAL Two"). The aims of TALT are threefold: First, TALT is intended to provide sufficient expressive power to serve as the target language for certifying compilers for expressive, high-level languages (in particular, Standard ML and Popcorn (a safe dialect of C) [18]). Second, TALT's operational semantics is intended to account, to the greatest extent possible, for the central issues of execution on the actual hardware, so that the second stage of the foundational safety proof discussed above is a simple (albeit lengthy and tedious) simulation argument. Third, TALT is intended to be fully formalized and enjoy a machine-checkable type safety proof.

We begin our discussion with an overview of how TALT accomplishes each of these aims:

Expressiveness In order to provide the expressive power necessary to compile practical, high-level programming languages, TALT follows TALx86 [18] in adding support for heterogeneous tuples (*i.e.,* tuples in which not all fields have the same size), recursive types, disjoint sums, and arrays. (Due to space considerations, discussion of arrays is omitted in this paper, and appears instead in the companion technical report [10].) Unlike TALx86, however, TALT supports these constructs using a formalized type theory, rather than using specialized code in the type checker.

TALT provides all the expressive power of the original published TALx86, with the exception of TAL's initialization flags, which are replaced by another, slightly less general mechanism (Section 2.2.2). Most later enhancements of TALx86 are not supported in TALT, although many could be added without difficulty.

Completeness The operational semantics of TAL is specified in terms of a low-level abstract machine, including an explicit register file, memory, and (in stack-based TAL [19]) stack. This brings TAL nearly to as low a level as possible without committing to the details of an architecture. However, the TAL machine model did not include a program counter; instead, it included a stream of input instructions that served as a surrogate for the program counter. On any branch instruction, the machine would discard the current stream of instructions and copy in the instructions at the branch address. TALT, on the other hand, includes an explicit program counter.

A program counter must be part of the model to address a few important issues: On the concrete architecture, it is possible in principle to write into the code just ahead of the program counter. This is rarely permitted in practice, of course, but even so, since programs are in fact stored in memory, it is an essential part of a foundational safety proof to explicitly address the issue. The requisite argument is not difficult (TALT follows most standard practice and makes the code read-only), but the machine model must include an explicit program counter to make the argument at all.

An explicit program counter is also necessary to account for the notion of relative addressing, which is particularly important on the IA-32 where most control transfers (including all conditional jumps) are relative. Finally, an explicit program counter, or, more precisely, the fact that each instruction is individually addressable, allows for a true `call` instruction in which the return address is taken from the program counter.

Our system includes a conservative garbage collector [4] as part of its trusted computing base. This makes the garbage collector effectively a part of the architecture. Accordingly, there is no need to prove that the collector itself is safe, but it is necessary to show that programs adhere to the invariants that the collector requires [3, 2]. The TALT type system accounts for these invariants, and using those invariants we prove that type safety is not disrupted by garbage collection (Section 4).

In the interest of simplicity, the version of TALT in this paper provides only a few representative arithmetic operations (add, subtract, and compare). We also omit any discussion of floating point; this is because the IA-32's implementation of floating point (using register stacks) is very idiosyncratic. We foresee no fundamental difficulties in extending TALT in either direction.

TALT also follows stack-based TAL in viewing the stack as a separate unbounded resource, rather than as just a designated area of memory. This dramatically simplifies the type system [19], but requires the second stage of the foundational safety proof to show that the TALT stack can be mapped onto the concrete stack. The principal issue in doing so is stack overflow, which we address in Section 6.

Machine-Checkable Proofs TALT is formalized in LF [14] as mechanized in the Twelf system [24, 25]. In accordance with the usual LF methodology, TALT typing judgements correspond to LF types, typing rules correspond to LF constants, and TALT typing derivations correspond to LF terms. The validity of a typing derivation can then be verified by type checking.

Of central importance to this effort are the type safety meta-theorems (progress, type preservation, and GC safety). These are expressed in Twelf in relational form as logic programs [22, 23]. In support of this, Twelf provides a totality checker [27, 25] that ensures that the relations represent total functions, and are therefore valid meta-proofs. This is discussed in detail in Section 5.

It is important to note that decidable checking of typability is *not* an aim for TALT. Unlike TAL and TALx86, TALT is a type assignment (or Curry-style) system; values contain none of the type annotations they carry in explicitly typed (or Church-style) systems such as TAL. This means that checking the typability of TALT programs requires type inference, and since TALT's type system is polymorphic, this means that typability checking is almost certainly undecidable [31].

Consequently, the safety certificate for a TALT program is an entire typing derivation, rather than—as it is in TALx86—a collection of type annotations from which one can reconstruct a typing derivation. However, this is largely just a matter of presentation; a TALT typing derivation contains little information that would not appear in type annotations, and we conjecture that what overhead remains can be eliminated using Necula and Rahul's technique [21].

$$
\begin{array}{lrcl}
\text{unit values} & u & ::= & b \mid \texttt{junk} \mid a{:}i \mid \iota^k{:}i \\
\text{values} & v & ::= & \langle u_1, \ldots, u_n \rangle \\
\\
\text{addresses} & a & ::= & \ell{+}n \\
\text{cond. codes} & cc & ::= & \{\texttt{cf} \mapsto \mathit{bit}, \texttt{zf} \mapsto \mathit{bit}\} \\
\\
\text{memories} & H & ::= & \{\ell_1 \mapsto v_1, \ldots, \ell_n \mapsto v_n\} \\
\text{register files} & R & ::= & \{\texttt{r1} \mapsto v_1, \ldots, \texttt{rN} \mapsto v_N, \texttt{fl} \mapsto cc\} \\
\text{states} & M & ::= & (H, R, a)
\end{array}
$$

Figure 1: Untyped TALT Syntax

This paper is organized as follows: We begin in Section 2 by presenting the basic TALT language, and in Section 3 we present extensions supporting disjoint sums and relative addressing. In Section 4 we give our formalization of GC safety. Section 5 discusses the pragmatics of our machine-checked proofs. Concluding remarks follow in Section 6.

2 Basic TALT

We begin our presentation of the basic TALT language by presenting its untyped fragment in four parts: First we present the structure of values, second we present the machine model, third we discuss the instruction set, and fourth we give its operational semantics. Following this, we present the type system. Throughout this section we will remark on the differences between TALT and TAL for the benefit of those readers familiar with the latter; readers unfamiliar with TAL may skip these remarks.

We adopt the following notational conventions: W stands for the size of the machine word (measured in bytes, 4 for the IA-32); i, j, k, m and n range over nonnegative integers; bit ranges over $\{0, 1\}$; b ranges over bytes (*i.e.,* integers modulo 256); and B ranges over words (*i.e.,* integers modulo 2^{8W}). As usual, we consider alpha-equivalent expressions to be identical. However, section identifiers (ℓ) and registers (r) will not be considered variables and will not alpha-vary; the only variables appearing in TALT will be type constructor variables (α).

2.1 Untyped TALT

Values TAL separated its collection of values into two syntactic classes, called "heap" and "word" values.[2] This syntactic distinction determined what could fit into a register. In TALT, we combine these into a single syntactic class of values that may vary in size, and use types rather than syntax to determine how they may be used.

The syntax of type-free TALT appears in Figure 1. Values are simply sequences of zero or more *unit values,* each of which is exactly one byte in size. We construct values from unit values to ensure that any byte (including bytes in the middle of an atomic object) may be addressed, and to make it easy to compute the size of any value. Note that it is not merely a cosmetic change to eliminate word values but add unit values; unlike TAL's word values, unit values are

[2]TAL also included a third class called "small" values. These were not truly values at all, and were renamed "operands" in later work [19], as they are called here.

never objects of interest by themselves and are used only to construct values.

By a convenient abuse of notation, we will also often write values in the form $\langle v_1, \ldots, v_n \rangle$, representing the appending of several values.

There are four forms of value that we may consider atomic: literal bytes b, junk values, addresses a (which are W bytes in length), and instruction encodings ι^k. The metavariable ι^k ranges over all k-byte instruction encodings. At this level of abstraction, it is not appropriate to specify the possible values for ι^k; instead, instead we assume the existence of a decoding function $[\![\iota^k]\!]$ that takes instruction encodings to instructions (ι), which we discuss a bit later. Note that $[\![-]\!]$ need not be injective, as a single instruction can have multiple encodings, nor need it be surjective.

Literals and junk values are one byte in size, and are represented directly as unit value constructs. However, addresses and (usually) instruction encodings have size greater than one byte. Thus, the latter two forms of unit value represents bytes taken from larger atomic values: $a{:}i$ represents the ith byte of the address a, and $\iota^k{:}i$ represents the ith byte of the instruction encoding ι^k. When i is out of the appropriate range, these constructs are considered junk values.

We will rarely need to deal with subcomponents of addresses and instructions, so for most purposes a and ι^k can be viewed as value forms, using the abbreviations:

$$\begin{array}{rcl} \overline{a} & \stackrel{\text{def}}{=} & \langle a{:}0, \ldots, a{:}(W{-}1) \rangle \\ \overline{\iota^k} & \stackrel{\text{def}}{=} & \langle \iota^k{:}0, \ldots, \iota^k{:}(k{-}1) \rangle \end{array}$$

We also write $\overline{B}$ for the encoding of B in bytes; on a little-endian architecture such as the IA-32, $\overline{B} = \langle b_{W-1}, \ldots, b_0 \rangle$ where B is $b_0 b_1 \ldots b_{W-1}$.

Machine Model Like TAL, the TALT machine model contains a memory and register file, but it replaces TAL's stream of input instructions with an explicit program counter. We also augment the state with a collection of condition codes (indicating the carry and zero flags here, but it would not be difficult to add more), which it is convenient to consider as part of the register file, contained in a flag (`fl`) register. The TALT register file has a fixed collection of N general-purpose registers, `r1` through `rN`.[3] (TAL, on the other hand, assumed an infinite supply of registers.)

The TALT memory is broken into *sections,* identified by section identifiers ℓ. A section represents an area of memory that is known to be contiguous. Distinct sections, however, appear in an unknown order in memory, possibly with intervening gaps. Thus, we view the TALT memory as a mapping of section identifiers to values (since values are simply sequences of bytes). Any byte in memory can be addressed (subject to the limitations of the garbage collector) by combining a section identifier ℓ with a numeric offset n into the section to produce an address $\ell{+}n$.

When we consider garbage collection in Section 4, it will be necessary for us to distinguish between the heap (which the collector manages) and the remainder of memory. We refer to these portions as the heap and code segments. In anticipation of this development, we make a few remarks now. We view the segment that a section resides in to be an intrinsic property of the section that may be determined from its identifier. This is formalized using two predicates; $\texttt{hseg}(\ell)$ holds if ℓ resides in the heap segment and conversely for `cseg`. The heap segment is traced by the collector and is read/write; the code segment is not traced and is read-only. Newly allocated sections always appear in the heap segment.

[3] On the IA-32, N is 7, since the stack pointer is treated specially (Section 2.3).

$$\begin{array}{llcl}
\text{registers} & r & ::= & \texttt{r1} \mid \cdots \mid \texttt{rN} \\
\text{operands} & o & ::= & \texttt{im}(v) \mid \texttt{rco}(r) \mid \texttt{mco}(m, o, n) \\
\text{destinations} & d & ::= & \texttt{rdest}(r) \mid \texttt{mdest}(m, o, n) \\
\text{conditions} & \kappa & ::= & \texttt{eq} \mid \texttt{neq} \mid \texttt{lt} \mid \texttt{lte} \mid \texttt{gt} \mid \texttt{gte} \\
 & & & \\
\text{instructions} & \iota & ::= & \texttt{add}\ d, o_1, o_2 \\
 & & \mid & \texttt{cmp}\ o_1, o_2 \\
 & & \mid & \texttt{jcc}\ \kappa, o \\
 & & \mid & \texttt{jmp}\ o \\
 & & \mid & \texttt{malloc}\ n, d \\
 & & \mid & \texttt{mov}\ d, o \\
 & & \mid & \texttt{sub}\ d, o_1, o_2 \\
 & & & \\
\text{code} & I & ::= & \epsilon \mid \iota; I
\end{array}$$

Figure 2: Instructions

The Instruction Set The basic TALT instruction set is given in Figure 2. With the exception of `malloc` (which is implemented by the run-time system, rather than the architecture), the instructions should be familiar to any programmer of the IA-32. Note that instructions are given in Intel-style (destination first) notation. Note also that the load and store instructions of TAL are omitted; they are replaced by the `mov` instruction used with appropriate addressing modes. Code sequences (I) are not used in the TALT operational semantics, but will arise in the type checking rules.

Our notation for operands and destinations is more novel. The operand $\texttt{im}(v)$ indicates an immediate operand, and the operand $\texttt{rco}(r)$ indicates that the operand is the contents of the register r. The contents of memory are obtained using the operand $\texttt{mco}(m, o, n)$, where o is a sub-operand providing an address; n indicates a fixed offset from that address at which to read, and m indicates the size of the operand to be read (typically W). Similarly, $\texttt{rdest}(r)$ indicates that the destination for the instruction is the register r, and $\texttt{mdest}(m, o, n)$ indicates that the destination (of size m) is in memory, with the address indicated by the operand o plus offset n.

In order to make TALT as elegant and as general as possible, no effort has been made to limit the available operands or operand/instruction combinations to those actually supported on the IA-32 (or any other) architecture. (For example, no real architecture permits arbitrary chaining of indirection through nesting of memory operands as TALT does; and few sizes of memory reads are permitted.) It is more elegant to include a variety of unsupported possibilities, and to note that these unsupported instructions are simply not in the range of the decoding function.

Operational Semantics The operational semantics are given in Figure 3. The main judgement is $M \mapsto M'$, indicating that the machine state M steps to the machine state M'.

$M \mapsto M'$ **when** $M = (H, R, a)$ $H(a) = \langle \overline{\iota^k}, \ldots \rangle$ and:		
if $[\![\iota^k]\!]$ is:	then M' is:	provided that:
$\texttt{add}\ d, o_1, o_2$	$(H', R'\{\texttt{fl} \mapsto cc\}, a + k)$	$M \triangleright o_1 \Downarrow \overline{B_1}$ $M \triangleright o_2 \Downarrow \overline{B_2}$ $B_1 + B_2 = B_3(cc)$ $M \triangleright d := \overline{B_3} \Downarrow (H', R')$
$\texttt{cmp}\ o_1, o_2$	$(H, R\{\texttt{fl} \mapsto cc\}, a + k)$	$M \triangleright o_1 \Downarrow \overline{B_1}$ $M \triangleright o_2 \Downarrow \overline{B_2}$ $B_1 - B_2 = B_3(cc)$
$\texttt{jcc}\ \kappa, o$	(H, R, a')	$R(\texttt{fl}) \vDash \kappa$ $M \triangleright o \Downarrow \overline{a'}$
$\texttt{jcc}\ \kappa, o$	$(H, R, a + k)$	$\neg\ R(\texttt{fl}) \vDash \kappa$
$\texttt{jmp}\ o$	(H, R, a')	$M \triangleright o \Downarrow \overline{a'}$
$\texttt{malloc}\ n, d$	$(H', R', a + k)$	ℓ is fresh and $\texttt{hseg}(\ell)$ $(H\{\!\{\ell \mapsto \texttt{junk}^n\}\!\}, R, a) \triangleright d := \ell{+}0 \Downarrow (H', R')$
$\texttt{mov}\ d, o$	$(H', R', a + k)$	$M \triangleright o \Downarrow v$ $M \triangleright d := v \Downarrow (H', R')$
$\texttt{sub}\ d, o_1, o_2$	$(H, R'\{\texttt{fl} \mapsto cc\}, a + k)$	$M \triangleright o_1 \Downarrow \overline{B_1}$ $M \triangleright o_2 \Downarrow \overline{B_2}$ $B_1 - B_2 = B_3(cc)$ $M \triangleright d := \overline{B_3} \Downarrow (H', R')$

Operand Resolution

$$\frac{}{M \triangleright \texttt{im}(v) \Downarrow v} \qquad \frac{}{(H, R, a) \triangleright \texttt{rco}(r) \Downarrow R(r)} \qquad \frac{(H, R, a) \triangleright o \Downarrow \overline{a'} \quad H(a') = \langle v_1, v_2, v_3 \rangle \quad |v_1| = n \quad |v_2| = m}{(H, R, a) \triangleright \texttt{mco}(m, o, n) \Downarrow v_2}$$

Destination Propagation

$$\frac{|v| = W}{(H, R, a) \triangleright \texttt{rdest}(r) := v \Downarrow (H, R\{r \mapsto v\})}$$

$$\frac{(H, R, a) \triangleright o \Downarrow \overline{\ell{+}n'} \quad H(\ell) = \langle v_1, v_2, v_3 \rangle \quad |v_1| = n' + n \quad |v| = |v_2| = m \quad \texttt{hseg}(\ell)}{(H, R, a) \triangleright \texttt{mdest}(m, o, n) := v \Downarrow (H\{\ell \mapsto \langle v_1, v, v_3 \rangle\}, R)}$$

Condition Satisfaction

$cc \vDash \texttt{eq}$	iff	$cc(\texttt{zf}) = 1$	$cc \vDash \texttt{neq}$	iff	$cc(\texttt{zf}) = 0$
$cc \vDash \texttt{lt}$	iff	$cc(\texttt{cf}) = 1$	$cc \vDash \texttt{gte}$	iff	$cc(\texttt{cf}) = 0$
$cc \vDash \texttt{lte}$	iff	$cc(\texttt{cf}) = 1 \lor cc(\texttt{zf}) = 1$	$cc \vDash \texttt{gt}$	iff	$cc(\texttt{cf}) = 0 \land cc(\texttt{zf}) = 0$

Definitions

$$\begin{array}{lcll}
|\langle u_1, \ldots, u_n \rangle| & \overset{\text{def}}{=} & n & \\
H(\ell{+}n) & \overset{\text{def}}{=} & v_2 & \text{where } H(\ell) = \langle v_1, v_2 \rangle \text{ and } |v_1| = n \\
(\ell{+}n) + k & \overset{\text{def}}{=} & \ell{+}(n + k) & \\
v^n & \overset{\text{def}}{=} & \underbrace{\langle v, \ldots, v \rangle}_{n \text{ times}} &
\end{array}$$

Figure 3: Operational Semantics

kinds	K	$::=$	T	types
		$\mid$	Ti	types of size i
types	τ	$::=$	α	type variable
		$\mid$	$\texttt{B0}$	null value
		$\mid$	$\texttt{B1}$	bytes
		$\mid$	$\tau_1 \times \tau_2$	tuples
		$\mid$	$\texttt{box}(\tau)$	pointers
		$\mid$	$\texttt{mbox}(\tau)$	mutable pointers
		$\mid$	$\texttt{code}(\Gamma)$	code values
		$\mid$	$\Gamma \to 0$	code pointers
		$\mid$	$\forall\alpha{:}K.\tau$	universal quant.
		$\mid$	$\exists\alpha{:}K.\tau$	existential quant.
		$\mid$	$\tau_1 \wedge \tau_2$	intersection type
		$\mid$	$\tau_1 \vee \tau_2$	union type
		$\mid$	$\texttt{ns}$	top (nonsense) type
		$\mid$	$\texttt{void}$	bottom (empty) type
		$\mid$	$\mu\alpha.\tau$	recursive type
r. f. types	Γ	$::=$	$\{\texttt{r1} : \tau_1, \ldots, \texttt{rN} : \tau_N\}$	
mem. types	Ψ	$::=$	$\{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\}$	
contexts	Δ	$::=$	$\alpha_1{:}K_1, \ldots, \alpha_n{:}K_n$	

Figure 4: Type Syntax

There are three main auxiliary judgements to discuss; with them established, and using some auxiliary definition given in the figure, the operational semantics should be entirely unsurprising.

The resolution of operands is formalized by the judgement $M \rhd o \Downarrow v$, which indicates that in machine state M, the operand o resolves to value v. Propagation of computed results to their destinations is formalized by the judgement $M \rhd d := v \Downarrow (H, R)$, which indicates that in machine state M, when a destination d is assigned the value v, the resulting memory and register file are H and R. Finally, conditions (for the conditional jump `jcc`) are handled by the proposition $cc \vDash \kappa$, which is true if and only if the condition codes cc satisfy the condition κ.

A few other points of notation merit comment. Note that the arithmetic equations in the semantics specify a condition code result as well as the numeric result. The condition code results are determined in the usual manner: the carry flag (`cf`) is set when the operation generates a carry or borrow (formally, when the denormalized result lies outside the allowable range), and the zero flag (`zf`) is set when the result is zero. Map update is written $H\{\ell \mapsto v\}$ for memories and similarly for register files. Map extension (for memories only) is written $H\{\!\{\ell \mapsto v\}\!\}$, when ℓ is fresh.

2.2 The Type System

The types of TALT appear in Figure 4. The base types are `B0` which is the type of the zero-length value ($\langle\rangle$), and `B1` which is the type of byte literals. The most important type constructor is the product space, $\tau_1 \times \tau_2$, which contains values consisting of two adjacent values, having types τ_1 and τ_2. Since appending of values is associative and has a unit (namely $\langle\rangle$), products are also associative and have a (left and right) unit (namely `B0`); these equivalences are realized by a collection of subtyping rules.

$$\begin{aligned} \tau^0 &\overset{\text{def}}{=} \texttt{B0} \\ \tau^{n+1} &\overset{\text{def}}{=} \tau \times \tau^n \\ \texttt{int} &\overset{\text{def}}{=} \texttt{B1}^W \\ \texttt{nsw} &\overset{\text{def}}{=} \texttt{ns}^W \end{aligned}$$

Figure 5: Type Abbreviations

Note that unlike in TAL, a product space does *not* contain pointers to tuples; in TALT the pointer is made explicit with its own type constructor (`box`, below). This is one factor necessary to combine TAL's heap and word values; since tuples and pointers to tuples have different types, they need not be distinguished by syntactic class, as in TAL.

Three different types are provided for pointers: $\texttt{box}(\tau)$ is the type of ordinary pointers to τ, $\texttt{mbox}(\tau)$ is the type of mutable pointers, and $\Gamma \to 0$ is the type of code pointers. Each pointer type provides different privileges: `box` is covariant but immutable, `mbox` is mutable but invariant, and only code pointers may be jumped to. A subtyping rule is provided for promoting mutable pointers to covariant ones.

A series of instructions can be given the type $\texttt{code}(\Gamma)$ when it is executable provided the register file has type Γ. The function space $\Gamma \to 0$ contains executable code pointers; $\Gamma \to 0$ is like $\texttt{box}(\texttt{code}(\Gamma))$, except that the pointer must point into the code segment.

Universal and existential quantifiers, intersection types, union types, and recursive types are standard (for a Curry-style system). The top type (`ns`) contains any value of size 1. (Wider top types may be defined by exponents of `ns`.) The bottom type (`void`) is empty. Several useful abbreviations (including a word-sized top `nsw`) are defined in Figure 5.

There are two kinds for TALT types: the kind T contains all types, and the sized kind Ti contains those types whose elements all have size i. For example, `B0` belongs to $T0$, `B1` belongs to $T1$, and $\tau \times \tau'$ belongs to $T(i+j)$ if $\tau : Ti$ and $\tau' : Tj$. The various pointer types all belong to TW. Uninhabited types may vacuously belong to more than one sized kind (*e.g.,* $\texttt{B0} \wedge \texttt{B1}$ belongs to $T0$ and $T1$, and `void` vacuously belongs to any Ti), but any inhabited type has at most one sized kind.

Register file types give the type for each register (other than `fl`). In a well-formed register file type, each register's type must have kind TW. Every register must be included, so no width subtyping rules need be included. Instead, when a register's value is irrelevant it may be given the type `nsw`, and promotion to `nsw` can be done using depth subtyping. Heap types give a type for each section.

2.2.1 Static Semantics

The collection of judgements in the TALT static semantics is given in Figure 6. There are thirteen total judgements. Due to space considerations, the complete set of rules is not given here, but appears in the companion technical report [10]. Three judgements are for type formation; these rules are unsurprising and are given only in the Appendix. Two judgements are for subtyping, one for ordinary types and one for register file types; the latter simply applies the former pointwise to each register's type.

$\boxed{\vdash M}$

$$\frac{\epsilon \vdash \Psi \qquad \epsilon;\Psi \vdash H : \Psi \qquad \epsilon;\Psi \vdash R : \Gamma \qquad \texttt{cseg}(\ell) \qquad \epsilon;\Psi;\Gamma \vdash [\![H(\ell{+}n)]\!]}{\vdash (H, R, \ell{+}n)}$$

where:

$$[\![\langle \overline{\iota_1^{k_1}}, \ldots, \overline{\iota_n^{k_n}} \rangle]\!] \overset{\text{def}}{=} [\![\iota_1^{k_1}]\!]; \ldots ; [\![\iota_n^{k_n}]\!];$$

$\boxed{\Delta;\Psi \vdash v : \tau}$

$$\frac{}{\Delta;\Psi \vdash \langle\rangle : \texttt{B0}} \qquad \frac{}{\Delta;\Psi \vdash \langle b\rangle : \texttt{B1}} \qquad \frac{}{\Delta;\Psi \vdash \langle u\rangle : \texttt{ns}}$$

$$\frac{\Delta;\Psi \vdash v_1 : \tau_1 \quad \Delta;\Psi \vdash v_2 : \tau_2}{\Delta;\Psi \vdash \langle v_1, v_2\rangle : \tau_1 \times \tau_2} \qquad \frac{\Delta \vdash \Gamma \quad \Delta;\Psi;\Gamma \vdash [\![v]\!]}{\Delta;\Psi \vdash v : \texttt{code}(\Gamma)}$$

$$\frac{\Delta \vdash \Psi(\ell) \leq \tau_1 \times \tau_2 \qquad \Delta \vdash \tau_1 \times \tau_2 \leq \Psi(\ell) \qquad \Delta \vdash \tau_1 : Tn \qquad \Delta \vdash \tau_2 : T \qquad \texttt{hseg}(\ell)}{\Delta;\Psi \vdash \overline{\ell{+}n} : \texttt{mbox}(\tau_2)}$$

$$\frac{\Delta \vdash \Psi(\ell) \leq \tau_1 \times \tau_2 \quad \Delta \vdash \tau_1 : Tn \quad \Delta \vdash \tau_2 : T}{\Delta;\Psi \vdash \overline{\ell{+}n} : \texttt{box}(\tau_2)}$$

$$\frac{\Delta \vdash \Psi(\ell) \leq \tau \times \texttt{code}(\Gamma) \quad \Delta \vdash \tau : Tn \quad \Delta \vdash \Gamma \quad \texttt{cseg}(\ell)}{\Delta;\Psi \vdash \overline{\ell{+}n} : \Gamma \to 0}$$

$$\frac{(\Delta, \alpha{:}K);\Psi \vdash v : \tau}{\Delta;\Psi \vdash v : \forall\alpha{:}K.\tau} \qquad \frac{\Delta;\Psi \vdash v : \tau_1 \quad \Delta;\Psi \vdash v : \tau_2}{\Delta;\Psi \vdash v : \tau_1 \wedge \tau_2}$$

$$\frac{\Delta;\Psi \vdash v : \tau_1 \quad \Delta \vdash \tau_1 \leq \tau_2}{\Delta;\Psi \vdash v : \tau_2}$$

$\boxed{\Delta;\Psi \vdash H : \Psi'}$

$$\frac{\Delta;\Psi \vdash v_i : \tau_i \quad (\text{for } 1 \leq i \leq n)}{\Delta;\Psi \vdash \{\ell_1 \mapsto v_1, \ldots, \ell_n \mapsto v_n\} : \{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\}}$$

$\boxed{\Delta;\Psi \vdash R : \Gamma}$

$$\frac{\Delta;\Psi \vdash v_i : \tau_i \quad \Delta \vdash \tau_i : TW \quad (\text{for } 1 \leq i \leq N)}{\Delta;\Psi \vdash \{\texttt{r1} \mapsto v_1, \ldots, \texttt{rN} \mapsto v_N, \texttt{fl} \mapsto cc\} : \{\texttt{r1} : \tau_1, \ldots, \texttt{rN} : \tau_N\}}$$

$\boxed{\Delta;\Psi;\Gamma \vdash o : \tau}$

$$\frac{\Delta;\Psi \vdash v : \tau}{\Delta;\Psi;\Gamma \vdash \texttt{im}(v) : \tau} \qquad \frac{\Gamma(r) = \tau}{\Delta;\Psi;\Gamma \vdash \texttt{rco}(r) : \tau}$$

$$\frac{\Delta;\Psi;\Gamma \vdash o : \texttt{box}(\tau_1 \times \tau_2 \times \tau_3) \quad \Delta \vdash \tau_1 : Tn \quad \Delta \vdash \tau_2 : Tm}{\Delta;\Psi;\Gamma \vdash \texttt{mco}(m, o, n) : \tau_2}$$

$$\frac{\Delta;\Psi;\Gamma \vdash o : \tau_1 \quad \Delta \vdash \tau_1 \leq \tau_2}{\Delta;\Psi;\Gamma \vdash o : \tau_2}$$

$\boxed{\Delta;\Psi;\Gamma \vdash d : \tau \to \Gamma'}$

$$\frac{\Delta \vdash \tau : TW}{\Delta;\Psi;\Gamma \vdash \texttt{rdest}(r) : \tau \to \Gamma\{r{:}\tau\}}$$

$$\frac{\Delta;\Psi;\Gamma \vdash o : \texttt{mbox}(\tau_1 \times \tau_2 \times \tau_3) \quad \Delta \vdash \tau_1 : Tn \quad \Delta \vdash \tau_2 : Tm}{\Delta;\Psi;\Gamma \vdash \texttt{mdest}(m, o, n) : \tau_2 \to \Gamma}$$

Figure 7: Typing Rules (except instructions)

Judgement	Interpretation
$\Delta \vdash \tau : K$	τ is well-formed with kind K
$\Delta \vdash \Gamma$	Γ is well-formed
$\Delta \vdash \Psi$	Ψ is well-formed
$\Delta \vdash \tau_1 \leq \tau_2$	τ_1 is a subtype of τ_2
$\Delta \vdash \Gamma_1 \leq \Gamma_2$	Γ_1 is a subtype of Γ_2
$\Delta;\Psi \vdash v : \tau$	value v has type τ
$\Delta;\Psi;\Gamma \vdash o : \tau$	operand o has type τ
$\Delta;\Psi;\Gamma \vdash d : \tau \to \Gamma'$	propagation of a value having type τ to destination d results in a register file with type Γ'
$\Delta;\Psi;\Gamma \vdash I$	code I is executable when the register file has type Γ
$\Delta;\Psi;\Gamma \vdash I$ inits $r{:}\texttt{mbox}(\tau)$	discussed in Section 2.2.2
$\Delta;\Psi \vdash H : \Psi'$	H is well-formed with type Ψ'
$\Delta;\Psi \vdash R : \Gamma$	R is well-formed with type Γ
$\vdash M$	M is well-formed

Figure 6: Static Judgements

The remaining judgements are for typing. Three are for typing values, operands, and destinations; two are for typing code (the second is an auxiliary judgement for typing tuple initialization code); and the final three judgements are for typing machine states and their top-level components. Since pointers may appear anywhere and the type of a pointer depends on the type of the value it points to, each typing judgement (except state typing[4]) includes a memory type as part of the context. Operands, destinations, and code can also refer to registers, so their typing judgements also include a register file type in the context. The TALT typing rules appear in Figures 7 and 8 and are discussed below.

Machine States The typing rules for machine states are essentially the same as those of TAL. A state is well-formed if there exists a memory type Ψ and register file type Γ with respect to which the memory, register file, and pro-

[4]The state typing judgement omits a memory type because the full machine state is closed. Although the full memory is also closed, the memory typing judgement still includes a memory type as part of the context because it is occasionally necessary to type check a memory under weakened assumptions.

gram counter may be consistently typed. To type check the program counter, the current code I is looked up in memory at the program counter's address and type checked. The memory's series of bytes is converted to a code sequence by the function $[\![-]\!]$, which lifts the instruction decode function from single instructions to series of instructions.

Two aspects of this will change in future developments. First, we will add a second state typing rule for typing the intermediate states of tuple initialization (Section 2.2.2). Second, the decode function $[\![-]\!]$ will be generalized to account for relative addressing (Section 3.2).

Values Most of the value typing rules are self-explanatory, but the rules for pointers merit discussion. A pointer with offset n may be given the type $\texttt{box}(\tau_2)$ when the section it points to can be given type $\tau_1 \times \tau_2$ and all elements of τ_1 have size n (*i.e.,* they are skipped by the offset). Pointers ℓ into the heap segment may be given the stronger type $\texttt{mbox}(\tau_2)$, provided that $\Psi(\ell)$ is a subtype *and* a supertype of $\tau_1 \times \tau_2$. This ensures that when a τ_2 is written back into the heap object, the object still has type $\Psi(\ell)$, so the heap's type is unchanged.

The value typing rules tell far from the entire story. Most of the flexibility of the TALT type system is provided by its subtyping rules. Subtyping provides the introduction facility for existential, union, and recursive types; the elimination facility for universal, intersection, and recursive types; associativity and identity rules for products; and distributivity rules for intersection and union types and void. There are about fifty subtyping rules in all, listed in Appendix A; in the interest of brevity, we will discuss individual rules only as they arise.

Operands and Destinations Since an operand may be drawn from a register, the operand typing judgement includes the register file's type in its context. Similarly, since a destination may be a register, propagation of a value to a destination can change the register file type, and hence the destination typing judgement includes register file types for before and after.

The typing rules for memory operands and destinations are novel. Using the associativity and identity subtyping rules for products,[5] one casts the pointer's type in the form $(\texttt{m})\texttt{box}(\tau_1 \times \tau_2 \times \tau_3)$, where τ_2 is the operand's type, and τ_1 and τ_3 are the types of the left and right residual values. The sizes are checked using kinds: τ_1 must have kind Tn to match the offset, and τ_2 must have kind Tm to match the operand size. Note that unlike register destinations, memory destinations are forbidden to change the data's type; allowing changes would be unsound due to the possibility of aliasing.

Instructions With much of the type system moved into the rules for operands and destinations, the typing rules for instructions given in Figure 8 should be largely as expected. The final two rules merit comment. The first is a subtyping rule for code; it states that if I type checks under assumptions Γ' then it also type checks under the stronger assumptions Γ. The second rule is the elimination rule for existential types. When a register contains a value of existential type, it allows that value to be unpacked in place.

[5]Although mbox is invariant, it does respect symmetric subtyping (that is, $\texttt{mbox}(\tau) \leq \texttt{mbox}(\tau')$ when $\tau \leq \tau'$ and $\tau' \leq \tau$), so associativity and identity for products can be applied beneath an mbox.

$$\frac{\begin{array}{cc}\Delta;\Psi;\Gamma \vdash o_1 : \texttt{int} & \Delta;\Psi;\Gamma \vdash o_2 : \texttt{int} \\ \Delta;\Psi;\Gamma \vdash d : \texttt{int} \to \Gamma' & \Delta;\Psi;\Gamma' \vdash I\end{array}}{\Delta;\Psi;\Gamma \vdash \texttt{add}\ d, o_1, o_2; I}$$

$$\frac{\Delta;\Psi;\Gamma \vdash o_1 : \texttt{int} \quad \Delta;\Psi;\Gamma \vdash o_2 : \texttt{int} \quad \Delta;\Psi;\Gamma \vdash I}{\Delta;\Psi;\Gamma \vdash \texttt{cmp}\ o_1, o_2; I}$$

$$\frac{\Delta;\Psi;\Gamma \vdash o : \Gamma \to 0 \quad \Delta;\Psi;\Gamma \vdash I}{\Delta;\Psi;\Gamma \vdash \texttt{jcc}\ \kappa, o; I}$$

$$\frac{\Delta;\Psi;\Gamma \vdash o : \Gamma \to 0}{\Delta;\Psi;\Gamma \vdash \texttt{jmp}\ o; I}$$

$$\frac{\Delta;\Psi;\Gamma \vdash o : \tau \quad \Delta;\Psi;\Gamma \vdash d : \tau \to \Gamma' \quad \Delta;\Psi;\Gamma' \vdash I}{\Delta;\Psi;\Gamma \vdash \texttt{mov}\ d, o; I}$$

$$\frac{\begin{array}{cc}\Delta;\Psi;\Gamma \vdash o_1 : \texttt{int} & \Delta;\Psi;\Gamma \vdash o_2 : \texttt{int} \\ \Delta;\Psi;\Gamma \vdash d : \texttt{int} \to \Gamma' & \Delta;\Psi;\Gamma' \vdash I\end{array}}{\Delta;\Psi;\Gamma \vdash \texttt{sub}\ d, o_1, o_2; I}$$

$$\frac{\Delta;\Psi;\Gamma' \vdash I \quad \Delta \vdash \Gamma \leq \Gamma'}{\Delta;\Psi;\Gamma \vdash I}$$

$$\frac{\Gamma(r) = \exists\alpha{:}K.\tau \quad (\Delta,\alpha{:}K);\Psi;\Gamma\{r{:}\tau\} \vdash I}{\Delta;\Psi;\Gamma \vdash I}$$

Figure 8: Instruction Typing Rules (except allocation)

(The TAL instruction unpack is obtained by combining a mov instruction with this rule.)

2.2.2 Allocation and Initialization

The most complicated TALT typing rule is for malloc. A newly allocated tuple begins filled with junk and is then initialized field-by-field. As a tuple is initialized, its type must change to reflect its new components. This means that initialization is incompatible with our usual rule for memory writes.

TAL provided an elegant way to handle this using "initialization flags," in which the a newly allocated tuple is essentially stamped with its ultimate type and any initialization must move toward that ultimate type [11]. This addressed the aliasing problem because any aliases could have weaker views of a tuple, but not incompatible ones. Moreover, since TAL conflates the mechanisms of pointers and tuples, TAL can track the initialization of a tuple on a field-by-field basis. Thus, TAL permits the interleaving of initialization with other computation.

Unfortunately, (field-by-field) initialization flags are incompatible with TALT because of TALT's decoupling of pointers and tuples. Moreover, the flexibility afforded by initialization flags appears rarely to be used in practice. Therefore, as an expedient alternative, TALT requires that a newly allocated object be fully initialized before any other computation takes place. During this uninterrupted initialization, it is easy to maintain the invariant that no aliases to the new object exist. A more powerful account of initialization could likely be given using alias types [29], but we have not explored such an extension.

$$\frac{\Delta;\Psi;\Gamma\{r{:}\mathtt{nsw}\}\vdash I \text{ inits } r{:}\mathtt{mbox}(\mathtt{ns}^n)}{\Delta;\Psi;\Gamma\vdash \mathtt{malloc}\ n,\mathtt{rdest}(r);I}$$

$$\frac{\begin{array}{c}\Delta\vdash\tau\le\tau_1\times\tau_2\times\tau_3\\ \Delta\vdash\tau_1:Tn\qquad\Delta\vdash\tau_2:Tm\qquad\Delta\vdash\tau_2':Tm\\ \Delta;\Psi;\Gamma\vdash o:\tau_2'\qquad\Delta;\Psi;\Gamma\vdash I\text{ inits }r{:}\mathtt{mbox}(\tau_1\times\tau_2'\times\tau_3)\end{array}}{\Delta;\Psi;\Gamma\vdash\mathtt{mov}\ \mathtt{mdest}(m,\mathtt{rco}(r),n),o;I\text{ inits }r{:}\mathtt{mbox}(\tau)}$$

$$\frac{\Delta;\Psi;\Gamma\{r{:}\mathtt{mbox}(\tau)\}\vdash I}{\Delta;\Psi;\Gamma\vdash I\text{ inits }r{:}\mathtt{mbox}(\tau)}$$

Figure 9: Allocation/Initialization Typing Rules

The rules implementing this mechanism appear in Figure 9. The first rule provides that when an object is allocated and its pointer is placed in register r, type checking moves into a special mode in which r's type can be updated, but r cannot be used for any other purpose (the latter is enforced by giving it type `nsw`). After a series of initializing moves, the final rule is used to make r an ordinary pointer and shift back into the ordinary type checking mode. An additional (ugly) rule for type checking machine states in the midst of initialization appears in the technical report [10]; it expresses the invariants of initialization so that the Type Preservation property can be established.

2.3 Stacks

TALT supports stacks using type mechanism essentially identical to those of stack-based TAL [19], although a variety of its special instructions have been folded into TALT's operand/destination mechanism. The full details appear in the technical report [10]. For reasons of brevity, we omit any discussion of stacks in the body of this paper beyond a few remarks here.

Since TALT already has the notions of a null object ($\langle\rangle$) and of concatenation of objects in memory ($\times$), there is no need to introduce special types (`nil` and `::`) as in stack-based TAL. There is, however, a need for a special kind: since the stack's type must determine its size, and TALT types do not always do so (*e.g.,* $\mathtt{B0}\vee\mathtt{B1}$), we require a kind of types whose size are determined (though possibly unknown, as for variables of kind TD). This kind, called TD, is rarely useful aside from stacks, so for practical purposes it serves as the kind of stack types.

2.4 Multiple Entry Points

One facility provided by TALT that may not immediately be obvious is the ability to jump into the middle of a block of code, which is impossible in TAL. Operationally it is clear that one can jump to any valid address. What is less obvious is that such jumps can be well-typed. Suppose

$$H(\ell)=\langle v_{\mathtt{code}},v_{\mathtt{morecode}}\rangle$$

where, for simplicity, $\langle v_{\mathtt{code}},v_{\mathtt{morecode}}\rangle$ and $v_{\mathtt{morecode}}$ are each executable with register file type Γ, and suppose $|v_{\mathtt{code}}|=n$. We wish to assign ℓ a type so that both $\ell{+}0$ and $\ell{+}n$ have

$$\text{types}\quad\tau\quad::=\quad\cdots\mid\mathtt{set}_=(B)\mid\mathtt{set}_<(B)\mid\mathtt{set}_>(B)$$

$$\frac{}{\Delta;\Psi\vdash\overline{B}:\mathtt{set}_=(B)}$$

$$\frac{B<B'}{\Delta;\Psi\vdash\overline{B}:\mathtt{set}_<(B')}\qquad\frac{B>B'}{\Delta;\Psi\vdash\overline{B}:\mathtt{set}_>(B')}$$

$$\begin{array}{lcll}
\mathtt{set}_=(B) & \le & \mathtt{int} & \\
\mathtt{set}_<(B) & \le & \mathtt{int} & \\
\mathtt{set}_>(B) & \le & \mathtt{int} & \\
\mathtt{set}_=(B)\wedge\mathtt{set}_<(B) & \le & \mathtt{void} & \\
\mathtt{set}_=(B)\wedge\mathtt{set}_>(B) & \le & \mathtt{void} & \\
\mathtt{set}_<(B)\wedge\mathtt{set}_>(B) & \le & \mathtt{void} & \\
\mathtt{set}_<(B_1) & \le & \mathtt{set}_<(B_2) & (\text{if } B_1\le B_2)\\
\mathtt{set}_>(B_1) & \le & \mathtt{set}_>(B_2) & (\text{if } B_1\ge B_2)\\
\mathtt{set}_=(B_1) & \le & \mathtt{set}_<(B_2) & (\text{if } B_1< B_2)\\
\mathtt{set}_=(B_1) & \le & \mathtt{set}_>(B_2) & (\text{if } B_1> B_2)
\end{array}$$

Figure 10: Integer Ranges

type $\Gamma\rightarrow 0$. This is possible by giving ℓ the type:

$$\mathtt{code}(\Gamma)\wedge(\mathtt{ns}^n\times\mathtt{code}(\Gamma))$$

This makes it possible to structure TALT code with any desired forward and backward jumps, without needing to insert arbitrary breaks for typing purposes. More importantly, since any executable address can be given a type in this manner, TALT can support a true `call` instruction in which the return address is obtained from the program counter. Details of the `call` instruction appear in the technical report [10].

3 Extensions

3.1 Disjoint Sums

The standard implementation of a disjoint sum type, say $\mathtt{int}+(\mathtt{int}\times\mathtt{int})$, is as a pointer to a tuple whose leading field is a tag identifying the arm of the sum. In TALx86, this idiom is supported directly by a special type for disjoint sums. In TALT, we decompose it into its primitive components:

First we add the notion of a singleton type, written $\mathtt{set}_=(B)$, containing only the integer $\overline{B}$. With it, we can easily construct a type that faithfully characterizes this encoding:

$$\mathtt{box}(\mathtt{set}_=(0)\times\mathtt{int})\vee\mathtt{box}(\mathtt{set}_=(1)\times\mathtt{int}\times\mathtt{int})\qquad(*)$$

An element of this type is either a pointer to an `int` following a zero tag, or a pointer to an $\mathtt{int}\times\mathtt{int}$ following a one tag.

Figure 10 gives the typing rules and some pertinent subtyping rules for $\mathtt{set}_=$, and also for the upper and lower subrange types $\mathtt{set}_<$ and $\mathtt{set}_>$, which will be useful shortly. By themselves, these rules are not sufficient, however, because although the type above accurately expressed the members of a disjoint sum, it is not immediately useful.

The problem is the absence of an elimination rule for union types. We may load the tag word from offset 0 into

$$\text{instructions} \quad \iota \quad ::= \quad \cdots \mid \texttt{cmpjcc}\ o_1, o_2, \kappa, o_3$$

$$\begin{array}{lllll}
\tau_{sat}^{\texttt{eq},B} & = & \tau_{unsat}^{\texttt{neq},B} & = & \texttt{set}_{=}(B) \\
\tau_{sat}^{\texttt{neq},B} & = & \tau_{unsat}^{\texttt{eq},B} & = & \texttt{set}_{<}(B) \vee \texttt{set}_{>}(B) \\
\tau_{sat}^{\texttt{lt},B} & = & \tau_{unsat}^{\texttt{gte},B} & = & \texttt{set}_{<}(B) \\
\tau_{sat}^{\texttt{lte},B} & = & \tau_{unsat}^{\texttt{gt},B} & = & \texttt{set}_{<}(B) \vee \texttt{set}_{=}(B) \\
\tau_{sat}^{\texttt{gt},B} & = & \tau_{unsat}^{\texttt{lte},B} & = & \texttt{set}_{>}(B) \\
\tau_{sat}^{\texttt{gte},B} & = & \tau_{unsat}^{\texttt{lt},B} & = & \texttt{set}_{>}(B) \vee \texttt{set}_{=}(B)
\end{array}$$

$$\frac{\begin{array}{lll}
\Delta;\Psi;\Gamma & \vdash o_1 : \texttt{int} & (1) \\
\Delta;\Psi;\Gamma & \vdash o_2 : \texttt{set}_{=}(B) & (2) \\
\Delta;\Psi;\Gamma & \vdash \texttt{rco}(r) : \tau_1 \vee \tau_2 & (3) \\
\Delta & \vdash \tau_1 \vee \tau_2 : TW & \\
\Delta;\Psi;\Gamma\{r{:}\tau_1\} & \vdash o_1 : \tau_1' & (5) \\
\Delta;\Psi;\Gamma\{r{:}\tau_2\} & \vdash o_1 : \tau_2' & \\
\Delta & \vdash \tau_1' \wedge \tau_{unsat}^{\kappa,B} \leq \texttt{void} & (7) \\
\Delta & \vdash \tau_2' \wedge \tau_{sat}^{\kappa,B} \leq \texttt{void} & \\
\Delta;\Psi;\Gamma & \vdash o_3 : (\Gamma\{r{:}\tau_1\}) \rightarrow 0 & (9) \\
\Delta;\Psi;\Gamma\{r{:}\tau_2\} & \vdash I & (10)
\end{array}}{\Delta;\Psi;\Gamma \vdash \texttt{cmpjcc}\ o_1, o_2, \kappa, o_3; I}$$

Figure 11: Union Elimination

the disjoint union, since:

$$\begin{array}{l}
\texttt{box}(\texttt{set}_{=}(0) \times \texttt{int}) \vee \texttt{box}(\texttt{set}_{=}(1) \times \texttt{int} \times \texttt{int}) \\
\leq \hfill \text{(distributing } \vee \text{ over } \texttt{box}) \\
\texttt{box}((\texttt{set}_{=}(0) \times \texttt{int}) \vee (\texttt{set}_{=}(1) \times \texttt{int} \times \texttt{int})) \\
\leq \hfill \text{(distributing } \vee \text{ over } \times) \\
\texttt{box}((\texttt{set}_{=}(0) \vee \texttt{set}_{=}(1)) \times (\texttt{int} \vee (\texttt{int} \times \texttt{int}))) \\
\leq \hfill \text{(promotion of } \texttt{set}_{=} \text{ to } \texttt{int}) \\
\texttt{box}(\texttt{int} \times (\texttt{int} \vee (\texttt{int} \times \texttt{int})))
\end{array}$$

Then, by comparing the tag to zero, we may determine (dynamically) which arm of the disjoint union the object belongs to. However, with the rules discussed so far, there is no way to take advantage of that information in the static typing, so there is no way to access the variant constituents of the disjoint union.

To make it possible to eliminate union types, we add a new instruction `cmpjcc`. Operationally, `cmpjcc` o_1, o_2, κ, o_3 is identical to the two-instruction sequence `cmp` o_1, o_2; `jcc` κ, o_3. What is special about this (pseudo-)instruction is its typing rule.

Suppose $r : \tau_1 \vee \tau_2$, and r is used in the comparison of o_1 to o_2. (In the example above, r has type $(*)$, and the comparison is of $\texttt{mco}(W, \texttt{rco}(r), 0)$ to $\texttt{im}(\overline{0})$ using condition `eq`.) The idea of the rule is to eliminate r's union type by providing a static proof that r cannot have type τ_2 when the branch is taken, and that r cannot have type τ_1 when the branch is skipped. It follows that r can be given type τ_1 for the branch, and τ_2 for the non-branch.

Consider the `cmpjcc` typing rule, given in Figure 11. Clauses 1 and 2 ensure that the comparison itself is permissible, and indicate that the value being compared against is the number B. In the event the branch is taken, o_1 must have the type $\tau_{sat}^{\kappa,B}$, and $\tau_{unsat}^{\kappa,B}$ if it is not.

Clause 3 identifies the union type of interest, $\tau_1 \vee \tau_2$. It follows that r has either type τ_1 or type τ_2. Clause 5 then re-types the first operand in the first hypothetical circumstance. Thus, if r has type τ_1, then o_1 has type τ_1'. However, if the branch is skipped, o_1 must also have the type $\tau_{unsat}^{\kappa,B}$, and clause 7 proves that both cannot simultaneously be true. Therefore, if the branch is skipped we may conclude that r has type τ_2 (clause 10). A similar argument shows that if the branch is taken, we may conclude that r has type τ_1 (clause 9).

For example, suppose r has type $(*)$ and we wish to jump if the tag is zero using

$$\texttt{cmpjcc}\ \texttt{mco}(W, \texttt{rco}(r), 0), \texttt{im}(\overline{0}), \texttt{eq}, o_{jump}$$

The first six clauses are easily established (using the argument above for clause 1); τ_1' and τ_2' work out to be $\texttt{set}_{=}(0)$ and $\texttt{set}_{=}(1)$. It remains to show that

$$\texttt{set}_{=}(0) \wedge (\texttt{set}_{<}(0) \vee \texttt{set}_{>}(0)) \leq \texttt{void}$$

and

$$\texttt{set}_{=}(1) \wedge \texttt{set}_{=}(0) \leq \texttt{void}$$

It is easy to obtain the first using distributivity and the contradiction subtyping rules $\texttt{set}_{=}(B) \wedge \texttt{set}_{<}(B) \leq \texttt{void}$ and $\texttt{set}_{=}(B) \wedge \texttt{set}_{>}(B) \leq \texttt{void}$. For the second, observe that $\texttt{set}_{=}(1) \leq \texttt{set}_{>}(0)$ and then the result follows by the latter contradiction subtyping rule. Thus, we may continue type checking with a refined type for r in each branch.

In the more general case of n-ary sums, we may wish to case-analyze a sum using binary search to execute $\log_2 n$ comparisons rather than $n - 1$. This can be done using the same rule by choosing an inequality comparison rather than equality. To set up the rule, it is necessary to use the associativity of union types to cast an n-ary sum in the form of a single union type in which one arm is incompatible with branching and the other with not branching.

3.2 Relative Addressing

Given the TALT machine model, PC-relative addressing is not difficult to add operationally. We simply add a new operand form $\texttt{pcrel}(\pm n)$ and the operand resolution rules:

$$\frac{}{(H, R, a) \rhd \texttt{pcrel}(+n) \Downarrow a + n}$$

and

$$\frac{a' + n = a}{(H, R, a) \rhd \texttt{pcrel}(-n) \Downarrow a'}$$

For the type system, however, matters are trickier. Relative addressing creates a situation in which code (and therefore values in general) may have a certain type only when it resides in a particular location, at least when viewed naively. This situation is undesirable for two reasons: First, making the typing rules aware of the locations in which values reside would require wide-scale changes to the type system that would lessen its elegance. Second, the resulting type system would be very unlike conventional type systems in which a value's type does not depend on where it is written.

To preserve the type system in its current form, we introduce a technical device we call *delocalization.* We employ a function `deloc` to convert relative addresses to absolute ones. For example:

$$\begin{array}{ll}
\texttt{deloc}(a, \texttt{pcrel}(+n)) & = \texttt{im}(a + n) \\
\texttt{deloc}(a, \texttt{pcrel}(-n)) & = \texttt{im}(a') \quad \text{(where } a' + n = a) \\
\texttt{deloc}(a, \texttt{im}(v)) & = \texttt{im}(v) \\
\texttt{deloc}(a, \texttt{rco}(r)) & = \texttt{rco}(r) \\
\texttt{deloc}(a, \texttt{mco}(m, o, n)) & = \texttt{mco}(m, \texttt{deloc}(a, o), n)
\end{array}$$

Code is always type checked in delocalized form, so no typing rule need be provided for relative operands. Consequently, the typing rules for instruction can be preserved without change.

The burden of delocalization is assumed in the `code` typing rule, where values are converted to code sequences for type checking. Recall the rule from Figure 8:

$$\frac{\Delta \vdash \Gamma \quad \Delta;\Psi;\Gamma \vdash [\![v]\!]}{\Delta;\Psi \vdash v : \mathtt{code}(\Gamma)} \text{ (old)}$$

First, we modify the decode function (now $[\![v]\!]a$) to account for delocalization:

$$\begin{aligned}[\![\langle \overline{\iota^k}, v\rangle]\!]a &= \mathtt{deloc}(a, [\![\iota^k]\!]); [\![v]\!](a+k)\\ [\![\langle\rangle]\!]a &= \epsilon\end{aligned}$$

Second, since the typing rule does not know where in memory v lies, it is permitted to assume any address (just as it may assume any register file type) and it records the assumption in the type:

$$\frac{\Delta \vdash \Gamma \quad \Delta;\Psi;\Gamma \vdash [\![v]\!]a}{\Delta;\Psi \vdash v : \mathtt{code}(a,\Gamma)} \text{ (new)}$$

The type $\mathtt{code}(a,\Gamma)$ should be read as "code that may be executed when the register file has type Γ, *provided* it resides at address a."

The obligation is discharged in the typing rule for code pointers, which admits only properly delocalized pointers as code pointers:

$$\frac{\begin{array}{c}\Delta \vdash \Psi(\ell) \leq \tau \times \mathtt{code}(\ell{+}n, \Gamma)\\ \Delta \vdash \tau : Tn \qquad \Delta \vdash \Gamma \qquad \mathtt{cseg}(\ell)\end{array}}{\Delta;\Psi \vdash \overline{\ell{+}n} : \Gamma \to 0} \text{ (new)}$$

Note that if a code block is delocalized at an address other than where it resides, that address cannot be given a code pointer type.

The only other change that must be made to the type system is that the top-level state typing rule must be modified in the obvious manner to account for delocalization:

$$\frac{\begin{array}{c}\epsilon \vdash \Psi \qquad \epsilon;\Psi \vdash H : \Psi \qquad \epsilon;\Psi \vdash R : \Gamma\\ \mathtt{cseg}(\ell) \qquad \epsilon;\Psi;\Gamma \vdash [\![H(\ell{+}n)]\!](\ell{+}n)\end{array}}{\vdash (H, R, \ell{+}n)}$$

4 GC Safety

The principal safety results for TALT, as usual, are Progress and Type Preservation:

Theorem 4.1 (Progress) *If $\vdash M$ then $M \mapsto M'$, for some M'.*

Theorem 4.2 (Type Preservation) *If $\vdash M$ and $M \mapsto M'$ then $\vdash M'$.*

The Progress proof is by induction on the typing derivation. The Type Preservation proof is by case analysis on the evaluation derivation, with an outer induction on the typing derivation (to handle the register file subsumption and existential unpacking rules).

$$\frac{\Delta;\Psi^\circ \vdash v : \tau}{\Delta;\Psi;\Gamma \vdash \mathtt{im}(v) : \tau}$$

$$\frac{\begin{array}{ll}\Delta;\Psi \vdash v_i : \tau_i & (\text{for } 1 \leq i \leq n \text{ where } \mathtt{hseg}(\ell_i))\\ \Delta;\Psi^\circ \vdash v_i : \tau_i & (\text{for } 1 \leq i \leq n \text{ where } \mathtt{cseg}(\ell_i))\end{array}}{\Delta;\Psi \vdash \{\ell_1 \mapsto v_1, \ldots, \ell_n \mapsto v_n\} : \{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\}}$$

$$\text{where } \Psi^\circ = \Psi \restriction \{\ell \in \mathrm{dom}(\Psi) \mid \mathtt{cseg}(\ell)\}$$

Figure 12: Modified Rules for GC Safety

Additionally, since the TALT operational semantics does not account for garbage collection, we must prove an additional result showing that garbage collection does not disrupt typability.

Like TALx86, TALT is designed for use with the Boehm-Demers-Weiser conservative garbage collector [4]. Therefore there is no need to maintain any tagging invariants in order to assist the collector in distinguishing pointers from integers. However, one *GC safety* condition must be maintained for the collector to function properly [3]:

> Every live heap object must be reachable by a chain of pointers from the root set.

A pointer is permitted to reach an object by pointing anywhere from the object's beginning to its end.[6] Our root set is the collection of values in the register file and stack.

If this condition is not maintained, the only consequence is that live objects may be garbage collected. Therefore, for our purposes we define a *live* object to be one whose presence is required for the state to be well-typed. Our main GC result then will be a proof that garbage collecting unreachable objects cannot break typability of the state.

In order to satisfy the GC safety condition, we impose the following two invariants on well-typed TALT code:

1. The code segment contains no (non-accidental) pointers into the heap segment.
2. Immediate operands contain no (non-accidental) pointers into the heap segment.

Invariant 1 is necessary because the code segment is not part of the collector's root set. Invariant 2 is not stricly speaking necessary but is not burdensome either; if an operand is part of an instruction in the code segment then invariant 2 follows from invariant 1, and if not the instruction is useless anyway. The invariant is imposed because it dramatically simplifies the development to follow, since it means there is no need to formalize how to look for pointers in code values.

These invariants are achieved by replacing the typing rules for immediate operands and memories with the rules in Figure 12, which strip all heap sections from the memory type when typing code sections and immediate operands.

To formalize the GC safety theorem, we need a definition of unreachability:

[6]Interior pointers are not permitted by Boehm and Chase's specification [3], but are permitted by the collector's implementation.

Definition 4.3 Suppose S is a set of heap section identifiers. Then S is *H-reachable* from v if v can be written in the form $\langle v_1, \overline{\ell+n}, v_2\rangle$ for some $\ell \in S$ and $n \leq |H(\ell)|$.

A section is unreachable from the memory as a whole if every section that can reach it is either untraced (*i.e.,* in the code segment) or unreachable itself:

Definition 4.4 Suppose S is a set of heap section identifiers. Then S is *unreachable* from H if for every $\ell \in \mathrm{dom}(H)$, either $\mathtt{cseg}(\ell)$ or $\ell \in S$ or S is not H-reachable from $H(\ell)$.

Definition 4.5 Suppose S is a set of heap section identifiers. Then S is *unreachable* in (H, R, a) if:

- S is unreachable from H, and
- for every r, S is not H-reachable from $R(r)$, and
- S is not H-reachable from $R(\mathtt{sp})$.

Note that the definition of unreachability is not deterministic. When S is unreachable in M, S can be as small as the empty set. The definition is crafted in this manner because we cannot predict what objects will actually be collected by a conservative collector. Instead, we prove our result for all unreachable sets, thereby covering whatever the collector turns out to do.

Theorem 4.6 (GC Safety) *Suppose S is unreachable in (H, R, a) and suppose that $\vdash (H, R, a)$. Then $\vdash (H \setminus S, R, a)$, where $H \setminus S = H \upharpoonright (\mathrm{dom}(H) \setminus S)$.*

The proof is by induction on the typing derivation.

5 Machine-Checked Proofs

The results of this paper are formalized as machine-checked proofs in the Twelf system [24, 25]. The first step in this formalization is to encode the TALT syntax, type system, and operational semantics as an LF signature [14] wherein judgements become types and derivations become terms. This process is standard, so we will not belabor it here.

Once the language is formalized, we can state and prove the principal meta-theorems: Progress, Type Preservation, and GC Safety. A meta-theorem statement is encoded as a relation between derivations [22, 23]. For example, the principal meta-theorem statements become:

```
progress : machineok M
           -> stepsto M M' -> type.
%mode progress +D1 -D2.

preservation : machineok M
               -> stepsto M M'
               -> machineok M' -> type.
%mode preservation +D1 +D2 -D3.

collect_ok : collect M M'
             -> machineok M
             -> machineok M' -> type.
%mode collect_ok +D1 +D2 -D3.
```

For instance, the `progress` theorem is a relation between `machineok M` derivations and `stepsto M M'` derivations, and so forth. The `%mode` declaration following each statement indicates the input and output arguments of the relation [26].

Proof Components	
lines	purpose
210	Properties of canonical forms
252	Properties of conditions
320	Properties of memory lookup/update/extension
349	Instruction decode lemmas
361	Size lemmas
372	Properties of register file lookup/update
399	Properties of values
407	Equality lemmas
429	Properties of natural number arithmetic
515	Validity (well-behavedness of derivations)
537	Canonical Forms lemma
548	Lemmas regarding type formation
592	Progress
713	Weakening and strengthening of memory types
844	Operand/destination lemmas
925	Type Preservation
1119	GC Safety
1245	Properties of binary arithmetic

Table 1: Safety Proof Breakdown

In each of these cases, the last argument (with the "`-`" mode) is the sole output argument.

A proof of a meta-theorem so encoded is a logic program whose type is the relational encoding of the theorem. The program can then (in principle) be given derivations in the input arguments and unification variables in the output arguments and executed to obtain the resulting derivations. A logic program represents a valid meta-proof if the execution always runs to a successful conclusion.

The Twelf totality checker verifies that the logic program is in fact total, with assistance from the programmer in identifying the induction variable(s). This consists of checking three facts:

1. Mode checking: output arguments (and input arguments to subcalls) are fully determined.
2. Termination checking: the induction variable(s) decrease in all recursive calls.
3. Totality checking: in every case analysis, all cases are covered.

For details, the reader is referred to Pfenning and Schürmann [25] or Schürmann [27].

The specification of TALT in LF consists of 2081 lines of Twelf code, and the complete proof of Theorems 4.1, 4.2 and 4.6 consists of 10137 lines of Twelf code (including comments). A breakdown of the proof code for the interested reader is given in Table 1. The full proof takes approximately three minutes to check in Twelf 1.3R3 on a Pentium 3 with 128 megs of RAM.

6 Conclusion

TALT provides an elegant, expressive, and fully formalized type system for assembly language. The machine-checkable safety theorems of Section 5 provide a complete safety argument for TALT programs in terms of a safety policy expressed (by the operational semantics) at the assembly language level. To complete a foundational TAL system, this work must be combined with a proof that the TALT operational semantics maps correctly on the concrete architecture. TALT is designed so that this latter stage of the proof implementation is a type-free simulation argument.

TALT is also designed to be easily adaptable to other architectures. The general architectural constants (big-endian vs. little-endian, the number of registers, and the size of the machine word) are parameters to the language and can easily be changed. New instructions can also easily be added; in fact, given the generality of the TALT operand/destination mechanism, many "new" instructions may already be present. This means that the primary burden of specializing TALT to an architecture is accounting for its idiosyncrasies (*e.g.,* the IA-32's treatment of floating point, or the delay slots following jumps on many RISC architectures).

One issue not addressed in TALT is the issue of processor exceptions and faults (*e.g.,* division by zero, stack overflow, or various floating-point exceptions). These can be prohibitively expensive to prevent dynamically, and are difficult to prevent statically. Fortunately, for our purposes they can be safely ignored, assuming that the occurrence of an exception aborts the program, because a program that is no longer running is certainly safe.

The issue of stack overflow is a little bit more involved, because we must ensure that any stack that overflows will cause the stack overflow exception. We can do so by following the standard practice of preceding[7] the stack in memory by at least one (typically many) unmapped memory pages. Any `push` instruction that overflows the stack will therefore hit an unmapped page and cause the exception. The `salloc` instruction, which increases the stack's size by an arbitrary amount, is a bit trickier because it could skip all the unmapped pages and land in accessible memory. We can prevent this by limiting `salloc` to one page at a time and forcing it to touch the top of the stack as `push` does. This ensures that any overflowing `salloc` will hit the unmapped page.

To assist in the development of TALT programs, we have also designed an explicitly typed variant of TALT, called XTALT, that enjoys decidable type checking. XTALT adds type annotations to sections and replaces TALT's subtyping with explicit coercions. We are implementing an assembler that typechecks XTALT programs, generates the corresponding TALT derivations, and compresses them using Necula and Rahul's technique [21]. Also under development is a Popcorn [18] to XTALT compiler. These tools can be combined with Dunfield's Standard ML to Popcorn compiler [12] (based on RML [30] and MLton [6]) to provide a complete certifying compiler from Standard ML to TALT. A direct Standard ML to TALT compiler is also underway.

[7]assuming a downward growing stack

References

[1] Andrew W. Appel and Amy P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *Twenty-Seventh ACM Symposium on Principles of Programming Languages*, pages 243–253, Boston, January 2000.

[2] Hans-J. Boehm. Simple garbage-collector safety. In *1996 SIGPLAN Conference on Programming Language Design and Implementation*, pages 89–98, Philadelphia, Pennsylvania, May 1996.

[3] Hans-J. Boehm and David Chase. A proposal for garbage-collector-safe C compilation. *The Journal of C Language Translation*, 4(2):126–141, December 1992.

[4] Hans-J. Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, September 1988.

[5] Rajkumar Buyya and Mark Baker, editors. *First International Workshop on Grid Computing*, volume 1971 of *Lecture Notes in Computer Science*, Bangalore, India, December 2000. Springer-Verlag.

[6] Henry Cejtin, Suresh Jagannathan, and Stephen Weeks. Flow-directed closure conversion for typed languages. In *European Symposium on Programming*, March 2000.

[7] Bor-Yuh Evan Chang, Karl Crary, Margaret DeLap, Robert Harper, Jason Liszka, Tom Murphy VII, and Frank Pfenning. Trustless grid computing in ConCert. Technical Report CMU-CS-02-152, Carnegie Mellon University, School of Computer Science, June 2002.

[8] Christopher Colby, Peter Lee, George Necula, and Fred Blau. A certifying compiler for Java. In *2000 SIGPLAN Conference on Programming Language Design and Implementation*, pages 95–107, Vancouver, British Columbia, June 2000.

[9] ConCert. `http://www.cs.cmu.edu/~concert`, September 2001.

[10] Karl Crary. Toward a foundational typed assembly language. Technical Report CMU-CS-02-196, Carnegie Mellon University, School of Computer Science, December 2002.

[11] Karl Crary and Greg Morrisett. Type structure for low-level programming langauges. In *Twenty-Sixth International Colloquium on Automata, Languages, and Programming*, volume 1644 of *Lecture Notes in Computer Science*, pages 40–54, Prague, Czech Republic, July 1999. Springer-Verlag.

[12] Joshua Dunfield. Personal communication.

[13] Nadeem Hamid, Zhong Shao, Valery Trifonov, Stefan Monnier, and Zhaozhong Ni. A syntactic approach to foundational proof-carrying code. In *Seventeenth IEEE Symposium on Logic in Computer Science*, Copenhagen, Denmark, July 2002.

[14] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, January 1993.

[15] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual*, 2001. Order numbers 245470–245472.

[16] Craig Lee, editor. *Second International Workshop on Grid Computing*, volume 2242 of *Lecture Notes in Computer Science*, Denver, Colorado, November 2001. Springer-Verlag.

[17] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[18] 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 Workshop on Compiler Support for System Software*, Atlanta, May 1999.

[19] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. *Journal of Functional Programming*, 12(1):43–88, January 2002. An earlier version appeared in the 1998 Workshop on Types in Compilation, volume 1473 of Lecture Notes in Computer Science.

[20] 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. An earlier version appeared in the 1998 Symposium on Principles of Programming Languages.

[21] George Necula and S. P. Rahul. Oracle-based checking of untrusted software. In *Twenty-Eighth ACM Symposium on Principles of Programming Languages*, pages 142–154, London, January 2001.

[22] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[23] Frank Pfenning and Ekkehard Rohwedder. Implementing the meta-theory of deductive systems. In *Eleventh International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Computer Science*, pages 537–551, Saratoga Springs, New York, June 1992. Springer-Verlag.

[24] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logic framework for deductive systems. In *Sixteenth International Conference on Automated Deduction*, volume 1632 of *Lecture Notes in Computer Science*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag.

[25] Frank Pfenning and Carsten Schürmann. *Twelf User's Guide, Version 1.3R4*, 2002. Available electronically at `http://www.cs.cmu.edu/~twelf`.

[26] Ekkehard Rohwedder and Frank Pfenning. Mode and termination checking for higher-order logic programs. In *European Symposium on Programming*, volume 1058 of *Lecture Notes in Computer Science*, pages 296–310, Linköping, Sweden, April 1996. Springer-Verlag.

[27] Carsten Schürmann. *Automating the Meta Theory of Deductive Systems*. PhD thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania, October 2000.

[28] SETI@Home. `http://setiathome.ssl.berkeley.edu`, November 2000.

[29] Frederick Smith, David Walker, and Greg Morrisett. Alias types. In *European Symposium on Programming*, Berlin, Germany, March 2000.

[30] Andrew Tolmach and Dino P. Oliva. From ML to Ada: Strongly-typed language interoperability via source translation. *Journal of Functional Programming*, 8(4):367–412, July 1998.

[31] J. B. Wells. Typability and type checking in System F are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98(1–3):111–156, 1999.

A Type Formation and Subtyping Rules

$\boxed{\Delta \vdash c : K}$

$$\frac{}{\Delta \vdash \alpha : \Delta(\alpha)} \qquad \frac{}{\Delta \vdash \texttt{B0} : \mathit{T0}} \qquad \frac{}{\Delta \vdash \texttt{B1} : \mathit{T1}} \qquad \frac{}{\Delta \vdash B : \mathit{Num}} \qquad \frac{\Delta \vdash \tau_1 : T \quad \Delta \vdash \tau_2 : T}{\Delta \vdash \tau_1 \times \tau_2 : T} \qquad \frac{\Delta \vdash \tau_1 : \mathit{TD} \quad \Delta \vdash \tau_2 : \mathit{TD}}{\Delta \vdash \tau_1 \times \tau_2 : \mathit{TD}}$$

$$\frac{\Delta \vdash \tau_1 : \mathit{Ti} \quad \Delta \vdash \tau_2 : \mathit{Tj}}{\Delta \vdash \tau_1 \times \tau_2 : T(i+j)} \qquad \frac{\Delta \vdash \tau : T}{\Delta \vdash \texttt{box}(\tau) : \mathit{TW}} \qquad \frac{\Delta \vdash \tau : T}{\Delta \vdash \texttt{mbox}(\tau) : \mathit{TW}} \qquad \frac{\Delta \vdash \tau : \mathit{TD}}{\Delta \vdash \texttt{sptr}(\tau) : \mathit{TW}}$$

$$\frac{\Delta \vdash \Gamma}{\Delta \vdash \texttt{code}(a, \Gamma) : T} \qquad \frac{\Delta \vdash \Gamma}{\Delta \vdash \Gamma \to 0 : \mathit{TW}} \qquad \frac{\Delta \vdash x : \mathit{Num}}{\Delta \vdash \texttt{set}_=(x) : \mathit{TW}} \qquad \frac{\Delta \vdash x : \mathit{Num}}{\Delta \vdash \texttt{set}_<(x) : \mathit{TW}} \qquad \frac{\Delta \vdash x : \mathit{Num}}{\Delta \vdash \texttt{set}_>(x) : \mathit{TW}}$$

$$\frac{\Delta, \alpha{:}K \vdash \tau : T}{\Delta \vdash \forall \alpha{:}K.\tau : T} \qquad \frac{\Delta, \alpha{:}K \vdash \tau : T \quad \Delta \vdash c : K \quad \Delta \vdash \tau[c/\alpha] : K'}{\Delta \vdash \forall \alpha{:}K.\tau : K'} \ (K' \in \{\mathit{TD}, \mathit{Ti}\}) \qquad \frac{\Delta, \alpha{:}K \vdash \tau : T}{\Delta \vdash \exists \alpha{:}K.\tau : T} \qquad \frac{\Delta, \alpha{:}K \vdash \tau : \mathit{Ti}}{\Delta \vdash \exists \alpha{:}K.\tau : \mathit{Ti}}$$

$$\frac{}{\Delta \vdash \texttt{ns} : \mathit{T1}} \qquad \frac{}{\Delta \vdash \texttt{void} : \mathit{Ti}} \qquad \frac{\Delta, \alpha{:}T \vdash \tau : T}{\Delta \vdash \mu\alpha.\tau : T} \qquad \frac{\Delta, \alpha{:}T \vdash \tau : T \quad \Delta \vdash \tau[\mu\alpha.\tau/\alpha] : K}{\Delta \vdash \mu\alpha.\tau : K} \ (K \in \{\mathit{TD}, \mathit{Ti}\})$$

$$\frac{\Delta \vdash \tau_1 : T \quad \Delta \vdash \tau_2 : T}{\Delta \vdash \tau_1 \wedge \tau_2 : T} \qquad \frac{\Delta \vdash \tau_1 : \mathit{TD} \quad \Delta \vdash \tau_2 : T}{\Delta \vdash \tau_1 \wedge \tau_2 : \mathit{TD}} \qquad \frac{\Delta \vdash \tau_1 : T \quad \Delta \vdash \tau_2 : \mathit{TD}}{\Delta \vdash \tau_1 \wedge \tau_2 : \mathit{TD}} \qquad \frac{\Delta \vdash \tau_1 : \mathit{Ti} \quad \Delta \vdash \tau_2 : T}{\Delta \vdash \tau_1 \wedge \tau_2 : \mathit{Ti}}$$

$$\frac{\Delta \vdash \tau_1 : T \quad \Delta \vdash \tau_2 : \mathit{Ti}}{\Delta \vdash \tau_1 \wedge \tau_2 : \mathit{Ti}} \qquad \frac{\Delta \vdash \tau_1 : T \quad \Delta \vdash \tau_2 : T}{\Delta \vdash \tau_1 \vee \tau_2 : T} \qquad \frac{\Delta \vdash \tau_1 : \mathit{Ti} \quad \Delta \vdash \tau_2 : \mathit{Ti}}{\Delta \vdash \tau_1 \vee \tau_2 : \mathit{Ti}} \qquad \frac{\Delta \vdash \tau : \mathit{TD}}{\Delta \vdash \tau : T} \qquad \frac{\Delta \vdash \tau : \mathit{Ti}}{\Delta \vdash \tau : \mathit{TD}}$$

$\boxed{\Delta \vdash \tau_1 \le \tau_2}$

$$\frac{}{\Delta \vdash \tau \le \tau} \qquad \frac{\Delta \vdash \tau_1 \le \tau_2 \quad \Delta \vdash \tau_2 \le \tau_3}{\Delta \vdash \tau_1 \le \tau_3} \qquad \frac{\Delta \vdash \tau_1 \le \tau_1' \quad \Delta \vdash \tau_2 \le \tau_2'}{\Delta \vdash \tau_1 \times \tau_2 \le \tau_1' \times \tau_2'}$$

$$\frac{\Delta \vdash \Gamma' \le \Gamma}{\Delta \vdash \texttt{code}(a, \Gamma) \le \texttt{code}(a, \Gamma')} \qquad \frac{\Delta \vdash \Gamma' \le \Gamma}{\Delta \vdash \Gamma \to 0 \le \Gamma' \to 0} \qquad \frac{\Delta \vdash \tau \le \tau'}{\Delta \vdash \texttt{box}(\tau) \le \texttt{box}(\tau')} \qquad \frac{\Delta \vdash \tau \le \tau' \quad \Delta \vdash \tau' \le \tau}{\Delta \vdash \texttt{mbox}(\tau) \le \texttt{mbox}(\tau')}$$

$$\frac{\Delta \vdash \tau \le \tau' \quad \Delta \vdash \tau : \mathit{TD} \quad \Delta \vdash \tau' : \mathit{TD}}{\Delta \vdash \texttt{sptr}(\tau) \le \texttt{sptr}(\tau')} \qquad \frac{\Delta, \alpha{:}K \vdash \tau \le \tau'}{\Delta \vdash \forall \alpha{:}K.\tau \le \forall \alpha{:}K.\tau'} \qquad \frac{\Delta, \alpha{:}K \vdash \tau \le \tau'}{\Delta \vdash \exists \alpha{:}K.\tau \le \exists \alpha{:}K.\tau'}$$

$$\frac{\Delta, \alpha{:}K \vdash \tau : T \quad \Delta \vdash c : K}{\Delta \vdash \forall \alpha{:}K.\tau \le \tau[c/\alpha]} \qquad \frac{\Delta, \alpha{:}K \vdash \tau : T \quad \Delta \vdash c : K}{\Delta \vdash \tau[c/\alpha] \le \exists \alpha{:}K.\tau} \qquad \frac{}{\Delta \vdash \tau \le \forall \alpha{:}K.\tau} \ (\alpha \notin \tau) \qquad \frac{}{\Delta \vdash \exists \alpha{:}K.\tau \le \tau} \ (\alpha \notin \tau)$$

$$\frac{\Delta \vdash \tau : \mathit{Ti}}{\Delta \vdash \tau \le \texttt{ns}^i} \qquad \frac{\Delta \vdash \tau : T}{\Delta \vdash \texttt{void} \le \tau} \qquad \frac{\Delta, \alpha{:}T \vdash \tau : T}{\Delta \vdash \tau[\mu\alpha.\tau/\alpha] \le \mu\alpha.\tau} \qquad \frac{\Delta, \alpha{:}T \vdash \tau : T}{\Delta \vdash \mu\alpha.\tau \le \tau[\mu\alpha.\tau/\alpha]}$$

$$\frac{\Delta \vdash \tau \le \tau_1 \quad \Delta \vdash \tau \le \tau_2}{\Delta \vdash \tau \le \tau_1 \wedge \tau_2} \qquad \frac{\Delta \vdash \tau_2 : T}{\Delta \vdash \tau_1 \wedge \tau_2 \le \tau_1} \qquad \frac{\Delta \vdash \tau_1 : T}{\Delta \vdash \tau_1 \wedge \tau_2 \le \tau_2} \qquad \frac{\Delta \vdash \tau_2 : T}{\Delta \vdash \tau_1 \le \tau_1 \vee \tau_2}$$

$$\frac{\Delta \vdash \tau_1 : T}{\Delta \vdash \tau_2 \le \tau_1 \vee \tau_2} \qquad \frac{}{\Delta \vdash \tau \wedge (\tau_1 \vee \tau_2) \le (\tau \wedge \tau_1) \vee (\tau \wedge \tau_2)} \qquad \frac{\Delta \vdash \tau_1 : \mathit{Ti} \quad \Delta \vdash \tau_1' : \mathit{Ti}}{\Delta \vdash (\tau_1 \times \tau_2) \wedge (\tau_1' \times \tau_2') \le (\tau_1 \wedge \tau_1') \times (\tau_2 \wedge \tau_2')}$$

$$\frac{}{\Delta \vdash \tau \times (\tau_1 \vee \tau_2) \le (\tau \times \tau_1) \vee (\tau \times \tau_2)} \qquad \frac{}{\Delta \vdash (\tau_1 \vee \tau_2) \times \tau \le (\tau_1 \times \tau) \vee (\tau_2 \times \tau)}$$

$$\frac{\Delta \vdash \tau : T}{\Delta \vdash \tau \times \texttt{void} \le \texttt{void}} \qquad \frac{\Delta \vdash \tau : T}{\Delta \vdash \texttt{void} \times \tau \le \texttt{void}} \qquad \frac{}{\Delta \vdash \texttt{mbox}(\tau) \le \texttt{box}(\tau)} \qquad \frac{}{\Delta \vdash \tau_1 \times (\tau_2 \times \tau_3) \le (\tau_1 \times \tau_2) \times \tau_3}$$

$$\frac{}{\Delta \vdash (\tau_1 \times \tau_2) \times \tau_3 \le \tau_1 \times (\tau_2 \times \tau_3)} \qquad \frac{}{\Delta \vdash \tau \le \texttt{B0} \times \tau} \qquad \frac{}{\Delta \vdash \texttt{B0} \times \tau \le \tau} \qquad \frac{}{\Delta \vdash \tau \le \tau \times \texttt{B0}} \qquad \frac{}{\Delta \vdash \tau \times \texttt{B0} \le \tau}$$

$$\frac{}{\Delta \vdash \texttt{set}_=(B) \le \texttt{int}} \qquad \frac{}{\Delta \vdash \texttt{set}_<(B) \le \texttt{int}} \qquad \frac{}{\Delta \vdash \texttt{set}_>(B) \le \texttt{int}}$$

$$\frac{}{\texttt{set}_=(B) \wedge \texttt{set}_<(B) \le \texttt{void}} \qquad \frac{}{\texttt{set}_=(B) \wedge \texttt{set}_>(B) \le \texttt{void}} \qquad \frac{}{\texttt{set}_<(B) \wedge \texttt{set}_>(B) \le \texttt{void}} \qquad \frac{}{\Delta \vdash \texttt{int} \le \exists \alpha{:}\mathit{Num}.\texttt{set}_=(\alpha)}$$

$$\frac{B_1 \le B_2}{\texttt{set}_<(B_1) \le \texttt{set}_<(B_2)} \qquad \frac{B_1 \ge B_2}{\texttt{set}_>(B_1) \le \texttt{set}_>(B_2)} \qquad \frac{B_1 < B_2}{\texttt{set}_=(B_1) \le \texttt{set}_<(B_2)} \qquad \frac{B_1 > B_2}{\texttt{set}_=(B_1) \le \texttt{set}_>(B_2)}$$

$\boxed{\Delta \vdash \Gamma \le \Gamma'}$

$$\frac{\Delta \vdash \tau_i \le \tau_i' \quad (\text{for } 1 \le i \le N) \quad \Delta \vdash \tau \le \tau'}{\Delta \vdash \{\texttt{r1}{:}\tau_1, \ldots, \texttt{rN}{:}\tau_N, \texttt{sp}{:}\tau\} \le \{\texttt{r1}{:}\tau_1', \ldots, \texttt{rN}{:}\tau_N', \texttt{sp}{:}\tau'\}}$$