An Introduction to Typed Assembly Language

David Walker
Department of Computer Science
Princeton University

March 2, 2002
Acknowledgments

- These notes started as a lecture given by Greg Morrisett, July 2001 [10] and have since been extended and edited.

- They give readers a simple introduction to many of the core elements of the Cornell Typed Assembly Language project.


  - See http://www.cs.cornell.edu/talc

- Suggested Reading


- A more complete bibliography appears at the end of these notes.
Safety through Types

• An architecture for safe mobile code:
  – Download code and typing annotations from untrusted code producer
  – Verify untrusted code using trusted type checker
  – Link verified code into extensible system & run without error

• Security hinges on an understanding of programming language structure
  – We must be able to reason precisely about what programs do.
  – We must be able to define the “good” and “bad” behaviors.
  – We must be able to identify and rule out (mechanically) those programs that might exhibit “bad” behaviors.

• Typed Assembly Language (TAL) is the language technology we will use to accomplish the goals.
Outline

- TAL-0: Assembly Language Control Flow and Basic Types
- TAL-1: Parametric Polymorphism
- TAL-2: Stack Types
- Type-Directed Compilation: From Tiny to TAL-2
- TAL-3: Data Structures
- TAL-4: Dependency
- TAL-5: Modularity and Linking
- References
What Is TAL?

- In theory:
  - An idealized RISC-style assembly language and formal operational semantics for a simple abstract machine
  - A formal type system (collection of type systems) that captures properties of processor register state, stack and memory
  - Rigorous proofs demonstrate that TAL enforces important safety guarantees in assembly language programs

- In practice [20, 11]:
  - A type checker for almost all of the Intel Pentium IA32 architecture
  - Tools for assembly, disassembly, and linking of TAL binaries (a pair of machine code segment and types segment)
  - A prototype compiler for a safe imperative language (Popcorn)

- These notes concentrate on the development of the theory of TAL and type-directed compilation. This presentation streamlines the formal work from past papers.
Example Assembly Language Program

High-level code:

\[
\text{fact}(n,a) = \\
\quad \text{if } (n \leq 0) \text{ then} \\
\quad \quad a \\
\quad \text{else} \\
\quad \quad \text{fact}(n-1,a \times n)
\]

Assembly language code:

\% \ r_1 \ holds \ n, \ r_2 \ holds \ a, \ r_{31} \ holds \ return \ address \n\% \ which \ expects \ the \ result \ in \ r_1

\textit{fact}:
\begin{align*}
\text{ble} \ r_1, L2 & \quad \% \ if \ n \leq 0 \ goto \ L2 \\
\text{mul} \ r_2, r_2, r_1 & \quad \% \ a := a \times n \\
\text{sub} \ r_1, r_1, 1 & \quad \% \ n := n-1 \\
\text{jmp} \ \text{fact} & \quad \% \ goto \ \text{fact}
\end{align*}

\textit{L2}:
\begin{align*}
\text{mov} \ r_1, r_2 & \quad \% \ result := a \\
\text{jmp} \ r_{31} & \quad \% \ jump \ to \ return \ address
\end{align*}
TAL-0

Syntax of a simple RISC-like assembly language.

- Registers: \( r \in \{ r_1, r_2, r_3, \ldots \} \)
- Labels: \( L \in \text{Identifier} \)
- Integers: \( n \in [-2^{k-1}..2^{k-1}) \)
- Blocks: \( B ::= \text{jmp } v | i; B \)
- Instrs: \( i ::= \text{aop } r_d, r_s, v | \text{bop } r, v | \text{mov } r, v \)
- Operands: \( v ::= r | n | L \)
- Arithmetic Ops: \( \text{aop} ::= \text{add} | \text{sub} | \text{mul} | \cdots \)
- Branch Ops: \( \text{bop} ::= \text{beq} | \text{bgt} | \cdots \)
TAL-0 Abstract Machine

- Model evaluation as a transition function mapping machine states to machine states: $\Sigma \mapsto \Sigma$

- Machine: $\Sigma = (H, R, B)$

- $H$ is a partial map from labels to basic blocks $B$.

- $R$ maps registers to values (ints $n$ or labels $L$). Notation:

  $$
  \begin{align*}
  R(n) &= n \\
  R(L) &= L \\
  R(r) &= v & \text{if } R = \{ \ldots, r \mapsto v, \ldots \}
  \end{align*}
  $$

- $B$ is a basic block (corresponding to the current program counter.)
Operational Semantics

\[(H, R, \text{mov} \ r_d, v; B) \mapsto (H, R[r_d := R(v)], B)\]

\[(H, R, \text{add} \ r_d, r_s, v; B) \mapsto (H, R[r_d := n], B)\]
where \(n = R(v) + R(r_s)\)

\[(H, R, \text{jmp} \ v) \mapsto (H, R, B)\]
where \(R(v) = L\) and \(H(L) = B\)

\[(H, R, \text{beq} \ r, v; B) \mapsto (H, R, B)\]
where \(R(r) \neq 0\)

\[(H, R, \text{beq} \ r, v; B) \mapsto (H, R, B')\]
where \(R(r) = 0, R(v) = L, \) and \(H(L) = B'\)

The other instructions (\text{sub}, \text{bgt}, etc.) follow a similar pattern.
Error Conditions

- The abstract machine is \textit{stuck} if there is no transition from the current state to some next state.
- The \textit{stuck states} define the “bad” things that may happen.
- Our type system will ensure that the machine never gets stuck.
- Example stuck states:
  - \((H, R, \text{add} r_d, r_s, v; B)\) and \(r_s\) or \(v\) aren’t ints
  - \((H, R, \text{jmp} v)\) and \(v\) isn’t a label, or
  - \((H, R, \text{beq} r, v; B)\) and \(r\) isn’t an int or \(v\) isn’t a label
- To distinguish between integers and labels, we require a type system.
Types

Basic types:

- \( \tau ::= \text{int} \mid \Gamma \to \{ \} \)
- \( \Gamma ::= \{ r_1: \tau_1, r_2: \tau_2, \ldots \} \)

Code types:

- Code labels have type \( \{ r_1: \tau_1, r_2: \tau_2, \ldots \} \to \{ \} \).
- The order that register names appear in a code type is irrelevant.
- To jump to code with this type, register \( r_1 \) must contain a value of type \( \tau_1 \), register \( r_2 \) must contain \( \ldots \)
- Intuitively, code labels are functions that take a record of arguments
- The function never returns — the code block always ends with a jump to another label
Example Program with Types

% $r_1$ holds $n$, $r_2$ holds $a$, $r_{31}$ holds return address
% which expects the result in $r_1$

\[
\text{fact: } \{r_1:i, r_2:i, r_{31}:r_1:i\} \rightarrow \{ \} \rightarrow \{ \} \\
\text{ble } r_1, L2 \quad \% \text{ if } n \leq 0 \text{ goto } L2 \\
\text{mul } r_2, r_2, r_1 \quad \% a := a \times n \\
\text{sub } r_1, r_1, 1 \quad \% n := n-1 \\
\text{jmp fact} \quad \% \text{ goto fact}
\]

\[
\text{L2: } \{r_2:i, r_{31}:r_1:i\} \rightarrow \{ \} \rightarrow \{ \} \\
\text{mov } r_1, r_2 \quad \% \text{ result := a} \\
\text{jmp } r_{31} \quad \% \text{ jump to return address}
\]
Mis-typed Program

\[ \text{fact: } \{ r_1: \text{int}, r_{31}: \{ r_1: \text{int} \} \rightarrow \{ \} \} \rightarrow \{ \} \]

\text{ble } r_1, L2
\text{mul } r_2, r_2, r_1 \quad \% \text{ ERROR! } r_2 \text{ doesn’t have a type}
\text{mov } r_1, r_3
\text{jmp } L1 \quad \% \text{ ERROR! no such label}

\[ \text{L2: } \{ r_2: \text{int}, r_{31}: \{ r_1: \text{int} \} \rightarrow \{ \} \} \rightarrow \{ \} \]
\text{mov } r_{31}, r_2
\text{jmp } r_{31} \quad \% \text{ ERROR! } r_{31} \text{ is not a label}
Type Checking Basics

• We need to keep track of:
  – the types of the registers at each point in the code (type-states)
  – the types of the labels on the code

• Heap Types: $\Psi$ will map labels to label types.

• Register Types: $\Gamma$ will map registers to types.
Typing Operands

• integer literals are ints:

\[ \Psi; \Gamma \vdash n : \text{int} \]

• lookup register types in \( \Gamma \):

\[ \Psi; \Gamma \vdash r : \Gamma(r) \]

• lookup label types in \( \Psi \):

\[ \Psi; \Gamma \vdash L : \Psi(L) \]
Subtyping

- Our program will never crash if the register file contains more values than necessary to satisfy some typing precondition.

- In other words, a register file type with more components is a *subtype* of a register file containing fewer components.

\[
\{r_1:\tau_1, \ldots, r_{i-1}:\tau_{i-1}, r_i:\tau_i\} \subseteq \{r_1:\tau_1, \ldots, r_{i-1}:\tau_{i-1}\}
\]

- Notice the similarity to record subtyping: a record with more fields is a subtype of a record with fewer fields.

- On the other hand, label type subtyping works in the opposite direction. A label that only requires \(r_1\) and \(r_2\) to contain integers may be used as a label that requires \(r_1\), \(r_2\) and \(r_3\) to contain integers.

- Label types, like ordinary function types, obey *contravariant* subtyping rules in their argument types:

\[
\begin{align*}
\Gamma' \leq \Gamma \\
\Gamma \rightarrow \{\} \leq \Gamma' \rightarrow \{\}
\end{align*}
\]

- Subtyping is also reflexive and transitive.

- A subsumption rule allows a value to be used at a supertype:

\[
\begin{align*}
\Psi; \Gamma \vdash v : \tau_1 & \quad \tau_1 \leq \tau_2 \\
\Psi; \Gamma \vdash v : \tau_2
\end{align*}
\]
Typing Instructions

- The judgment for instructions looks like:

  \[ \Psi \vdash i : \Gamma_1 \rightarrow \Gamma_2 \]

- \( \Gamma_1 \) describes the registers on input to the instruction (a typing precondition)

- \( \Gamma_2 \) describes the registers on output (a typing postcondition)

- \( \Psi \) is invariant. The types of heap objects will not change as the program executes (at least for now,...).
Typing Instructions

- Arithmetic operations:

\[
\Psi; \Gamma \vdash r_s : \text{int} \quad \Psi; \Gamma \vdash v : \text{int} \\
\Psi \vdash \text{aop} r_d, r_s, v : \Gamma \rightarrow \Gamma[r_d := \text{int}]
\]

- Conditional branches:

\[
\Psi; \Gamma \vdash r : \text{int} \quad \Psi; \Gamma \vdash v : \Gamma \rightarrow \{ \} \\
\Psi \vdash \text{bop} r, v : \Gamma \rightarrow \Gamma
\]

- Data movement:

\[
\Psi; \Gamma \vdash v : \tau \\
\Psi \vdash \text{mov} r, v : \Gamma \rightarrow \Gamma[r_d := \tau]
\]
Basic Block Typing

- All basic blocks end in the jump instruction:

\[
\Psi; \Gamma \vdash v : \Gamma \rightarrow \{ \} \\
\Psi \vdash \text{jmp} \, v : \Gamma \rightarrow \{ \}
\]

Since a \text{jmp} never returns/falls through to the following instruction, we may choose the return context arbitrarily. For simplicity, we choose \{ \} and make that the return context for all blocks.

- Instruction sequences:

\[
\Psi \vdash i : \Gamma_1 \rightarrow \Gamma_2 \quad \Psi \vdash B : \Gamma_2 \rightarrow \{ \} \\
\Psi \vdash i; B : \Gamma_1 \rightarrow \{ \}
\]

- Subtyping is an admissible rule for basic blocks:

**Lemma: Admissibility of Basic Block Subtyping** If \( \Psi \vdash B : \Gamma_2 \rightarrow \{ \} \) and \( \Gamma_1 \leq \Gamma_2 \) then \( \Psi \vdash B : \Gamma_1 \rightarrow \{ \} \).

**Proof:** By induction on the typing derivation for basic blocks and instructions.
Machine Typing

- Heap typing:

\[
\begin{align*}
\text{Dom}(H) &= \text{Dom}(\Psi) & \forall L \in \text{Dom}(H). \Psi \vdash H(L) : \Psi(L) \\
\vdash H : \Psi
\end{align*}
\]

- Register file typing:

\[
\begin{align*}
\forall r \in \text{Dom}(\Gamma). \Psi; \{\} & \vdash R(r) : \Gamma(r) \\
\Psi & \vdash R : \Gamma
\end{align*}
\]

- Machine typing:

\[
\begin{align*}
\vdash H : \Psi & \quad \Psi \vdash R : \Gamma & \quad \Psi \vdash B : \Gamma \to \{} \\
\vdash (H, R, B)
\end{align*}
\]
Type Safety

We have designed the type system so that it satisfies the following property:

- **Theorem: Type Safety.** If $\vdash \Sigma$ and $\Sigma \rightarrow^* \Sigma'$ then $\Sigma$ is not stuck.

Proof by induction on the length of the instruction sequence, following Wright and Felleisen [26] and Harper [7].

- (Preservation) Each step in evaluation preserves typing.
- (Progress) If a state is well-typed then it is not stuck.

Corollaries:

- All jumps are to valid labels (control-flow safety)
- All arithmetic is done with integers (not labels)
Proof: Canonical Forms

Before proving Progress and Preservation, we must be able to characterize the shape and properties of a value based upon its type.

Lemma: Canonical Forms. If $\vdash H : \Psi$ and $\Psi \vdash R : \Gamma$ and $\Psi ; \Gamma \vdash v : \tau$ then

- $\tau = \text{int}$ implies $R(v) = n$.

- $\tau = \{r_1 : \tau_1, \ldots, r_n : \tau_n\} \rightarrow \{\}$ implies $R(v) = L$.

Moreover, $H(L) = B$ and $\Psi \vdash B : \{r_1 : \tau_1, \ldots, r_n : \tau_n\} \rightarrow \{\}$

Proof: By induction on the value typing derivation. [Exercise: fill in the details.]
Proof: Progress

**Lemma: Progress.** If $\vdash \Sigma_1$ then there exists a $\Sigma_2$ such that $\Sigma_1 \rightarrow \Sigma_2$.

Proof: By cases on the form of the code block in $\Sigma_1$.

Example case: $\Sigma_1 = (H, R, \text{jmp \ } v)$. We are given the derivation:

\[
\begin{array}{c}
\vdash H : \Psi \\
\vdash R : \Gamma \\
\Psi \vdash \text{jmp \ } v : \Gamma \rightarrow \{ \}
\end{array}
\]

\[
\vdash (H, R, \text{jmp \ } v)
\]

By inspection of the typing rules for blocks, the third premise above must be a derivation that ends in the jump rule:

\[
\begin{array}{c}
\Psi ; \Gamma \vdash v : \Gamma \\
\vdash \text{jmp \ } v : \Gamma \rightarrow \{ \}
\end{array}
\]

By Canonical Forms, $R(v) = L$ and $L \in \text{Dom}(H)$. Therefore, the operational rule for jumps applies and $\Sigma_1$ is not stuck: $(H, R, \text{jmp \ } v) \rightarrow (H, R, H(L))$
Proof: Preservation

**Lemma: Preservation.** If $\vdash \Sigma_1$ and $\Sigma_1 \rightarrow \Sigma_2$ then $\vdash \Sigma_2$.
Proof: By cases on the form of $\Sigma_1$.

Example case: $\Sigma_1 = (H, R, \text{jmp } v)$. We are given the derivation:

$$
\begin{array}{c}
\vdash H : \Psi \\
\Psi \vdash R : \Gamma \\
\Psi \vdash \text{jmp } v : \Gamma \rightarrow \{ \}
\end{array}
$$

$$\vdash (H, R, \text{jmp } v)$$

and the operational rule must be:

$$(H, R, \text{jmp } v) \rightarrow (H, R, B)$$

where $R(v) = L$ and $H(L) = B$

Hence, we must prove that $\vdash (H, R, B)$. As in the proof of Progress, we may deduce that the third premise of the typing derivation ends in an application of the jump rule:

$$
\begin{array}{c}
\Psi ; \Gamma \vdash v : \Gamma \rightarrow \{ \}
\end{array}
$$

$$\Psi \vdash \text{jmp } v : \Gamma \rightarrow \{ \}$$

Therefore, by Canonical Forms, we know

$$\Psi \vdash B : \Gamma \rightarrow \{ \}$$

and hence

$$
\begin{array}{c}
\vdash H : \Psi \\
\Psi \vdash R : \Gamma \\
\Psi \vdash B : \Gamma \rightarrow \{ \}
\end{array}
$$

$$\vdash (H, R, B)$$
Proof Summary

- The Type Safety theorem is relatively straightforward to prove using Canonical Forms, Progress and Preservation lemmas.

- Proofs almost always reveal flaws in initial design and clearly specify the properties that the language enforces.

- As we scale the programming language up, these proof techniques are remarkably robust. However, the proofs quickly become very detailed and tedious.

- **Open research problem**: How can we automate generation of these proofs? Some initial results from Schürmann and Pfenning [17, 14].
Scaling It up

The simple abstract machine and type system can be scaled up in many directions:

- more primitive types and options (e.g., floats, jal, complex instruction set operations, etc.) [20]
- a control stack for procedures [12]
- more polymorphism [13]
- a module system, link checker and dynamic linker [5]
- memory-allocated values (e.g., tuples and arrays) and explicit memory management [24, 19, 25, 23]
- objects for object-oriented programming [4]
- types for concurrency control
- dependent types for expressing more complex access control and security properties[22, 27]
- intentional type analysis [3, 2]

Over the next few lectures we will work through many of these topics.
Outline

- TAL-0: Assembly Language Control Flow and Basic Types
- TAL-1: Parametric Polymorphism
- TAL-2: Stack Types
- Type-Directed Compilation: From Tiny to TAL-2
- TAL-3: Data Structures
- TAL-4: Dependency
- TAL-5: Modularity and Linking
- References
TAL-1: Polymorphism

- Changes to types:
  - Add type variables to types: $\alpha$
    * Type variables are treated abstractly
    * Allow code reuse
    * As we’ll see they come in handy elsewhere...
  - Label types can be polymorphic:

    $\forall \alpha, \beta. \{ r_1 : \alpha, r_2 : \beta, r_3 : \{ r_1 : \beta, r_2 : \alpha \} \rightarrow \{ \} \} \rightarrow \{ \}$

    * Describes a function that swaps the values in registers $r_1$ and $r_2$, for values of any two types.
    * Register $r_3$ contains the return address which expects the values to be swapped.

- Changes to operands:
  - To jump to polymorphic functions, we explicitly instantiate type variables, calling for a new form of operand: $v[\tau]$
  - We write $v[\tau_1, \ldots, \tau_n]$ for $v[\tau_1] \cdots [\tau_n]$. 
Example Polymorphism

\[
\begin{align*}
\text{swap: } & \forall \alpha, \beta. \{ r_1 : \alpha, r_2 : \beta, r_{31} : \{ r_1 : \beta, r_2 : \alpha \} \rightarrow \{ \} \} \rightarrow \{ \} \\
& \text{mov } r_3, r_1 \quad \% \quad \{ r_1 : \alpha, r_2 : \beta, r_{31} : \{ r_1 : \beta, r_2 : \alpha \} \rightarrow \{ \}, r_3 : \alpha \} \\
& \text{mov } r_1, r_2 \\
& \text{mov } r_2, r_3 \\
& \text{jmp } r_{31}
\end{align*}
\]

\[
\begin{align*}
\text{swap\_ints: } & \{ r_1 : \text{int}, r_2 : \text{int}, r_{31} : \{ r_1 : \text{int}, r_2 : \text{int} \} \rightarrow \{ \} \} \rightarrow \{ \} \\
& \text{jmp swap[int, int]}
\end{align*}
\]

\[
\begin{align*}
\text{swap\_int\_and\_label: } & \{ r_1 : \text{int}, r_2 : \{ r_2 : \text{int} \} \rightarrow \{ \} \} \rightarrow \{ \} \\
& \text{mov } r_{31}, L \\
& \text{jmp swap[int, \{ r_2 : \text{int} \} \rightarrow \{ \}]}
\end{align*}
\]

\[
\begin{align*}
\text{L: } & \{ r_1 : \{ r_2 : \text{int} \} \rightarrow \{ \}, r_2 : \text{int} \} \rightarrow \{ \} \\
& \text{jmp } r_1
\end{align*}
\]
Callee-Saves Registers

• A common register-allocation strategy:
  
  – When calling a function, save the contents of some registers (caller-saves registers) onto the stack. When the function returns, restore the contents of these registers from the stack.

  – Allow the callee to save (and restore) the contents of other designated registers (callee-saves registers).

  – If the callee does not use all registers, the cost of saving and restoring is not incurred.

• Correctness criterion: the callee must return to the caller with the same values in the callee-saves registers
Callee-saves Registers Example

\[\text{callee: } \forall \alpha. \{r_1 : \text{int}, r_5 : \alpha, r_{31} : \{r_1 : \text{int}, r_5 : \alpha\} \rightarrow \{\}\} \rightarrow \{\}\]
\[\text{mov } r_4, r_5 \quad \% \text{ save register } r_5\]
\[\text{mov } r_5, 7 \quad \% \text{ use register } r_5 \text{ for other work}\]
\[\text{add } r_1, r_1, r_5\]
\[\text{mov } r_5, r_4 \quad \% \text{ restore register } r_5\]
\[\text{jmp } r_{31}\]

\[\text{caller: } \text{mov } r_5, 255 \quad \% \text{ will need } r_5 \text{ callee returns}\]
\[\text{mov } r_1, 5\]
\[\text{mov } r_{31}, L\]
\[\text{jmp callee[\text{int}]} \quad \% \text{ callee[\text{int}]} :\]
\[
\quad \% \quad \{r_1 : \text{int}, r_5 : \text{int}, r_{31} : \{r_1 : \text{int}, r_5 : \text{int}\} \rightarrow \{\}\}\]

\[L: \quad \{r_1 : \text{int}, r_5 : \text{int}\} \rightarrow \{\}\]
\[\text{mul } r_3, r_1, r_5\]
\[\ldots\]

\[\text{\text{\ldots}}\]
Callee-saves Registers Bug

\[\text{callee: } \forall \alpha. \{r_1 : \text{int}, r_5 : \alpha, r_{31} : \{r_1 : \text{int}, r_5 : \alpha\} \rightarrow \{ \} \} \rightarrow \{ \} \]
\[\text{mov } r_4, r_5\]
\[\text{mov } r_5, 7\]
\[\text{add } r_1, r_1, r_5\]
\[\text{jmp } r_{31} \quad \% \text{ ERROR! } r_5 : \text{int}\]

\[\text{caller: } \text{mov } r_5, 255\]
\[\text{mov } r_1, 5\]
\[\text{mov } r_{31}, L\]
\[\text{jmp callee[\text{int}] }\]

\[L : \quad \{r_1 : \text{int}, r_5 : \text{int}\} \rightarrow \{ \}\]
\[\text{mul } r_3, r_1, r_5\]
\[
\ldots
\]

- We can actually prove formally that \textit{callee} preserves the values of its callee-saves registers. This fact is a property of \textit{callee}'s polymorphic type! (See Reynolds [15] and Crary [1])

- Moral: polymorphism can be used for more than just code reuse. It can force a procedure to "behave well" in some circumstances.
Operational Semantics

- In order to prove our Type Preservation result, we must make a couple of minor changes in our operational semantics.

  - Heaps $H$ now map labels to type-labeled blocks:
    \[ H(L) = \forall \alpha_1, \ldots, \alpha_n. \Gamma \rightarrow \{ \}. B \]

  - Type variables $\alpha_1, \ldots, \alpha_n$ appear free both in $\Gamma$ and $B$

  - Control-flow operations substitute arguments types for type variables:

    \[
    (H, R, \text{jmp } v[\tau_1, \ldots, \tau_n]) \mapsto (H, R, B[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n])
    \]

    where $R(v) = L$ and $H(L) = \forall \alpha_1, \ldots, \alpha_n. \Gamma \rightarrow \{ \}. B$

    \[
    (H, R, \text{beq } r, v[\tau_1, \ldots, \tau_n]; B) \mapsto (H, R, B'[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n])
    \]

    where $R(r) = 0$, $R(v) = L$, and $H(L) = \forall \alpha_1, \ldots, \alpha_n. \Gamma \rightarrow \{ \}. B'$
Polymorphic Typing

- Since types may now contain variables, we must ensure they only contain properly declared variables. The following judgment states that a type is well-formed (i.e., it makes sense):

\[
\frac{\text{FreeVars}(\tau) \subseteq \Delta}{\Delta \vdash \tau}
\]

where \( \Delta = \alpha_1, \ldots, \alpha_n \)

- We also modify the operand and instruction typing judgments to account for the type variables in scope:

\[
\Psi; \Delta; \Gamma \vdash v : \tau
\]

\[
\Psi; \Delta \vdash i : \Gamma_1 \to \Gamma_2
\]
Polymorphic Typing

- We have a typing rule for our new sort of operand

\[
\frac{\Psi; \Delta; \Gamma \vdash v : \forall \alpha_1, \alpha_2, \ldots, \alpha_n. \Gamma \rightarrow \{ \} \quad \Delta \vdash \tau}{\Psi; \Delta; \Gamma \vdash v[\tau] : (\forall \alpha_2, \ldots, \alpha_n. \Gamma \rightarrow \{ \})[\tau/\alpha_1]} \]

- We change heap typing slightly in order to introduce the bound type variables:

\[
\forall L \in \text{Dom}(H). \Psi; \alpha_1, \ldots, \alpha_n \vdash B : \Gamma \rightarrow \{ \} \quad H(L) = \forall \alpha_1, \ldots, \alpha_n. \Gamma. B \quad \text{(for all } L) \\
\Psi(L) = \forall \alpha_1, \ldots, \alpha_n. \rightarrow \{ \} \\
\vdash H : \Psi
\]
Type Safety

- The type safety proof follows the same Progress and Preservation formula as before.

- We need one central addition to the proof: The Substitution Lemma.

  If $\Psi; \alpha_1, \ldots, \alpha_n \vdash B : \Gamma \rightarrow \{\}$ and $\vdash \tau_i$ for $i = 1..n$ then $\Psi; \cdot \vdash B[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n] : \Gamma[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n] \rightarrow \{\}$

- Exercise: Prove the Substitution Lemma and Preservation for TAL-1.
Outline

- TAL-0: Assembly Language Control Flow and Basic Types
- TAL-1: Parametric Polymorphism
- TAL-2: Stack Types
- Type-Directed Compilation: From Tiny to TAL-2
- TAL-3: Data Structures
- TAL-4: Dependency
- TAL-5: Modularity and Linking
- References
The Run-time Stack

- Almost every compiler uses a stack
  - A consecutive sequence memory addresses with one end designated the top
  - Values are stored on the stack and later retrieved
  - The compiler can grow the stack to store more values and later shrink the stack, explicitly deallocating the topmost values.

- Uses:
  - To store temporary values/result of intermediate computations when we run out of registers
  - To store the return address and local variables of recursive functions before a recursive function call.
TAL-2: Add a stack

- Machine states:
  - $M ::= (H, R, S, B)$

- Stacks are modelled as a list of values:
  - $S ::= \text{nil} \mid v :: S$

- New instructions:
  - $i ::= \text{salloc} \ n \mid \text{sfree} \ n \mid \text{sld} \ rd, n \mid \text{sst} \ rs, n$

- Error conditions:
  - If we free too much or read/write locations too deep in the stack, the machine will get stuck.
Remarks

- The stack operations have a 1-to-1 correspondence with RISC instructions.

- A designated register $sp$ points to the top of the stack.
  - $\text{salloc}$ corresponds to subtracting $n$ from a stack-pointer register (e.g. $\text{sub } sp, sp, n$)
  - $\text{sfree}$ corresponds to adding $n$ to the stack pointer (e.g. $\text{add } sp, sp, n$)
  - $\text{sst}$ corresponds to writing a value into offset $n$ from the stack pointer (e.g. $\text{st } sp(n), r$)
  - $\text{sld}$ corresponds to reading a value from offset $n$ relative to the stack pointer (e.g. $\text{ld } r, sp(n)$)

- CISC-like instructions (e.g. push/pop) can be synthesized.
  - $\text{push } v = \text{salloc } 1; \text{sst } v, 1$
  - $\text{pop } r = \text{sld } r, 1; \text{sfree } 1$
Simple Stack-Based Program

- A recursive version of the factorial function:

\[
\text{factrec}(n) = \\
\quad \text{if } n \leq 0 \text{ then } 1 \\
\quad \text{else } n \times \text{factrec } (n - 1)
\]

\text{factrec}: \% r_1 \text{ holds argument } n, r_{31} \text{ holds return address} \\
\% \text{ which expects the result in } r_1

\begin{align*}
\text{bgt } r_1, L1 & \quad \% n > 0, \text{ goto } L1 \\
\text{mov } r_1, 1 & \\
\text{jmp } r_{31} & \quad \% n \leq 0, \text{ return } 1
\end{align*}

\text{L1: } \text{salloc } 2 \quad \% \text{ allocate space for frame} \\
\text{sst } r_{31}, 1 \quad \% \text{ save return address} \\
\text{sst } r_1, 2 \quad \% \text{ save } n \\
\text{sub } r_1, r_1, 1 \quad \% n := n - 1 \\
\text{mov } r_{31}, RA \quad \% \text{ return address } := RA \\
\text{jmp } \text{factrec} \quad \% \text{ do recursive call, result in } r_1

\text{RA: } \% \text{ result in } r_1
\begin{align*}
\text{sld } r_2, 2 & \quad \% \text{ restore } n \text{ into } r_2 \\
\text{sld } r_{31}, 1 & \quad \% \text{ restore return address} \\
\text{mul } r_1, r_1, r_2 & \quad \% \text{ result } := n \times \text{fact}(n - 1) \\
\text{jmp } r_{31} & \quad \% \text{ return}
\end{align*}
Semantics for Stack Operations

- As before, the operational semantics maps machine states to machine states.

- After a sequence of new locations have been allocated at the top of the stack, they will initially be filled with garbage.
  
  - The junk value \(?\) models uninitialized/garbage stack slots.
  
  - It is introduced exclusively for the operational semantics. Programmers will not manipulate junk.

\[(H, R, S, \text{salloc } n; B) \rightarrow (H, R, \underbrace{\text{?} :: \cdots :: \text{?}}_{n} :: S, B)\]

\[(H, R, v_1 :: \cdots :: v_n :: S, \text{sfree } n; B) \leftrightarrow (H, R, S, B)\]

\[(H, R, S, \text{sldr } n; B) \rightarrow (H, R[r := v_n], S, B)\]
where \(S = v_1 :: \cdots :: v_n :: S'\)

\[(H, R, S_1, \text{sstr } n; B) \leftrightarrow (H, R, S_2, B)\]
where \(S_1 = v_1 :: \cdots :: v_{n-1} :: v_n :: S'\)
and \(S_2 = v_1 :: \cdots :: v_{n-1} :: R(r) :: S'\)
Typing the Stack

- Stack types:
  \[ \sigma ::= \text{nil} \mid \tau :: \sigma \mid \rho \]

- The \text{nil} type represents the empty stack.

- The type \( \tau :: \sigma \) represents a stack \( v :: S \) where \( \tau \) is the type of \( v \) and \( \sigma \) is the type of \( S \).

- The type \( \rho \) is a stack type variable that describes some unknown "tail" in the stack.

- Register file types contain a special register \( sp \) that is mapped to the type of the current stack:
  \[ \{ sp : \text{int} :: \rho, r_1 : \text{int}, \ldots \} \]

- In addition, we’ll let label types be polymorphic over stack types:
  \[ \forall \rho. \{ sp : \text{int} :: \rho, r_1 : \text{int} \} \rightarrow \{ \} \]

- Type contexts may now contain stack variables:
  \[ \Delta ::= \cdot \mid \Delta, \alpha \mid \Delta, \rho \]

- Junk values have junk type: \( ? \)
Stack Instruction Typing

As before, instruction typing judgments have the form

$$\Psi; \Delta \vdash i : \Gamma_1 \rightarrow \Gamma_2$$

- Stack allocation:

$$\Psi; \Delta \vdash \text{salloc } n : \Gamma[sp := \sigma] \rightarrow \Gamma[sp := ? :: \cdots :: ? :: \sigma]$$

- Stack free:

$$\Psi; \Delta \vdash \text{sfree } n : \Gamma[sp := \tau_1 :: \cdots :: \tau_n :: \sigma] \rightarrow \Gamma[sp := \sigma]$$

- Stack load:

$$\Gamma(sp) = \tau_1 :: \cdots :: \tau_n :: \sigma$$

$$\Psi; \Delta \vdash \text{sld } r, n : \Gamma \rightarrow \Gamma[r := \tau_n]$$

- Stack store:

$$\Psi; \Delta; \Gamma \vdash v : \tau$$

$$\Gamma(sp) = \tau_1 :: \cdots :: \tau_n :: \sigma$$

$$\Psi; \Delta \vdash \text{sst } v, n : \Gamma \rightarrow \Gamma[sp := \tau_1 :: \cdots :: \tau :: \sigma]$$
Typing Factrec (Bug)

type $\tau_\rho = \{r_1 : \text{int}, sp : \rho\} \rightarrow \{\}$

$\text{factrec} : \forall \rho. \{sp : \rho, r_1 : \text{int}, r_{31} : \tau_\rho\} \rightarrow \{\}$

- `bgt r_1, L1[\rho]`
- `mov r_1, 1`
- `jmp r_{31}`

$L1: \quad \forall \rho. \{sp : \rho, r_1 : \text{int}, r_{31} : \tau_\rho\} \rightarrow \{\}$

- `salloc 2` % $sp : ? :: ? :: \rho$
- `sst r_{31}, 1` % $sp : \tau_\rho :: ? :: \rho$
- `sst r_1, 2` % $sp : \tau_\rho :: \text{int} :: \rho$
- `sub r_1, r_{31}, 1`
- `mov r_{31}, RA[\rho]` % $r_{31} : \{sp : \tau_\rho :: \text{int} :: \rho, r_1 : \text{int}\} \rightarrow \{\}$
- `jmp factrec[\tau_\rho :: \text{int} :: \rho]`

$RA: \quad \forall \rho. \{sp : \tau_\rho :: \text{int} :: \rho, r_1 : \text{int}\} \rightarrow \{\}$

- `sld r_2, 2` % $r_2 : \text{int}$
- `sld r_{31}, 1` % $r_{31} : \tau_\rho$
- `mul r_1, r_1, r_2`
- `jmp r_{31}` % ERROR! $sp : \tau_\rho :: \text{int} :: \rho$
Typing Factrec Corrected

\[
type \ \tau_\rho = \{ r_1 : \text{int, } sp : \rho \} \rightarrow \{ \} \]

\[
factrec : \forall \rho. \{ sp : \rho, r_1 : \text{int}, r_{31} : \tau_{\rho} \} \rightarrow \{ \} \\
  \text{bgt } r_1, L1[\rho] \\
  \text{mov } r_1, 1 \\
  \text{jmp } r_{31} \\
L1: \ \forall \rho. \{ sp : \rho, r_1 : \text{int}, r_{31} : \tau_{\rho} \} \rightarrow \{ \} \\
  \text{salloc } 2 \\
  \text{sst } r_{31}, 1 \\
  \text{sst } r_1, 2 \\
  \text{sub } r_1, r_1, 1 \\
  \text{mov } r_{31}, RA[\rho] \\
  \text{jmp } factrec[\tau_\rho :: \text{int} :: \rho] \\
RA: \ \forall \rho. \{ sp : \tau_\rho :: \text{int} :: \rho, r_1 : \text{int} \} \rightarrow \{ \} \\
  \text{sld } r_2, 1 \% r_2 : \text{int} \\
  \text{sld } r_{31}, 2 \% r_{31} : \tau_\rho \\
  \text{mul } r_1, r_1, r_2 \\
  \text{sfree } 2 \% sp : \rho \\
  \text{jmp } r_{31} \]
Another Example

- The callee can’t mess with the caller’s stack frame:

```plaintext
caller: ∀ρ'. { sp : τ_code :: ρ' } → { }  
salloc
mov r1, 17
sst r1, 1
mov r31, RA[ρ']
jmp callee[τ_code :: ρ']
callee: ∀ρ. { sp : int :: ρ, r31 : { sp : ρ, r1 : int } → { } } → { }  
sld r1, 1
add r1, r1, r1
sst r1, 2  % ERROR!
sfreel
jmp r31

RA:  ∀ρ'. { sp : τ_code :: ρ', r1 : int } → { }  
...
```

- Polymorphism protects the stack.
The Theorems Carry Over

- Typing ensures we don’t get stuck.
  - e.g. try to write off the end of the stack
  - But it doesn’t ensure the stack stays within some quota

- With a bit more complication, we can deal with exceptions (See Morrisett et al. [12])
Things to Note

- We didn’t have to bake in a notion of procedure call/return. Jumps were good enough.
  - Side effect: tail calls are a non-issue.

- Polymorphism and polymorphic recursion are crucial for encoding standard procedure call/return.

- When combined with the callee-saves trick, we can code up calling conventions.
  - Arguments on stack or in registers?
  - Results on stack or in registers?
  - Return address? Caller pops? Callee pops?
  - Caller saves? Callee saves?

- It’s the orthogonal combination of typing features that makes things scale well.
Values of Different Size

- In high-level languages such as ML, all values have uniform size
  - The natural native representations of high-level values may have different sizes (64-bit floats vs. 32-bit integers).
  - To handle the size mismatch, an ML compiler will box floating-point values (represent them as a 32-bit pointer to a float).
- In low-level languages, we must handle values with non-uniform size.
  - There is no assembly language compiler to insert boxing coercions!
  - We must know how much space a value takes up on the stack so the type checker can verify that stack access is done properly.
  - We must know which values are small enough to fit into (32-bit) registers.
  - In summary, we need a function that computes the size of an object with type \( \tau \):
    \[
    \begin{align*}
    \text{size}(\text{int}) &= 1 \\
    \text{size}(\text{float}) &= 2 \\
    \text{size}(\forall \alpha_1, \ldots, \alpha_n. \Gamma \rightarrow \{ \}) &= 1 \\
    \text{size}(?_{32}) &= 1 \\
    \text{size}(?_{64}) &= 2
    \end{align*}
    \]
  - But how do we compute the size of an abstract type \( \alpha \)?
Kinds and Types

- Solution: we classify all types according to the size of the objects that inhabit them.

- Generally, when we need to establish properties of types, we will use a system of *kinds*

- Kinds classify types just as types classify expressions.

- Here, a kind can specify the size of the values in a particular type:

  \[ \kappa ::= \text{Sz}(i) \mid T \]

- Type contexts \( \Delta \) map type variables to their kinds:

  \[ \Delta ::= \cdot \mid \Delta, \alpha :: \kappa \]
Kinds and Types

- A judgment assigns each type a kind that reflects its size:

\[
\Delta \vdash \text{int} :: \text{Sz}(1) \quad \Delta \vdash \text{float} :: \text{Sz}(2)
\]

\[
\Delta \vdash \text{nil} :: \text{Sz}(0) \quad \Delta \vdash \tau :: \text{Sz}(i) \quad \Delta \vdash \sigma :: \text{Sz}(j) \\
\Delta \vdash (\tau :: \sigma) :: \text{Sz}(i + j)
\]

\[
\Delta, \alpha :: \kappa \vdash \alpha :: \kappa
\]

\[
\Delta \vdash \tau :: \text{Sz}(i) \quad \Delta \vdash \tau :: \text{T} \quad \Delta \vdash \tau :: \sigma :: \text{T} \\
\Delta \vdash \tau :: \text{T} \quad \Delta \vdash \tau :: \sigma :: \text{T}
\]

- Modified stack load:

\[
\Gamma(sp) = \tau_1 :: \cdots :: \tau_m :: \sigma
\]

\[
\Delta \vdash (\tau_1 :: \cdots :: \tau_{m-1} :: \text{nil}) :: \text{Sz}(n - 1) \quad \Delta \vdash \tau_m :: \text{Sz}(1)
\]

\[
\Psi; \Delta \vdash \text{sd} r, n : \Gamma \rightarrow \Gamma[r := \tau_m]
\]

- The load selects object \( m \) off the stack
- That object must fit inside a register (have kind \( \text{Sz}(1) \))

- \texttt{x86 fld} (load value onto floating point stack) will be similar but require the object have kind \( \text{Sz}(2) \)
Outline

- TAL-0: Assembly Language Control Flow and Basic Types
- TAL-1: Parametric Polymorphism
- TAL-2: Stack Types
- Type-Directed Compilation: From Tiny to TAL-2
- TAL-3: Data Structures
- TAL-4: Dependency
- TAL-5: Modularity and Linking
- References
Certified Code Systems

• A complete system for certified code contains three parts:
  – A strongly-typed source programming language.
  – A type-preserving compiler.
  – A strongly-typed target language.

• TAL will serve as our target language

• In this lecture, we will
  – Develop a very simple strongly-typed source language.
  – Explore the compilation process.
Source language: Tiny

- A simply-typed functional language.
  - Integer expressions
  - Conditionals
  - Recursive functions
  - Function pointers (no closures)
  - A strong type system

- An example program:

```tiny
letrec
  fun fact (n:int) : int =
    if n = 0 then 1 else n * fact(n - 1)
  in
  fact 6
```
Tiny Syntax

- Types:

\[ \tau ::= \text{int} \mid \tau_1 \rightarrow \tau_2 \]

- Expressions:

\[ e ::= x \mid f \mid n \mid e_1 + e_2 \mid e_1 \cdot e_2 \mid \text{if } e_1 = 0 \text{ then } e_2 \text{ else } e_3 \mid \text{let } x = e_1 \text{ in } e_2 \]

- Function declarations:

\[ d ::= \text{fun } f(x:\tau_1) : \tau_2 = e \]

- Programs:

\[ P ::= \text{letrec } d_1 \cdots d_n \text{ in } e \]
A Tiny Type System

- Type checking occurs in a context $\Phi$ which maps function variables $f$ and expression variables $x$ to types

Expressions:

$$\Phi \vdash x : \Phi(x)$$

$$\Phi \vdash f : \Phi(f)$$

$$\Phi \vdash n : \text{int}$$

$$\Phi \vdash e_1 : \text{int} \quad \Phi \vdash e_2 : \text{int} \quad \Phi \vdash e_1 + e_2 : \text{int}$$

$$\Phi \vdash e_1 : \tau_1 \to \tau_2 \quad \Phi \vdash e_2 : \tau_1 \quad \Phi \vdash e_1 e_2 : \tau_2$$

$$\Phi \vdash e_1 : \text{int} \quad \Phi \vdash e_2 : \tau \quad \Phi \vdash e_3 : \tau \quad \Phi \vdash \text{if } e_1 = 0 \text{ then } e_2 \text{ else } e_3 : \tau$$

$$\Phi \vdash e_1 : \tau_1 \quad \Phi, x:\tau_1 \vdash e_2 : \tau_2 \quad \Phi \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2$$
Typing Tiny Programs

Declarations:

\[
\Phi, x: \tau_1 \vdash e : \tau_2
\]

\[
\Phi \vdash \text{fun } f(x: \tau_1) : \tau_2 = e : (f: \tau_1 \rightarrow \tau_2)
\]

Programs:

\[
\Phi = f_1: \tau_{1,1} \rightarrow \tau_{1,2}, \ldots, f_n: \tau_{n,1} \rightarrow \tau_{n,2}
\]

\[
\Phi \vdash d_i : (f_i: \tau_{i,1} \rightarrow \tau_{i,2}) \quad \Phi \vdash e : \text{int}
\]

\[
\vdash \text{letrec } d_1 \ldots d_n \text{ inc e}
\]

- All Tiny programs return an integer as their final result
- Exercise: verify that the factorial program is well-typed
Type-Preserving Compilation

- A compiler for a realistic language normally consists of a series of type-preserving transformations
  - After each transformation, we can type check the code to help detect compilers.
- Every transformation in type-preserving compiler has two parts:
  - A type translation from source types to target types
  - A term translation from source types and terms to target terms
- The compiler described here is derived from the original implementation of our Popcorn compiler [20, 11].
The Type Translation

- The type translation \( T[\cdot] \) maps Tiny types to TAL types
- Integers:

\[
T[int] = int
\]

- Function types:
  - The translation of function types fixes the calling convention that the compiler will use.
    * The caller pushes the argument and then the return address onto the stack.
    * The callee pops the argument and return address. The result is placed in register \( r_a \).

\[
T[\tau_1 \rightarrow \tau_2] = \forall \rho. \{ sp : K[\tau_2, \rho] :: T[\tau_1] :: \rho \rightarrow \{ \}\}
\]

where

\[
K[\tau, \sigma] = \{ sp : \sigma, r_a : T[\tau] \rightarrow \{ \}\}
\]
Expression Translation

- To keep the translation simple, we will use the stack extensively:
  - The values of all expression variables are kept on the stack
    * $M$ maps expression variables to stack offsets
    * $I(M)$ increments the stack offset associated with each variable in the domain of $M$
  - To compute the value of an expression, we first compute the values of its subexpressions and push them on the stack.
  - We return the value of an expression in the register $r_a$

- In all, we use 3 registers and the stack

- The shape formal translation is $\mathcal{E}[e]_{M,\sigma} = J$ where $J$ is a sequence of labels (and their types) and instructions.

- For each function $f$, we assume there is a TAL label $L_f$

- $\mathsf{T}(e)$ is the source type of expression $e$
  - Technically, we should thread the Tiny typing context $\Phi$ through the translation to make it possible to construct the type of an expression $e$. For the sake of brevity, we elide this detail.
Expression Translation

• Expression variables:

\[ \mathcal{E}[[x]]_{M, \sigma} = \text{sl}d \ r_a, M(x) \]

• Function variables:

\[ \mathcal{E}[[f]]_{M, \sigma} = \text{mov} \ r_a, L_f \]

• Integer constants:

\[ \mathcal{E}[[n]]_{M, \sigma} = \text{mov} \ r_a, n \]

• Addition:

\[ \mathcal{E}[[e_1 + e_2]]_{M, \sigma} = \]
\[ \mathcal{E}[[e_1]]_{M, \sigma} \]
\[ \text{push} \ r_a \]
\[ \mathcal{E}[[e_2]]_{I(M), \text{int}::\sigma} \]
\[ \text{pop} \ r_t \]
\[ \text{add} \ r_a, r_t, r_a \]
Expression Translation

- Function Call:

\[ E[e_1 \ e_2]_{M,\sigma} = \]
\[ E[e_1]_{M,\sigma} \]
\[ \text{push } r_a \]
\[ E[e_2]_{I(M),T[\tau_1 \rightarrow \tau_2]::\sigma} \]
\[ \text{pop } r_t \]
\[ \text{push } r_a \]
\[ \text{push } L_r[\rho] \]
\[ \text{jmp } r_t[\sigma] \]
\[ L_r : \forall \rho.K[\tau_2, \sigma] \]

where \( T(e_1) = \tau_1 \rightarrow \tau_2 \)
and \( L_r \) is fresh

- Conditional:

\[ E[\text{if } e_1 = 0 \text{ then } e_2 \text{ else } e_3]_{M,\sigma} = \]
\[ E[e_1]_{M,\sigma} \]
\[ \text{bneq } r_a, L_{else}[\rho] \]
\[ E[e_2]_{M,\sigma} \]
\[ \text{jmp } L_{end}[\rho] \]
\[ L_{else} : \forall \rho.\{sp : \sigma\} \]
\[ E[e_3]_{M,\sigma} \]
\[ \text{jmp } L_{end}[\rho] \]
\[ L_{end} : \forall \rho.K[\tau, \sigma] \]

where \( T(e_2) = \tau \)
and \( L_{else}, L_{end} \) are fresh

- Exercise: Translate the let-expression
Program Translation

- Function translation:

\[
\mathcal{F}[\text{fun } f(x:\tau_1) : \tau_2 = e] = \\
L_f : \mathcal{T}[\tau_1 \rightarrow \tau_2] \\
\mathcal{E}[e]|_{x:=2}, \mathcal{K}[\tau_2, \rho] :: \mathcal{T}[\tau_1] :: \rho \\
\text{pop } r_t \\
\text{sfree l} \\
\text{jmp } r_t
\]

- Program translation:

\[
\mathcal{P}[\text{letrec } d_1 \ldots d_n \text{ in } e] = \\
\mathcal{F}[d_1] \\
\ldots \\
\mathcal{F}[d_n] \\
L_{\text{main}} : \forall \rho. \{ sp : \mathcal{K}[\text{int}, \rho] :: \rho \} \\
\mathcal{E}[e], \mathcal{K}[\text{int}, \rho] :: \rho \\
\text{pop } r_t \\
\text{jmp } r_t
\]

- To run the program, jump to \( L_{\text{main}} \) after pushing the return address on the stack.
- Expect the program result in register \( r_a \).
Example: Compiling Fact

- Recall the fact function in Tiny:

```plaintext
letrec
  fun fact (n:int) : int =
    if n = 0 then 1 else n * fact(n - 1)
in
  fact 6
```
Example: Compiling Fact

$L_{fact}$: $\forall \rho. \{ sp : \mathcal{K}[\text{int}] :: \text{int} :: \rho \}$

  sld $r_a, 2$ % load argument
  bneq $r_a, L_{else}[\rho]$ % $n = 0$?
  mov $r_a, 1$ % return 1
  jmp $L_{end}$

$L_{else}$: $\forall \rho. \{ sp : \mathcal{K}[\text{int}] :: \text{int} :: \rho \}$

  sld $r_a, 2$ % begin multiplication (load $n$)
  push $r_a$
  mov $r_a, L_{fact}$ % begin $\text{fact}$ call sequence
  push $r_a$
  sld $r_a, 4$ % begin subtraction (load $n$)
  push $r_a$
  mov $r_a, 1$
  pop $r_t$
  sub $r_a, r_t, r_a$ % $n - 1$
  pop $r_t$ % load $L_{fact}$
  push $L_r[\rho]$
  jmp $r_t[\text{int} :: \mathcal{K}[\text{int}, \rho] :: \text{int} :: \rho]$

$L_r$: $\forall \rho. \{ sp : \text{int} :: \mathcal{K}[\text{int}, \rho] :: \text{int} :: \rho, r_a :: \text{int} \}$

  pop $r_t$ % load $n$
  mul $r_a, r_t, r_a$ % $n \times fact(n - 1)$
  jmp $L_{end}[\rho]$

$L_{end}$: $\forall \rho. \{ sp : \mathcal{K}[\text{int}, \rho] :: \text{int} :: \rho, r_a :: \text{int} \}$

  pop $r_t$ % pop return address
  sfree $l$ % throw away argument
  jmp $r_t$ % return
Optimizations

- Almost any compiler will produce better code than ours!
  - But how many compilers can you fit on three slides?
- Our type system makes it possible to generate much better code and to implement many standard optimizations:
  - Instruction selection optimizations
  - Common subexpression elimination
  - Register allocation
  - Redundant load and store elimination
  - Instruction scheduling optimizations
  - Strength reduction
  - Loop-invariant removal
  - Tail-call optimizations
  - And others.
- As demonstrated by the TIL/TILT compilers, types do not interfere with most common optimizations [21]
Instruction Selection

- Design principal: instruction sequences with the same operational behavior should have the same static behavior.
  - Unattainable in general, but something to strive for.
- We can synthesize the typing rule for `push` from a stack allocation and store since `push v = alloc l; sst v, l`
  - First, we write down the typing rules for the sequence, specialized to specific operands:

\[
\frac{\Psi; \Delta \vdash \text{alloc} l : \Gamma[sp := \sigma] \rightarrow \Gamma[sp := ? :: \sigma]}{\Psi; \Delta \vdash \text{alloc} l; sst v, l : \Gamma[sp := \sigma] \rightarrow \Gamma[sp := \tau :: \sigma]} \quad D
\]

\[
\Psi; \Delta; \Gamma[sp := ? :: \sigma] \vdash v : \tau
\]

\[D = \Psi; \Delta \vdash sst v, l : \Gamma[sp := ? :: \sigma] \rightarrow \Gamma[sp := \tau :: \sigma] \]

- Then we extract the premises at the leaves of the derivation, removing the intermediate states:

\[
\frac{\Psi; \Delta; \Gamma[sp := ? :: \sigma] \vdash v : \tau}{\Psi; \Delta \vdash \text{push} v : \Gamma[sp := \sigma] \rightarrow \Gamma[sp := \tau :: \sigma]}
\]
Instruction Selection

- Since \texttt{push}\,v is statically equivalent to \texttt{salloc1;sst v, 1}, a compiler writer can always replace one with the other
  - To optimize instruction encoding size
  - To optimize execution efficiency
  - To enable other optimizations

- Example:

  \begin{verbatim}
  push 7
  push 8
  push 9
  \end{verbatim}

  Can be replaced by:

  \begin{verbatim}
  salloc 1
  sst 7, 1
  salloc 1
  sst 8, 1
  salloc 1
  sst 9, 1
  \end{verbatim}

  Which can be further reduced to:

  \begin{verbatim}
  salloc 3
  sst 7, 1
  sst 8, 1
  sst 9, 1
  \end{verbatim}
Tail-Call Optimizations

- A crucial optimization for functional languages
- Applies when the final operation in a function \( f \) is a function call to \( g \)
- Rather than have \( f \) push the return address and engage in the normal calling sequence, \( f \) will pop all of its temporary values and jump directly to \( g \), never to return
- Example:

Without tail-call optimization:

\[
L_f: \quad \ldots
\]

\[
\forall \rho. \{ sp : K[\tau_{\text{return}}, \rho] :: \tau_{f-\text{arg}} :: \rho, r_a : \tau_{g-\text{arg}} \} \rightarrow \{ \}
\]

salloc 2

sst \( L_r \) \hspace{1cm} \% \text{push return address}

sst \( r_a, 2 \) \hspace{1cm} \% \text{push argument}

jmp \( L_g[\tau_{\text{raddr}} :: \tau_{f-\text{arg}} :: \rho] \)

\[
L_r: \quad \forall \rho. \{ sp : \tau_{\text{raddr}} :: \tau_{f-\text{arg}} :: \rho, r_a : \tau_{\text{ret}} \} \rightarrow \{ \}
\]

pop \( r_t \) \hspace{1cm} \% \text{pop return address}

sfreel \hspace{1cm} \% \text{throw away } \( f \)'s \text{ argument}

jmp \( r_t \) \hspace{1cm} \% \text{return}

With tail-call optimization:

\[
L_f: \quad \ldots
\]

\[
\forall \rho. \{ sp : \tau_{\text{raddr}} :: \tau_{f-\text{arg}} :: \rho, r_a : \tau_{g-\text{arg}} \} \rightarrow \{ \}
\]

sst \( r_a, 2 \)

jmp \( L_g[\rho] \) \hspace{1cm} \% \text{ } g \text{ will return to } \( f \)'s \text{ caller}
What optimizations can’t we handle?

The version of TAL discussed so far provides no mechanisms for the following source of optimizations:

- Optimizations that alter the code stream: run-time code generation, run-time code optimization
  
  - Smith, Hornoff, Jim, and Morrisett have designed a system for safe run-time code generation (see Smith’s thesis [18])

- Various stack-allocation strategies
  
  - Our type system can’t represent pointers deep into the stack
  
  - Morrisett et al. [12] extend the stack typing discipline, but more work needs to be done here

- Optimizations that rely upon properties of values that are not reflected in the type structure:
  
  - Arithmetic properties of integers (eg: \( n = 17 \)), which are useful for reasoning about arrays and pointer arithmetic (coming in a following section)
  
  - Aliasing properties of pointers in heap-allocated data structures (coming in a following section)
Properties of the Compiler

- Our compiler is type-preserving:
  
  If $P$ is a well-typed Tiny program: $\vdash P$ then the compiled program is also well-typed: $\vdash \mathcal{P}[P] : \Psi$ for some $\Psi$.

- The proof would proceed by induction on the structure of the program $P$.

- Each optimization phase and compiler transformation respects this property.

- To detect errors in our compiler’s implementation we can run the compiler and type check the output.
Practical Compiler Issues

- As you translate from a high-level language to a low-level TAL-like language, the types must encode the structural information lost in the translation

- Result: by the time we have compiled to assembly, the types encode lots of data

- Careful engineering is required to enable efficient code size and type checking time
  
  - The Popcorn Compiler (PII266):
  
    - Object code: 0.55MB, 39 modules
  
    - Naive encoding: 4.50MB, checking time: 750s
  
    - Optimized encoding: 0.27MB, checking time: 22s

    - Checking time scales linearly with code size

    - Likely more optimization possible
Popcorn Example

- **Source Type:**

\[
\text{int} \rightarrow \text{bool}
\]

- **TAL Type:**

\[
\text{All } a:T, b:T, c:T, r1:S, r2:S, e1:C, e2:C. \\
\{\text{ESP: } \{\text{EAX: bool, M:e1+e2, EBX:a, ESI:b, EDI:c,} \\
\text{ESP:int::r1@}\{\text{EAX:exn,ESP:r2,M:e1+e2::r2}\::\text{int::r1@} \\
\text{ESP:exn,ESP:r2,M:e1+e2::r2,} \\
\text{EBP: sptr}\{\text{EAX:exn,ESP:r2,M:e1+e2::r2,} \\
\text{EBX:a, ESI:b, EDI:c, M:e1+e2}\}
\]

- Types for higher-order functions can require pages to write them down!
Compressing Types

- Gzip:
  - Effective for reducing binary size over the wire
  - No help during verification

- Tailor types to the language being compiled/the compiler
  - eg: fix the calling convention
  - Restricts interoperability/language and compiler evolution

- Higher-order type constructors
  - Fairly effective, useful for compiler debugging/code readability

- Hash-cons (ie: use graphs to represent types)
  - Highly effective, fast type equality
  - A significant engineering investment

- Type reconstruction/type inference
  - Can be very efficient with respect to both space and time
  - Must take care to avoid increasing trusted computing base

Summary of Type-Directed Compilation

- Type-directed and type-preserving compilation provides an \textit{automatic} way to generate certifiable low-level code
- We can prove that the compiler produces well-typed assembly code from any well-typed source language program
- Programmers can program as they normally do in their favorite strongly typed high-level language
- Constructing a type-preserving compiler takes more work initially but the result is more robust:
  - Compiler writers must transform both types and terms
  - Special care must be taken to compress type information
  - Type checking intermediate program representations can detect compiler errors
- Most conventional compiler optimizations are naturally type-preserving, so using a typed target language has little impact (if any) on compiler performance
Outline

- TAL-0: Assembly Language Control Flow and Basic Types
- TAL-1: Parametric Polymorphism
- TAL-2: Stack Types
- Type-Directed Compilation: From Tiny to TAL-2
- TAL-3: Data Structures
- TAL-4: Dependency
- TAL-5: Modularity and Linking
- References
Data Structures

• The register file and stack give us some local storage for word-sized values
  
  – Stack space can be recycled for values of different types
  – Critical trick: can’t create pointers to these values
  – The trick prevents code from seeing two different views of the stack (through different pointers/aliases). It is simple to ensure that the single view of the stack is accurate.

• What about aggregates?
  
  – eg: tuples, records, arrays, objects, datatypes, etc.
  – TAL puts these “large” values in the heap and refers to them via pointers.
  – This introduces aliasing and the potential for multiple views/access paths for the same data structure
  – Recycling heap memory is not as easy
TAL-3: Add Tuples

- Let heap $H$ map labels to either blocks of code or tuples of values: $\langle v_1, \ldots, v_n \rangle$
- The values $v_i$ are either integers or labels
- The labels are abstract (no pointer arithmetic)
- Tuple instructions:
  - Allocate tuple: $\text{malloc} \, r_d, n$
  - Load from $k^{th}$ component of the tuple: $\text{ld} \, r_d, r_s(k)$
  - Store into $k^{th}$ component of the tuple: $\text{st} \, r_d(k), r_s$
- Tuple types: $\langle \tau_1, \ldots, \tau_n \rangle$
Tuple Operational Semantics

- Allocation:

\[
(H, R, v_1 :: \cdots :: v_n :: S, \text{malloc}_{r_d, n}; B) \mapsto (H[L \equiv \langle v_1, \ldots, v_n \rangle], R[r_d := L], S, B)
\]
where \( L \) is a fresh label (ie: not in \( \text{Dom}(H) \))

- Load:

\[
(H, R, S, \text{ld}_{r_d, r_s(k)}; B) \mapsto (H, R[r_d := v_k], S, B)
\]
where \( H(R(r_s)) = \langle v_1, \ldots, v_n \rangle \) and \( 1 \leq k \leq n \)

- Store:

\[
(H[L \equiv \langle v_1, \ldots, v_n \rangle], R, S, \text{str}_{r_d(k), r_s}; B) \mapsto (H[L \equiv \langle v_1, \ldots, v_{k-1}, R(r_s), v_{k+1}, \ldots, v_n \rangle], R, S, B)
\]
where \( R(r_d) = L \)
Tuple Typing

- Allocation:

\[
\Gamma(sp) = \tau_1 : : \tau_2 : : \cdots : : \tau_n : : \sigma \\
\Psi; \Delta \vdash \text{malloc } r_d, n : \Gamma \rightarrow \Gamma[sp := \sigma, r_d := \langle \tau_1, \tau_2, \ldots, \tau_n \rangle]
\]

- Load:

\[
\Psi; \Delta; \Gamma \vdash r_s : \langle \tau_1, \ldots, \tau_n \rangle \quad 1 \leq k \leq n \\
\Psi; \Delta \vdash \text{l}d r_d, r_s(k) : \Gamma \rightarrow \Gamma[r_d := \tau_k]
\]

- Store:

\[
\Psi; \Delta; \Gamma \vdash r_d : \langle \tau_1, \ldots, \tau_n \rangle \\
\Psi; \Delta; \Gamma \vdash r_s : \tau_k \quad 1 \leq k \leq n \\
\Psi; \Delta \vdash \text{s}t r_d(k), r_s : \Gamma \rightarrow \Gamma
\]
Remarks

• The load and store operations correspond to conventional RISC instructions.

• The malloc instruction does not.
  – Typically, this would be implemented by a call into the run-time to atomically allocate and initialize the tuple.
  – Atomic allocation and initialization interferes with our ability to compile common C-style programming idioms
  – Interferes with instruction selection and scheduling
  – The advantage is a simple design where we need not reason about pointers and aliasing.

• There’s no way to explicitly deallocate heap memory
  – TAL relies upon a garbage collector to reclaim all heap storage.
  – Remember, the garbage collector is another element of our trusted computing base.

• The types of tuples are invariant.
  – You can’t update a component in the tuple with a value of a different type
  – The same is true for code and other heap objects

• In summary, TAL has the memory model of a high-level programming language
Arrays

- Hard issues:
  - Need to allocate and initialize storage of unknown size.
  - Each array subscript operation must be in bounds.
  - In general, this implies we need size information at run time.

- Simple solution: special operations:
  - `new_array r_a, r_size, r_item`
  - `asub r_item, r_a(r_i)`
  - `aupd r_a(r_i), r_item`
  - The disadvantage is that this fixes array representations and makes interoperation with other languages difficult/costly. There is some overhead to performing the array-bounds checks.
Outline

- TAL-0: Assembly Language Control Flow and Basic Types
- TAL-1: Parametric Polymorphism
- TAL-2: Stack Types
- Type-Directed Compilation: From Tiny to TAL-2
- TAL-3: Data Structures
- TAL-4: Dependency
- TAL-5: Modularity and Linking
- References
TAL-4: A Refined Memory Model

- Machine states now have the form \((H_U; H_M; S; R; B)\) where 
  \(H_M\) is memory managed explicitly by the TAL program

- In order to check programs that explicitly manage memory 
  (as most C programs do) we will reason about the shape 
  of memory using a simple logic

- \(C ::= \{\ell \mapsto (\tau_1, \ldots, \tau_n)\} \mid \mathbf{1} \mid C_1 \otimes C_2 \mid \epsilon\)

- \(\epsilon\) is a logic variable

- \(\ell\) is a label: either a label variable \(\phi\) or a concrete label \(L\)

- We also introduce a new type of managed pointers: \(S(\ell)\)
  
  - Only label \(L\) has type \(S(L)\)
  
  - When two labels have type \(S(\phi)\), we do not know 
    which labels they are, but we do know that they are 
    the same label (they are *aliases*)
Well-formed Stores

- The judgment $\Psi \vdash H : C$ states that a heap $H$ is well-formed and is described by the formula $C$.

- We specify a nondeterministic merge of two stores $H_1$ and $H_2$ using the notation $H_1 \bowtie H_2$. It requires that the domains of the stores $H_1$ and $H_2$ be disjoint.

\[ \Psi \vdash \{ \} : 1 \]

\[ \Psi \vdash H_1 : C_1 \quad \Psi \vdash H_2 : C_2 \]

\[ \Psi \vdash H_1 \bowtie H_2 : C_1 \otimes C_2 \]

\[ \Psi; \cdot \vdash v_i : \tau_i \quad \text{for } 1 \leq i \leq n \]

\[ \Psi \vdash \{L \mapsto \langle v_1, \ldots, v_n \rangle\} : \{L \mapsto \langle \tau_1, \ldots, \tau_n \rangle\} \]

- Example:

\[ L_1: \quad L_2: \quad L_3:\]

\[ \{L_1 \mapsto \langle S(L_3), S(L_2) \rangle\} \otimes \]

\[ \{L_2 \mapsto \langle S(L_3) \rangle\} \otimes \]

\[ \{L_3 \mapsto \langle \text{int} \rangle\} \]
Using Store Types

- New instructions:
  - `malloc φ, r, n`
  - `free r`
- Our old load and store instructions will have overloaded typing rules
- Code types are extended with an extra field to describe the shape the store must have before we jump to the code:
  - `{hp : C, sp : σ, r_1 : τ_1, ..., r_n : τ_n} → { }`
Examples

foo: $\forall \epsilon, \rho. \{ hp : \epsilon, sp : \rho, r_1 : \text{int},$
    \hspace{1cm} $r_{31} : \{ hp : \epsilon, sp : \rho, r_1 : \text{int} \} \rightarrow \{ \} \rightarrow \{ \}$
    \hspace{1cm} $\text{mmalloc } \phi, r_2, n \% \ hp : \epsilon \otimes \{ \phi \mapsto \langle ?, ? \rangle \}, r_2 : S(\phi)$
    \hspace{1cm} $\text{mov } r_7, r_2 \ % \ r_7 : S(\phi)$
    \hspace{1cm} $\text{st } r_7[1], r_1 \ % \ hp : \epsilon \otimes \{ \phi \mapsto \langle \text{int}, ? \rangle \}$
    \hspace{1cm} $\text{st } r_2[2], r_1 \ % \ hp : \epsilon \otimes \{ \phi \mapsto \langle \text{int}, \text{int} \rangle \}$
    \hspace{1cm} $\text{free } r_2 \ % \ hp : \epsilon$
    \hspace{1cm} $\text{jmp } r_{31}$

An error:

foo: $\forall \epsilon, \rho. \{ hp : \epsilon, sp : \rho, r_1 : \text{int},$
    \hspace{1cm} $r_{31} : \{ hp : \epsilon, sp : \rho, r_1 : \text{int} \} \rightarrow \{ \} \rightarrow \{ \}$
    \hspace{1cm} $\text{mmalloc } \phi, r_2, n \% \ hp : \epsilon \otimes \{ \phi \mapsto \langle ?, ? \rangle \}, r_2 : S(\phi)$
    \hspace{1cm} $\text{mov } r_7, r_2 \ % \ r_7 : S(\phi)$
    \hspace{1cm} $\text{st } r_7[1], r_1 \ % \ hp : \epsilon \otimes \{ \phi \mapsto \langle \text{int}, ? \rangle \}$
    \hspace{1cm} $\text{st } r_2[2], r_1 \ % \ hp : \epsilon \otimes \{ \phi \mapsto \langle \text{int}, \text{int} \rangle \}$
    \hspace{1cm} $\text{jmp } r_{31} \ % \ \text{ERROR! Memory leak.}$
Heap Logic: Details

- To type check code, we must use the entailment relation from our heap logic: $C \vdash C'$

- More generally, entailment has the form $L \vdash C$ where $L$ is a sequence of assumptions $C$

- This logic is a tiny fragment of linear logic and the sequent calculus rules follow.

\[
\frac{}{\vdash 1}
\]

\[
\frac{L, L' \vdash C}{L, 1, L' \vdash C}
\]

\[
\frac{L, C, C', L' \vdash C''}{L, C \otimes C', L' \vdash C''}
\]

\[
\frac{L \vdash C \quad L' \vdash C'}{L \otimes L' \vdash C \otimes C'}
\]

\[
\frac{\{\phi \mapsto (\tau_1, \ldots, \tau_n)\} \vdash \{\phi \mapsto (\tau_1, \ldots, \tau_n)\}}{\epsilon \vdash \epsilon}
\]

- These rules are sound with respect to our heap model and entailment is decidable. Prove these facts as an exercise.
Subtyping

- We fold the logic into our type system by extending the subtyping relation:

\[
\frac{C \vdash C'}{\Gamma[hp := C] \leq \Gamma[hp := C']}
\]
New Judgments and Block Typing

- Extended instruction typing judgment:

\[
\Psi; \Delta \vdash i : \Gamma \rightarrow [\Delta']\Gamma'
\]

- may be read as “given a managed heap type \(\Psi\) and the type variables \(\Delta\), instruction \(i\) has register file precondition \(\Gamma\) and there exist types \(\Delta'\) such that the postcondition \(\Gamma'\) will be satisfied upon execution of the instruction.

- The block typing judgment is as before:

\[
\Psi; \Delta \vdash B : \Gamma \rightarrow \{ \}
\]

- But the rules for stringing together instructions change slightly:

\[
\begin{align*}
\Psi; \Delta \vdash i : \Gamma \rightarrow [\Delta']\Gamma' & \quad \Psi; \Delta, \Delta' \vdash B : \Gamma' \rightarrow \{ \} \\
\Psi; \Delta \vdash i; B : \Gamma \rightarrow \{ \}
\end{align*}
\]

- The rule for typing jumps does not change, but remember that register file typings now contain more information (the type of the managed heap).

\[
\begin{align*}
\Psi; \Gamma \vdash v : \Gamma \rightarrow \{ \}
\end{align*}
\]

\[
\Psi \vdash \text{jmp} v : \Gamma \rightarrow \{ \}
\]
Instruction Typing Rules

\[
\Gamma(hp) = C \quad \Gamma' = \Gamma[hp := C \otimes \{\phi \mapsto \langle ?, \ldots, ? \rangle\}] [r := S(\phi)]
\]

\[
\Psi; \Delta \vdash \text{mmalloc} \phi, r, n : \Gamma \rightarrow [\phi] \Gamma'
\]

\[
\Gamma(hp) = C \otimes \{\ell \mapsto \langle \tau_1, \ldots, \tau_n \rangle\} \quad \Gamma' = \Gamma[hp := C]
\]

\[
\Psi; \Delta \vdash \text{free} r : \Gamma \rightarrow [] \Gamma'
\]

\[
\Gamma(hp) = C \otimes \{\ell \mapsto \langle \tau_1, \ldots, \tau_k, \ldots, \tau_n \rangle\} \quad \Gamma' = \Gamma[hp := C \otimes \{\ell \mapsto \langle \tau_1, \ldots, \tau, \ldots, \tau_n \rangle\}]
\]

\[
\Psi; \Delta \vdash \text{st} r_d(k), r_s : \Gamma \rightarrow [] \Gamma'
\]

\[
\Gamma(hp) = C \otimes \{\ell \mapsto \langle \tau_1, \ldots, \tau_k, \ldots, \tau_n \rangle\}
\]

\[
\Psi; \Delta \vdash \text{id} r_d, r_s(k) : \Gamma \rightarrow [] \Gamma
\]

The store type may not match a given instruction precondition syntactically, so we must introduce the following rule to prove the store has the form required at different program points.

\[
\Gamma \leq \Gamma'
\]

\[
\Psi; \Delta; \Gamma \vdash i : \Gamma \rightarrow [] \Gamma'
\]
Comments

• *Singleton types* allow us to identify pointers and their aliases.

• *Label polymorphism* allows us to abstract away from the specific name of a label but retain the aliasing structure of the heap

• *Heap polymorphism* allows us to abstract away from the size and shape of a portion of the heap

• With recursive and existential types, we can encode linear lists and trees. (See Walker and Morrisett [25])

• We can extend our type system to incorporate a Turing-complete logic provided we annotate our programs with explicit proofs of the entailment relation. (See Reynolds [16] and Ishtiaq and O’Hearn [9])
Arrays

- Often, using some simple arithmetic facts we can prove that an array access is in bounds at compile time, eliminating the need for a check at run time

- Following Xi, Pfenning and Harper ([28, 27]), we may extend the type checker with a (classical) logic for reasoning about arithmetic, just as we used a (linear) logic for reasoning about the heap

- Arithmetic expressions:

\[ a ::= i \mid n \mid a_1 +_{32} a_2 \mid a_1 -_{32} a_2 \mid a_1 \times_{32} a_2 \mid a_1 \text{ xor } a_2 \mid \cdots \]

  - \( i \) is a 32-bit number variable
  - \( n \) is a 32-bit constant
  - All expressions have machine semantics

- Logical connectives:

\[ P ::= p \mid \text{true} \mid \text{false} \mid a_1 \preceq u a_2 \mid P_1 \supset P_2 \mid P_1 \land P_2 \mid \neg P \mid \cdots \]

- New types:
  
  - Singleton integers: \( S(a) \)
  
  - Array types: \( \tau \text{ array}(a) \)
Refined Operand Typing

- New type contexts:

  \[ \Delta ::= \cdot \mid \Delta, \alpha :: \kappa \mid \Delta, P \]

- New operands: \( v[\text{proof}] \)
  - \( v \) must be code with a logical precondition: \( \forall [P, \Delta'].\Gamma' \)
  - \( v[\text{proof}] \) has type \( \forall [\Delta'].\Gamma' \) provided that \( \text{proof} \) is a proof of \( P \) in the current context:

    \[
    \Psi; \Delta; \Gamma \vdash v : \forall [P, \Delta'].\Gamma' \rightarrow \{ \} \quad \Delta \vdash \text{proof} : P \text{ true} \\
    \hline
    \Psi; \Delta; \Gamma \vdash v[\text{proof}] : \forall [\Delta'].\Gamma' \rightarrow \{ \} 
    \]

  - For the sake of brevity, we will omit such proofs from our examples (alternatively, we could assume that a theorem prover is able to reconstruct the proof without help)
  - we write instead

    \[ v[\cdot] \]

- We give constant integers a more refined type:

  \[ \Psi; \Delta; \Gamma \vdash n : S(n) \]
Refined Instruction Typing

- Instruction typing judgment:

$$\Psi; \Delta \vdash i : \Gamma \rightarrow [\Delta'] \Gamma'$$

- Addition:

$$\Psi; \Delta; \Gamma \vdash r_2 : S(a_2) \quad \Psi; \Delta; \Gamma \vdash r_3 : S(a_3)$$

$$\Psi; \Delta \vdash \text{add} r_1, r_2, r_3 : \Gamma \rightarrow \Gamma[r_1 := S(a_2 + 32 a_3)]$$

- Array access:

$$\Psi; \Delta; \Gamma \vdash r_2 : \tau \text{array}(a) \quad \Psi; \Delta; \Gamma \vdash r_3 : S(a_3)$$

$$\Delta \vdash a_3 \leq_u a \text{ true}$$

$$\Psi; \Delta \vdash \text{ld} r_1, r_2(r_3) : \Gamma \rightarrow \Gamma[r_1 := \tau]$$

- As with operands, we could annotate load instructions with a proof of the arithmetic inequality above:

$$\text{ld} r_1, r_2(r_3)[\text{proof}]$$

- Conditional branches

$$\Psi; \Delta; \Gamma \vdash v : \forall [P].\Gamma \rightarrow \{ \} \quad \Psi; \Delta; \Gamma \vdash r : S(a)$$

$$\Delta, a \leq 0 \vdash P \text{ true}$$

$$\Psi; \Delta \vdash \text{ble} r, v : \Gamma \rightarrow [a > 0] \Gamma$$
Outline

- TAL-0: Assembly Language Control Flow and Basic Types
- TAL-1: Parametric Polymorphism
- TAL-2: Stack Types
- Type-Directed Compilation: From Tiny to TAL-2
- TAL-3: Data Structures
- TAL-4: Dependency
- TAL-5: Modularity and Linking
- References
Separate Compilation and Linking

- TAL provides mechanisms that allow program parts to be compiled separately, checked for compatibility and linked together to form an executable

- Such functionality is important in almost any programming environment but indispensable in a setting of mobile code and extensible systems

- TAL provides facilities for static linking (all components are assembled before executing the program)
  
  - See Glew and Morrisett [5]

- TAL also provides facilities for dynamic linking (components are loaded into a running program)
  
  - See Hicks, Weirich and Crary [8]

- Here, we concentrate on static linking
Linking Diagram
Example

fact_e.tali:

VAL factrec: $\forall \rho. \{ sp : \rho, r_1 : \text{int},$

$\quad r_{31} : \{ r_1 : \text{int}, sp : \rho \} \rightarrow \{ \} \} \rightarrow \{ \}$

fact.tal:

EXPORT fact_e.tali

factrec: $\forall \rho. \{ sp : \rho, r_1 : \text{int},$

$\quad r_{31} : \{ r_1 : \text{int}, sp : \rho \} \rightarrow \{ \} \} \rightarrow \{ \}$

sub $r_3, r_1, 1$

ble $r_3, L1[\rho]$

jmp $r_{31}$

$L1: \quad \forall \rho. \{ sp : \rho, r_1 : \text{int}, r_3 : \text{int},$

$\quad r_{31} : \{ r_1 : \text{int}, sp : \rho \} \rightarrow \{ \} \} \rightarrow \{ \}$

salloc 2

sst $r_{31}, 0$

...
Example Continued

stdio_e.tali:

TYPE file
VAL fprintf: ...
...

main_i.tali:

TYPE file
VAL fprintf: ...
VAL factrec: ...

main_e.tali:

VAL main: ...

main.tal:

IMPORT main_i.tali
EXPORT main_e.tali

main: ...
...
...  jmp factrec
Comments

- At the assembly language level:
  - Each implementation file (.tal file) defines a collection of types and values.
  - Each implementation file also declares a collection of imports and exports.
  - Each interface file (.tali file) declares a collection of values with their types and types with their kinds.
  - Our convention is that foo_i.tal files contain the imports needed by foo.tal and foo_e.tal files contain the exports.

- At the machine code level:
  - .tal files are replaced by .o files, which contain binary code and data and .to files, which contain a compressed binary representation of the associated typing annotations.
Link Checking

- Before linking, we check:
  - If one file imports a value labeled *foo* and the other file exports a value labeled *foo*, does *foo* have the type expected by the importing file?
  - Similarly, do import and export type declarations with the same name have the same kind (in our simple case: do stack types match stack types and ordinary types match ordinary types)?
  - Are there any import/export name clashes?
  - Note that unexported labels will not clash with labels from other files since they alpha-vary

- Before attempting execution, we check:
  - Are there any remaining types or values to import?
References


