**Typed Assembly Languages**

COS 441, Fall 2004
Frances Spalding

Based on slides from Dave Walker and Greg Morrisett

---

**Type Safety**

- **Progress**: if a state is well-typed then it is not stuck
  
  \[ \Gamma \vdash e : \tau \quad \text{then either } e \text{ is a value or} \exists e' \text{ such that } e \rightarrow^* e' \]

- **Preservation**: each step in evaluation preserves typing
  
  \[ \Gamma \vdash e : \tau \quad \text{and} \quad e \rightarrow^* e' \quad \text{then} \quad \Gamma \vdash e' : \tau \]

---

**High Level vs. Low Level**

Type Safe Source Code

- If the source language has been proved to be type safe, what do we know about the safety of the code we actually run?
- Nothing!
- What if the compiler produced a typed assembly language?

---

**Proof Carrying Code**

- The code producer provides a proof of safety along with the executable
- The code consumer checks that proof before executing the code
- A typing derivation of the low-level code can be a safety proof

---

**High-level code for factorial**

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

---

**Assembly code for factorial**

```
; r1 holds n, r2 holds a, r3 holds return address
; i.e. expects the result in r3
fact:  
exr1, r2     ; if n <= 0 go to l2
mul r2, r1, r1 ; n := n \times n
sub r1, r1, l 
jeq fact ; n := n-1
jrp fact ; i.e. jump to return address
l2:  
exr1, r2     ; a := r1
jrp r3     ; i.e. jump to return address
```
### TAL₀ Syntax
- **Registers:** \( r_1, r_2, \ldots \)
- **Labels:** \( L, \ldots \)
- **Integers:** \( n \)
- **Operands:** \( v ::= r | n | L \)
- **Arithmetic Ops:** \( \text{aop} ::= \text{add} | \text{sub} | \text{mul} | \ldots \)
- **Branch Ops:** \( \text{bop} ::= \text{beq} | \text{bgt} | \ldots \)
- **Instructions:** \( i ::= \text{aop } r, r_v | \text{bop } r, v | \text{mov } r, v \)
- **Blocks:** \( B ::= \text{jmp } v | 1; B \)

### TAL₀ Dynamic Semantics
- **Machine State** \( \Sigma = (H, R, B) \)
- \( H \) maps labels to basic blocks
- \( R \) maps registers to values (integers and labels)
- \( B \) is a basic block (corresponding to the current program counter)
- Model evaluation as a transition function mapping machine states to machine states: \( \Sigma \mapsto \Sigma \)

### TAL₀ Static Semantics
- \( \tau ::= \text{int} | \Gamma \rightarrow \{ \} \)
- \( \Gamma ::= \{ r_1 : \tau_1, r_2 : \tau_2, \ldots \} \)
- Code labels have type \( \{ r_1 : \tau_1, r_2 : \tau_2, \ldots \} \rightarrow \{ \} \)
- To jump to code with this type, \( r_i \) must contain a value of type \( \tau \), etc
- Notice that functions never return – the code block always ends with a jump to another label

### Example Program with Types
```
% r3 holds n, r2 holds a, r1 holds return address
% which segfaults the compiler in r2
fact:  \{ r2 : int, r3 : int, r4 : \{ r1 : int \} \} \rightarrow \{ \}
      blt r2, L2 % if a <= 0 goto L2
      add r2, r2, r1 % a := a * x
      sub r2, r2, 1 % a := a - 1
      jmp fact % goto fact
L2:  \{ r1 : int, r2 : \{ r1 : int \} \} \rightarrow \{ \}
      mov r1, r2 % return := x
      jmp r31 % jump to return address
```
Basic Typing Judgments

- $\Psi; \Gamma \vdash n : \text{int}$
- $\Psi; \Gamma \vdash r : \Gamma(r)$
- $\Psi; \Gamma \vdash L : \Psi(L)$

Subtyping on Register Files

- Our program will never crash if the register file contains more values than necessary to satisfy some typing precondition
- Width subtyping:
  \[
  \{r_1 : \tau_1, \ldots, r_{i-1} : \tau_{i-1}, r_i : \tau_i\} < \{r_1 : \tau_1, \ldots, r_{i-1} : \tau_{i-1}\}
  \]

Subtyping on Code Types

- Like ordinary function types, code types are contravariant

Typing Instructions

- The typing judgment for instructions describes the registers on input to the function and the registers on output

Typing Instructions

- All basic blocks end with jump instructions

Instruction Sequences

- $\Psi \vdash \text{jmp } v : \Gamma \Rightarrow \{\}$

- $\Psi \vdash \text{jmp } v : \Gamma \Rightarrow \{\}$

Basic Block Typing

- Instruction Sequences

- $\Psi \vdash \text{jmp } v : \Gamma \Rightarrow \{\}$

- $\Psi \vdash \text{jmp } v : \Gamma \Rightarrow \{\}$
Heap and Register File Typing

- **Heap Typing**
  \[
  \text{Dom}(H) = \text{Dom}(\Psi) \quad \forall L \in \text{Dom}(H) \cdot \Psi \vdash H(L) : \Psi(L)
  \]

- **Register File Typing**
  \[
  \forall r \in \text{Dom}(\Gamma) \cdot \Psi, \Gamma \vdash R(r) : \Gamma(r)
  \]

Machine Typing

- **Main Typing Judgment**
  \[
  \vdash H : \Psi \\
  \vdash R : \Gamma \\
  \vdash B : \Gamma \rightarrow \{\}
  \]

Type Safety for TAL\(\alpha\)

- **Progress:**
  If \(\vdash \Sigma_1\) then \(\exists \Sigma_2\) such that \(\Sigma_1 \rightarrow \Sigma_2\)

- **Preservation**
  If \(\vdash \Sigma_1\) and \(\Sigma_1 \rightarrow \Sigma_2\) then \(\vdash \Sigma_2\)

Limitations of TAL\(\alpha\)

- **What about situations where we don’t care which specific type is in a register?**
  - Eg. A function that swaps the contents of two registers

- **We can use polymorphic types to express this.**

TAL\(\alpha\) : Polymorphism

- \(\tau ::= \ldots | \alpha\)

- \(\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 \(r_1\) and \(r_2\) for values of any two types

- **To jump to polymorphic functions, first explicitly instantiate type variables: \(v[\tau]\)**

Polymorphism Example

- \(v[\mu_\alpha \beta. (r_1 : m, r_2 : n, r_3 : k, r_4 : l) \rightarrow \{\} \rightarrow \{\}
  \]
  - Describes a function that swaps the values in \(r_1\) and \(r_2\) for values of any two types

- **To jump to polymorphic functions, first explicitly instantiate type variables: \(v[\tau]\)**
Callee-Saved Registers

- We can use polymorphism to implement callee-saved registers
  - We'd like to be able to enforce that a function doesn't change certain registers. In other words, when it returns, certain registers have the same values as when it started.
  \[ \forall \alpha. \{ r_2: \text{int}, ..., r_5: \alpha, ..., r_{31}: \{ r_1: \text{int}, ..., r_5: \alpha, ... \} \} \rightarrow \{ \} \rightarrow \{ \} \]
- Why does this enforce the invariant we want?

Full TAL Syntax

- Types: \( \tau \equiv \alpha \mid \text{int}[\alpha] \mid \{\tau_1, \ldots, \tau_n\} \mid \text{int} \)
- Executing flags: \( \nu \equiv 0 \mid 1 \)
- Local types: \( \Psi \equiv \{ r_{i_1}, \ldots, r_{i_k} \} \)
- Register file types: \( \Gamma \equiv \{ r_{i_1}, \ldots, r_{i_k} \} \)
- Type contexts: \( \Delta \equiv \delta \)
- Registers: \( r \equiv \{ r_{i_1}, r_{i_2}, \ldots \} \)
- Variables: \( x \equiv \{ \tau \mid \nu \} \cdot \text{pred} \{ r_\alpha \} \mid \tau \}
- Heap made: \( h \equiv (a_1, \ldots, a_k) \mid \text{caddr}\{ \tau \} \)
- Heap contexts: \( H \equiv (r_1 = h_1, \ldots, r_k = h_k) \)
- Register file contexts: \( R \equiv (r_1 = r_2, \ldots, r_k = r_{k-1}) \)
- Instructions: \( t \equiv \text{salloc} n; \text{sld} r_d n \mid \text{sld} r_s n \mid \text{sst} v \mid \text{sam} \mid \text{bfree} \mid \text{bld} \)
- Instruction sequences: \( S \equiv i_j \mid \text{push} \mid \text{bjump} \)
- Programs: \( P \equiv (H, R, S) \)

TAL Semantics

- Changes to the static semantics to support polymorphism
  - Add the judgment \( \Delta \vdash \tau \text{ ok} \)
  - Modify the other judgments to take this into account (just like homework 8!)
- Changes to the dynamic semantics
  - Add rule to do instantiation

Limitations of TAL

- Almost every compiler uses a stack
  - As storage space for local variables and other temporaries when we run out of space in registers
  - To keep track of control flow
    - Control Links
    - Activation Links

STAL: Add a Stack

  - Stack \( S := \text{nil} \mid v :: S \)
  - A designated register sp points to the top of the stack
- 1. \(:= \text{salloc} n \mid \text{sfree} n \mid \text{sld} r_d n \mid \text{sst} r_s n \)
  - push \( v \equiv \text{salloc} 1; \text{sld} v, 1 \)
  - pop \( v \equiv \text{sld} r_1, 1 \); \text{sfree} 1

Factorial Example

- STAL code:
  - \text{salloc} n \text{ sld} r_d n \text{ sst} r_s n \text{ push} v \text{ sld} r_d 1 \text{ sfree} v

Full TAL Syntax

- Types: \( \tau \equiv \alpha \mid \text{int}[\alpha] \mid \{\tau_1, \ldots, \tau_n\} \mid \text{int} \)
- Executing flags: \( \nu \equiv 0 \mid 1 \)
- Local types: \( \Psi \equiv \{ r_{i_1}, \ldots, r_{i_k} \} \)
- Register file types: \( \Gamma \equiv \{ r_{i_1}, \ldots, r_{i_k} \} \)
- Type contexts: \( \Delta \equiv \delta \)
- Registers: \( r \equiv \{ r_{i_1}, r_{i_2}, \ldots \} \)
- Variables: \( x \equiv \{ \tau \mid \nu \} \cdot \text{pred} \{ r_\alpha \} \mid \tau \}
- Heap made: \( h \equiv (a_1, \ldots, a_k) \mid \text{caddr}\{ \tau \} \)
- Heap contexts: \( H \equiv (r_1 = h_1, \ldots, r_k = h_k) \)
- Register file contexts: \( R \equiv (r_1 = r_2, \ldots, r_k = r_{k-1}) \)
- Instructions: \( t \equiv \text{salloc} n; \text{sld} r_d n \mid \text{sld} r_s n \mid \text{sst} v \mid \text{sam} \mid \text{bfree} \mid \text{bld} \)
- Instruction sequences: \( S \equiv i_j \mid \text{push} \mid \text{bjump} \)
- Programs: \( P \equiv (H, R, S) \)
Extensions for STAL

- Stack types \( \sigma ::= \text{nil} | \tau : \sigma | \rho \)
  - \( \text{nil} \) represents the empty stack
  - \( \tau : \sigma \) represents a stack \( v : S \) where \( v : \tau \) and \( S : \sigma \)
  - \( \rho \) is a stack variable which we can use to describe an unknown stack tail
- Use polymorphism
  \[ \forall \rho. \{ \text{sp} : \text{int} ; r_1 : \text{int} \} \rightarrow \{ \} \]

Stack Instruction Typing

- Stack load
  \[ \Gamma(\text{sp}) = \tau_1 :: \ldots :: \tau_n :: \sigma \]
  \[
  \Psi; \Delta \vdash \text{sld} \ r, \ n : \Gamma \rightarrow \Gamma[\text{r} := \tau_n]
  \]

- Stack store
  \[ \Psi; \Delta; \Gamma \vdash \text{sst} \ v, \ n : \Gamma(\text{sp}) = \tau_1 :: \ldots :: \tau_n :: \sigma \]
  \[
  \Psi; \Delta \vdash \text{sst} \ v, \ n : \Gamma[\text{sp} := \tau_1 :: \ldots :: \tau_n :: \sigma] \rightarrow \Gamma[\text{sp} := \sigma]
  \]

STAL Features

- Polymorphism and polymorphic recursion are crucial for encoding standard procedure call/return
  - We didn’t have to bake in the notion of procedure call/return. Basic jumps were good enough
  - Can combine features to encode up a variety of calling conventions
    - Arguments on stack or in registers?
    - Results on stack or in registers?
    - Return address? Caller pops? Callee pops?
    - Caller saves? Callee saves?

STAL Syntax

- Types
  \[ \tau ::= \epsilon | \text{T} \]
- Stack types
  \[ \sigma ::= \rho | \text{nil} | \tau : \sigma | \rho \]
- Register assignments
  \[ \Delta ::= \beta \]
- Register file
  \[ R ::= (r_1 := w_1, \ldots, r_n := w_n, \text{sp} := s) \]
- Stacks
  \[ S ::= \text{nil} | \text{sp} : S \]
- Instructions
  \[ i ::= \text{salloc} \ n | \text{sfree} \ a | \text{sp} \ r_1, \text{sp} \ r_2 | \text{not} \ r_1, \text{sp} \ r_2 | \text{add} \ r_1, \text{sp} \ r_2 \]

General Limitations of TAL/STAL

- More functionality can be added to TAL
- There are still some limitations
  - Must know the order of data allocated on the stack
    - sp: \( r_1 \), sp: \( r_2 \), sp: \( r_3 \)
  - Must distinguish between heap and stack pointers
    - Can’t have a function that takes in two generic int pointers and adds their contents
For more information on TAL/STAL

- **From System F to Typed Assembly Language**, G. Morrisett, D. Walker, K. Crary, and N. Glew.
- **Stack-Based Typed Assembly Language**, G. Morrisett, K. Crary, N. Glew, D. Walker.

What Makes Stack Typing So Tricky?

- In the heap, we use the abstraction that a location never changes type
  - The garbage collector handles the reuse
- When modeling instructions like push/pop, we have to break this abstraction and use strong updates to change types
- Aliasing can make this unsound if we’re not careful

My Research

- A different approach to typed assembly language
- Use logic to describe what memory looks like before and after each instruction
  
  \[
  (r_1 \Rightarrow \text{int}) \otimes (r_2 \Rightarrow S(f)) \otimes (f \Rightarrow \text{bool}) \\
  \text{load}[r_3], r_1 \\
  (r_1 \Rightarrow \text{int}) \otimes (r_2 \Rightarrow S(f)) \otimes (f \Rightarrow \text{int})
  \]

Modeling Stack “Evolution”

- Keep track of the different “versions” of each stack location and how they are related
- We can do this using a pair of a tree and a pointer to a node in the tree

Tagged Locations

- All locations are “tagged” with their version number
  - \( (r_1 \Rightarrow S(k_2, f_2)) \) is different from \( (r_1 \Rightarrow S(k_2, f_2)) \)
- The tags act as “guards” on the location
  - Before accessing a tagged location, you must prove that it is “live”
  - In other words, the tag must be on the path between the root and Top